Universität des Saarlandes
Naturwissenschaftlich-Technische Fakultät I
Fachrichtung Informatik

Abstracting Cryptographic Protocols

Esfandiar Mohammadi

Dissertation
zur Erlangung des Grades
des Doktors der Naturwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Saarbrücken
Eingereicht: Oktober 2014

Thesis for obtaining the title of Doctor of Natural Sciences of the Faculties of Natural Sciences and Technology of Saarland University

Dissertation zur Erlangung des Grades des Doktors der Naturwissenschaften der Naturwissenschaftlich-Technischen Fakultäten der Universität des Saarlandes

Reporters / Berichterstattende

Prof. Dr. Michael Backes (Universität des Saarlandes & MPI-SWS)

Prof. Dr. Dominique Schröder (Universität des Saarlandes)

Prof. Dr. Dennis Hofheinz (Karlsruher Institut für Technologie)

Dean / Dekan

Prof. Dr. Markus Bläser

Examination Board / Prüfungsausschuss

Prof. Dr. Markus Bläser, chair (Universität des Saarlandes)

Prof. Dr. Michael Backes (Universität des Saarlandes & MPI-SWS)

Prof. Dr. Dominique Schröder (Universität des Saarlandes)

Prof. Dr. Dennis Hofheinz (Karlsruher Institut für Technologie)

Giancarlo Pellegrino, PhD

Date of the Colloquium / Tag des Kolloquiums

29 June 2015

Zusammenfassung

Kryptographische Protokolle haben den Zweck die Sicherheit von IT Systemen zu härten, allerdings sind diese Protokolle nur dann wirksam, wenn sie sorgfältig in ein System eingearbeitet werden. Da kryptographische Sicherheitsbeweise, eine Standardtechnik für Sicherheitsbeweise, inhärent komplex und fehleranfällig sind, gibt es erfolgreiche Ansätze zur modularen Abstraktion von kryptographischen Protokollen.

Die vorliegende Arbeit analysiert die Fehlerfreiheit von zwei Arten von Abstraktionen: ideale Funktionalitäten, ein Szenario in dem die ehrlichen Parteien als unkorrumpierbare Maschine mit einem geteilten Speicher dargestellt werden, und symbolische Abstraktionen, ein Szenario in dem als Berechnungsmodell ein symbolisches Kalkül betrachtet wird und kryptographische Operationen mittels simplen, formalen Regeln charakterisiert werden.

Die Fehlerfreiheit von symbolischen Abstraktionen wird Computational Soundness genannt. Wir präsentieren Resultate, um Wiederbenutzbarkeit von existierenden Computational Soundness Beweisen zu ermöglichen, insbesondere um Garantien für starke äquivalenzbasierte Eigenschaften zu erhalten.

Wir benutzen ideale Funktionalitäten, um anonyme Kommunikationstechniken zu analysieren. Wir analysieren die Sicherheit von Tor (dem meistegenutzten Anonymitätsnetzwerk) zugrundeliegenden Protokolls und schlagen Techniken zur Verbesserung der Leistung des Tor Netzwerkes vor.

Abstract

Cryptographic protocols can be used to harden the security of IT systems, but only if these protocols are carefully woven into the system. Thus, a security analysis of the overall system is necessary to ensure that the cryptographic protocols are effectively applied. The classical proof technique for computational security proofs (reduction proofs) are, due to their complexity, hard to automatically derive, already for medium-sized IT systems. Hence, there is a successful line of research about abstracting cryptographic protocols.

In this thesis, we consider two kinds of abstractions: *i*) symbolic abstractions, where formal rules characterize the attacker's capabilities and the functionality of the cryptographic operations, and *ii*) ideal functionalities, where all honest parties are replaced by one single incorruptible machine.

For symbolic abstractions, we study their accuracy (called computational soundness), i.e., under which conditions these abstractions capture all cryptographic attacks. We establish a computational soundness result for malleable zero-knowledge proofs, and we establish two results for re-using existing computational soundness proofs, in particular for obtaining strong equivalence properties.

Additionally, we devise an ideal functionality for the most-widely used anonymous communication protocol Tor and (using the UC framework) prove its accuracy. For improving Tor's performance, we moreover propose a novel, provably secure key-exchange protocol.

Background of this Dissertation

The present thesis is based on papers that were written during my doctorate under the supervision of Michael Backes, in the IS&C group of CISPA at the Saarland University. The thesis is based on six research papers [BBMMP15; BGKM12; BKM12; BMM10; BMR14; BMR15]. Whenever a chapter of this thesis is based on a publication, a brief disclaimer [in square parenthesis and marked in green] mentions the respective publication and clarifies my contribution to the part was that is presented in the respective chapter.

The paper on computational soundness for malleable zero-knowledge proofs [BBMMP15] extends the work from the present thesis (in Chapter 3) to equivalence properties and uses a more restrictive and simpler symbolic model from the PhD thesis of Kim Pecina. The work in the present thesis is in turn a substantial extension of the my master's thesis [Moh09] to a significantly more general setting and more precise symbolic abstraction. Moreover, the work on generalizing computational soundness for interactive primitives [BMR15] (on which Chapter 5 is based) builds on the bachelor's thesis of Tim Ruffing [Ruf12], which I supervised.

The work on on anonymous communication [BGKM12] (on which Chapter 7 is based) serves as a foundational building block of anonymity quantifications for three papers to which I contributed [BKMM14; BKMMM13; BMM14].

Research papers on which this thesis is based

[BMM10] M. Backes, M. Maffei, and E. Mohammadi. "Computationally Sound Abstraction and Verification of Secure Multi-Party Computations". In: *Proc. 30th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 352–363.

[BGKM12] M. Backes, I. Goldberg, A. Kate, and E. Mohammadi. "Provably Secure and Practical Onion Routing". In: *Proc. 25th IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society Press, 2012, pp. 369–385.

[BKM12] M. Backes, A. Kate, and E. Mohammadi. "Ace: an efficient key-exchange protocol for onion routing". In: *Proc. 11th annual ACM Workshop on Privacy in the Electronic Society (WPES)*. ACM Press, 2012, pp. 55–64.

[BMR14] M. Backes, E. Mohammadi, and T. Ruffing. "Computational Soundness Results for ProVerif". In: *Proc. 3rd Conference on Principles of Security and Trust (POST)*. Springer, 2014, pp. 42–62.

[BBMMP15] M. Backes, F. Bendun, M. Maffei, E. Mohammadi, and K. Pecina. "A Computationally Sound, Symbolic Abstraction for Malleable Zero-knowledge Proofs". In: *Proc. 28th IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society Press, 2015, pp. 412–480.

[BMR15] M. Backes, E. Mohammadi, and T. Ruffing. "Computational Soundness for Interactive Primitives for Equivalence Properties". In: *Proc. 20th European Symposium on Research in Computer Security (ESORICS)*. Springer, 2015, pp. 125–145.

Other research papers of the author

[BFM13] M. Backes, D. Fiore, and E. Mohammadi. "Privacy-Preserving Accountable Computation". In: *Proc. 18th European Symposium on Research in Computer Security (ESORICS)*. Springer, 2013, pp. 38–56.

[BKMMM13] M. Backes, A. Kate, P. Manoharan, S. Meiser, and E. Mohammadi. "AnoA: A Framework For Analyzing Anonymous Communication Protocols". In: *Proc. 26th IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society Press, 2013, pp. 163–178.

[BKMM14] M. Backes, A. Kate, S. Meiser, and E. Mohammadi. "(Nothing else) MATor(s): Monitoring the Anonymity of Tor's Path Selection". In: *Proc. 21st ACM Conference on Computer and Communication Security (CCS)*. ACM Press, 2014, pp. 513–524.

[BMM14] M. Backes, P. Manoharan, and E. Mohammadi. "TUC: Time-sensitive and Modular Analysis of Anonymous Communication". In: *Proc. 27th IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society Press, 2014, pp. 383–397.

[PMP14] K. Pecina, E. Mohammadi, and C. Pöpper. "Zero-Communication Seed Establishment for Anti-Jamming Techniques". In: *Proc. 1st NDSS Workshop on Security of Emerging Networking Technologies (SENT)*. Internet Society, 2014.

Related Bachelor's & Master's Thesis

[Moh09] E. Mohammadi. *Computational Soundness for Symbolic Zero - Knowledge Proofs Against Active Attackers under Relaxed Assumptions*. Master's Thesis. 2009.

[Ruf12] T. Ruffing. *Computational Soundness of Interactive Primitives*. Bachelor's Thesis. 2012.

Acknowledgements

First and foremost, I would like to thank my wife, Melanie Mohammadi, for all the patient support and love throughout the last years.

During my doctorate, my supervisor Michael Backes constantly pushed forward and supported my work and ideas. He clarified to me time and time again that successful long-term research needs a clear vision as a steering wheel. Thank you for all of that and for your huge support.

The group's secretary, Bettina Balthasar, was one of the most valuable persons during my doctorate. Her mindful and caring support protected me numerous times from the university's bureaucracy. She always had an open door for any non-scientific problems, and brightened every day with her good mood and colorful (and tasteful) clothing.

I would like to thank Dominique Unruh for teaching me the handicraft of weaving complex proofs. In numerous enlightening discussions Dominique convinced me that all too often corner cases elude a hand-wavy argumentation and that a solid proof typically has a clear line of argumentation.

Rose Hoberman taught me the handicraft of writing. Countless times she intensely explained to me why some write-up was confusing or lacked emphasis. Her explanations were priceless. Thank you for these invaluable lessons.

The entire area of anonymous communication was introduced to me by Aniket Kate. He illustrated to me in various discussions that anonymous communication is an inherently hard problem but fundamental for numerous applications. Thank you, and thank you also for your steady support when you, without hesitation, introduced me to various interesting fellow researchers, be it on conferences or when we had visitors.

Like a second supervisor Matteo Maffei always had an open door for research ideas and problems that I encountered. Thank you for your support and for the countless discussions.

Special thanks are due to my long-standing office-mate Raphael Reischuk. I enjoyed our time together, and I will miss our inspiring discussions. I learnt much from you.

I am grateful to have met my colleague Sebastian Meiser in my undergraduate studies. We have not only designed – in our (almost non-existent) research-free time – one of the best strategic board games that I am aware of but also had numerous invaluable scientific discussions that were extremely fruitful.

I would also like to thank my other co-authors, Praveen Manoharan, Fabian Bendun, Tim Ruffing, Kim Pecina, Christina Pöpper, Ian Goldberg, Dario Fiore, Christian Rossow, and Simon Koch, for all the insightful and productive discussions. In addition, I would like to thank all my colleagues in the CISPA for the good working environment and for many inspiring discussions. I also would like to thank my fellow students, Arnd Hartmanns and Sebastiano Barbieri, for all the evening we spent together. In particular, thank you Arnd for the curly-leafed kale evenings. Your curly-leafed kale dish is unchallenged.

Of course, I also thank my family for all the support. It is hard to be so far away from you. I hope that future generations have such a thorough connectivity such that they will not have to leave their families for the working place.

Last, thank you Saarland for the warm embrace and the investment into my education and for giving me such a good starting point for my career.

Contents

Chapter 1.

Introduction

Modern IT systems tremendously increase the cost- and time-efficiency of even sensitive tasks, such as online transactions, e-voting, and sharing health-related information. In particular for such sensitive tasks, however, the trust into these IT systems is essential. Cryptographic protocols, e.g., TLS [DR06] or Kerberos [MIT], are a fundamental building block for strengthening the security of IT systems and thereby increasing the trust in them.

Interactive (i.e., cryptographic protocols) and non-interactive cryptographic primitives (e.g., encryption) have to be carefully incorporated into a system. Recent events [App14; Duc14] have shown that already minor mistakes in the usage of cryptographic primitives can have disastrous effects on the security of a system. Due to the randomized nature and the complex mathematical operations involved in cryptographic primitives, it is highly non-trivial to conduct a security proof of already medium-sized systems.

There is a successful line of work that abstracts cryptographic primitives as simpler, idealized operations that typically do not involve any cryptography. These abstractions are often deterministic and limit the attacker's capabilities to a small set of actions. For cryptographic primitives there are two kinds of abstractions: ideal functionalities, i.e, incorruptible machines where the involved parties have a shared memory, or symbolic abstractions in a symbolic model, e.g., the applied π-calculus, where, e.g., the encryption of a message m with a key k, is a symbolic term $enc(k, m)$ and decryption follows simple rules ($dec(k, enc(k, m)) = m$). For symbolic abstractions the security proofs state that the symbolic abstraction captures all attacks. This accuracy of an abstraction is called *computational soundness*.

While symbolic abstractions enable the automated verification of entire classes of protocols that use the abstracted cryptographic primitive, the symbolic verification tools for security verification are in some cases not sufficiently expressive: e.g., symbolic models, used for security proofs, typically only characterize computations without randomized decisions. There are important classes of cryptographic protocols that include randomized decisions, such as the anonymous communication network Tor. Anonymous communication constitutes a core building block for privacy preserving IT systems. For the abstraction of such a cryptographic protocol, an ideal functionality is needed.

This thesis contains proofs of security for both kinds of abstractions: Dolev-Yao-style abstractions and ideal functionalities. In Part I we present five results on computational soundness, and in Part II we present two results on anonymous communication, using an ideal functionality.

Computational Soundness

A security proof typically reduces the security properties of a protocol, such as secrecy of RSA-OAEP, to hardness assumptions, such as factoring. These security proofs are typically manually conducted in the *computational model* in which both the protocol and the attacker are modeled as interactive polynomial-time Turing machines. Such manual security proofs are complex and error-prone, and vulnerabilities have accompanied the early design of security protocols [BCJSW06; Ble98; Fis03; NS78]. As these proofs are inherently complex and we increasingly rely on them, verification tools for proving the security of such protocols are indispensable.

Such verification tools have considerably advanced in recent years. A successful line of work on verification tools [BAF05; BHM08a; BJP02; BMU08; CCD10; CW12; DKR09; DKRS11; DY83; EG83; KMM94; MSCB13] uses, instead of the real underlying cryptographic primitive, an abstract representation, a so-called *symbolic model*: in a symbolic model all cryptographic primitives are represented as symbolic terms, such as $enc(k, m)$ for an encryption, and the attacker is reduced to a small set of simple rules, such as $dec(k, enc(k, m)) = m$ for decryption.

A symbolic model, however, is not necessarily sound with respect to the computational model: it might fail to model attacks that are possible in the computational model, such as computing the length of a plaintext from the length of a ciphertext. A symbolic model of a set of cryptographic primitives has *computational soundness* (CS) if all possible attacks in the computational model can be modeled in the symbolic model. In particular, a single CS result enables provable security results for an entire class of protocols: each successful verification, with verification tools like ProVerif [BAF05], Tamarin [MSCB13; SMCB12; SSCB14] or APTE [CCD10], yields a security result in the computational model.

Backes, Hofheinz, and Unruh developed a modular and generic framework for symbolic protocol analysis and computational soundness proofs: the CoSP framework [BHU09]. In CoSP, the embedding of the calculi (e.g., the applied π-calculus) is decoupled from computational soundness proofs of cryptographic primitives. As a result, CoSP enables proving x Dolev-Yao models computationally sound for y calculi with $x + y$ proofs instead of $x \cdot y$. This CoSP framework is the basis of this thesis.

The contribution of this thesis is fourfold: first, the first computational soundness result of a sophisticated primitive under weak cryptographic assumptions, malleable zero-knowledge proofs, second, the extension of the CoSP framework to equivalence properties, third, a computational soundness result that states that all symbolic models can be extended with composably secure interactive primitive, fourth the characterization of a property, called self-monitoring, that enables the conclusion that computational soundness for trace properties implies computational soundness for uniformity, i.e., the equivalence of protocols with the same control flow.

Computational Soundness for Malleable Zero-Knowledge Proofs. While symbolic models traditionally include only basic cryptographic primitives such as encryption and digital signatures, recent work has started to extend them to more sophisticated primitives, such as zero-knowledge (ZK) proofs, with unique security features. These security features go far beyond the traditional goal of cryptography to solely offer secrecy and authenticity of communication. ZK proofs [GMR89] arguably constitute the most

prominent such primitive (though not the only one[1]) and have become a central building block for a variety of modern security protocols. A zero-knowledge proof consists of a message or a sequence of messages that combines two seemingly contradictory properties: First, it constitutes a proof of a statement x (e.g, $x =$ "the message within this ciphertext begins with 0") that cannot be forged, i.e., it is impossible, or at least computationally infeasible, to produce a zero-knowledge proof of a wrong statement. Second, a zero-knowledge proof does not reveal any information other than the sole fact that x constitutes a valid statement.

In addition to these core properties, commonly used ZK proof schemes, such as the Groth-Sahai proof system [GS08], offer a novel type of cryptographic flexibility. First, a participant is able to re-randomize existing ZK proofs, which is fundamental for achieving unlinkability in anonymity protocols. Second, in order to adhere to individual privacy requirements, a participant can hide public parts of a ZK proof statement to selectively hide information of third-party proofs (e.g., this enables the design of privacy-preserving credentials for open-ended systems [BMP12; MP11; MPR13]). Third, a participant can logically compose ZK proofs in order to construct new proof statements. ZK proof systems that permit these transformations are called *malleable*. In addition to offering this extended functionality, malleable ZK constructions are often significantly more efficient than their non-malleable counterparts.

Existing symbolic abstractions are restricted to non-malleable ZK proofs, which model ZK proofs as monolithic building blocks that cannot be further transformed [BHM08b; BHM12; BMU08]. A symbolic model for malleable ZK proofs is intrinsically more difficult for automated verification techniques because the much more comprehensive adversary model that includes ZK transformations requires a significantly more involved symbolic analysis.

The symbolic models of non-malleable ZK proofs have been justified by computational soundness results, i.e., a successful symbolic analysis carries over to the corresponding cryptographic ZK realizations [BBU13; BU10]. The symbolic model of malleable ZK proofs imposes challenges for such a result due to the significantly more complex adversary model.

First, we provide a symbolic abstraction of malleable ZK (MZK) proofs by means of an equational theory.[2] The main conceptual challenge we faced when devising this abstraction was to identify a finite representation of the infinite number of possible transformations that are available to the adversary. Roughly, we categorize transformations as one of the three types: *re-randomizing*, *logical transformations* (used, e.g., to produce a proof of the statement $x \wedge y$ from independent proofs of x and y, or to prove $\exists w.x$ from a proof of x, thereby hiding the witness w in the statement x), and *functional transformations* (used, e.g., to prove $enc(k, x, r) = enc(k, y, r)$ from a proof of $x = y$ for two secret values x, y, key k, and randomness r).

The last category of transformations (i.e., functional transformations) is rarely used in the literature, but it is nevertheless available to the attacker, as shown by Fuchsbauer [Fuc10, Lemma 6]. Therefore, we present two variations of our symbolic model that only differ in this last category: the *fully MZK (FMZK) abstraction* grants the attacker the capability to apply transformations that modify the witnesses of a proof, which allows for weaker

[1] Examples for other primitives studied in symbolic models are blind-signatures [KR05], Diffie-Hellman-style exponentiation [AF01], and private contract signatures [KKW05].

[2] We further consider asymmetric encryptions and digital signatures, handled in a standard way [BHU09].

cryptographic realizations; the *controlled MZK (CMZK) abstraction* excludes this kind of transformations but requires a slightly less efficient cryptographic realization. Concerning automated verification, the CMZK abstraction is accessible to standard automated reasoning tools for equational theories, whereas reasoning about the FMZK abstraction additionally requires solving constraints, e.g., via a theorem prover.

Second, we prove the computational soundness of the FMZK and CMZK abstractions with respect to trace properties. We first identify the class of *MZK-safe* protocols, which basically disallows the reuse of randomness as well as revealing signature keys or decryption keys to the adversary. We then establish computational soundness of the FMZK abstraction for all MZK-safe protocols based on weak cryptographic definitions (non-interactive zero-knowledge arguments of knowledge). For establishing the computational soundness of the CMZK abstraction for all MZK-safe protocols, we leverage the cryptographic construction for controlled malleability proposed by Chase et al. [CKLM12]. These results are given in CoSP [BHU09], a modular and generic framework for symbolic protocol analysis and computational soundness proofs. The process of embedding calculi is decoupled from computational soundness proofs of cryptographic primitives. As a result, our work immediately entails a computationally sound symbolic model in the applied-pi calculus, and we show that our result also entails a computationally sound symbolic abstraction in ML (building on results from [BMU10]).

Extending the CoSP Framework to Equivalence Properties. A successful line of work showed over the past decade that numerous symbolic models are indeed computationally sound. Most of these CS results against active attacks, however, have been specific to the class of trace properties [BBU13; BCW13; BHU09; BU10; CKKW06; CW05; CW11; GGV08; JLM05; MW04], which is only sufficient as long as strong notions of privacy are not considered, e.g., in particular for establishing various authentication properties. Only few CS results are known for the class of equivalence properties against active attackers, which are restricted in of the following three ways: either they are restricted to a small class of simple processes, e.g., processes that do not contain private channels and abort if a conditional fails [CC08; CCS12; CH11], or they rely on non-standard abstractions for which it is not clear how to formalize any equivalence property beyond the secrecy of payloads [BL06; BP04; BPW03a], such as anonymity properties in protocols that encrypt different signatures, or existing automated tool support is not applicable [CHKS12; SBBPW06]. We are thus facing a situation where CS results, despite tremendous progress in the last decade, still fall short in comprehensively addressing the class of equivalence properties and protocols that state-of-the-art verification tools are capable to deal with. Moreover, it is unknown to which extent existing results on CS for trace properties can be extended to achieve more comprehensive CS results for equivalence properties.

The CoSP framework originally only modelled CS w.r.t. trace properties. As a basis for future work and our for other contributions, we extend this framework to equivalence properties.

A Composable Computational Soundness Result for Interactive Primitives. While Dolev-Yao models traditionally comprise only non-interactive cryptographic operations (i.e., cryptographic operations that produce a single message and do not involve any form of communication, such as encryption and digital signatures), recent cryptographic protocols rely on more sophisticated *interactive primitives* (i.e., cryptographic operations that involve several message exchanges among parties), with unique features that go far

beyond the traditional goals of cryptography to solely offer secrecy and authenticity of communication.

From previous [BBU13; BCW13; BHU09; BMM10; BU10; CKKW06; CW05; CW11; GGV08; JLM05; MW04], which is only sufficient as long as strong notions of privacy are not considered, e.g., in particular for establishing various authentication properties. Only few CS results are known for the class of equivalence properties against active attackers, and these results either do not cover interactive primitives [BL06; BP04; BPW03a; CC08; CCS12; CH11; CHKS12; SBBPW06] or do not allow to combine the DY model with non-interactive primitives [KTG12].

As a first step, we present an abstraction of a concrete interactive primitive, secure multi-party computation (SMPC), within the applied π-calculus [AF01]. In an SMPC, a number of parties $P_1, \ldots, P_n$ wish to securely compute the value $F(d_1, \ldots, d_n)$, for some well-known public function F, where each party P_i holds a private input d_i. This multi-party computation is considered secure if it does not divulge any information about the private inputs to other parties This abstraction consists of a process that receives the inputs from the parties involved in the protocol over private channels, computes the result, and sends it to the parties again over private channels, however augmented with certain details to enable computational soundness results, see below. This abstraction can be used to model and reason about larger cryptographic protocols that employ SMPC as a building block.

We establish computational soundness results (in the sense of preservation of trace properties) for protocols built upon our abstraction of SMPC. This result is obtained in essentially two steps: We first establish a connection between our symbolic abstraction of SMPC in the applied π-calculus (symbolic setting) and the notion of an ideal functionality for SMPC in the UC framework [CLOS02], which constitutes a low-level abstraction of SMPC that is defined based on bitstrings, Turing machines, etc. (cryptographic setting) Second, we build upon existing results on the secure realization of this functionality in the UC framework in order to obtain a secure cryptographic realization of our symbolic abstraction of SMPC. This computational soundness result holds for SMPC that involve arbitrary arithmetic operations; moreover, it is compositional, since the proof is parametric over the other (non-interactive) cryptographic primitives used in the symbolic protocol and within the SMPC itself. Computational soundness holds as long as these primitives are shown to be computationally sound (e.g., in the CoSP framework [BHU09]). We prove in particular the computational soundness of a Dolev-Yao model with public-key encryption, signatures, and the aforementioned arithmetic operations, leveraging and extending prior work in CoSP. Such a result allows for soundly modelling and verifying many applications employing SMPC as a building block, including the case studies considered in this paper.

As a second step, we generalize the previous result and formalize interactive primitives in CoSP. Then, we prove a sufficient conditions for computational soundness of interactive primitives w.r.t. uniformity: as long as these interactive primitive satisfy the cryptographic definition of universal composability [Can01] computational soundness for the symbolic representation of an ideal functionality holds.We stress that this result is parametric in the Dolev-Yao model, i.e., in the non-interactive primitives.

From Trace Properties to Equivalence Properties. As mentioned above, only limited CS results exist w.r.t. equivalence properties. Complimentarily to our other results, we identify and establish sufficient conditions under which CS results w.r.t. trace properties

imply CS results for uniformity, i.e., the equivalence of protocols with the same control flow. We illustrate this proof-technique by proving these sufficient conditions for a strong CS result for signatures and encryption. This result enables the reusability of numerous previous results.

Anonymous Communication

The onion routing (OR) network Tor [Tor03] has emerged as a successful technology for anonymous web browsing. It currently employs more than 6000 dedicated relays, and serves millions of users across the world. Its impact is also evident from the media coverage it has received over the last few years [Gre11]. Despite its success, the existing Tor network still lacks a rigorous security analysis. Previous analysis abstracted away from the cryptographic protocol underlying Tor [CL05; FJS07b; FJS11; MVV04], which lead to abstracted models are only accurate in very limited circumstances (see Section 7.7.1 for more details).

We extract from the specification of the Tor client and server protocol, an onion routing protocol in the universal composability framework [Can01]. Then, we devise an ideal representation of onion routing (in an ideal functionality), which assumes incorruptible parties, does not use any cryptographic operations, and restricts the adversary to a small set of actions. We prove that the onion routing protocol securely realizes this ideal functionality, i.e., we prove that all attacks that can be mounted against the onion routing protocol can be mounted against the ideal functionality. In contraposition, if there is no attack against the ideal functionality, then there is no attack against the onion routing protocol. We determine the exact security properties required for underlying cryptographic primitives (onion construction and processing algorithms, and a key exchange protocol) to achieve a provably secure OR protocol. We show that the currently deployed onion algorithms with slightly strengthened integrity properties can be used in a provably secure OR construction.

We furthermore present a novel key-exchange protocol that, compared to Tor's current key-exchange protocol improves on the computation costs without decreasing the communication costs. In numbers, the client has an efficiency improvement of 46% and the server of nearly 19%. The proposed protocol requires a client to send one additional group element to a server, compared to the *ntor* protocol. However, an additional group element easily fits into the 512 bytes fix-sized Tor packets (or cell) in the elliptic curve cryptography (ECC) setting. Consequently, our protocol does not produce a communication overhead in the Tor protocol. Moreover, we prove that our protocol satisfies the security definitions that we require in the realization proof for the onion routing protocol. Given that the ECC setting is under consideration for the Tor system, the improved computational efficiency, and the proven security properties make our 1W-AKE an ideal candidate for use in the Tor protocol.

Outline of this Thesis

Chapter 2 reviews the original CoSP framework (for trace properties) and presents the extension of the CoSP framework for equivalence properties. Moreover, we present how to embed the applied π-calculus into the CoSP framework such that uniformity is preserved. To this end, we first review the applied π-calculus. Chapter 3 presents the symbolic abstraction of malleable zero-knowledge proofs and its computational soundness

proof. Chapter 4 presents the symbolic abstraction of secure multi-party computation, its computational soundness proof, and a case study in the applied π-calculus. Chapter 5 generalizes the results of the previous chapter introducing a generic way to integrate an interactive primitive into the CoSP framework (for equivalence properties). Then, a sufficient conditions for computational soundness is proven: the well-known notion of universal composability [Can01]. Chapter 6 identifies and proves a sufficient conditions for connecting computational soundness w.r.t. trace properties and w.r.t. uniformity. We show that as long as the symbolic model allows for a technical condition that we call "self-monitoring" computational soundness w.r.t. trace properties of that symbolic model suffices to conclude computation soundness w.r.t. uniformity.

Chapter 7 extracts a cryptographic onion routing protocol from Tor's specification, and abstracts this onion routing protocol as an ideal functionality. We identify cryptographic properties for the cryptographic building blocks of the onion routing protocol under which this abstraction is UC-secure. (UC-security is a similar notion to computational soundness for equivalence properties.) Chapter 8 presents the *Ace* protocol: an computation efficient key-exchange protocol for Tor. We prove this protocol secure w.r.t. the cryptographic properties that we identified in Chapter 7. Moreover, we evaluate the performance of *Ace*.

Chapter 9 concludes the work, and Appendix A contains the source code of the case study from Chapter 4.

Part I.

Computational Soundness

Malleable Zero-Knowledge Proofs & Sufficient Conditions for Computational Soundness of Interactive & Non-Interactive Primitives

Chapter 2.

Theoretical Foundations: The CoSP Framework, The Applied π-calculus & Limitations of ProVerif

2.1. Motivation

This chapter presents the theoretical framework in which all results of this thesis are cast: the CoSP (Computational Soundness Proofs) framework [BHU09]. CoSP decouples the computational soundness (CS) of Dolev-Yao models from the embedding of the calculi, such as the applied π-calculus or RCF. As a result, CoSP enables proving x Dolev-Yao models computationally sound for y calculi by only having $x + y$ proofs instead of $x \cdot y$.

Section 2.2 reviews the original CoSP framework [BHU09]. This version, however, is restricted to trace properties, such as authentication, and does not include the essential class of equivalence properties, such as vote privacy (in e-voting schemes) or strong secrecy. Section 2.3 extends this original version to equivalence properties. For this generalized framework, we show that the applied π-calculus can be soundly embedded. First, Section 2.4 briefly reviews the applied π-calculus (also needed in other chapters of the present thesis), and second, Section 2.5 proves the existence of an embedding from the applied π-calculus to the generalized CoSP framework. We show that this embedding preserves the uniformity of bi-processes, using a slight variation of the already existing embedding for trace properties. Finally, we additionally observe that the automated verification tool ProVerif is not able to cope with symbolic models that involve length functions. Section 2.6 shows how computationally sound automated analyses can still be achieved in those situations in which ProVerif does not manage to terminate whenever the Dolev-Yao model supports a length function. We propose to proceed in two steps. First, a stripped-down version of the protocol without length functions is fed to ProVerif, and ProVerif then yields a result concerning the uniformity of bi-processes, but only for this stripped-down protocol. Second, the original protocol is fed to the APTE tool by Cheval, Cortier, and Plet [CCP13], which is specifically tailored to length functions. This yields a result for the original protocol but only concerning trace equivalences, which are not covered by the computational soundness results of Chapter 5 and Chapter 6. It is shown that both results can be combined to achieve uniformity of bi-processes for the original protocol, which are covered by our computational soundness results.

2.2. The CoSP Framework for Trace Properties

Backes, Hofheinz, and Unruh presented a framework for conducting computational soundness proofs [BHU09]. This framework separates the task of proving computational soundness for a set of (non-interactive) primitives against Dolev-Yao attackers from the task of embedding a calculus into a setting with a symbolic Dolev-Yao attacker.

CoSP compares two models: the symbolic and the computational model. In Section 2.2.1, we review the symbolic model, which specifies the notion of a protocol, the set of constructors and destructors that abstract the cryptographic primitives, and the symbolic counterpart to the execution of a program. In Section 2.2.2, we review the computational model, and in Section 2.2.3 we review the notion of computational soundness, which compares the symbolic and the computational model. Finally, in Section 2.2.4 we review a sufficient condition for computational soundness by Backes, Hofheinz, and Unruh [BHU09], which we satisfy in Chapter 3.

All definitions are copied verbatim from the original CoSP paper [BHU09], marked with a reference to the paper in the name of the definition.

2.2.1. Symbolic Model

We begin the review of CoSP with the symbolic model, including the symbolic attacker, the notion of a protocol in CoSP, and the notion of a symbolic execution in CoSP. A symbolic model includes the symbolic abstraction of the set of functions that can be used in the protocols that are considered and that are used to characterize the symbolic attacker (also called the Dolev-Yao attacker). This set of functions consists of a set of uninterpreted functions, called *constructors*, and a set of partial functions, called *destructors*. The set of symbolic terms is induced by the set of constructors and destructors. The symbolic attacker is a relation, called the *deduction relation*, over symbolic terms that characterizes the set of terms that the symbolic attacker can deduce from another term.

Definition 1 (Symbolic model [BHU09])**.** *A* constructor C *is a symbol with an arity. We write* $C/n \in \boldsymbol{C}$ *to say that the set* $\boldsymbol{C}$ *contains a constructor* C *with arity* n*. A* nonce N *is a symbols with zero arity. A* message type $\boldsymbol{T}$ *over* $\boldsymbol{C}$ *and* $\boldsymbol{N}$ *is a set of terms over constructors* $\boldsymbol{C}$ *and nonces* $\boldsymbol{N}$*. A* destructor D *of arity* n *over a message type* $\boldsymbol{T}$ *is a partial map* $\boldsymbol{T}^n \to \boldsymbol{T}$*. If* D *is undefined on* $\underline{t}$*, we write* $D(\underline{t}) = \bot$*. A* deduction relation $\vdash$ *over a message type* $\boldsymbol{T}$ *is a relation between* $2^{\boldsymbol{T}}$ *and* $\boldsymbol{T}$*.*

A symbolic model $\boldsymbol{M} = (\boldsymbol{C}, \boldsymbol{N}, \boldsymbol{T}, \boldsymbol{D}, \vdash)$ *consists of a set of constructors* $\boldsymbol{C}$*, a set of nonces* $\boldsymbol{N}$*, a message type* $\boldsymbol{T}$ *over* $\boldsymbol{C}$ *and* $\boldsymbol{N}$ *with* $\boldsymbol{N} \subseteq \boldsymbol{T}$*, a set of destructors* $\boldsymbol{D}$ *over* $\boldsymbol{T}$*, and a deduction relation* $\vdash$ *over* $\boldsymbol{T}$*.*

A protocol is in CoSP defined as an infinite tree with labelled nodes and edges. The tree has a distinguished root and otherwise consists of five kind of nodes: *computation nodes* for producing terms by applying constructors or destructors, *input and output nodes* for receiving and sending messages to the adversary, *control nodes* for enabling the adversary to influence a protocol's control flow (e.g., if the adversary is used to schedule concurrent threads), and *nondeterministic nodes* for nondeterministic choices in the protocol (meant for modeling probabilistic choices for which the probability distribution is not known a-priori). The labels at the nodes and the edges carry various kinds of additional data needed for the

execution of the protocol, e.g., each nodes-label carries a unique identifier, a computation node is additionally labelled with an identifier for the constructor or destructor, and edges are used to mark distinct branches of a protocol such as a *yes* and a *no* label after each computation node.

Definition 2 (CoSP protocol [BHU09])**.** *A* CoSP protocol Π_{s} *is a tree with a distinguished root and labels on both edges and nodes. Each node has a unique identifier N and one of the following types:*

- Computation nodes *are annotated with a constructor, nonce, or destructor F/n together with the identifiers of n (not necessarily distinct) nodes. Computation nodes have exactly two successors; the corresponding edges are labeled with yes and no, respectively.*
- Output nodes *are annotated with the identifier of one node. An output node has exactly one successor.*
- Input nodes *have no further annotation. An input node has exactly one successor.*
- Control nodes *are annotated with a bitstring l. A control node has at least one and up to countably many successors annotated with distinct bitstrings $l' \in \{0,1\}^*$. (We call l the out-metadata and l' the in-metadata.)*
- Nondeterministic nodes *have no further annotation. Nondetermininistic nodes have at least one and at most finitely many successors; the corresponding edges are labeled with distinct bitstrings.*

We assume that the annotations are part of the node identifier N. If a node N contains an identifier N' in its annotation, then N' has to be on the path from the root to N (including the root, excluding N), and N' must be a computation node or input node. In case N' is a computation node, the path from N' to N has to additionally go through the outgoing edge of N' with label yes.

Nondeterministic nodes are an abstraction, modelling protocol with random choices for any probability distribution of the random choices. Real protocols do not include such nondeterministic choices. Hence, CoSP introduces an intermediate protocol, *probabilistic CoSP protocols*, where every nondeterministic node has a probability distribution labelling its outgoing edges. By removing the probability distributions from the labels of the nondeterministic nodes, there is a canonical CoSP protocol Π_{s} for every probabilistic CoSP protocol Π_{p}. We say that Π_{s} is the *symbolic protocol that corresponds to* Π_{p}.

Definition 3 (Probabilistic CoSP protocol [BHU09])**.** *A* probabilistic CoSP protocol Π_{p} *is a CoSP protocol, where each nondeterministic node is additionally annotated with a probability distribution over the labels of the outgoing edges.*

So far, a CoSP protocol is not necessarily poly-time computable, some are not even computable at all. Hence, CoSP introduces a notion of an *efficient protocol* for CoSP protocols. If there is one poly-time algorithm that enables a iterative computation of the protocol, the protocol is efficient.

Definition 4 (Efficient Protocol [BHU09])**.** *We call a CoSP protocol* efficient *if:*

- *There is a polynomial p such that for any node N, the length of the identifier of N is bounded by $p(m)$ where m is the length (including the total length of the edge-labels) of the path from the root to N.*

- *There is a deterministic polynomial-time algorithm that, given the identifiers of all nodes and the edge labels on the path to a node N, computes the identifier of N.*
- *There is a deterministic polynomial-time algorithm that, given the identifier of a control node N, the identifiers of all nodes and all edge labels on the path to N, computes the lexicographically smallest label of an edge (i.e., the in-metadata) of all edges that lead from N to one of its successors.*

Next, CoSP includes a symbolic counterpart to the actual execution of a protocol, called the *symbolic execution*. It represents a trace of the execution of the protocol. Technically, the symbolic execution is a sequence of triples (S, ν, f) with S modeling the knowledge of the adversary if the execution is at the node ν and with a partial mapping f from node identifiers to messages, needed, for computation nodes, to keep track of a computation node's result and, for input nodes, to fix the messages that have been sent by the symbolic adversary (i.e., terms that adhere the deduction relation $\vdash$).

The following notation makes the definition of a symbolic execution more succinct: for a constructor or nonce F $eval_F(t_1, \ldots, t_n) := F(\underline{t})$ denotes $F(\underline{t}) \in \mathbf{T}$ and $eval_F(\underline{t}) := \bot$ otherwise. For a destructor F $eval_F(\underline{t}) := F(\underline{t})$ denotes $F(\underline{t}) \neq \bot$ and $eval_F(\underline{t}) := \bot$ otherwise.

Definition 5 (Symbolic execution [BHU09])**.** *Let a symbolic model $(\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D}, \vdash)$ and a CoSP protocol Π_s be given. A* full trace *is a (finite) list of tuples (S_i, ν_i, f_i) such that the following conditions hold:*

- Correct start*: $S_1 = \varnothing$, ν_1 is the root of Π_s, f_1 is a totally undefined partial function mapping node identifiers to terms.*
- Valid transition*: For every two consecutive tuples (S, ν, f) and (S', ν', f') in the list, let $\underline{\tilde{\nu}}$ be the node identifiers in the annotation of ν and define $\underline{\tilde{t}}$ through $\tilde{t}_j := f(\tilde{\nu}_j)$. We have:*
  - *If ν is a computation node with constructor, destructor or nonce F, then $S' = S$. If $m := eval_F(\underline{\tilde{t}}) \neq \bot$, ν' is the yes-successor of ν in Π_s, and $f' = f(\nu := m)$. If $m = \bot$, then ν' is the no-successor of ν and $f' = f$.*
  - *If ν is an input node, then $S' = S$ and ν' is the successor of ν in Π_s and there exists an m with $S \vdash m$ and $f' = f(\nu := m)$.*
  - *If ν is an output node, then $S' = S \cup \{\tilde{t}_1\}$, ν' is the successor of ν in Π_s and $f' = f$.*
  - *If ν is a control or a nondeterministic node, then ν' is a successor of ν and $f' = f$ and $S' = S$.*

A list of node identifiers (ν_i) is a node trace *if there is a full trace with these node identifiers.*

2.2.2. Computational Model

CoSP defines a computational execution of a protocol, which represents the execution of a protocol in a cryptographic setting. For defining the notion of computational soundness, CoSP compares the symbolic execution (see Definition 5) and this computational execution.

Before arriving at the notion of a computational execution, we say when an algorithm is suitable for realizing a constructor, a destructor, or drawing a nonce in the cryptographic setting. CoSP calls these algorithms *computational implementations*. Such a computational

implementation needs to be poly-time computable and all implementations need to be deterministic, except for the the algorithm A_N for drawing a nonce. Requiring determinism is not a severe restriction, since most cryptographic algorithms can be modeled by adding A_N as an argument.[1]

Definition 6 (Computational implementation [BHU09])**.** *Let a symbolic model* $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D}, \vdash)$ *be given. A* computational implementation of $\mathbf{M}$ *is a family of functions* $A = (A_x)_{x \in \mathbf{C} \cup \mathbf{D} \cup \mathbf{N}}$ *such that* A_F *for* $F/n \in \mathbf{C} \cup \mathbf{D}$ *is a partial deterministic function* $\mathbb{N} \times (\{0,1\}^*)^n \to \{0,1\}^*$*, and* A_N *for* $N \in \mathbf{N}$ *is a total probabilistic function with domain* $\mathbb{N}$ *and range* $\{0,1\}^*$ *(i.e., it specifies a probability distribution on bitstrings that depends on its argument). The first argument of* A_F *and* A_N *represents the security parameter.*

All functions A_F *have to be computable in deterministic polynomial-time, and all* A_N *have to be computable in probabilistic polynomial-time.*[2]

The computational execution is a probabilistic algorithm that executes the interaction between a (probabilistic) CoSP protocol and some ppt machine, which represents the adversary. The computational execution traverses the CoSP protocol, runs for each computation node the corresponding computational implementation, sends messages to the adversary, runs the adversary, and stores the responses of the adversary.

Definition 7 (Computational execution [BHU09])**.** *Let a symbolic model* $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D}, \vdash)$*, a computational implementation* A *of* $\mathbf{M}$*, and a probabilistic CoSP protocol* Π_p *be given. Let a probabilistic polynomial-time interactive machine* E *(the adversary) be given (polynomial-time in the sense that the number of steps in all activations are bounded in the length of the first input of* E*), and let* p *be a polynomial. We define a probability distribution* $Nodes^p_{\mathsf{M},A,\Pi_\mathsf{p},E}(k)$*, the* computational node trace*, on (finite) lists of node identifiers* (ν_i) *according to the following probabilistic algorithm (both the algorithm and* E *are run on input* k*):*

- *Initial state:* $\nu_1 := \nu$ *is the root of* Π_p*. Let* f *be an initially empty partial function from node identifiers to bitstrings, and let* n *be an initially empty partial function from* $\mathbf{N}$ *to bitstrings.*
- *For* $i = 2, 3, \ldots$ *do the following:*
  - *Let* $\underline{\tilde{\nu}}$ *be the node identifiers in the annotation of* ν*.* $\tilde{m}_j := f(\tilde{\nu}_j)$*.*
  - *Proceed depending on the type of node* ν*:*
    - *If* ν *is a computation node with nonce* $N \in \mathbf{N}$*: Let* $m' := n(N)$ *if* $n(N) \neq \bot$ *and sample* m' *according to* $A_N(k)$ *otherwise. Let* ν' *be the yes-successor of* ν*,* $f' := f(\nu := m')$*, and* $n' := n(N := m')$*. Let* $\nu := \nu'$*,* $f := f'$ *and* $n := n'$*.*
    - *If* ν *is a computation node with constructor or destructor* F*, then* $m' := A_F(k, \underline{\tilde{m}})$*. If* $m' \neq \bot$*, then* ν' *is the yes-successor of* ν*, if* $m' = \bot$*, then* ν' *is the no-successor of* ν*. Let* $f' := f(\nu := m')$*. Let* $\nu := \nu'$ *and* $f := f'$*.*
    - *If* ν *is an input node, ask for a bitstring* m *from* E*. Abort the loop if* E *halts. Let* ν' *be the successor of* ν*. Let* $f := f(\nu := m)$ *and* $\nu := \nu'$*.*

[1] As an example, if A_N uniformly draws randomness, all cryptographic algorithms that use uniform randomness can be modelled.

[2] More precisely, there has to exist a single uniform probabilistic polynomial-time algorithm A that, given the name of $C \in \mathbf{C}$, $D \in \mathbf{D}$, or $N \in \mathbf{N}$, together with an integer k and the inputs $\underline{m}$, computes the output of A_C, A_D, and A_N or determines that the output is undefined. This algorithm must run in polynomial-time in $k + |\underline{m}|$ and may not use random coins when computing A_C and A_D.

  - *If ν is an output node, send $\tilde{m}_1$ to E. Abort the loop if E halts. Let ν' be the successor of ν. Let $\nu := \nu'$.*
  - *If ν is a control node, annotated with out-metadata l, send l to E. Abort the loop if E halts. Upon receiving an answer l', let ν' be the successor of ν along the edge labeled l' (or the lexicographically smallest edge if there is no edge with label l'). Let $\nu := \nu'$.*
  - *If ν is a nondeterministic node, let $\mathcal{D}$ be the probability distribution in the annotation of ν. Pick ν' according to the distribution $\mathcal{D}$, and let $\nu := \nu'$.*
- *Let $\nu_i := \nu$.*
- *Let len be the number of nodes from the root to ν plus the total length of all bitstrings in the range of f. If len $> p(k)$, stop.*

2.2.3. Computational Soundness

For defining computational soundness for trace properties, we define trace properties in a way that they are compatible with the computational and the symbolic execution: traces are sequences of protocol states (technically nodes) of one protocol run and trace properties are prefix-closed sets of such traces.

Definition 8 (Trace property [BHU09])**.** *A* trace property $\wp$ *is an efficiently decidable and prefix-closed set of (finite) lists of node identifiers.*

Let $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D}, \vdash)$ *be a symbolic model and* Π_s *a CoSP protocol. Then* Π_s symbolically satisfies *a trace property* $\wp$ *in* $\mathbf{M}$ *iff every node trace of* Π_s *is contained in* $\wp$*. Let* Impl *be a computational implementation of* $\mathbf{M}$ *and let* Π_p *be a probabilistic CoSP protocol. Then* $(\Pi_\mathsf{p}, \mathrm{Impl})$ computationally satisfies *a trace property* $\wp$ *in* $\mathbf{M}$ *iff for all probabilistic polynomial-time interactive machines E and all polynomials p, the probability is overwhelming that* $Nodes^p_{\mathsf{M},\mathrm{Impl},\Pi_\mathsf{p},E}(k) \in \wp$.

Finally, we are in a position to define computational soundness for trace properties.

Definition 9 (Computational soundness for trace properties [BHU09])**.** *A computational implementation* Impl *of a symbolic model* $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D}, \vdash)$ *is* computationally sound *for a class P of CoSP protocols iff for every trace property $\wp$ and for every efficient probabilistic CoSP protocol* Π_p*, we have that* $(\Pi_\mathsf{p}, \mathrm{Impl})$ *computationally satisfies $\wp$ whenever the corresponding CoSP protocol* Π_s *of* Π_p *symbolically satisfies $\wp$ and* $\Pi_\mathsf{s} \in P$.

2.2.4. A Sufficient Criterion for Soundness

The CoSP framework provides a technical tool for proving soundness: a simulator that translates between bitstrings and symbolic terms and that satisfies two important properties. First, the simulator has to adhere to the deduction relation that models the symbolic attacker, i.e., all terms that the simulator produces have to be deducable in the sense of the deduction relation from Definition 1. This property is called *Dolev-Yaoness*. Second, at the same time the simulator has to behave towards the adversary in a manner such that the resulting distribution of node traces are indistinguishable. This property is a sanity condition that forces the simulator to translate all attacks against trace properties of the adversary (or an indistinguishable version of them) to symbolic terms. This property is called *indistinguishability*.

For the sake of brevity, we use an arbitrary by fixed symbolic model $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D}, \vdash)$ and computational implementation Impl of $\mathbf{M}$. Furthermore, we make the following assumption: each term or node identifier that a machine sends can be encoded as a bitstring.

We start with presenting syntactic conditions on interactive machines for the notion of a simulator.

Definition 10 (Simulator [BHU09])**.** *A* simulator *is an interactive machine Sim that satisfies the following syntactic requirements:*

- *When activated without input, it replies with a term $m \in \mathbf{T}$. (This corresponds to the situation that the protocol expects some message from the adversary.)*
- *When activated with some $t \in \mathbf{T}$, it replies with an empty output. (This corresponds to the situation that the protocol sends a message to the adversary.)*
- *When activated with $(\texttt{info}, \nu, t)$ where ν is a node identifier and $t \in \mathbf{T}$, it replies with* $(\texttt{proceed})$.
- *At any point (in particular instead of sending a reply), it may terminate.*

The simulator communicates with a symbolic execution and behaves like a computational execution towards the adversary. However, since the symbolic execution is not an interactive machine, CoSP introduces an interactive machine, called the hybrid execution, that executes the symbolic execution.

Definition 11 (Hybrid execution [BHU09])**.** *Let Π_{p} be a probabilistic CoSP protocol, and let Sim be a simulator. We define a probability distribution H-Trace$_{\mathsf{M},\Pi_{\mathsf{p}},Sim}(k)$ on (finite) lists of tuples (S_i, ν_i, f_i) called the* full hybrid trace *according to the following probabilistic algorithm Π^C, run on input k, that interacts with Sim. (Π^C is called the hybrid protocol machine associated with Π_{p} and internally runs a symbolic simulation of Π_{p} as follows:)*

- Start*: $S_1 := S := \varnothing$, $\nu_1 := \nu$ is the root of Π_{p}, and $f_1 := f$ is a totally undefined partial function mapping node identifiers to $\mathbf{T}$. Run Π_{p} on ν.*
- Transition*: For $i = 2, 3, \ldots$ do the following:*
  - *Let $\underline{\tilde{\nu}}$ be the node identifiers in the label of ν. Define $\underline{\tilde{t}}$ through $\tilde{t}_j := f(\tilde{\nu}_j)$.*
  - *Proceed depending on the type of ν:*
    - *If ν is a computation node with constructor, destructor, or nonce F, then let $m := eval_F(\underline{\tilde{t}})$. If $m \neq \bot$, let ν' be the yes-successor of ν and let $f' := f(\nu := m)$. If $m = \bot$, let ν' be the no-successor of ν and let $f' := f$.*
    - *If ν is an output node, send $\tilde{t}_1$ to Sim (but without handing over control to Sim). Let ν' be the unique successor of ν. Set $\nu := \nu'$.*
    - *If ν is an input node, hand control to Sim, and wait to receive $m \in \mathbf{T}$ from Sim. Let $f' := f(\nu := m)$, and let ν' be the unique successor of ν. Set $f := f'$ and $\nu := \nu'$.*
    - *If ν is a control node labeled with out-metadata l, send l to Sim, hand control to Sim, and wait to receive a bitstring l' from Sim. Let ν' be the successor of ν along the edge labeled l' (or the lexicographically smallest edge if there is no edge with label l'). Let $\nu := \nu'$.*
    - *If ν is a nondeterministic node, sample ν' according to the probability distribution specified in ν. Let $\nu := \nu'$.*
  - *Send $(info, \nu, t)$ to Sim. When receiving an answer $(proceed)$ from Sim, continue.*
  - *If Sim has terminated, stop. Otherwise let $(S_i, \nu_i, f_i) := (S, \nu, f)$.*

The probability distribution of the (finite) list $\nu_1, \ldots$ produced by this algorithm we denote H-Nodes$_{\mathsf{M},\Pi_\mathsf{p},Sim}(k)$. We call this distribution the hybrid node trace.

If we write $Sim + \Pi^C$, we mean the execution of *Sim* and Π^C.

As a next step, we define the two semantic properties of the simulator: Dolev-Yaoness and indistinguishability. Dolev-Yaoness requires the simulator to only produce terms that adhere to the deduction relation of Definition 1.

Definition 12 (Dolev-Yao style simulator [BHU09])**.** *A simulator Sim is* Dolev-Yao style *(short:* DY*) for* M *and* Π_p*, if with overwhelming probability the following holds:*

In an execution of $Sim + \Pi^C$, for each ℓ, let $m_\ell \in \mathbf{T}$ be the ℓ-th term sent (during processing of one of Π^C's input nodes) from Sim to Π^C in that execution. Let $T_\ell \subseteq \mathbf{T}$ the set of all terms that Sim has received from Π^C (during processing of output nodes) prior to sending m_ℓ. Then we have $T_\ell \vdash m_\ell$.

The indistinguishability of a simulator is a sanity condition. It requires that while satisfying Dolev-Yaoness the simulator is not allowed to change the behavior of the adversary in the following sense: the distribution of node trace of the hybrid execution and of the computational execution have to be indistinguishable.

Definition 13 (Indistinguishable simulator)**.** *A simulator Sim is* indistinguishable *for* M*,* Π_p*, an implementation* Impl*, an adversary E, and a polynomial p, if*

$$Nodes^p_{\mathsf{M},\mathrm{Impl},\Pi_\mathsf{p},E}(k) \overset{C}{\approx} \textit{H-Nodes}_{\mathsf{M},\Pi_\mathsf{p},Sim}(k),$$

i.e., if the computational node trace and the hybrid node trace are computationally indistinguishable.

We define the following abbreviation.

Definition 14 (Good simulator)**.** *A simulator is* good *for* M*,* Π_p*,* Impl*, E, and p if it is Dolev-Yao style for* M*, and* Π_p*, and indistinguishable for* M*,* Π_p*,* Impl*, E, and p.*

A good simulator is sufficient to achieve computational soundness for trace properties.

Theorem 1 (Good simulator implies soundness [BHU09])**.** *Let* $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D}, \vdash)$ *be a symbolic model, let P be a class of CoSP protocols, and let* Impl *be a computational implementation of* M*. Assume that for every efficient probabilistic CoSP protocol* Π_p *(whose corresponding CoSP protocol is in P), every probabilistic polynomial-time adversary E, and every polynomial p, there exists a good simulator for* M*,* Π_p*,* Impl*, E, and p. Then* Impl *is computationally sound for protocols in P.*

2.3. The CoSP Framework for Equivalence properties

The original CoSP framework presented so far (Section 2.2) is only capable of handling CS with respect to trace properties, i.e., properties that can be formulated in terms of a single trace. Typical examples include the non-reachability of a certain "bad" protocol state, in that the attacker is assumed to have succeeded (e.g., the protocol never reveals a secret), or correspondence properties such as authentication (e.g., a user can access a resource only after proving a credential). However, many interesting protocol properties cannot be expressed in terms of a single trace. For instance, strong secrecy or anonymity are properties that are, in the computational setting, usually formulated by means of a game in which the attacker has to distinguish between several scenarios.

To be able to handle the class of equivalence properties, we extend the CoSP framework to support equivalence properties. First, we recall the basic definitions of the original framework. Dolev-Yao models are formalized as follows in CoSP.

2.3.1. Symbolic Indistinguishability

In this section, we define a symbolic notion of indistinguishability. First, we model the capabilities of the symbolic attacker. Operations that the symbolic attacker can perform on terms are defined as follows, including the destruction of already known terms and the creation of new terms.[3]

Definition 15 (Symbolic Operation)**.** *Let* $\mathsf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D})$ *be a symbolic model. A* symbolic operation O/n (of arity n) on $\mathbf{M}$ *is a finite tree whose nodes are labeled with constructors from* $\mathbf{C}$*, destructors from* $\mathbf{D}$*, nonces from* $\mathbf{N}$*, and formal parameters* x_i *with* $i \in \{1, \ldots, n\}$*. For constructors and destructors, the children of a node represent its arguments (if any). Formal parameters* x_i *and nonces do not have children.*

We extend the evaluation function to symbolic operations*. Given a list of terms* $\underline{t} \in \mathbf{T}^n$*, the evaluation function* $eval_O : \mathbf{T}^n \to \mathbf{T}$ *recursively evaluates the tree* O *starting at the root as follows: The formal parameter* x_i *evaluates to* t_i*. A node with* $F \in \mathbf{C} \cup \mathbf{N}_E \cup \mathbf{D}$ *evaluates according to* $eval_F$*. If there is a node that evaluates to* $\bot$*, the whole tree evaluates to* $\bot$*.*

We stress that the identity function is included in the set of symbolic operations. It is the tree that contains only x_1 as node.

A symbolic execution of a protocol is a valid path through the protocol tree together with a communication with the attacker. A *view* only consists of the communication with the attacker. Equivalence of two protocols is formalized by requiring that the set of possible views looks the same for the symbolic adversary.

Definition 16 (Symbolic Execution)**.** *Let a symbolic model* $\mathsf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D})$ *and a CoSP protocol* Π *be given. A* symbolic view *of the protocol* Π *is a (finite) list of triples* (V_i, ν_i, f_i) *with the following conditions:*

[3] We deviate from the definition in the original CoSP framework [BHU09], where a deduction relation describes which terms the attacker can deduce from the already seen terms. This modification is not essential; all results for trace properties that have been established in the original framework so far are compatible with our definition.

For the first triple, we have $V_1 = \varepsilon$, ν_1 is the root of Π, and f_1 is an empty partial function, mapping node identifiers to terms. For every two consecutive tuples (V, ν, f) and (V', ν', f') in the list, let $\underline{\tilde{\nu}}$ be the nodes referenced by ν and define $\underline{\tilde{t}}$ through $\tilde{t}_j := f(\tilde{\nu}_j)$. We conduct a case distinction on ν.

- *ν **is a computation node with constructor, destructor or nonce** F. Let $V' = V$. If $m := eval_F(\underline{\tilde{t}}) \neq \bot$, ν' is the* `yes`*-successor of ν in Π, and $f' = f(\nu := m)$. If $m = \bot$, then ν' is the* `no`*-successor of ν, and $f' = f$.*
- *ν **is an input node.** If there exists a term $t \in \mathbf{T}$ and a symbolic operation O on M with $eval_O(V_{Out}) = t$, let ν' be the successor of ν in Π, $V' = V :: (\mathtt{in}, (t, O))$, and $f' = f(\nu := t)$.*
- *ν **is an output node.** Let $V' = V :: (\mathtt{out}, \tilde{t}_1)$, ν' is the successor of ν in Π, and $f' = f$.*
- *ν **is a control node with out-metadata** l. Let ν' be the successor of ν with the in-metadata l' (or the lexicographically smallest edge if there is no edge with label l'), $f' = f$, and $V' = V :: (\mathtt{control}, (l, l'))$.*

Here, V_{Out} denotes the list of terms in V that have been sent at output nodes, i.e., the terms t contained in entries of the form $(\mathtt{out}, t)$ in V. Analogously, $V_{Out\text{-}Meta}$ denotes the list of out-metadata in V that has been sent at control nodes.

The set of all symbolic views of Π is denoted by $SViews(\Pi)$. Furthermore, V_{In} denotes the partial list of V that contains only entries of the form $(\mathtt{in}, (*, O))$ or $(\mathtt{control}, (*, l'))$ for some symbolic operation O and some in-metadata l', where the input term and the out-metadata have been masked with the symbol $*$. The list V_{In} is called attacker strategy*. We write $[V_{In}]_{SViews(\Pi)}$ to denote the class of all views $U \in SViews(\Pi)$ with $U_{In} = V_{In}$.*

The knowledge of the attacker comprises the results of all the symbolic tests the attacker can perform on the messages output by the protocol. To define the attacker knowledge formally, we have to pay attention to two important details. First, we concentrate on whether a symbolic operation fails or not, i.e., if it evaluates to $\bot$ or not; we are not interested in the resulting term in case the operation succeeds. The following example illustrates why. Let us consider a protocol Π_1 that does nothing more than sending a ciphertext c to the attacker, whereas another protocol Π_2 sends a different ciphertext c' (with the same plaintext length) to the attacker. Assume that the decryption key is kept secret. This pair of protocols should be symbolically indistinguishable. More precisely, the attacker knowledge in Π_1 should be statically indistinguishable from the attacker knowledge in the Π_2. Recall that $O = x_1$ is the symbolic operation that just returns the first message received by the attacker. If the result of O were part of the attacker knowledge, the knowledge in Π_1 (containing c) would differ from the knowledge in the Π_2 (containing c'), which is not what we would like to express. On the other hand, our definition, which only cares about the failure or success of a operation, requires that the symbolic model contains an operation *equals* to be reasonable. This operation *equals* allows the attacker to test equality between terms: consider the case where Π_2 sends a publicly known term t instead of c', but still of the same length as c. In that case the attacker can distinguish this pair of protocols with the help of the symbolic operation $equals(t, x_1)$.

The second observation is that the definition should cover the fact that the attacker knows which symbolic operation leads to which result. This is essential to reason about indistinguishability: consider a pair of protocols Π_1 and Π_2 such that the first protocol

Π_1 sends the pair (n, t), but the second protocol Π_2 sends the pair (t, n), where t is again a publicly known term and n is a fresh protocol nonce. The two protocols do not differ in the terms that the attacker can deduce after their execution; the deducible terms are all publicly known terms as well as n. Still, the protocols are trivially distinguishable by the symbolic operation $equals(O_t, snd(x_1))$ because $equals(O_t, snd((n, t))) \neq \bot$ but $equals(t, snd((t, n))) = \bot$, where snd returns the second component of a pair and O_t is a symbolic operation that constructs t.

Definition 17 (Symbolic Knowledge)**.** *Let* $\mathbf{M}$ *be a symbolic model. Given a view* V *with* $|V_{Out}| = n$*, a* symbolic knowledge function $f_V : SO(\mathsf{M})_n \to \{\top, \bot\}$ *is a partial function from symbolic operations (see Definition 15) of arity* n *to* $\{\top, \bot\}$*. The* full symbolic knowledge function *is a total symbolic knowledge function is defined by*

$$K_V(O) := \begin{cases} \bot & \text{if } eval_O(V_{Out}) = \bot \\ \top & \text{otherwise.} \end{cases}$$

Intuitively, we would like to consider two views *equivalent* if they look the same for a symbolic attacker. Despite the requirement that they have the same order of output, input and control nodes, this is the case if they agree on the out-metadata (the control data sent by the protocol) as well as the symbolic knowledge that can be gained out of the terms sent by the protocol.

Definition 18 (Equivalent Views)**.** *Let two views* V, V' *of the same length be given. We denote their* i*th entry by* V_i *and* V'_i*, respectively.* V *and* V' *are* equivalent *(*$V \sim V'$*), if the following three conditions hold:*

1. *(Same structure)* V_i *is of the form* $(s, \cdot)$ *if and only if* V'_i *is of the form* $(s, \cdot)$ *for some* $s \in \{\texttt{out}, \texttt{in}, \texttt{control}\}$*.*
2. *(Same out-metadata)* $V_{Out\text{-}Meta} = V'_{Out\text{-}Meta}$*.*
3. *(Same symbolic knowledge)* $K_V = K_{V'}$*.*

Finally, we define two protocols to be symbolically indistinguishable if its two variants lead to equivalent views when faced with the same attacker strategy. Thus, a definition of "symbolic indistinguishability" should compare the symbolic knowledge of two protocol runs only if the attacker behaves identically in both runs.

Definition 19 (Symbolic Indistinguishability)**.** *Let* M *be a symbolic model and* P *be a class of protocols on* M*. Given an attacker strategy* V_{In} *(in the sense of Definition 16), two protocols* $\Pi_1, \Pi_2 \in \mathrm{P}$ *are* symbolically indistinguishable under V_{In} *if for all views* $V_1 \in [V_{In}]_{SViews(\Pi_1)}$ *of* Π_1 *under* V_{In}*, there is a view* $V_2 \in [V_{In}]_{SViews(\Pi_2)}$ *of* Π_2 *under* V_{In} *such that* $V_1 \sim V_2$*, and vice versa.*

Two protocols $\Pi_1, \Pi_2 \in \mathrm{P}$ *are* symbolically indistinguishable *(*$\Pi_1 \approx_s \Pi_2$*), if* Π_1 *and* Π_2 *are indistinguishable under all attacker strategies.*

2.3.2. Computational Indistinguishability

The computational challenger of a protocol is a randomized interactive machine that runs against a ppt attacker $\mathcal{A}$. This interaction is called the computational execution. The transcript of the execution is the computational counterparts of a symbolic view.

Definition 20 (Computational Challenger)**.** *Let* $\mathbf{A}$ *be a computational implementation of the symbolic model* $\mathsf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D})$ *and* Π *be a CoSP protocol. Let* $\mathcal{A}$ *be a ppt machine. For a security parameter* k*, the* computational challenger $Exec_{\mathsf{M},\mathrm{Impl},\Pi}(k)$ *is the following interactive machine:*

Initially, let ν *be the root of* Π*. Let* f *and* n *be empty partial functions from node identifiers to bitstrings and from* $\mathbf{N}$ *to bitstrings, respectively. Enter a loop and proceed depending on the type of* ν*:*

- ν ***is a computation node with nonce*** $N \in \mathbf{N}$**.** *If* $n(N) \neq \bot$*, let* $m' := n(N)$*; otherwise sample* m' *according to* $A_N(k)$*. Let* ν' *be the yes-successor of* ν*. Let* $f := f(\nu := m')$*,* $n := n(N := m')$*, and* $\nu_i := \nu'$*.*
- ν ***is a computation node with constructor or destructor*** F**.** *Let* $\underline{\tilde{\nu}}$ *be the nodes referenced by* ν *and* $\tilde{m}_j := f(\tilde{\nu}_j)$*. Then,* $m' := A_F(k, \underline{\tilde{m}})$*. If* $m' \neq \bot$*, then* ν' *is the* `yes`*-successor of* ν*, if* $m' = \bot$*, then* ν' *is the* `no`*-successor of* ν*. Let* $f := f(\nu := m')$ *and* $\nu_i := \nu'$*.*
- ν ***is an input node.*** *Ask the adversary* $\mathcal{A}$ *for a bitstring* m*. Let* ν' *be the successor of* ν*. Let* $f := f(\nu := m)$ *and* $\nu_i := \nu'$*.*
- ν ***is an output node.*** *Send* $\tilde{m}_1$ *to* $\mathcal{A}$*. Let* ν' *be the successor of* ν*, and let* $\nu_i := \nu'$*.*
- ν ***is a control node with out-metadata*** l**.** *Send* l *to* $\mathcal{A}$*. Upon receiving in-metadata* l'*, let* ν' *be the successor of* ν *along the edge labeled* l' *(or the edge with the lexicographically smallest label if there is no edge with label* l'*). Let* $\nu_i := \nu'$*.*

Definition 21 (Computational Execution)**.** *The interaction between the challenger* $Exec_{\mathsf{M},\mathbf{A},\Pi}(k)$ *and the adversary* $\mathcal{A}(k)$ *is called the* computational execution*, denoted by* $\langle Exec_{\mathsf{M},\mathbf{A},\Pi}(k) \mid \mathcal{A}(k)\rangle$*. It stops whenever one of the two machines stops, and the output of* $\langle Exec_{\mathsf{M},\mathbf{A},\Pi}(k) \mid \mathcal{A}(k)\rangle$ *is the output of* $\mathcal{A}(k)$*.*

We do not use the standard notion of computational indistinguishability for defining the indistinguishability in the computational model, as it has severe limitations for computational soundness. To understand these limitation, consider a pair of protocols that is symbolically indistinguishable from a uniformity enforcing class of pairs of protocols. After receiving an input, the first protocol Π_1 immediately produces an output, and the second protocol Π_2 enters a loop that takes a superpolynomial number of steps but afterwards produces the same output. While these two protocols are symbolically indistinguishable (as the symbolic model abstracts away from running time), a polynomial-time distinguisher can observe that the second protocol Π_2 prematurely halts, i.e., the protocols are computationally distinguishable. Thus, computational soundness cannot hold in this case.

Another problem arises because many reactive systems, such as file servers, have actually no a-priori bound for their running time and that protocols that could (potentially) run in superpolynomial time prevent us of from using computational assumptions. Unruh [Unr11] gives the following example. Consider a pair of protocols Π_1, Π_2 that both describe a file server that allows clients to store values to files, read (parts of) files, store the concatenation of two files as a new file, and store the encryption of a files as a new file. Both protocols use an IND-CCA secure public encryption scheme and have only access to the public key; the secret key is never used. In Π_1, the encryption uses the actual contents of the file of length n, and in Π_2, the encryption instead uses 0^n. Now consider the following polynomial-time distinguisher $\mathcal{A}$: First, $\mathcal{A}$ creates a file f_0 with content 1. Second, $\mathcal{A}$ creates k files $f_1, \ldots, f_k$

by concatenation operations such that $f_i = 1^{2^i}$, where k is the security parameter. Third, $\mathcal{A}$ creates a file f by letting the file server encrypt the file f_k. Finally, $\mathcal{A}$ reads the first bit of f. Since IND-CCA security does not consider input messages of superpolynomial size (e.g., 1^{2^k}), nothing is guaranteed about the resulting ciphertext f. Indeed, the first bit of f could the first bit of f_k, in which case $\mathcal{A}$ can easily distinguish Π_1 from Π_2.

As a remedy, Unruh introduced a notion, called tic-indistinguishability, which defines indistinguishability up to running time tests. With tic-indistinguishability we are able to formulate a meaningful CS result, because it guarantees indistinguishability as long as the protocol does not long actually run in a superpolynomial time, while it ignores the situation after a superpolynomial number of steps.

Note that both these problems could be solved also by restricting the protocol class to pairs of protocols that always run in polynomial time. In that case, we could use normal computational indistinguishability. However, this way comes with drastic restrictions: First, it excludes many plausible protocols like the protocol described above. Second, to enable mechanized verification, it is highly desirable to define the class of pairs of protocols captured by the CS result using statically-verifiable conditions, e.g., syntactic conditions such as disallowing loops. (This way is taken by Comon-Lundh and Cortier [CC08] as well as Comon-Lundh, Cortier and Scerri [CCS12].) This, however, comes with even more restrictions, because such syntactic restrictions are typically only a rough over-approximation of polynomial-time protocols. Thus, even more protocols are excluded.

Definition 22 (Tic-indistinguishability [Unr11])**.** *Given two machines M, M' and a polynomial p, we write $\Pr[\langle M \mid M'\rangle \Downarrow_{p(k)} x]$ for the probability that the interaction between M and M' terminates within $p(k)$ steps and M' outputs x. We call two machines A and B* termination-insensitively computationally indistinguishable for a machine $\mathcal{A}$ *($A \approx^{\mathcal{A}}_{tic} B$) if for all polynomials p, there is a negligible function μ such that for all $z, a, b \in \{0,1\}^*$ with $a \neq b$,*

$$\Pr[\langle A(k) \mid \mathcal{A}(k,z)\rangle \Downarrow_{p(k)} a] + \Pr[\langle B(k) \mid \mathcal{A}(k,z)\rangle \Downarrow_{p(k)} b] \leq 1 + \mu(k).$$

Here, z represents an auxiliary string. Additionally, we call A and B termination-insensitively computationally indistinguishable *($A \approx_{tic} B$) if we have $A \approx^{\mathcal{A}}_{tic} B$ for all polynomial-time machines $\mathcal{A}$.*

Termination insensitive computational indistinguishability is not transitive, i.e., $A \approx_{tic} B$ and $B \approx_{tic} C$ does not imply $A \approx_{tic} C$ in general (see [Unr11] for more details). Due to this limitation, we additionally use in the proofs the standard notion of computational indistinguishability, denoted as $A \approx_{comp} B$. Moreover, we also use a notion of *time-equivalence*, denoted as $A \approx_{time} B$, that states that the output of two machines A and B is statistically indistinguishable, as long as only the output that A or B produces after super-polynomially many steps is ignored.[4]

We refer the reader to Unruh [Unr11] for a further discussion of tic-indistinguishability and a precise technical definition of the underlying machine model.

Given this definition, computational indistinguishability for pairs of protocols is naturally defined. A pair of protocols is indistinguishable if its challengers are computationally indistinguishable for every ppt attacker $\mathcal{A}$.

[4]Technically, the definition formalizes this intuition by only considering outputs generated after a polynomially bounded number of steps. For the full definition, we refer to the original paper [Unr11].

Definition 23 (Termination Insensitive Computational Indistinguishability)**.** *Let* Π_1 *and* Π_2 *be an efficient CoSP protocol and let* Impl *be a computational implementation of* $\mathbf{M}$*.* Π_1 *and* Π_2 *are* termination-insensitively computationally indistinguishable *if for all ppt attackers* $\mathcal{A}$ *and for all polynomials* p*,*

$$Exec_{\mathbf{A},\mathsf{M},\Pi_1} \approx_{tic} Exec_{\mathbf{A},\mathsf{M},\Pi_2}.$$

2.3.3. Computational Soundness

The previous definitions culminate in the definition of CS for equivalence properties. It states that the symbolic indistinguishability of a pair of protocols implies its computational indistinguishability. In other words, it suffices to check the security of the symbolic pair of protocols, e.g., using mechanized protocol verifiers such as ProVerif.

Definition 24 (Computational Soundness for Equivalence Properties)**.** *Let a symbolic model* M *and a class* P *of efficient protocols be given. An implementation* Impl *of* M *is* computationally sound (w.r.t. equivalence properties) for M and P *if for every pair* $\Pi_1, \Pi_2 \in \mathrm{P}$*, we have that* Π_1 *and* Π_2 *are computationally indistinguishable whenever* Π_1 *and* Π_2 *are symbolically indistinguishable.*

2.3.4. Bi-Protocols

In order to compare two variants of a protocol, we consider bi-protocols, which rely on the same idea as bi-processes in the applied π-calculus [BAF05]. Bi-protocols are pairs of protocols that only differ in the messages they operate on.

Definition 25 (CoSP Bi-protocol)**.** *A* CoSP bi-protocol Π *is defined like a protocol but uses bi-references instead of references. A* bi-reference *is a pair* $(\nu_{left}, \nu_{right})$ *of node identifiers of two (not necessarily distinct) nodes in the protocol tree. In the* left protocol *left*(Π) *the bi-references are replaced by their left components; the* right protocol *right*(Π) *is defined analogously.*

Efficiency for CoSP bi-protocols is defined as for CoSP protocols.

Definition 26 (Efficient Bi-Protocol)**.** *We call a CoSP bi-protocol* efficient *if:*

- *There is a polynomial* p *such that for any node* N*, the length of the identifier of* N *is bounded by* $p(m)$ *where* m *is the length (including the total length of the edge-labels) of the path from the root to* N*.*
- *There is a deterministic polynomial-time algorithm that, given the identifiers of all nodes and the edge labels on the path to a node* N*, computes the identifier of* N*.*
- *There is a deterministic polynomial-time algorithm that, given the identifier of a control node* N*, the identifiers of all nodes and all edge labels on the path to* N*, computes the lexicographically smallest label of an edge (i.e., the in-metadata) of all edges that lead from* N *to one of its successors.*

Definition 27 (Symbolic Indistinguishability for Bi-Protocols)**.** *For a bi-protocol* Π*, we say that* Π *is symbolically indistinguishable if left*$(\Pi) \approx_s$ *right*(Π)*.*

2.4. Review of the Applied π-calculus

In this section, we review the syntax and semantics of the applied π-calculus in a very brief manner.

Figure 2.1 depicts the syntax of the applied π-calculus. The set of terms in the applied π-calculus is the free algebra according to the grammar in Figure 2.1. The applied π-calculus is parametric in a set of uninterpreted function symbols, called *constructors*. As an example, constructors modeling symbolic public-key encryption could contain the constructor $enc_{/3}$ with arity 3 and the constructor $\mathsf{pk}_{/1}$ with arity 1. The term $enc(M, \mathsf{pk}(K), L)$ would represent the result of an encryption operation applied to the message (term) M, the public-key (term) $\mathsf{pk}(K)$, and the randomness (term) L. As a notation, we use $\underline{M}$ for denoting a tuple $M_1, \ldots, M_n$. Moreover, applied π-calculus is parametric in partial functions, called *destructors*, from terms to terms. Whenever a destructor is not defined, we say it fails (denoted as $d(\underline{M}) = \bot$). For symbolic public-key encryption, a destructor *dec* could be defined as $dec(enc(M, \mathsf{pk}(K), L), K) = M$.

Having defined terms in the applied π-calculus, we define the notion of processes, whose grammar is depicted in Figure 2.1 and whose inference rules for the operational semantics is depicted in Figure 2.2. We begin with *plain processes*. The empty process denotes that a process terminates at that point; as syntactic sugar, it is typically omitted from the code. A name restriction $\nu n.P$ produces a fresh name n and afterwards behaves as P with the additional fresh name n. $a(x).P$ expects a message N from the channel a and afterwards behaves as $P\{N/x\}$. $\overline{a}\langle N\rangle.P$ sends a message N on channel a and afterwards behaves as P.[5] $P \mid Q$ models two parallel processes P and Q. $P \mid Q$ nondeterministically behaves for one step as either P or Q and keeps a separate state (i.e., distinct local variable and name scopes) for both processes. $!P$ acts as unboundedly many number of copies of P that are executed in parallel; For $\mathsf{let}\ x = D\ \mathsf{then}\ P\ \mathsf{else}\ Q$ it is checked whether D is of the form $d(\underline{M})$ for a destructor d and terms $\underline{M}$. If D is of that form and if the application fails $\bot = d(\underline{M})$, the processes $\mathsf{let}\ x = D\ \mathsf{then}\ P\ \mathsf{else}\ Q$ behaves as the process Q. If D is of the form $d(\underline{M})$ and the application results in a term $N = d(\underline{M})$ or if $D = N$, the process $\mathsf{let}\ x = D\ \mathsf{then}\ P\ \mathsf{else}\ Q$ behaves as $P\{N/x\}$.[6]

Name restrictions, inputs, and lets restrict the scope of names and variables. In a process P, we write for free variables $fv(P)$ and for free names $fn(P)$. We call a term without any variables a *ground term*. We call a process without free variables a *closed process*. We call a process with a hole $\bullet$ a *context*. We call a context whose hole is not under replication, a conditional (i.e., a let), an input, or an output, an *evaluation context*.

As depicted in Figure 2.2, the operational semantics of the applied π-calculus includes a so-called *structural equivalence* relation ($\equiv$) and a so-called *internal reduction* relation ($\rightarrow$). Processes that are equivalent up to syntactic re-arrangement are *structural equivalence*, and the actual way how processes are executed and synchronized is defined by the *internal reduction*.

Safety properties. As in the work of Fournet, Gordon, and Maffeis [FGM07], we assume that points in the protocol that are security-relevant are annotated with logical predicates,

[5] For the sake of brevity, we use $a(x)$ and $\overline{a}\langle N\rangle$ as input and output command, but the actual applied π-calculus (hence also our case study in Appendix A) uses `in(a,x)` and `out(a,N)`, respectively.

[6] For sets of destructors that contains a destructor `equal` that checks term equality (i.e., $\forall M.\texttt{equals}(M, M) = M$), we write `if` $a = b$ `then` P `else` Q for `let equals`$(a, b) = a$ `in` P `else` Q.

$M, N ::=$	terms
x, y, z	variables
a, b, c	names
$f(M_1, \ldots, M_n)$	constructor application
$D ::=$	destructor term
M	terms
$d(M_1, \ldots, M_n)$	destructor application
$P, Q ::=$	processes
$\overline{M}\langle N\rangle.P$	output
$M(x).P$	input
0	*empty*
$P \mid Q$	parallel composition
$!P$	replication
$\nu a.P$	restriction
let $x = D$ *in* P *else* Q	let
`assume` F	assumption
`assert` F	assertion where F is a formula in first order logic

Figure 2.1.: Syntax of the Applied π-calculus

and security requirements, such as authorization policies, are formulated as logical formulas. Formally, we introduce two processes `assume` F and `assert` F, F being a formula in first order logic. Assumptions and assertions are not further executed; their sole purpose is to express security properties. On a high level, safety holds if all assertions are implied by assumptions in every protocol execution.

Definition 28 (Safety)**.** *A closed process P is* safe *if and only if for every F and Q such that $P \rightarrow^* \nu\underline{a}.(\texttt{assert}\ F \mid Q)$, there exists an evaluation context $E[\bullet] = \nu\underline{b}.\bullet \mid Q'$ such that $Q \equiv E[\texttt{assume}\ F_1 \mid \ldots \mid \texttt{assume}\ F_n]$, $fn(F) \cap \underline{b} = \emptyset$, and we have that $F_1 \wedge \ldots \wedge F_n \implies F$.*

A process that is safe when run in parallel with any *opponent* is called *robustly safe.*

Definition 29 (Opponent)**.** *A closed process is an* opponent *without any* assert*.*

Definition 30 (Robust Safety)**.** *A closed process P is* robustly safe *if and only if $P \mid O$ is safe for every opponent O.*

$$\frac{}{P \mid 0 \equiv P} \qquad \frac{}{P \equiv P} \qquad \frac{}{P \mid Q \equiv Q \mid P}$$

$$\frac{}{(P \mid Q) \mid R \equiv P \mid (Q \mid R)} \qquad \frac{P \equiv Q \qquad Q \equiv R}{P \equiv R}$$

$$\frac{}{\nu a.\nu b.P \equiv \nu b.\nu a.P} \qquad \frac{P \equiv Q}{P \mid R \equiv Q \mid R} \qquad \frac{a \notin fn(P)}{\nu a.(P \mid Q) \equiv P \mid \nu a.Q}$$

$$\frac{P \equiv Q}{\nu a.P \equiv \nu a.Q} \qquad \frac{N \approx N'}{\overline{N}\langle M\rangle.Q \mid N'(x).P \to Q \mid P\{M/x\}}$$

$$\frac{d(\underline{M}) = N \neq \bot}{let\ x = d(\underline{M})\ in\ P\ else\ Q \to P\{N/x\}}$$

$$\frac{d(\underline{M}) = \bot}{let\ x = d(\underline{M})\ in\ P\ else\ Q \to Q} \qquad \frac{}{let\ x = N\ in\ P\ else\ Q \to P\{N/x\}}$$

$$\frac{}{!P \to P \mid\ !P} \qquad \frac{P \to Q}{P \mid R \to Q \mid R \qquad \nu a.P \to \nu a.Q}$$

$$\frac{P' \equiv P \qquad P \to Q \qquad Q \equiv Q'}{P' \to Q'} \qquad \frac{P \to Q}{P \mid R \to Q \mid R \qquad \nu a.P \to \nu a.Q}$$

$$\frac{P' \equiv P \qquad P \to Q \qquad Q \equiv Q'}{P' \to Q'}$$

Figure 2.2.: Semantics of the applied π-calculus

2.5. Embedding from the Applied π-calculus Calculus

In this section, we present the connection between uniform bi-processes in the applied π-calculus and our CS result in CoSP, namely that the applied π-calculus can be embedded into the extended CoSP framework. In contrast to previous work [CC08; CCS12; CH11], we consider CS for bi-protocols from the full applied π-calculus. In particular, we also consider private channels and non-determinate processes.

In Section 2.4, the applied π-calculus for trace properties is reviewed. In this chapter, we consider the variant of the applied π-calculus for bi-protocols (as described in [BAF05]), and thus we only discuss the differences to Section 2.4.

A uniform bi-process [BAF05] in the applied π-calculus is the counterpart of a uniform bi-protocol in CoSP. A bi-process is a pair of processes that only differ in the terms they operate on. Formally, they contain expressions of the form $choice[a, b]$, where a is used in

the left process and b is used in the right one. A bi-process Q can only reduce if both its processes can reduce in the same way.

Definition 31 (Uniform Bi-process)**.** *A bi-process Q in the applied π-calculus is* uniform *if $left(Q) \to R_{left}$ implies that $Q \to R$ for some bi-process R with $left(R) \equiv R_{left}$, and symmetrically for $right(Q) \to R_{right}$ with $right(R) \equiv R_{right}$.*

2.5.1. Embedding into CoSP

The execution of an process P in the applied π-calculus can be modeled as a CoSP protocol $e(P)$ as defined by Backes, Hofheinz, and Unruh [BHU09]. The function e is called embedding. $e(P)$ sends the current state of P to the attacker, who replies with a the description of an evaluation context E that determines which part of P should be reduced next. Thus the attacker is given control over the whole scheduling of the process.

The state of P is used as a node identifier. An execution of the process performs only the following operations on terms: Applying constructors (this includes nonces) and destructors, comparing using *equals*,[7] and sending and receiving terms. However, terms are not encoded directly into the node identifier; instead, the node in which they were created (or received) is referenced instead. This is due to the fact that a CoSP protocol allows to treat terms only as black boxes. Note that the process P and the terms occurring within P will be encoded in the node identifier (encoded as bitstrings). Operations on messages can then be performed by using constructor and destructor nodes, and the input and output of terms is handled using input/output nodes. If the attacker wants to send or receive on a public channel, she is forced to produce the term that corresponds to the channel.

To ensure that the resulting protocols are uniformity-enforcing, we have to modify the embedding slightly as discussed in Section 6.2.1. The modification is highlighted in blue. This modification clearly does not influence the validity of the computational soundness result of the applied π-calculus that has been established in the original CoSP framework [BHU09] for trace properties.

Definition 32 (Symbolic Execution of a Process in the Applied π-calculus (based on [BHU09]))**.** *Let P_0 be a closed process, and let $\mathcal{A}$ be an interactive machine called the attacker. We define the* symbolic π-execution *as an interactive machine* SExec_{P_0} *that interacts with $\mathcal{A}$:*

- Start*: Let $P := P_0$ (where we rename all bound variables and names such that they are pairwise distinct and distinct from all unbound ones). Let η be a totally undefined partial function mapping variables to terms, let μ be a totally undefined partial function mapping names to terms. Let $a_1, \ldots, a_n$ denote the free names in P_0. For each i, choose a different $r_i \in \mathbf{N}_P$ Set $\mu := \mu(a_1 := r_1, \ldots, a_n := r_n)$. Send $(r_1, \ldots, r_n)$ to $\mathcal{A}$.*
- Main loop: *Send P to the attacker and include the address of the resulting control node in the protocol tree in its out-metadata. Expect an evaluation context E from the attacker and distinguish the following cases:*
  - *$P = E[M(x).P_1]$: Request two CoSP-terms c, m from the attacker. If $c \cong \mathrm{eval}^{\mathsf{fw}}(M\eta\mu)$, set $\eta := \eta(x := m)$ and $P := E[P_1]$.*

[7] For instance, this is used to determine if two processes can communicate, i.e., if a channel on which a message should be sent matches a channel on which a message should be received.

- $P = E[\nu a.P_1]$: *Choose* $r \in \mathbf{N}_P \setminus range(\mu)$, *set* $P := E[P_1]$ *and* $\mu := \mu(a := r)$.
- $P = E[\overline{M_1}\langle N\rangle.P_1][M_2(x).P_2]$: *If* $\text{eval}^{\mathsf{fw}}(M_1)\eta\mu \cong \text{eval}^{\mathsf{fw}}(M_2\eta\mu)$ *then set* $P := E[P_1][P_2]$ *and* $\eta := \eta(x := \text{eval}^{\mathsf{fw}}(N\eta\mu))$.
- $P = E[let\ x = D\ in\ P_1\ else\ P_2]$: *If* $m := \text{eval}^{\mathsf{fw}}(D\eta\mu) \neq \bot$, *set* $\eta := \eta(x := m)$ *and* $P := E[P_1]$. *Otherwise set* $P := E[P_2]$.
- $P = E[!P_1]$: *Rename all bound variables of* P_1 *such that they are pairwise distinct and distinct from all variables and names in* P *and in the domains of* η *and* μ, *yielding a process* $\tilde{P}_1$. *Set* $P := E[\tilde{P}_1 \mid !P_1]$.
- $P = E[\overline{M}\langle N\rangle.P_1]$: *Request a CoSP-term* c *from the attacker. If* $c \cong \text{eval}^{\mathsf{fw}}(M\eta\mu)$, *set* $P := E[P_1]$ *and send* $\text{eval}^{\mathsf{fw}}(N\eta\mu)$ *to the attacker.*
- *In all other cases, do nothing.*

Modifications for Bi-processes. If P is the left or the right variant of a bi-process in the applied π-calculus, expressions of the form $choice[a, b]$ are translated to subtrees in the CoSP protocol that compute both a and b entirely. References to such expressions are translated to bi-references in the natural way. When a description of the process P is sent to the attacker, expressions of the form $choice[a, b]$ are sent untouched; if only a would be sent left and only b right, the attacker could trivially distinguish the bi-protocol.

Relating CoSP and the Applied π-calculus. The following lemma connects uniformity in the applied π-calculus to uniformity in CoSP by demonstrating that uniform bi-processes in the applied π-calculus correspond to uniform bi-protocols in CoSP.

Lemma 1 (Uniformity in CoSP and the Applied π-calculus)**.** *Let a bi-process* Q *in the applied* π*-calculus be given. There is an embedding* e *from bi-processes in the applied* π*-calculus to CoSP bi-protocols with the following property: If* $left(Q)$ *and* $right(Q)$ *are uniform, then* $left(e(Q)) \approx_s right(e(Q))$ *and* $range(e)$ *is uniformity-enforcing.*

Proof. We show the contrapositive. Suppose that we have $left(e(Q)) \not\approx_s right(e(Q))$. Then there there is a distinguishing attacker strategy V_{In}, i.e., there is a view $V \in [V_{In}]_{SViews(left(e(Q)))}$ such that for all $V' \in [V_{In}]_{SViews(right(e(Q)))}$ we have $V \not\sim V'$ (the symmetric case is completely analogous).

Let V, V' be defined accordingly. As $V \sim V'$, there is a shortest prefix v such that for the prefix v' of V' (of the same as v), we have $v \not\sim v'$. Let v_{In} be the corresponding prefix of the attacker strategy V_{In}. Recall that v_{In} is a list that contains operations to create input terms to the protocol and in-metadata to schedule the execution of the protocol. This suffices to construct an evaluation context E_{In} from v_{In}: By construction of e, input nodes are used in $e(Q)$ to give the attacker the possibility to send or receive on a channel. If the attacker is able to produce the channel term, the corresponding action is performed; in the send case the attacker has to provide the term additionally. The operations in v_{In} are used in E_{In} to produce the required channel name as well as the term to be sent. Moreover, the in-meta data yields a unique reduction t for $left(Q)$ such that $E_{In}[left(Q)] \to^t P_{left}$.

We distinguish three cases. First, the shape is different (1): there is an i such that $v_i = (s, x)$ and $v'_i = (s', y)$ and $s \neq s'$ $(s, s' \in \{\texttt{out}, \texttt{in}, \texttt{control}\})$. Second, the out-metadata is different (2): $v_{Out\text{-}Meta} \neq v'_{Out\text{-}Meta}$. Third, static equivalence fails (3): $K(v_{Out}) \neq K(v'_{Out})$.

In case (1), different shapes in the left and right protocol correspond to different protocol actions and thus to different cases in the main loop of the symbolic execution of the process in the applied π-calculus. By construction of this execution, we conclude that the left protocol has received other in-metadata that the right one. This contradicts the fact that V and V' are views under the same attacker strategy V_{In}.

For case (2), recall that the out-metadata at control nodes is only used to send a state of the executed process to the attacker, and to raise events. Thus, if the out-metadata differs, then the left and the right process have reduced differently. More formally, we have $\mathit{left}(Q) \to P_{\mathit{left}}$ but there is no bi-process P such that $Q \to P$ and $\mathit{left}(P) = P_{\mathit{left}}$ because otherwise there would be a view v' of $\mathit{right}(e(Q))$ inducing a reduction $\mathit{right}(Q) \to P_{\mathit{right}}$ with $P_{\mathit{right}} \equiv \mathit{right}(P)$.[8] This shows that Q is not uniform.

In the remaining case (3) we construct an evaluation context E_u that breaks the observational equivalence (and thus the uniformity) of P. We know that there is a symbolic operation O such that $K_V(O) \neq K'_V(O)$. In other words, $O(V_{Out}) = \bot$ and $O(V'_{Out}) \neq \bot$ (or vice versa). This induces the context E_u that executes the constructors and destructors corresponding to O and branches depending on the result of the whole operation. Finally, the combined context $E_u[E_{In}[\cdot]]$ distinguishes $\mathit{left}(Q)$ and $\mathit{right}(Q)$. Thus Q is not uniform. □

[8] Note that the CoSP bi-protocol is deterministic, because it is efficient in the sense of Definition 26. Hence it additionally renames bound variables and chooses nonces $r_i \in \mathbf{N_P}$ consistently.

2.6. Equivalence Notions

ProVerif is not able to handle complicated destructors such as *len*, which are typically defined recursively, e.g., $\mathit{len}(\mathit{pair}(t_1, t_2)) = \mathit{len}(t_1) + \mathit{len}(t_2)$. Recent work by Cheval, Cortier, and Plet [CCP13] extends the automated protocol verifier APTE [CCD11; Che], which is capable of proving trace equivalence of two processes in the applied π-calculus (without replication), to support such length functions. However, trace equivalence is a weaker notion than uniformity, i.e., there are bi-processes that are trace equivalent but not uniform, our computational soundness result does not carry over to APTE.[9] Due to the lack of a tool that is able to check (only) uniformity as well as to handle length functions properly, we explain how APTE can be combined with ProVerif to make protocols on the symbolic model of our case study amendable to automated verification.

In [CCP13], the notion of trace equivalence w.r.t. length is defined. Particularized for a bi-process Q, the definition states essentially that Q is trace-equivalent w.r.t. length if for any sequence of input or output actions that an attacker can reach with $\mathit{left}(Q)$, the same sequence is reachable in $\mathit{right}(Q)$ such that the resulting attacker knowledge is statically equivalent, and vice versa. Here, static equivalence means that the attacker has no test (built from destructors and constructors) that tells the knowledge reached in $\mathit{left}(Q)$ apart from the knowledge reached in $\mathit{right}(Q)$. This holds in particular for the test that returns the length associated with a term in the knowledge of the attacker. We refer the reader to [CCP13] for a precise definition. Since any *len* destructor as introduced in the symbolic model $\mathbf{M}$ in Section 6.3 is linear, it can be regarded as the mentioned length test. Consequently, the analysis of APTE leads to guarantees that are meaningful with respect to the established computational soundness result.

On the other hand, ProVerif is able to decide whether the following holds on a input bi-process Q:

1. if $Q' \equiv K'[\overline{N}\langle M\rangle.P \mid N'(x).R]$ then $\mathit{left}(N) = \mathit{left}(N')$ if and only if $\mathit{right}(N) = \mathit{right}(N')$,
2. if $Q' \equiv K'[\mathit{let}\ x = d\ \mathit{in}\ P\ \mathit{else}\ R]$ then $\mathit{eval}(\mathit{left}(d)) = \bot$ if and only if $\mathit{eval}(\mathit{right}(d)) = \bot$.

In this case, we say that Q is *PV-uniform*. Blanchet, Abadi, and Fournet [BAF05] prove that PV-uniformity implies uniformity.

Our goal is to show that if a bi-process Q is PV-uniform w.r.t. $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$ with $\mathbf{D}'_{pi} = \mathbf{D}_{pi} \setminus \{\mathit{len}/1\}$, and its left and right variant are trace equivalent w.r.t. $\mathbf{C}_{pi} \cup \mathbf{D}_{pi}$, then it is also PV-uniform w.r.t $\mathbf{C}_{pi} \cup \mathbf{D}_{pi}$. To this end, we prove the following slightly more general lemma.

Lemma 2. *Let Q be a bi-process in the applied π-calculus and $\mathbf{C}_{pi}$ be a set of constructors and $\mathbf{D}_{pi}$ be a set of destructors. If*

(i) Q only uses the constructors $\mathbf{C}_{pi}$ and the destructors $\mathbf{D}'_{pi} \subseteq \mathbf{D}_{pi}$,

(ii) Q is PV-uniform with $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$, and

(iii) for all $d \in \mathbf{D}_{pi} \setminus \mathbf{D}'_{pi}$ and for all terms t, the existence of a term t' such that

[9] Cheval, Cortier, and Delaune [CCD13] show that trace equivalence implies observational equivalence for so-called determinate processes. Recall that the even stronger notion of uniformity, which is only defined for bi-processes, was introduced as a means to prove observational equivalence.

$eval(d(t)) = t'$ implies that t' can be built using constructors only, and

(iv) $left(Q)$ and $right(Q)$ are trace equivalent,

then Q is PV-uniform against an attacker that uses $\mathbf{C}_{pi} \cup \mathbf{D}_{pi}$.

Proof. Assume towards contradiction that Q is not PV-uniform against an attacker that uses $\mathbf{C}_{pi} \cup \mathbf{D}_{pi}$, but Q is PV-uniform against an attacker that uses $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$, Q only uses $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$, and assume that $left(Q)$ and $right(Q)$ are trace equivalent.

Recall that if Q is not PV-uniform (against an attacker that uses $\mathbf{C}_{pi} \cup \mathbf{D}_{pi}$), then there are evaluation contexts K, K' and a bi-process Q' such that $K[Q] \to^* Q'$ and one of the following two properties are violated:

1. if $Q' \equiv K'[\overline{N}\langle M\rangle.P \mid N'(x).R]$ then $left(N) = left(N')$ if and only if $right(N) = right(N')$,
2. if $Q' \equiv K'[let\ x = d\ in\ P\ else\ R]$ then $eval(left(d)) = \bot$ if and only if $eval(right(d)) = \bot$.

W.l.o.g. we can assume that Q' is the first process in the reduction sequence $K[Q] \to^* Q'$ that violates the PV-uniformity-properties.

Case 1: In this case, we have w.l.o.g. $left(N) = left(N')$ but $right(N) \neq right(N')$. Since Q' is the first process that violates the PV-uniformity properties, 1 and 2 hold in all previous reduction steps. We distinguish two cases. First, *the protocol communicates with the attacker*, i.e., either $\overline{N}\langle M\rangle.P$ or $N'(x).R$ is not a residue process of the protocol Q. Then, the trace of $left(Q)$ and $right(Q)$ shows whether the communication over N and N' succeeds. Consequently, $right(N) \neq right(N')$ contradicts the trace equivalence of $left(Q)$ and $right(Q)$, i.e., assumption (iv).

Second, *the protocol performs an internal communication*, i.e., both $\overline{N}\langle M\rangle.P$ and $N'(x).R$ are residues processes of the protocol Q. In this case, we know by assumption (iii) that the attacker context K (which uses $\mathbf{C}_{pi} \cup \mathbf{D}_{pi}$) together with the protocol process Q cannot produce more terms than a corresponding context K'' that only uses $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$ together with Q. Even though K with Q can branch differently than K'' with Q, we know by assumption that all previous reduction steps satisfied the PV-uniformity property 2. Hence, there is a context K'' that only uses $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$ but has a reduction sequence $K'' \to^* Q''$ such that

$$Q'' \equiv K'''[\overline{N}\langle M\rangle.P \mid N'(x).R]$$

Then, however, $right(N) \neq right(N')$ contradicts the PV-uniformity of Q against an attacker that only uses $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$, i.e., assumption (ii).

Case 2: In this case, we assume w.l.o.g. that $eval(left(d)) = \bot$ and $eval(right(d)) \neq \bot$. We distinguish two cases. First, *a protocol test fails in $left(Q)$ but not in $right(Q)$*, i.e.,

$$let\ x = d\ in\ P\ else\ R$$

is inside a residue process of the protocol Q. In this case, (with the same argumentation as above) there is, by assumption iii, a context K'' that only uses $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$ but has a reduction sequence $K'' \to^* Q''$ such that

$$Q'' = K'''[let\ x = d\ in\ P\ else\ R]$$

$eval(left(d)) = \bot$ and $eval(right(d)) \neq \bot$, however, contradicts the PV-uniformity of Q against attackers that only use $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$, i.e., assumption (ii).

Second, *the attacker distinguishes left*(Q) *and right*(Q) *with the test* d, i.e.,

$$\mathit{let}\ x = d\ \mathit{in}\ P\ \mathit{else}\ R$$

is not inside a residue process of the protocol Q and $\mathit{eval}(\mathit{left}(d)) = \bot$ and $\mathit{eval}(\mathit{right}(d)) \neq \bot$. Then, $\mathit{eval}(\mathit{left}(d)) = \bot$ and $\mathit{eval}(\mathit{right}(d)) \neq \bot$ breaks the static equivalence property that the trace equivalence requires, i.e., assumption (iv). □

The lemma allows us to prove that the combined analysis using ProVerif and APTE is sound.

Theorem 2. *Let* $\mathbf{C}_{pi} \cup \mathbf{D}_{pi}$ *be the symbolic model defined in Section 6.3. Let* Q *be a bi-process in the applied* π*-calculus that uses only* $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$ *with* $\mathbf{D}'_{pi} = \mathbf{D}_{pi} \setminus \{\mathit{len}\}$*. If ProVerif proves uniformity for* Q *with* $\mathbf{C}_{pi} \cup \mathbf{D}'_{pi}$ *and APTE proves trace-equivalence w.r.t. length with* $\mathbf{C}_{pi} \cup \mathbf{D}_{pi}$*, then* Q *is uniform with* $\mathbf{C}_{pi} \cup \mathbf{D}_{pi}$*.*

Proof. The theorem follows from Lemma 2. Condition (iii) of the lemma is satisfied, because the destructor *len* outputs only length specifications, which can be built using the constructors O and S. The other conditions hold by assumption. □

Note. Assumption (iii) in Lemma 2, i.e., that the destructors in $\mathbf{D}_{pi} \setminus \mathbf{D}'_{pi}$ do not allow the attacker to learn new terms is essential, because otherwise the following counter-example breaks the uniformity of Q with $\mathbf{C}_{pi} \cup \mathbf{D}_{pi}$:

$$\begin{aligned} Q :=\ & \overline{c}\langle \mathit{hidden}(\mathit{magic})\rangle.c(x). \\ & \quad \mathit{let}\ y = \mathit{equals}(x, \mathit{magic})\ \mathit{in} \\ & \quad\ \ \mathit{let}\ z = \mathit{equals}(a, \mathit{choice}[a, b]) \\ & \quad\ \ \ \ \mathit{in}\ 0\ \mathit{else}\ 0 \\ & \quad \mathit{else}\ P \\ K[\bullet] :=\ & \nu c.\big(c(x).\mathit{let}\ y = \mathit{showme}(\mathit{hidden}(x))\ \mathit{in}\ \overline{c}\langle y\rangle \mid \bullet\big) \end{aligned}$$

Here, $\mathit{showme}(\mathit{hidden}(m)) = m$ and $\mathit{showme} \in \mathbf{D}_{pi} \setminus \mathbf{D}'_{pi}$.

Chapter 3.

A Computationally Sound Symbolic Abstraction for Malleable Zero-knowledge Proofs

[This chapter is based on a work with Michael Backes, Fabian Bendun, Matteo Maffei, and Kim Pecina [BBMMP15]. I am the main contributor to all parts that occur in this chapter.]

3.1. Motivation

While symbolic models traditionally only cover basic cryptographic primitives such as encryption and signatures, recent work started incorporating more sophisticated primitives, such as zero-knowledge (ZK) proofs [GMR89]. Zero-knowledge (ZK) proofs prove computational statements while hiding parts of that statement, making it an enabling building block for applications that require more than solely secrecy or authenticity of communication. In more detail, a ZK proof combines two properties that might occur contradictory. On the one hand, a ZK proof proves of a statement, e.g., "the plaintext of this ciphertext constitutes a signature of the certificate X" (for some certificate X), and forging this statement is either impossible or computationally hard. On the other hand, a ZK proof does not leak anything more than the sole fact that the statement is valid.

In addition to these core properties, commonly used ZK proof schemes, such as the Groth-Sahai proof system [GS08], offer a novel type of cryptographic flexibility. First, a participant is able to re-randomize existing ZK proofs, which is fundamental for achieving unlinkability in anonymity protocols. Second, in order to adhere to individual privacy requirements, a participant can hide public parts of a ZK proof statement to selectively hide information of third-party proofs (e.g., this enables the design of privacy-preserving credentials for open-ended systems [BMP12; MP11; MPR13]). Third, a participant can logically compose ZK proofs in order to construct new proof statements. ZK proof systems that permit these transformations are called *malleable*. In addition to offering this extended functionality, malleable ZK constructions are often significantly more efficient than their non-malleable counterparts.

The flexibility and efficiency of malleable ZK proofs is highly desirable in practice. But existing symbolic abstractions are restricted to non-malleable ZK proofs, which model ZK proofs as monolithic building blocks that cannot be further transformed [BHM08b; BHM12; BMU08]. A symbolic model for malleable ZK proofs is intrinsically more difficult for automated verification techniques because the much more comprehensive adversary

model that includes ZK transformations requires a significantly more involved symbolic analysis.

The symbolic models of non-malleable ZK proofs have been justified by computational soundness results, i.e., a successful symbolic analysis carries over to the corresponding cryptographic ZK realizations [BBU13; BU10]. The symbolic model of malleable ZK proofs imposes challenges for such a result due to the significantly more complex adversary model.

Our Contribution. First, we provide a symbolic abstraction of malleable ZK (MZK) proofs by means of an equational theory.[1] The main conceptual challenge we faced when devising this abstraction was to identify a finite representation of the infinite number of possible transformations that are available to the adversary. Roughly, we categorize transformations as one of the three types: *re-randomizing*, *logical transformations* (used, e.g., to produce a proof of the statement $x \wedge y$ from independent proofs of x and y, or to prove $\exists w.x$ from a proof of x, thereby hiding the witness w in the statement x), and *functional transformations* (used, e.g., to prove $enc(k, x, r) = enc(k, y, r)$ from a proof of $x = y$ for two secret values x, y, key k, and randomness r).

The last category of transformations (i.e., functional transformations) is rarely used in the literature, but it is nevertheless available to the attacker, as shown by Fuchsbauer [Fuc10, Lemma 6]. Therefore, we present two variations of our symbolic model that only differ in this last category: the *fully MZK (FMZK) abstraction* grants the attacker the capability to apply transformations that modify the witnesses of a proof, which allows for weaker cryptographic realizations; the *controlled MZK (CMZK) abstraction* excludes this kind of transformations but requires a slightly less efficient cryptographic realization. Concerning automated verification, the CMZK abstraction is accessible to standard automated reasoning tools for equational theories, whereas reasoning about the FMZK abstraction additionally requires solving constraints, e.g., via a theorem prover.

Second, we prove the computational soundness of the FMZK and CMZK abstractions with respect to trace properties. We first identify the class of *MZK-safe* protocols, which basically disallows the reuse of randomness as well as revealing signature keys or decryption keys to the adversary. We then establish computational soundness of the FMZK abstraction for all MZK-safe protocols based on weak cryptographic definitions (non-interactive zero-knowledge arguments of knowledge). For establishing the computational soundness of the CMZK abstraction for all MZK-safe protocols, we leverage the cryptographic construction for controlled malleability proposed by Chase et al. [CKLM12]. These results are given in CoSP [BHU09], a modular and generic framework for symbolic protocol analysis and computational soundness proofs. The process of embedding calculi is decoupled from computational soundness proofs of cryptographic primitives. As a result, our work immediately entails a computationally sound symbolic model in the applied-pi calculus, and we show that our result also entails a computationally sound symbolic abstraction in ML (building on results from [BMU10]).

Outline of this chapter. Our symbolic abstraction of malleable ZK proofs is presented in Section 3.2 and shown computationally sound in Section 3.3. Section 3.7 concludes the chapter.

[1] We further consider asymmetric encryptions and digital signatures, handled in a standard way [BHU09].

$$
\begin{aligned}
dec(dk(t_1), enc(ek(t_1), m, t_2)) &= m & fst(pair(t_1, t_2))) &= t_1\\
ver_{sig}(vk(t_1), sig(sk(t_1), t_2, t_3)) &= t_2 & snd(pair(t_1, t_2)) &= t_2\\
ekof(enc(ek(t_1), t_2, t_3)) &= ek(t_1) & unstring_1(string_1(s)) &= s\\
vkof(sig(sk(t_1), t_2, t_3)) &= vk(t_1) & unstring_0(string_0(s)) &= s\\
& & equals(x, x) &= x
\end{aligned}
$$

Figure 3.1.: Definition of $\mathbf{D}'$: basic destructors

3.2. Symbolic abstraction of Malleable ZK Proofs

This section presents our symbolic abstraction for malleable zero-knowledge (MZK) proofs by means of an equational theory. We introduce two variations of our symbolic model: *fully-malleable zero-knowledge proofs* (the FMZK variation) and *controlled-malleable zero-knowledge proofs* (the CMZK variation). The two variations differ only in their degree of malleability: the FMZK variation also permits functional transformations, whereas the CMZK variation excludes this case and is therefore better suited for automated verification tools.

For the sake of presentation, we first present a standard symbolic model for digital signatures and public-key encryptions, which closely resembles previous work [BBU13; BHU09]. Thereafter, we extend this symbolic abstraction with MZK proofs. Our symbolic abstraction constitutes a symbolic model $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D}, \vdash)$. In what follows, we write $\underline{x}$ for a tuple $(x_i)_{i=1}^n$.

3.2.1. The basic symbolic model

Terms. In the basic symbolic model, messages are modeled as *basic terms* $\mathbf{T}'$; the grammar for this set is presented below. Nonces are modeled as distinct terms from a countably infinite set $\mathbf{N} \subset \mathbf{T}'$, with two disjoint subsets for protocol nonces $\mathbf{N}_P \subset \mathbf{N}$ and attacker nonces $\mathbf{N}_E \subset \mathbf{N}$. Let $N, N' \in \mathbf{N}$ and $M, M' \in \mathbf{T}'$. Then, pairs are represented as $pair(M, M')$, an encryption key as $ek(N)$, a decryption key as $dk(N)$, the encryption of a message M with the key $ek(N)$ and randomness N' as $enc(ek(N), M, N')$, a signature key as $sk(N)$, a verification key as $vk(N)$, the signature of a message M with the key $sk(N)$ and randomness N' as $sig(sk(N), M, N')$. Moreover, we also incorporate into the symbolic model payloads, expressed by the three constructors ε, $string_0$ and $string_1$. We define the set $\mathbf{C}' := \{enc, ek, dk, sig, vk, sk, pair, \varepsilon, string_0, string_1\}$ of *basic constructors*. The grammar of terms, which subsumes the grammar of basic terms, is depicted in Figure 3.3 (excluding technical terms, see Figure 3.7 for the full grammar).

Explicitly modeling randomness. Cryptographic terms such as encryptions and signatures include an explicit randomness term in our symbolic model. Beside making the model more accurate, such randomness terms allow modeling MZK proofs about ciphertexts. Moreover, this randomness allows MZK proofs that prove ownership of a ciphertext.

Destructors. Operations on messages are modeled as partial functions $f : \mathbf{T}'^n \to \mathbf{T}'$. Whenever such a function f, called destructor, is undefined on some $\underline{t}$ we write $f(\underline{t}) = \bot$. As an example, consider equality, which is modeled as the 2-ary destructor $equals := \{((x, x), x) \mid x \in \mathbf{T}'\}$. This set $\mathbf{D}'$ of *basic destructors* is defined in Figure 3.1.

$$\frac{m \in S}{S \vdash m} \qquad \frac{S \vdash \bar{t} \quad \bar{t} \in \mathbf{T} \quad F \in \mathbf{C}' \cup \mathbf{D}' \quad eval_F(\bar{t}) \neq \bot}{S \vdash eval_F(\bar{t})} \qquad \frac{N \in \mathbf{N}_E}{S \vdash N}$$

Figure 3.2.: Knowledge relation for the basic model

Symbolic attacker in the basic model. The main advantage of the symbolic model is that the attacker is restricted to a small set of well-defined actions. These actions are formalized by a deduction relation $\vdash$: $2^{\mathbf{T}'} \times \mathbf{T}'$, called the *knowledge relation*, where $S \vdash t$ characterizes the messages t that the attacker can compute given some knowledge $S \subseteq \mathbf{T}'$. This knowledge relation is canonically defined over the set of basic constructors $\mathbf{C}'$ and destructors $\mathbf{D}'$ (see Figure 3.2). For a knowledge set $S \subset \mathbf{T}'$ the expression $S \vdash m$ denotes that the attacker can compute m out of his knowledge S.

Sort-destructors & garbage terms. For the sake of brevity, we omitted two kinds of terms and destructors in the presentation of the basic symbolic model. The full grammar can be found in Figure 3.7. First, in our complete symbolic model (see Figure 3.6) we include for every constructor f a so-called *sort-destructor* isF, defined as $isF(f(x)) = f(x)$ for all terms x. Second, since we require (potentially forgeable) message types for each cryptographic primitive, we have to model ill-typed messages, such as signatures that do not pass verification. In order to achieve computational soundness, we model these wrongly-typed messages in our symbolic model via so-called *garbage constructors*. At this point, we stress that in contrast to previous work, we do not include a ZK-garbage constructor, since an ill-types ZK-proof will have garbage commitments (see Section 3.6.4).

3.2.2. Symbolic MZK proofs

We first present the syntax of our model. Thereafter, we discuss how we define the validity of symbolic MZK proofs, in particular how we prevent an MZK proof of a disjunction $A \vee B$ from leaking the validity A or B alone. Then, we introduce a finite set of destructors that characterizes all possible transformations that a cryptographic attacker (against trace properties) can perform. Finally, we describe the symbolic attacker. For the sake of readability, we postponed the full specifications for some parts of the symbolic model to the appendix (Section 3.8).

3.2.2.1. Terms and statements

A MZK proof is represented symbolically as a term $ZK(s, r, N)$, where ZK is a constructor, s is the statement, r is auxiliary verification information, and N is a randomness symbol. A statement is basically a propositional logic formula over equations of terms. In order to give a finite characterization of all possible transformations of MZK proofs in our setting, we explicitly model cryptographic commitments that are typically used in a cryptographic realization of MZK proofs. A symbolic commitment on a term t with randomness r' and the CRS[2] $crs(r)$ is represented as $com(crs(r), t, r')$. For a commitment $com(crs(r), t, r')$, the corresponding unveiling information is modeled as $uv(t, r')$, and a destructor *open* such that $open(com(crs(r), t, r'), uv(t, r')) = t$. A witnesses w is represented as a commitment

[2]The common reference string (CRS) is a central assumption in the construction of many commitment schemes that are used in zero-knowledge proofs

$$
\begin{aligned}
T ::= \;& enc(ek(N), T, N) \mid ek(N) \mid dk(N) \mid pair(T, T) \mid \\
& sig(sk(N), T, N) \mid vk(N) \mid sk(N) \mid S \mid N \mid crs(N) \\
& ZK(S, R, M) \mid com(crs(N), T, M) \mid uv(T, M) \\
Q ::= \;& \varepsilon \mid string_0(Q) \mid string_1(Q)
\end{aligned}
$$

$$
\begin{aligned}
S ::= \;& S \wedge S \mid S \vee S \mid TX = TX \mid TX \neq TX \\
TX ::= \;& \langle com(crs(n), TY, M), U \rangle \mid F\langle \underbrace{TX, \ldots, TX}_{m-times} \rangle \\
R ::= \;& R \wedge R \mid R \vee R \mid RX = RX \mid RX \neq RX \\
RX ::= \;& M \mid F\langle \underbrace{RX, \ldots, RX}_{m-times} \rangle \qquad U ::= uv(TY, M) \mid \varepsilon \\
TY ::= \;& T \mid N \qquad M ::= N \mid rand(M, M) \qquad N ::= n
\end{aligned}
$$

where $F \in StF$, $n \in \mathbf{N}$ and m is the arity of F.

Figure 3.3.: The syntax of terms (selection)

$com(crs(r), w, r')$ in the statement. Public information is also represented as a commitment $com(crs(r), m, r')$; however, in contrast to witnesses, public information in the statement also carries its unveiling information $uv(m, r')$, hence it is possible to retrieve m.

Analogous to common cryptographic MZK proof schemes (e.g., the Groth-Sahai scheme [GS08]), we represent public information as commitments together with unveiling information.

Statements. A symbolic statement is the encoding of a propositional logic formula over atomic statements. Atomic statements are equalities and negated equalities of terms. For simplifying the automated verification of MZK transformations, we require these logical formulas to be in conjunctive normal form.

We represent statements as encodings of contexts with commitments instead of holes. The tags AND, OR, COM, $EQUALS$, $NEQUALS$ are encoded using $string_1$, $string_0$, ε. We introduce for every constructor and destructor $F \in StF := \{ver_{sig}, fst, snd, enc, pair, sig\}$ a tag $\hat{F}$. We use the following abbreviations, where for $A_1, \ldots, A_n, A, B$ we write $\langle A_1, \ldots, A_n \rangle$ for $pair(A_1, \ldots pair(A_{n-1}, A_n) \ldots)$, $A \wedge B$ for $\langle AND, A, B \rangle$, $A \vee B$ for $\langle OR, A, B \rangle$, $A = B$ for $\langle EQUALS, A, B \rangle$, $A \neq B$ for $\langle NEQUALS, A, B \rangle$, and $F\langle A_1, \ldots, A_n \rangle$ for $\langle \hat{F}, A_1, \ldots, A_n \rangle$, where $F \in StF$.

Terms. In addition to the basic symbolic model from the previous section, the set $\mathbf{T}$ of terms contains commitments and MZK proofs. MZK proofs are modeled as terms $ZK(t, r, N)$ that consist of three parts: a statement t as described above, a so-called *randomness tree* r, which is basically a commitment to the commitments in a statement (see Section 3.2.2.2), and a randomness nonce N.

Formally, the set of *terms* $\mathbf{T}$ is generated by the grammar from Figure 3.7 for the non-terminal T and for which the logical formulas that correspond to the statements t inside a term $ZK(t, r, N)$ are in CNF. The set of *constructors* is defined as $\mathbf{C} := \mathbf{C}' \cup \{ZK, com, crs, uv, rand\}$.

Example 1: Anonymous webs of trust. We will illustrate our approach on the anonymous webs of trust protocol [BLMP10], which can be seen as a form of anonymous delegatable

$$
\begin{aligned}
ver_{zk}(crs, ZK(t,r,N)) &= check_{zk}(crs,t,r)\\
getPub(ZK(t_1,r_1,N)) &= t_1\\
setPub(ZK(t_1,r_1,N),t_2,N') &= ZK(t_2,r_1,rand(N,N'))\\
setPubA(ZK(t_1,r_1,N),t_2) &= ZK(t_2,r_1,N)\\
and_{ZK}(ZK(t_1,r_1,N),ZK(t_2,r_2,N')) &= ZK(t_1\wedge t_2, r_1\wedge r_2, rand(N,N'))\\
splitAnd(ZK(t_1\wedge t_2,r_1\wedge r_2,rand(N,N'))) &= pair(ZK(t_1,r_1,N),ZK(t_2,r_2,N'))\\
or_{ZK}(ZK(t_1,r_1,N),ZK(t_2,r_2,N')) &= ZK(t_1\vee t_2, r_1\vee r_2, rand(N,N'))\\
commute(ZK(t_1\vee t_2,r_1\vee r_2,N)) &= ZK(t_2\vee t_1, r_2\vee r_1, N)\\
rer_{zk}(ZK(t_1,r_1,N),r_2,N') &= ZK(rerPr(t_1,r_2,N'),rerPr(r_1,r_2,N'),\\
&\qquad rand(N,N'))\\
open(crs(t_1),com(crs(t_1),t_2,t_3),uv(t_2,t_3)) &= t_2\\
crsof(com(crs(t_1),t_2,t_3)) &= crs(t_1)\\
rer_{com}(com(crs(t_1),t_2,t_3),t_4) &= com(crs(t_1),t_2,rand(t_3,t_4))\\
applyF(com(crs(t_1),t_2,t_3),t_4) &= com(crs(t_1),f(t_2,t_4),t_3) \text{ for } f\in\mathbf{D}\cup\mathbf{C}
\end{aligned}
$$

Figure 3.4.: The set $\mathbf{D}''$ of destructors for ZK proofs for the FMZK model. The CMZK model does not contain or_{ZK}, $applyF$, and $setPub$ can only be used to hide witnesses.

credentials [BCCKLS09] (also known as anonymous proxy signatures [FP09]). In webs of trust, a party A shows that it trusts a party B by signing the verification key of B ($sig(sk_A, vk_B, r)$). A chain of trust from A to B via C can be expressed as follows: $sig(sk_C, vk_B, r_1), sig(sk_A, vk_C, r_2)$.

Anonymous webs of trust additionally hide the identity of all parties except from the party to whom the message is sent, i.e., the statement of a proof that there is an anonymous chain of trust to A looks as follows:

$$\exists s_C, s_A, v_B, v_C. ver_{sig}(v_C, s_C) = v_B \wedge ver_{sig}(vk_A, s_A) = v_C$$

This statement corresponds to the following term:

$$\hat{ver}_{sig}\big\langle\langle c_{v_C},\varepsilon\rangle,\langle c_{s_C},\varepsilon\rangle\big\rangle = \langle c_{v_B},\varepsilon\rangle \wedge \hat{ver}_{sig}\big\langle\langle c_{vk_A}, uv(\mathrm{vk}_A, r_{c_{\mathrm{vk}_A}})\rangle,\langle c_{s_A},\varepsilon\rangle\big\rangle = \langle c_{v_C},\varepsilon\rangle$$

where c_t denotes the commitment to the message t. Such an anonymous chain of trust can be used by B to anonymously authenticate a message m with the trusted verification key v_B. Assuming B uses an anonymous channel (such as Tor), then B anonymously sends the authenticated message m to A. The authentication is achieved with an MZK proof for the following statement:

$$\exists s_C, s_B, s_A, v_B, v_C.\ ver_{sig}(v_B, s_B) = m\ \wedge ver_{sig}(v_C, s_C) = v_B \wedge ver_{sig}(vk_A, s_A) = v_C$$

Let S_{awot} be the corresponding term. ◇

3.2.2.2. Destructors for MZK Proofs

We consider three kinds of destructors for MZK proofs. First, our model allows for retrieving the public part of a MZK proof (via *getPub*), which outputs the statement of a ZK proof. Second, our model allows for checking the validity of an MZK proof (via ver_{zk}), which checks the validity by calling the function $check_{zk}$ (see below) on the CRS, the statement

and the randomness tree. Third, our model allows for transforming MZK proofs (via *setPub*, and_{ZK}, *splitAnd*, or_{ZK}, *commute*, rer_{zk}). The set of *destructors* is defined as $\mathbf{D} := \mathbf{D'} \cup \mathbf{D''}$, where $\mathbf{D'}$ are the destructors from the basic symbolic model, and $\mathbf{D''}$ are the destructors presented in Figure 3.4. Below, we describe these MZK transformations.

Validity of symbolic proofs: $check_{zk}$**.** A statement contains all information that a zero-knowledge proof carries. However, for dealing with equivalence properties, we need to capture the zero-knowledge property. As an example, consider a proof $ZK(A \vee B, r, N)$ of a disjunctive statement $A \vee B$. We have to ensure that $ZK(A \vee B, r, N)$ hides which branch (A or B) is valid. Even though the attacker should be able to retrieve the full statement $A \vee B$ from a proof $ZK(A \vee B, r, N)$, the attacker should not be able to learn whether A or whether B is true. In particular, the verification procedure should fail if the attacker tries to verify A or B outside of the original proof $ZK(A \vee B, r, N)$.

We achieve the zero-knowledge property of a disjunctive proof $ZK(A \vee B, r, N)$ by keeping track (in the auxiliary verification information r) of the randomness of the commitments, loosely speaking by adding a commitment to the commitments. This makes the auxiliary verification information r unique for $A \vee B$ and unguessable for anybody who did not create the proof. Technically, the term r has exactly the same shape as the statement except for the equalities and negated equalities over pairs of commitments and possible unveiling information terms.

This auxiliary verification information r differs in the FMZK model and the CMZK model. In the FMZK model, r only contains the randomness of the commitments, i.e., we only commit to the randomness of the commitments. Hence, $check_{zk}$ only checks whether the randomness in the auxiliary verification information r coincides with the randomness of the commitments. In the CMZK model, r contains the messages and the randomness of the commitments, i.e., we commit to the messages and to the randomness of the commitments. Hence, $check_{zk}$ checks whether the messages and the randomness in the auxiliary verification information r coincides with the messages and randomness of the commitments. Adding the messages to the auxiliary information and matching them with the witness in the proof suffices to prevent functional transformations.

We say that a symbolic proof is *valid* if the following partial function $check_{zk}$ outputs *true*[3]. $check_{zk}$ is recursively defined as depicted below. (The verification procedure additionally checks whether all commitments use the same CRS.) The partial function $extractW$ extracts the witness of a commitment.

$$check_{zk}(crs, S \wedge S', R \wedge R') \Leftrightarrow check_{zk}(crs, S, R) \text{ and } check_{zk}(crs, S', R')$$
$$check_{zk}(crs, S \vee S', R \wedge R') \Leftrightarrow check_{zk}(crs, S, R) \text{ or } check_{zk}(crs, S', R')$$
$$check_{zk}(crs, A = B, R = R') \Leftrightarrow \bot \neq extractW(crs, A, R) = extractW(crs, B, R) \neq \bot$$
$$check_{zk}(A \neq B, R \neq R') \Leftrightarrow \bot \neq extractW(crs, A, R) \neq extractW(crs, B, R') \neq \bot$$

where m is the arity of the constructor or destructor F.

Recall that the auxiliary verification information differs in the FMZK and the CMZK model. Hence, we have to adjust the witness extraction function $extractW$ so as to reflect the different structure of the auxiliary verification information.

[3] *true* and *false* are encoded using the payload constructors $\varepsilon, string_0, string_1$.

$$extractW(crs, F\langle \underline{t} \rangle, F\langle \underline{u} \rangle) = F(extractW(crs, t_1, u_1), \ldots, extractW(crs, t_m, u_m))$$
$$extractW(crs, \langle com(crs, t, M), U \rangle, \langle t, M \rangle) = t$$

In the FMZK model, the partial function *extractW* is defined in the same way, except that the last equation in the FMZK model merely checks the randomness:

$$extractW(crs, \langle com(crs, t, M), U \rangle, M) = t$$

MZK transformations. Malleable zero-knowledge proofs allow a recipient to transform a proof even if this recipient does not know anything about the witnesses of the proof. This flexibility has been used in previous work for re-randomizing proofs, in order to achieve unlinkability to previous proofs [BCCKLS09], and hiding statements from received proofs, allowing a user to selectively disclose information [BMP12]. We discuss below the various transformations.

Conjunctions. We realize the transformations for constructing conjunctions from two proofs, splitting conjunctions, and reordering conjunctions with the destructors and_{ZK} and *splitAnd*. For the destructor *splitAnd*, we need the randomness to be composed using *rand*.

Commuting disjunctions. Reordering disjunctions requires modifying the randomness tree r in a proof $ZK(t, r, N)$, which cannot be accessed. We thus introduce a destructor $commute(ZK(t_1 \vee t_2, r_1 \vee r_2, N))$, which simply swaps the two literals A and B and the corresponding randomness subtrees r_1 and r_2.

Re-randomization. Re-randomization needs to modify the auxiliary verification information r as well. Hence, we introduce a destructor $rerPr(ZK(t, r, N), r', N')$ that takes as input a tree r' and a randomness term N'. The former specifies the commitment that is to be re-randomized, while the latter specifies the randomness to be used.

Hiding public values. Public values occurring in the statement can be hidden (i.e., made private) by means of the destructor $setPub(ZK(t_1, r_1, N), t_2, N')$, which replaces the statement t_1 with the statement t_2 and re-randomizes the proof with N'. For technical reasons, we require that every honest party re-randomizes the proof after hiding public values. However, the attacker can hide values of the witness without re-randomizing the proof; hence we introduce a destructor $setPubA(ZK(t_1, r_1, N), t_2)$.

In the CMZK model, *setPub* is restricted to hiding witnesses. Formally, $setPub(ZK(t_1, r_1, N), t_2, N')$ outputs $\perp$ if t_2 differs from t_1 in more than merely omitting unveil information.

Functional transformations. These transformations are modeled by the destructors *applyF* (for any constructor $f \in \mathbf{D} \cup \mathbf{C}$), which are applicable to commitments such that $applyF(com(crs, t, R), x) = com(crs, f(t, x), R)$. *applyF* can only be applied by the symbolic attacker. Together with $getPub(z)$ and $setPub(z')$ the destructors *applyF* allow the symbolic attacker to modify a proofs witness. In the FMZK model, a proof that is transformed using *applyF* and *setPub* might still pass verification. In the CMZK model, a modified proof will not pass verification(even though commitments can be malleable), since the randomness tree additionally carries information about the original message.

Example 2: Usage of transformations. Recall the statement S_{awot} for Anonymous Webs of Trust from Example 1: an anonymously authenticated message m for the party A. Assume

that m requests A to extend the chain of trust to a party D and to send D a message m': $m = (\texttt{extend}, D, m')$. Moreover, assume that D trusts A. Yet, A wants to hide from anybody intercepting the ZK proof that it was A who extended the chain of trust to D. We can model this scenario by applying the following transformations. First, since D trusts A, A has a signature on its verification key from D: $s_D := sig(sk_D, vk_A, r_A)$. Second, D creates an atomic proof for the validity of s_A, i.e., a MZK proof $ZK(S_A, r_A, N_A)$ of the statement $\exists s_D.ver_{sig}(vk_D, s_D) = vk_A$, denoted as S_A:

$$\hat{ver}_{sig}\big\langle\langle c_{vk_D}, uv(\mathrm{vk}_D, r'_{c_{\mathrm{vk}_D}})\rangle, \langle c_{s_D}, \varepsilon\rangle\big\rangle = \langle c_{\mathrm{vk}_A}, uv(\mathrm{vk}_A, r_{c_{\mathrm{vk}_A}})\rangle$$

Third, A computes the conjunction of S_{awot} and S_A, i.e., A applies the transformation $and_{ZK}(ZK(S_{awot}, r, N), ZK(S_A, r_D, N_D)) =: z$. Fourth, A hides his own verification key vk_A in the statement $S := S_{awot} \wedge S_D$ by removing the unveiling information $uv(vk_A, r'_{c_{vk_A}})$ from S_A. Similarly, A removes the unveiling information $uv(vk_A, r_{c_{vk_A}})$ from S_{awot} (see above). Let S' be the modified statement with the removed unveiling information. Finally A applies $setPub(z, S') =: z'$, to obtain an MZK proof of the following statement:

$$\exists s_B, s_C, s_D, s_A, v_A, v_B, v_C.\ ver_{sig}(v_B, s_B) = m \wedge \\ ver_{sig}(v_C, s_C) = v_B \ \wedge ver_{sig}(v_A, s_A) = v_C \wedge \ ver_{sig}(vk_D, s_D) = v_A \qquad \diamond$$

3.2.2.3. ZK preservation

Before introducing the symbolic attacker for the FMZK and the CMZK variation, we characterize the functional transformations available to the attacker in the FMZK model. We introduce the notion of *ZK preserving functional transformations*. Consider an MZK proof z of the statement $\exists s, b.ver_{sig}(vk, s) = m \vee A$, where b occurs in A. By applying *getPub*, *applyPair*, and *setPubA* to z, an attacker can (in the symbolic model) produce an MZK proof z' of $\exists s, b.\ ver_{sig}(vk, s) = pair(m, m) \vee A$. A successful verification of z' leaks the validity of $\exists s, b.A$, which contradicts the zero-knowledge property. Hence, we only allow ZK preserving functional transformations.

We write $L(t)$ for denoting the logical formula of a statement t. Technically, $L(t) := \exists \underline{x}.L'(t)$, where $\underline{x}$ is the list of variables V in $L'(t)$ with $L'(t)$ being recursively defined as $L'(S_1 \vee S_2) := L'(S_1) \vee L'(S_2)$, analogous for $\vee, =, \neq$, and $\hat{F}$ (for $F \in StF$). The commitments with unveiling information are mapped to the message inside the commitment, and those without unveiling information are mapped to variables that are indexed by the message inside the commitment. A destructor application $f(z)$ is *ZK preserving* if the following logical formula holds for all ZK terms $z = ZK(t, r, N)$ and $L(t) = \exists \underline{x}.L'(t)$:

$$\forall \underline{x} \in \mathbf{T}^n.\big(L'(getPub(z)) \Rightarrow L'(getPub(f(z)))\big)$$

Intuitively, the formula above says that the statement of the original proof implies the statement of the one obtained by the transformation. This, in particular, rules out the aforementioned problem related to proofs of logical disjunctions.

A functional transformation *applyF* is *ZK preserving* if f is ZK preserving.

ZK preservation is decidable. Ramsey showed in 1929 that EPR formulas are decidable, i.e., all first-order logic formulas of the following form $\exists^* \underline{y} \forall^* \underline{x}.\phi(x, y)$ are decidable, where $\phi(x, y)$ is a propositional logic formula about relations [Ram29]. If we consider each atomic statement, i.e, each equation $C(\underline{x}) = C'(\underline{x})$ or $C(\underline{x}) \neq C'(\underline{x})$ as a relation $\phi'(\underline{x})$, checking ZK preservation amounts to checking an EPR formula. Hence, checking ZK

preservation is decidable. A verification tool needs to check that every applied functional transformation *applyF* is ZK preserving.

3.2.2.4. Symbolic attacker

The symbolic attacker is defined as for the basic symbolic model (see Figure 3.2), except that $\mathbf{C}'$ is replaced with $\mathbf{C}$ and $\mathbf{D}'$ with $\mathbf{D}$, with the additional condition (for the FMZK variation) that all destructor applications of the symbolic attacker have to be ZK preserving.

3.3. Computational soundness

In this section, we show the computational soundness of our symbolic model using the CoSP framework. While we carefully designed the symbolic abstraction such that it captures vulnerabilities that affect both trace and equivalence properties, our computational soundness result is cast for trace properties, which constitutes the current state-of-the-art in CoSP. We establish our computational soundness in three steps. We first identify necessary restrictions on the class of protocols for which we can show computational soundness , e.g., different encryptions always use different randomness(Section 3.4). We then introduce necessary implementation conditions for computationally sound realizations (Section 3.5). Finally, we conduct the actual proof of computational soundness (Section 3.6.1). Due to space constraints, we primarily elaborate on the protocol restrictions and the implementation conditions, as these illustrate the major insights how to achieve computational soundness for MZK proofs.

3.4. MZK-safe protocols

We first characterize the class of computationally sound protocols, which we call MZK-safe. Mainly, our protocol conditions exclude adaptive corruption and regulate the usage of randomness. In this section, we concentrate on the most insightful conditions. To prevent standard problems of adaptive corruption, we disallow decryption keys to be sent over the network. Similarly, we require that the plaintext message of a commitment is either publicly known or secret for the entire execution by imposing the condition that unveiling information $uv(m, r)$ of a commitment $com(crs(n), m, r)$ is only sent over the network as part of an MZK proof. In other words, we basically restrict the usage of commitments to MZK proofs. Moreover, for MZK proofs generated by honest protocol participants, we require that all commitments of one ZK proof use the same honestly generated CRS. Finally, we limit the way in which randomness can be reused. Recall that we distinguish nonces and randomness terms, which are used as randomness in the computation of cryptographic terms, such as encryptions, signatures, and MZK proofs. We only allow reusing randomness terms as witnesses of MZK proofs. For technical reasons, we exclude protocols that reuse the randomness of signatures as witnesses in MZK proofs. However, this restriction is not severe in practice, since we are not aware of any protocol that uses the signature randomness in an ZK proof.

Disallowing the re-usage of randomness completely, however, would result in excluding statements about ownership of ciphertexts, i.e., statements of the form $\exists r, m.c = enc(ek, m, r)$. There are IND-CCA secure encryption schemes that allow an attacker to

retrieve the plaintext once he knows the randomness of the ciphertext. Since we consider malleable zero-knowledge proofs, we cannot exclude that an attacker uses the randomness from some witness and a ciphertext to compute a proofs with the plaintext as a witness. Instead of granting the symbolic attacker this possibility, we require that every proof that carries a randomness terms N of some (protocol) ciphertext $enc(ek, m, N)$ also carries the corresponding plaintext m.[4] In summary, a randomness term r cannot be used for different terms unless the randomness belongs to a protocol ciphertext $enc(ek, m, r)$. In this case, r may additionally occur as a witness of a zero-knowledge proof if the same proof uses the corresponding plaintext m as a witness.

This list of protocol conditions is quoted from [BBU13] except for the parts that are marked in blue.

1. The annotation of each *crs*-node, each key-pair (ek, dk) and (vk, sk) is a fresh nonce, which does not occur anywhere else.
2. There is no node annotated with a $garb$, $garbEnc$, $garbSig$, or $N \in \mathbf{N}_E$ constructor in the protocol.
3. No commitment is sent to the attacker that is not inside ZK term, i.e., not the result of an term constructed by $mkZK$.
4. No com term that has been received over the network without being part of a ZK term and no $garbCom$ term is used in a constructor or destructor application.
5. For every commitment $com(c, m, N)$ that is published in a ZK proof it holds that if $com(c, m, N)$ is a witness commitment, $uv(m, N)$ is never revealed.
6. The constructor ZK and the destructors $setPubA$ and $applyF$ (for $F \in \mathbf{C} \cup \mathbf{D}$) is not used in the protocol.

   The last argument of a com, enc, sig constructor and of a $mkZK, rerPr$ destructor are fresh nonces. These nonces are not used anywhere else except in case of enc as part of a subterm of the second argument in a com-node if the following holds: Construct a symbolic knowledge S out of the path from the com-node to the root, including the com-node. Let N be the nonce of an enc constructor and m be the corresponding (symbolic) plaintext. Then, we require that if $S \vdash_w N$ holds true then also $S \vdash_w m$ holds true.
8. A dk-node is only used as first argument for dec-node or as subterm of the third argument in a ZK-node.
9. A sk-node is only used as first argument for sig-node or as subterm of the third argument in a ZK-node.
10. The first argument of a dec-computation node is a dk-node.
11. The first argument of a sig-computation node is a sk-node.
12. A $mkZK$-computation node is consistent in the following sense: for every com-computation node in the statement of $mkZK$ the destructor $crsof$ would not output $\perp$.
13. The first argument of a com-computation is a crs-computation node which is annotated by a nonce $N \in \mathbf{N}_P$. This nonce is only used as annotation of this crs node

[4] The alternative would be to introduce an attacker-specific destructor $decR(com(crs, N, r), com(crs, enc(ek, m, N), r')) = com(crs, m, rand(r, r'))$.

and nowhere else.

14. The first argument of a ver_{zk}-computation is a crs-computation node which is annotated by a nonce $N \in \mathbf{N}_P$. This nonce is only used as annotation of this crs node and nowhere else.
15. For the relation $R^{\text{sym}}_{\text{adv}}$ it holds: There is an efficient algorithm **SymbExtr**, that given a term M together with a set S of terms (which was generated according by any protocol satisfying the protocol conditions above), outputs a term N, such that there are $t, t' \in \mathbf{T}$ such that $S \vdash ZK(t, r, t')$ and $(N, M) \in R^{\text{sym}}_{\text{adv}}$ or outputs $\perp$ if there is no such term N. We call a relation satisfying this property symbolically extractable.
16. The relation $R^{\text{sym}}_{\text{adv}}$ is efficiently decidable.
17. All ZK-transformations are only applied to honestly generated proofs or to terms t for which $ver_{zk}(crs(t'), t) \neq false$ for some $t' \in \mathbf{N}_P$.
18. The protocol only uses polynomially many different nonces.
19. Every hide operation is followed by a re-randomize operation. Formally, a $setPub(ZK(t, r), t')$ application hides a witness w if t and t' both contain a commitment $com(crs, m, r')$ if $S \cup m \vdash w$ and $S \nvdash w$, however t carries the corresponding unveil information $uv(m, r')$ and t' does not. We require that every transformation $setPub(ZK(t, r), t') = ZK(t', r)$ that hides a witness w in $com(crs, m, r')$ is followed by a re-randomization $rer_{com}(com(crs, m, r'), r'')$ in $ZK(t', r)$ for fresh randomness r''.
20. The last node of a ZK-transformations is a fresh nonce. This nonce is not used anywhere else.

We will call a protocol satisfying these constraints a MZK-safe protocol. The class of MZK-safe protocols is the set of all protocols which are MZK-safe.

3.5. Implementation conditions

For the computational soundness result, we define necessary implementation conditions. The conditions can be partitioned into cryptographic requirements to cryptographic primitives and sanity conditions that ensure that the implementation behaves similar to the symbolic model.

Sanity conditions. We require that for each constructor and destructor $f \in \mathbf{C} \cup \mathbf{D}$, there is a polynomial-time computable, deterministic algorithm A_f and the algorithm A_N for drawing nonces is randomized. Moreover, for all algorithms the length of the output solely depends on the length of the input. Moreover, we require that all symbolic cancellation rules hold computationally as well, e.g., $fst(pair(x, y)) = x$ for all $x, y \in \{0, 1\}^*$.

Finally, we assume that all messages have an efficiently recognizable type. Given a message, we require that it is efficiently possible to recognize whether the message is a pair, a signature, a ciphertext, a commitment, a zero-knowledge proof, or a key, in particular which type of key. More specifically, we require that $A_{vkof}(m) \neq \perp$, $A_{ekof}(m) \neq \perp$, and $A_{crsof}(m) \neq \perp$ for a message m of type signature, ciphertext, and commitment, respectively.

Encryptions and signatures are secure. We require that the encryption algorithms A_{ek}, A_{dk}, A_{enc}, A_{dec} constitute an IND-CCA secure encryption scheme. Moreover, we require that whenever $A_{dec}(dk_N, c)$ succeeds, and that $A_{ekof}(c) = ek_N$ outputs the corre-

sponding public key.For the signature algorithms A_{sk}, A_{vk}, A_{sig}, $A_{ver_{sig}}$ we require that they constitute a CMA-existentially unforgeable signature scheme. We require that A_{sig} produces different signatures for different randomnesses.

Non-interactive zero-knowledge arguments of knowledge. We require the following properties from zero-knowledge proofs: (i) completeness (honest provers and honest verifiers succeed for true statements); (ii) zero-knowledge (given a simulation trapdoor, all valid proofs can be efficiently simulated without using the witness); (iii) extractability (given an extraction trapdoor, the witness is efficiently extractable from valid proofs); (iv) unpredictability (fresh proofs cannot be guessed even if the witness is known); (v) length-regularity (the length of the output only depends on the length of the input); (vi) deterministic verification and extraction. The conditions (i) to (iii) are the minimal requirements on proofs of knowledge. Conditions (iv) to (vi) are properties that we need for our computational soundness proof and that are easy to fulfill. For the FMZK realization, we require that $A_{ZK}, A_{ver_{zk}}, A_{crs}, A_{com}$ constitute a non-interactive argument of knowledge (NIZKAoK), and the same commitment algorithms $A_{crs}, A_{com}, A_{open}$ belong to an extractable non-interactive computationally hiding and binding commitment scheme. We also assume an algorithm A_{getPub} such that $A_{getPub}(A_{ZK}(t,r)) = r$ and we assume that the protocol transformations $setPub$, and_{ZK}, $splitAnd$, or_{ZK}, $commute$, rer_{zk}, rer_{com} have an implementation. These conditions are compatible with the controlled-malleability variant [CKLM12] of the widely deployed Groth-Sahai proofs [GS08]. The following definition is to a large extend quoted from [BBU13]. All parts that are modified are marked in blue.

Definition 33 (NIZK arguments of knowledge [BBU13])**.** *A non-interactive zero-knowledge argument of knowledge for relations $R_{\text{hon}}^{\text{comp}}, R_{\text{adv}}^{\text{comp}}$ is a tuple of polynomial-time algorithms $(\boldsymbol{K}, \boldsymbol{P}, \boldsymbol{V})$ such that there exist polynomial-time algorithms $(\boldsymbol{E}, \boldsymbol{S})$ and the following properties hold:*

• Completeness: *Let a polynomial-time adversary $\mathcal{A}$ be given. Let $(crs, simtd, extd) \leftarrow \boldsymbol{K}(1^\eta)$. Let $(x,w) \leftarrow \mathcal{A}(1^\eta, crs)$. Let $proof \leftarrow \boldsymbol{P}(x,w,crs)$. Then with overwhelming probability in η, it holds $(x,w) \notin R_{\text{adv}}^{\text{comp}}$ or $\boldsymbol{V}(x, proof, crs) = 1$.*

• Zero-Knowledge: *Fix a polynomial-time oracle adversary $\mathcal{A}$. For given $crs, simtd$, let $\mathcal{O}_{\boldsymbol{P},crs}(x,w) := \boldsymbol{P}(x,w,crs)$ if $(x,w) \in R_{\text{hon}}^{\text{comp}}$ and $\mathcal{O}_{\boldsymbol{P},crs}(x,w) := \perp$ otherwise, and let $\mathcal{O}_{\boldsymbol{S},crs,simtd}(x,w) := \boldsymbol{S}(x, crs, simtd)$ if $(x,w) \in R_{\text{hon}}^{\text{comp}}$ and $\mathcal{O}_{\boldsymbol{S},crs,simtd}(x,w) := \perp$ otherwise. Then*

$$\big|\Pr[\mathcal{A}^{\mathcal{O}_{\boldsymbol{P},crs}}(1^\eta, crs) = 1 : (crs, simtd, extd) \leftarrow \boldsymbol{K}(1^\eta)] - \Pr[\mathcal{A}^{\mathcal{O}_{\boldsymbol{S},crs,simtd}}(1^\eta, crs) = 1 : (crs, simtd, extd) \leftarrow \boldsymbol{K}(1^\eta)]\big|$$

is negligible in η.

• Extractability: *Let a polynomial-time oracle adversary $\mathcal{A}$ be given. Let $(crs, simtd, extd) \leftarrow \boldsymbol{K}(1^\eta)$. Let $(x, proof) \leftarrow \mathcal{A}(1^\eta, crs)$. Let $w \leftarrow \boldsymbol{E}(x, proof, extd)$. Then with overwhelming probability, if $\boldsymbol{V}(x, proof, crs) = 1$ then $R_{\text{adv}}^{\text{comp}}(x,w) = 1$.*

• Unpredictability: *Let a polynomial-time adversary $\mathcal{A}$ be given. Let $(crs, simtd, extd) \leftarrow \boldsymbol{K}(1^\eta)$. Let $(x, w, proof') \leftarrow \mathcal{A}(1^\eta, crs, simtd, extd)$. Then with overwhelming probability, it holds $proof' \neq \boldsymbol{P}(x,w,crs)$.*

• Length-regularity: *Let two witnesses w and w', and statements x and x' be given such that $|x| = |x'|$, and $|w| = |w'|$. Let $(crs, simtd, extd) \leftarrow \boldsymbol{K}(1^\eta)$. Then let $proof \leftarrow \boldsymbol{P}(x,w,crs)$ and $proof' \leftarrow \boldsymbol{P}(x',w',crs)$. Then we get $|proof| = |proof'|$ with probability 1.*

• Deterministic verification and extraction: *The algorithms V, E are deterministic.*[5]

3.5.1. Computational ZK Relation

For the computational soundness proof and for the computational implementation, we need four destructors $extrSta, extrWit, crsof, extrNon$, which traverse through the statement and extract the statement, the witness, the CRS, and the list of nonces, respectively (see Appendix 3.10). The computational ZK relation $R_{\text{hon}}^{\text{comp}}$ is then defined as follows:

$$\{(img_\eta(x), img_\eta(w)) \mid \begin{array}{l} t \text{ is a valid symbolic statement} \wedge \\ x = extrSta(t), w = extrWit(t) \\ \text{for a consistent } \eta\} \end{array}$$

We assume efficient encoding algorithms Let $e_\wedge, e_\vee, e_=, e_f$ ($f \in StF$, see Section 3.2.2.1). The function img_η depends on an environment η, a partial function $\mathbf{T} \to \{0,1\}^*$ that assigns bitstrings to nonces and adversary-generated terms. We use the definition of a *consistent environment* that lists various natural properties an environment will have (such as mapping ZK-terms to bitstrings of the right type).

We define img_η as follows:

$$\begin{array}{l} img_\eta(S_1 \wedge S_2) := e_\wedge(img_\eta(S_1), img_\eta(S_2)) \\ img_\eta(S_1 \vee S_2) := e_\vee(img_\eta(S_1), img_\eta(S_2)) \\ img_\eta(A = B) := e_=(img_\eta(A), img_\eta(B)) \\ img_\eta(f(x_1, \ldots, x_n)) := e_f(img_\eta(x_1), \ldots, img_\eta(x_n)) \\ \qquad \text{for } f \in StF \\ img_\eta(f(x_1, \ldots, x_n)) := A_f(\eta(x_1), \ldots, \eta(x_n)) \\ \qquad \text{for } f \in \mathbf{C} \cup \mathbf{D} \setminus StF \\ img_\eta(pair(com(crs(N), m, r), uv(m, r))) \\ := A_{pair}(A_{com}(crs(N), img_\eta(m), \eta(r)), A_{uv}(img_\eta(m), \eta(r))) \\ img_\eta(pair(com(crs(N), m, r), \varepsilon)) \\ := A_{pair}(A_{com}(crs(N), img_\eta(m), \eta(r)), A_\varepsilon()) \\ img_\eta(N) := \eta(N) \text{ for } N \in \mathbf{N} \end{array}$$

where X_m denotes the computational encoding of a variable with index $m \in \mathbf{T}$.

Definition 34 (Consistent environments [BBU13])**.** *Let $\mathcal{E}$ be the set of all partial functions $\eta : \boldsymbol{T} \to \{0,1\}^*$. We will call such an η an environment.*

Let an implementation A for the symbolic model by given. Define the partial function $img_\eta : \boldsymbol{T} \to \{0,1\}^*$ for $\eta \in \mathcal{E}$ by taking the first matching rule:

- *For a nonce N define $img_\eta(N) := \eta(N)$*
- *For a term $t = crs(N)$ define $img_\eta(crs(N)) := \eta(t)$*
- *For a term $t = ZK(x, w, M)$ define $img_\eta(t) := \eta(t)$*

[5] This is not a necessary requirement but we require them to be deterministic for the sake of simplifying the computational soundness proof.

- *Let C be a constructor from $\{ek, dk, vk, sk, enc, sig, crs, garbSig, garbEnc, garb\}$. For $t = C(t_1, \ldots, t_{n-1}, N)$ with $N \in \boldsymbol{N}_E$ define $img_\eta(t) := \eta(t)$.*
- *For a term $C(t_1, \ldots, t_n)$ define $img_\eta(C(t_1, \ldots, t_n)) := A_C(img_\eta(t_1), \ldots, img_\eta(t_n))$, if for all i we have $img_\eta(t_i) \neq \bot$, and $\bot$ otherwise.*

An environment η is consistent if the following conditions are satisfied: [6]

- *η is injective.*
  For each constructor C we require that the bitstring $img_\eta(C(t_1, \ldots, t_n))$ has the type as follows: The constructors $enc, garbEnc$ are mapped to type ciphertext, $sig, garbSig$ to signatures, ZK to ZK proofs, $com, garbCom$ to commitments, ek, dk, vk, sk to encryption, decryption, verification, signing key, respectively. crs to common reference string, pair to pair, $string_0, string_1, \varepsilon$ to payload-string, $\boldsymbol{N}$ to nonce, $garb$ has none of these types.

- $A_{ekof}(img_\eta(enc(ek(N), t, M))) = img_\eta(ek(N))$ *for all* $N, M \in \boldsymbol{N}_P$, $t \in \boldsymbol{T}$.
- *For all* $t = sig(sk(N), u, M)$ *with* $N, M \in \boldsymbol{N}, u \in \boldsymbol{T}$ *it holds:* $ver_{sig}(vkof(t), t) \neq \bot$ *implies that* $A_{ver_{sig}}(img_\eta(vkof(t)), img_\eta(t)) = img_\eta(u)$.
- *For* $t = ZK(x, w, M)$ *with* $M \in \boldsymbol{N}$ *holds:*
  1. $A_{ver_{zk}}(A_{crsof}(\eta(t)), \eta(t)) = 1$
  2. $A_{getPub}(\eta(t)) = img_\eta(x)$
  3. $A_{crsof}(\eta(t)) = A_{crsof}(\eta(c))$ *for all commitment terms c in x.*
- *For all* $t_1, t_2 \in \boldsymbol{T}$ *it holds that* $A_{ver_{sig}}(img_\eta(garbSig(t_1, t_2))) = \bot$
- *For all* $N, M \in \boldsymbol{N}$, $t \in \boldsymbol{T}$ *it holds that* $A_{dec}(img_\eta(dk(N)), img_\eta(enc(ek(N), t, M))) = img_\eta(t)$ *and* $img_\eta(t) \neq \bot$.
- *For all* $enc(ek(N), t, M) \in \boldsymbol{T}$ *it holds: If* $img_\eta(enc(ek(N), t, M)) =: c \neq \bot$, *then it follows* $A_{ekof}(c) = img_\eta(ek(N))$.
- *For all* $enc(ek(N), t, M) \in \boldsymbol{T}$ *it holds: If* $img_\eta(enc(ek(N), t, M)) \neq \bot$ *and* $d \in \{0,1\}^*$ *such that* $img_\eta(ek(N)) = p(d)$ [7], *then it follows that* $A_{dec}(d, img_\eta(enc(ek(N), t, M))) = img_\eta(t)$.
- *For all* $N, M \in \boldsymbol{N}$, $t \in \boldsymbol{T}$ *it holds that*

$$A_{open}(img_\eta(crs(N)), img_\eta(com(crs(N), t, M)), img_\eta(uv(t, M))) = img_\eta(t)$$

  and $img_\eta(t) \neq \bot$.
- *For all* $com(crs(N), t, M) \in \boldsymbol{T}$ *it holds: If* $img_\eta(com(crs(N), t, M)) =: u \neq \bot$, *then it follows* $A_{crsof}(u) = img_\eta(crs(N))$.

Given these notions, we formalize the conditions on $R^{\text{comp}}_{\text{hon}}, R^{\text{comp}}_{\text{adv}}$ in the following definition.

Definition 35 (Implementation of relations [BBU13])**.** *A pair of relations $R^{\text{comp}}_{\text{hon}}, R^{\text{comp}}_{\text{adv}}$ on $\{0,1\}^*$ implement a relation $R^{\text{sym}}_{\text{adv}}$ on $\boldsymbol{T}$ with usage restriction $R^{\text{sym}}_{\text{hon}}$ if the following conditions hold for any consistent $\eta \in \mathcal{E}$:*

(i) $(x, w) \in R^{\text{sym}}_{\text{hon}}$ *and* $img_\eta(x) \neq \bot \neq img_\eta(w) \implies (img_\eta(x), img_\eta(w)) \in R^{\text{comp}}_{\text{hon}}$

(ii) $(img_\eta(x), img_\eta(w)) \in R^{\text{comp}}_{\text{adv}} \implies (x, w) \in R^{\text{sym}}_{\text{adv}}$

[6] We consider a condition in which a term t occurs such that $img_\eta(t) = \bot$ as satisfied.
[7] Where p is the function defined in implementation condition 28.

(iii) $R^{\text{sym}}_{\text{hon}} \subseteq R^{\text{sym}}_{\text{adv}}$ *and* $R^{\text{comp}}_{\text{hon}} \subseteq R^{\text{comp}}_{\text{adv}}$

Computational statements. The computational ZK relation is the computational translation of the symbolic ZK relation. We implement a computational translation img_η that is parametric in the randomness assignment η. Basically, img_η is recursively defined over a symbolic statement. img_η uses an encoding of $\wedge, \vee, =$, and $\hat{f}$ for $f \in StF$, and applies for every $f \in \mathbf{C}$ the implementation A_f and assigns to every nonce N the bitstring $\eta(N)$.

For the computational soundness proof and for the computational implementation, we need four destructors *extrSta*, *extrWit*, *crsof*, *extrNon*, which traverse through the statement and extract the statement, the witness, the CRS, and the list of nonces, respectively. The computational ZK relation $R^{\text{comp}}_{\text{hon}}$ for the protocol is then defined as follows:

$$\left\{((t, img_\eta(x)), img_\eta(w)) \mid t \text{ symb. statement } \wedge x = extrSta(t), w = extrWit(t))\right\}$$

Additional conditions for the CMZK model. Soundly realizing the more restricted form of malleability mandated by the CMZK model requires additional strengthening of the ZK proofs. We use a recent construction for controlled malleability [CKLM12], which also applies to the Groth-Sahai proof scheme [GS08].

Efficient statements. In contrast to previous work on computationally symbolic ZK proofs [BBU13; BU10], the FMZK and the CMZK model have computationally sound realizations that allow efficient statements for encryption schemes and signature [CHKLN11; CK11; GS08].

MZK transformations. For MZK transformations, we basically require that a transformed proof looks like a freshly generated proof. We even require that a transformed proof is indistinguishable from a simulated proof. This property, called *strong derivation privacy*, was introduced in [CKLM12].

Moreover, we require that rer_{zk} and rer_{com} have indeed re-randomizing implementations. Given z the proof $z' \leftarrow A_{rer_{zk}}(z, r)$ is unpredictable, for a randomly chosen nonce r. Similarly, given c the commitment $c' \leftarrow A_{rer_{com}}(c, r)$ is unpredictable, for a randomly chosen nonce r.

3.5.2. List of implementation conditions

This list of implementation conditions is largely quoted from [BBU13]. All modifications and additions are marked in blue.

1. The implementation is an implementation according to Definition 6.

   There are disjoint and efficiently recognizable sets of bitstrings representing the node types nonce, ciphertext, encryption key, decryption key, signature, verification key, signing key, common reference string, zero-knowledge proof, commitment, pair, and payload-string.

   The images of A_N have type nonce (for all $N \in \mathbf{N}$), A_{enc} have type ciphertext, A_{ek} have type encryption key, A_{dk} have type decryption key, A_{sig} have type signature, A_{vk} have type verification key, A_{sk} have type signing key, A_{crs} have type common reference string, A_{mkZK} have type zero-knowledge proof, A_{com} have type commitment, A_{pair} have type pair, and $A_{string_0}, A_{string_1}, A_\varepsilon$ have type payload string.

3. The implementation A_N for nonces $N \in \mathbf{N}_P$ implement uniform distributions on $\{0,1\}^k$ where k is the security parameter.
4. If $A_{dec}(dk_N, m) \neq \perp$ then $A_{ekof}(m) = ek_N$, i.e. the decryption only succeeds if the corresponding encryption key can be extracted out of the ciphertext.
5. $A_{vkof}(A_{sig}(A_{sk}(x), y, z)) = A_{vk}(x)$ for all $y \in \{0,1\}^*$ and x, z nonces. If e is of type signature then $A_{vkof}(e) \neq \perp$, otherwise $A_{vkof}(e) = \perp$.
6. For all $m, k \in \{0,1\}^*$, k having type encryption key, and $r \neq r' \in \{0,1\}^*$ with $|r| = |r'|$ holds that $A_{enc}(k, m, r)$ and $A_{enc}(k, m, r')$ are equal with negligible probability.
7. For all $m, k \in \{0,1\}^*$, k having type signing key, and $r \neq r' \in \{0,1\}^*$ with $|r| = |r'|$ holds that $A_{sig}(k, m, r)$ and $A_{sig}(k, m, r')$ are equal with negligible probability.
8. The implementations A_{ek}, A_{dk}, A_{enc}, and A_{dec} belong to an encryption scheme $(KeyGen_{enc}, ENC, DEC)$ which is IND-CCA secure.
9. The implementations A_{vk},A_{sk},A_{sig}, and $A_{ver_{sig}}$ belong to a signature scheme $(KeyGen_{sig}, SIG, VER_{sig})$ which is strongly existential unforgeable.
10. All implementations are length regular, i.e. if the input has the same length the output will have the same too.
11. For $m_1, m_2 \in \{0,1\}^*$ holds $A_{fst}(A_{pair}(m_1, m_2)) = m_1$ and $A_{snd}(A_{pair}(m_1, m_2)) = m_2$
12. $A_{dec}(A_{dk}(r), A_{enc}(A_{ek}(r), m, r')) = m$ for all r, r' nonces.
13. Let $k \in \{0,1\}^*$ be an encryption key and $m, n \in \{0,1\}^*$ such that n is of type nonce. Then holds $A_{ekof}(A_{enc}(k, m, n)) = k$. If $c \in \{0,1\}^*$ is not of type ciphertext then $A_{ekof}(c) = \perp$.
14. Let $vk, sk \in \{0,1\}^*$ be a keypair, i.e. (vk, sk) is in the image of $KeyGen_{sig}$, then holds for all $m, n \in \{0,1\}^*$: $A_{vkof}(A_{sig}(sk, m, n)) = vk$.
15. $A_{ver_{sig}}(A_{vk}(r), A_{sig}(A_{sk}(r), m, r')) = m$ for all r, r' nonces.
16. For all $p, s \in \{0,1\}^*$ we have that $A_{ver_{sig}}(p, s) \neq \perp$ implies $A_{vkof}(s) = p$.
17. For $m \in \{0,1\}^*$ holds $A_{unstring_i}(A_{string_i}(m)) = m$ for $i \in \{0,1\}$ and $A_{string_0}(m) \neq A_{string_1}(m)$.
18. For all $m \in \{0,1\}^*$ of type zero-knowledge proof holds that $iszk(m) = m$ and if m has not type zero-knowledge proof, then $iszk(m) = \perp$. The same holds for $issig$ w.r.t. the type signature and $isenc$ w.r.t. the type ciphertext.
19. If $k \in \{0,1\}^*$ is not of the type encryption key then holds for all $m, n \in \{0,1\}^*$ that $A_{enc}(k, m, n) = \perp$. The same has to hold for the type signing key and the implementation of signatures.
20. There are algorithms $\mathbf{K}, \mathbf{P}, \mathbf{V}$ such that
    - $(crs, simtd, extd) = \mathbf{K}(1^\eta; r)$, where $A_{crs}(r) = crs$,
    - $A_{zk}(applyCom(x, w; r_1); r_2) = \mathbf{P}(x, w; r_1, r_2)$, and
    - $A_{ver_{zk}}(z) = \mathbf{V}(z)$,

    where $applyCom(x, w; r_1)$ traverses the statement tree and replaces each atomic leaf (e.g., the public messages or the placeholders for the witnesses) with pairs of commitments and their unveil information (for public messages), or an empty string (for witnesses). For producing the commitments $applyCom$ applies $A_{com}(crs, m; r_1)$, where m is either the public message, from the statement x, or from the witness w.

The tuple $(\mathbf{K}, \mathbf{P}, \mathbf{V})$ constitutes a non-interactive zero-knowledge argument of knowledge $(\mathbf{K}, \mathbf{P}, \mathbf{V})$, as defined in Definition 33, for the computational ZK relation $R_{\text{hon}}^{\text{comp}}$

$$\{(img_\eta(x), img_\eta(w)) \mid t \text{ is a valid symbolic statement} \wedge x = extrSta(t) \wedge w = extrWit(t) \wedge \text{ for a consistent } \eta\}$$

21. A_{crs}, A_{com}, A_{open}, A_{uv} belong to an extractable non-interactive computationally binding and hiding commitment scheme that is length regular.
22. For all $z \in \{0,1\}^*$ holds $A_{ver_{zk}}(crsof(z), z) \in \{0,1\}$, where $A_{ver_{zk}}(crsof(z), z) = true$ if and only if z is correct w.r.t. to the verifier of the proof system.
23. If $z \in \{0,1\}^*$ is not of type zero-knowledge, then $ver_{zk}(crsof(z), z) = false$.
24. For all $z \in \{0,1\}^*$ holds: If z is not of type zero-knowledge proof then $A_{crsof}(z) = \perp$.
25. If $z := A_{mkZK}(\bar{m}) \neq \perp$ then $A_{ver_{zk}}(A_{crsof}(z), z) = 1$.
26. If $(x, w) \notin R_{\text{hon}}^{\text{comp}}$ then for all $r \in \{0,1\}^*$, it holds $A_{mkZK}(x, w, r) = \perp$.
27. Let $x, w, n \in \{0,1\}^*$ such that $z = A_{mkZK}(x, w, n)$. If $z \neq \perp$ then holds that $x = A_{getPub}(z)$.
28. For $d \in \{0,1\}^*$ of type decryption key there is a efficiently computable function $p : \{0,1\}^* \to \{0,1\}^*$ such that for all $m, n \in \{0,1\}^*$, n of type nonce, it holds $A_{dec}(d, A_{enc}(p(d), m, n)) = m$, i.e. p computes the encryption key corresponding to d. The analogous statement has to holds for signing keys and verification keys.
29. Every transformation is strongly derivation private as defined my Chase, Kohlweiss, Lysyanskaya, Meiklejohn [CKLM12].

3.6. Complete proof of computational soundness

The computational soundness proof of our symbolic model first presents a simulator *Sim* (Section 3.6.4) and then shows that this simulator satisfies Dolev-Yaoness (Section 3.6.5) and indistinguishability (Section 3.6.5), as defined in Section 2.3. The computational soundness proof and the definition of a symbolic and computational ZK relation are based on the work of Backes, Bendun, and Unruh [BBU13]. To a large extend, the constructions, definitions and proofs from [BBU13] apply word for word. Section 3.6.4, 3.6.5, 3.6.6 are quoted from [BBU13], and all modifications and additions are marked in blue.

The overall goal of this section is proving the following theorem.

Theorem 3 (Computational soundness of MZK-safe protocols)**.** *Any computational implementation of the MZK-safe model* $\mathbf{M}$ *(see Section 3.4) that satisfies the implementation conditions for MZK-safe protocols (see Section 3.5) is computationally sound for the class of MZK-safe protocols (see Section 3.4).*

3.6.1. Proof overview

Due to the length of the full proof, first a brief overview of the computational soundness proof follows.

Recall that proving computational soundness in CoSP can be done by constructing a simulator that behaves as the computational execution towards a computational attacker and as a valid symbolic attacker towards the symbolic execution (see Theorem 1). Upon receiving a term from the symbolic execution, the simulator translates this term t as a bitstring $\beta(t)$ according to a function $\beta : \mathbf{T} \to \{0,1\}^*$ and forwards this bitstring $\beta(t)$ to the attacker. Upon receiving a bitstring m from the attacker, the simulator parses m according to a function $\tau : \{0,1\}^* \to \mathbf{T}$ and forwards $\tau(m)$ to the symbolic execution. Computational soundness follows if the simulator satisfies two properties: *Indistinguishability*, i.e., the interaction of the attacker with the simulator is indistinguishable for from attacker's interaction with the real computational execution of the protocol. And *Dolev-Yaoness*, i.e., for each message m from the attacker, and for the set S of terms received so far by the symbolic execution, $S \vdash \tau(m)$ holds with overwhelming probability.

The core of the simulator are the constructing function β and the parsing function τ. The constructing function β maintains a memory. Whenever a bitstring $\beta(t)$ for a term t is computed for the first time, $\beta(t)$ is stored; in all future calls $\beta(t)$ the stored bitstring is used. We recursively define β over terms, e.g., $\beta(enc(t_1, t_2, t_3)) := A_{enc}(\beta(t_1), \beta(t_2), \beta(t_3))$.

Dolev-Yaoness and Indistinguishability of *Sim*. The Dolev-Yaoness of *Sim* is proven by constructing a faking simulator Sim_f that fakes all honest encryptions, i.e., Sim_f instead computes encryptions of the constant-zero bitstring, and simulates all proofs and handles all transformations are simulated proofs. In this way no plaintext is used while constructing encryptions, and no witness is used while constructing zero-knowledge proofs. Additionally, we go one step further and do not even use the protocol randomness in the computation of β: Sim_f uses an encryption faking oracle for constructing ciphertexts and a simulation oracle for constructing zero-knowledge proofs. We then show that *Sim* and Sim_f are indistinguishable. This follows from the cryptographic properties of the encryptions scheme, the commitment scheme, and the ZK proof, and it can be shown using standard techniques. Since Sim_f satisfies *Dolev-Yaoness* by construction, this entails that *Sim* does as well.

The indistinguishability of *Sim* is proven in several steps. First, we show that parsing and constructing statements and witnesses preserves the validity of the statement-witness pairs. Then, we show that the probability that *Sim* aborts, i.e., that an extraction failure happens, is negligible. After these two properties have been shown, we can analyze the computational and the hybridexecution with *Sim* for fixed randomness of the attacker and the execution. We obtain that both executions are indistinguishable since $\beta(f(t_1, \ldots, t_n)) = A_f(\beta(t_1), \ldots, \beta(t_n))$.

3.6.2. Symbolic and computational ZK relation

Symbolically, we have the normal relation $R^{\text{sym}}_{\text{hon}}$ for protocols

$$\big\{(x,w) \mid \; L'(x)\,\{w_i/x_{w_i}\} \text{ is true, where } x = extrSta(t), w = extrWit(t))\big\}$$

$$
\begin{aligned}
L''(S_1 \vee S_2) &:= L''(S_1) \vee L''(S_2) \\
L''(S_1 \wedge S_2) &:= L''(S_1) \wedge L''(S_2) \\
L''(A = B) &:= L''(A) = L''(B) \\
L''(A \neq B) &:= L''(A) \neq L''(B) \\
L''(enc\langle X_1, X_2, X_n\rangle) &:= \exists r.F(L''(X_1), X_2, r) \\
L''(F\langle X_1, \ldots, X_n\rangle) &:= F(L''(X_1), \ldots, L''(X_n)) \\
n &= arity(F) \\
L''(\langle com(crs, m, r), \varepsilon\rangle) &:= x_m \text{ , for } x_m \in V \\
L''(\langle com(crs, m, r), uv(m, r)\rangle) &:= m
\end{aligned}
$$

Since parsing the correct randomness from attacker-generated encryptions is in general not possible, we have to allow the attacker to use any randomness. Hence, we relax the symbolic ZK relation for attacker messages in the ZK relation $R^{\text{sym}}_{\text{adv}}$. We define the following relaxed logical formulas:

$$
\begin{aligned}
L''(S_1 \vee S_2) &:= L''(S_1) \vee L''(S_2) \\
L''(S_1 \wedge S_2) &:= L''(S_1) \wedge L''(S_2) \\
L''(A = B) &:= L''(A) = L''(B) \\
L''(A \neq B) &:= L''(A) \neq L''(B) \\
L''(enc\langle X_1, X_2, X_n\rangle) &:= \exists r.F(L''(X_1), X_2, r) \\
L''(F\langle X_1, \ldots, X_n\rangle) &:= F(L''(X_1), \ldots, L''(X_n)), \text{ for } n = arity(F) \\
L''(\langle com(crs, m, r), \varepsilon\rangle) &:= x_m \text{ , for } x_m \in V \\
L''(\langle com(crs, m, r), uv(m, r)\rangle) &:= m
\end{aligned}
$$

Then, $R^{\text{sym}}_{\text{adv}}$ is defined as follows:

$$\big\{(x, w) \mid \ L''(x)\,\{w_i/x_{w_i}\} \text{ is true, where } x = extrSta(t), w = extrWit(t))\big\}$$

Computationally, recall $R^{\text{comp}}_{\text{hon}}$ is defined as follows:

$$\big\{(img_\eta(x), img_\eta(w)) \mid \ (x, w) \in R^{\text{sym}}_{\text{hon}} \text{ for a consistent } \eta\big\}$$

We stress that our definition is equivalent to the previous definition.

Analogously, $R^{\text{comp}}_{\text{adv}}$ is defined using $R^{\text{sym}}_{\text{hon}}$:

$$\big\{(img_\eta(x), img_\eta(w)) \mid \ (x, w) \in R^{\text{sym}}_{\text{adv}} \text{ for a consistent } \eta\big\}$$

By construction Definition 6 obviously holds.

3.6.3. Transparent hybrid executions

In the presence of zero-knowledge transformations it might happen that the simulator parses a zero-knowledge proof from the attacker, further transforms this proof, and sends this transformed proof back to the attacker. Then, the simulator in interaction with the hybrid execution would need to construct a proof even though some witnesses are only

known to the attacker. We remedy this problem by directly applying the implementations of the zero-knowledge transformations that lead to this term.

In order to be able to determine which zero-knowledge transformations have been applied to a term, we modify the hybrid execution to a so-called *transparent hybrid execution* that is defined as follows: the result of a destructor node is in the yes-branch instead of $f(\underline{t})$ the term $\hat{f}(\underline{t})$, where for every destructor f, we introduce a free constructor $\hat{f}$. We define the set of *extended terms* $\mathbf{T}'$ as the set of terms $\mathbf{T}$ that may also contain the constructors $\hat{f}$ for $f \in \mathbf{D}$. Moreover, instead of checking at a destructor node whether $f(\underline{t})$ outputs $\bot$ or another term, the symbolic execution first needs to get rid of the $\hat{f}$ constructors, by evaluating each t_i: $eval'\hat{f}(\underline{s}) := f(\underline{eval's})$ and $eval't := t$ otherwise. With this transparent hybrid execution the simulator is able to track the symbolically applied transformations and thereby to compute $\beta(\hat{f}(\underline{t}))$ as $A_{\hat{f}}(\underline{\beta(t)})$.

We generalize this idea and let for a set of destructors $\mathbf{D}_t$ the transparent hybrid execution only send terms for which every destructor application $f(\underline{t})$ with $f \in \mathbf{D}_t$ has been replaced by a distinguished constructor application $\hat{f}(\underline{t})$. For the definition of a transparent hybrid execution, we use the evaluation function $eval''$, where $\mathbf{T}''$ denotes the extended set of terms in which for each destructor d a constructor $\hat{d}$ has been introduced. We define $eval_d$ as in the CoSP paper: for $d \in \mathbf{C} \cup \{\hat{g} \mid g \in \mathbf{D}_t\}$ we have $eval_d(\underline{t}) := d(\underline{t})$ if $d(\underline{t}) \in \mathbf{T}$, otherwise $d(\underline{t}) := \bot$; for $d \in \mathbf{D} \setminus \mathbf{D}_t$ we have $eval_d(\underline{t}) := d(\underline{t})$ if $d(\underline{t}) \neq \bot$, otherwise $d(\underline{t}) := \bot$.

Recall that our symbolic abstraction explicitly models the commitments on the witnesses. Compared to previous work, we therefore need to be able to determine whether a commitment has been reused in an MZK proof. However, unless we assume extraction zero-knowledge we cannot use the extractor on simulated MZK proofs. As a consequence, we allow the simulator *Sim* to not only send terms but also variables and refine the assignment f from nodes to terms or variables upon every destructor application of the transparent hybrid execution. Let X be the set of variables. We define the set of terms as $\mathbf{T}'$ as $\mathbf{T}'' \cup X \cup \{\bot\}$. Moreover, we allow the simulator to send an assignment $\alpha : X \to \mathbf{T}'$ in order to refine the interpretation of previously sent terms. In detail, upon each destructor application the transparent hybrid execution queries the simulator *Sim* with (`update`) for a refinement α of the terms. Our destructors are defined by a set of conditional rewriting rules; hence we can naturally extend $eval''d(\underline{t})$ with $\underline{t} \in \mathbf{T}''^{|\underline{t}|}$ to $eval'd(\underline{t})$ with $\underline{t} \in \mathbf{T}'^{|\underline{t}|}$ where all variables x in t are treated as attacker nonces $n_x \in \mathbf{N}_E$.

We define the following function:

$$\begin{aligned} saturate(f, \alpha) &:= \lim_{i \to \infty} g_i \text{ where} \\ g_0 := f \text{ and } g_i &:= \big\{(n, t) \mid \exists C, \underline{y}.C[\underline{y}] = g_i(n) \\ &\qquad \wedge t = C[\alpha(y_1), \ldots, \alpha(y_n)]\big\} \end{aligned}$$

We stress that *saturate* is poly-time computable if α is the identity function up to polynomially many variables (which is how we use it in our proof).

Definition 36 (Transparent hybrid execution)**.** *Let Π_p be a probabilistic CoSP protocol, and let Sim be a simulator. We define a probability distribution TH-Trace$_{\boldsymbol{M},\boldsymbol{D}_t,\Pi_\mathsf{p},Sim}(k)$ on (finite) lists of tuples (S_i, ν_i, f_i) called the* full $\mathbf{D}_t$-transparent hybrid trace *according to the following probabilistic algorithm Π^T, run on input k, that interacts with Sim. (Π^T is called the transparent hybrid protocol machine associated with Π_p and internally runs a symbolic simulation of Π_p as follows:)*

- Start*: $S_1 := S := \varnothing$, $\nu_1 := \nu$ is the root of Π^T, and $f_1 := f$ is a totally undefined partial function mapping node identifiers to $\boldsymbol{T}$.*
- Transition*: For $i = 2, 3, \ldots$ do the following:*
  - *Let $\underline{\tilde{\nu}}$ be the node identifiers in the label of ν. Define $\underline{\tilde{t}}$ through $\tilde{t}_j := f(\tilde{\nu}_j)$.*
  - *Proceed depending on the type of ν:*
    - *If ν is a computation node with destructor d, then send* (`update`) *to Sim. Wait for a variable assignment $\alpha : X \to \boldsymbol{T}'$ as a response. Let id_S be the identity function of a set S. Update $f := saturate(f, \alpha)$ with α. Let $m := eval'_d(\underline{\tilde{t}})$. If $eval'd(\underline{\tilde{t}}) \neq \bot$, let ν' be the yes-successor of ν and let $f' := f(\nu := m)$. If $eval'f(\underline{\tilde{t}}) = \bot$, let ν' be the no-successor of ν and let $f' := f$. Set $f := f'$ and $\nu := \nu'$.*
    - *If ν is a computation node with constructor, destructor, or nonce d, then let $m := eval_d(\underline{\tilde{t}})$. If $eval'd(\underline{\tilde{t}}) \neq \bot$, let ν' be the yes-successor of ν and let $f' := f(\nu := m)$. If $eval'f(\underline{\tilde{t}}) = \bot$, let ν' be the no-successor of ν and let $f' := f$. Set $f := f'$ and $\nu := \nu'$.*
    - *If ν is an output node, send $\tilde{t}_1$ to Sim (but without handing over control to Sim). Let ν' be the unique successor of ν and let $S' := S \cup \{\tilde{t}_1\}$. Set $\nu := \nu'$ and $S := S'$.*
    - *If ν is an input node, hand control to Sim, and wait to receive $m \in \boldsymbol{T}$ from Sim. Let $f' := f(\nu := m)$, and let ν' be the unique successor of ν. Set $f := f'$ and $\nu := \nu'$.*
    - *If ν is a control node labeled with out-metadata l, send l to Sim, hand control to Sim, and wait to receive a bitstring l' from Sim. Let ν' be the successor of ν along the edge labeled l' (or the lexicographically smallest edge if there is no edge with label l'). Set $\nu := \nu'$.*
    - *If ν is a nondeterministic node, sample ν' according to the probability distribution specified in ν. Set $\nu := \nu'$.*
  - *Send $(info, \nu, t)$ to Sim. When receiving an answer (proceed) from Sim, continue.*
  - *If Sim has terminated, stop. Otherwise let $(S_i, \nu_i, f_i) := (S, \nu, f)$.*

The probability distribution of the (finite) list $\nu_1, \ldots$ produced by this algorithm we denote by $\text{TH-Nodes}'_{\boldsymbol{M},\boldsymbol{D}_t,\Pi_{\mathsf{p}},Sim}(k)$. By replacing every variable by distinct fresh nonce from $\boldsymbol{N}_E$, and by replacing $com(c, \bot, N)$ by $garbCom(c, N)$, we get (the distribution of traces) $\text{TH-Nodes}_{\boldsymbol{M},\boldsymbol{D}_t,\Pi_{\mathsf{p}},Sim}(k)$. We call this distribution the $\mathbf{D}_t$-transparent hybrid node trace*. In abuse of notation, we often omit the adjective transparent since we only consider transparent hybrid executions and traces.*

We stress that the definition of a good simulator can be extended to the transparent hybrid execution.

Analogous to the proof of the CoSP paper which shows that the existence of a good simulator implies computational soundness, it suffices to show the lemma about the hybrid node traces.

Lemma 3. *Consider a transparent hybrid execution of $Sim + \Pi^T$ in which Sim is DY, i.e., we have $\{\boldsymbol{T}_1, \ldots, \boldsymbol{T}_\ell\} \vdash m_\ell$ for all $\boldsymbol{T}_i$ and m_ℓ as in the definition of the symbolic execution in CoSP [BHU09] and all ℓ.*

Let tr be the full hybrid trace of that execution. Then tr is a full symbolic trace of Π_{s}.

Proof. We show that tr fulfills the conditions on full traces of the definition of the symbolic execution from the CoSP paper. [BHU09] This is clear for constructor, and control nodes, since the processing of these nodes in the hybrid setting of Definition 36 matches the one in the symbolic setting of the definition of symbolic execution in CoSP. For destructor nodes we have for all $\underline{t} \in \mathbf{T}^n$ $eval\hat{f}(\underline{t}) = \bot \Leftrightarrow eval_f(\underline{t}) = \bot$. Consequently, the same branches are taken in both settings.

Input/output nodes in tr consist of an extended term $t \in \mathbf{T}'$ sent from Π^T to Sim, or an extended term t' sent from Sim to Π^T. By the DY property of Sim, we know that $S \vdash eval't'$, where S denotes all terms (including t) sent from Π^T to Sim so far. Hence, the node satisfies the requirement for input/output nodes from the definition of symbolic execution in CoSP. This completes the proof of the lemma. □

The previous lemma immediately implies the following lemma.

Lemma 4 (Good simulator implies soundness)**.** *Let* $\mathbf{M} = (\boldsymbol{C}, \boldsymbol{N}, \boldsymbol{T}, \boldsymbol{D}, \vdash)$ *be a symbolic model, let* $\boldsymbol{D}_t \subseteq \boldsymbol{D}$ *be a set of destructors,let* P *be a class of CoSP protocols, and let* A *be a computational implementation of* $\boldsymbol{M}$*. Assume that for every efficient probabilistic CoSP protocol* Π_p *(whose corresponding CoSP protocol is in* P*), every probabilistic polynomial-time adversary* $\mathcal{A}$*, and every polynomial* p*, there exists a good simulator against the* $\boldsymbol{D}_t$*-transparent hybrid execution for* $\boldsymbol{M}$*,* Π_p*,* A*,* $\mathcal{A}$*, and* p*. Then* A *is computationally sound for all protocols in* P*.*

3.6.4. The simulator *Sim*

In the definition of the simulator we needed two functions β and τ to translate between the symbolic and computational worlds. Here is the full definition of the simulator. From here on (Section 3.6.4, 3.6.5, 3.6.6), large parts of the computational soundness proof is quoted from [BBU13]. All modifications are marked in blue.

The partial function $\beta : \mathbf{T} \to \{0,1\}^*$**.**

- $\beta(N) := r_N$ if $N \in \mathbf{N}_P$
- $\beta(N^m) := m$
- $\beta(enc(ek(N), t, M)) := A_{enc}(\beta(ek(N)), \beta(t), r_M)$ if $M \in \mathbf{N}_P$
- $\beta(enc(ek(M), t, N^m)) := m$ if $M \in \mathbf{N}_P$
- $\beta(ek(N)) := A_{ek}(r_N)$ if $N \in \mathbf{N}_P$
- $\beta(ek(N^m)) := m$
- $\beta(dk(N)) := A_{dk}(r_N)$ if $N \in \mathbf{N}_P$
- $\beta(dk(N^m)) := d$ such that $\tau(d) = dk(N^m)$ was computed earlier
- $\beta(sig(sk(N), t, M)) := A_{sig}(A_{sk}(r_N), \beta(t), r_M)$ if $N, M \in \mathbf{N}_P$
- $\beta(sig(sk(M), t, N^s)) := s$
- $\beta(vk(N)) := A_{vk}(r_N)$ if $N \in \mathbf{N}_P$
- $\beta(vk(N^m)) := m$
- $\beta(sk(N)) := A_{sk}(r_N)$ if $N \in \mathbf{N}_P$
- $\beta(sk(N^m)) := s$ such that $\tau(s) = sk(N^m)$ was computed earlier
- $\beta(crs(N)) := A_{crs}(r_N)$ if $N \in \mathbf{N}_P$
- $\beta(crs(N^c)) := c$
- $\beta(ZK(t_1, t_2, N_2)) := A_{mkZK}(\beta(crsof(t_1)), \beta(extrSta(t_1)), \beta(extrWit(t_1)), r_{N_2})$ if $N_2 \in \mathbf{N}_P$
- $\beta(ZK(t_1 \wedge t_1', t_2 \wedge t_2', rand(N, N'))) :=$
  $A_{and_{ZK}}(\beta(ZK(t_1, t_2, N)), \beta(ZK(t_1', t_2', N')))$ if $N \in \mathbf{N}_E$ or $N' \in \mathbf{N}_E$
- $\beta(ZK(t_1, t_2, N^s)) := s$

- $\beta(\hat{f}(t_1,\ldots,t_n)) := A_f(\beta(t_1),\ldots,\beta(t_n))$ if $f \in \hat{F}$
- $\beta(com(crs(N),m,M)) := A_{com}(\beta(crs(N)),\beta(m),r_M)$
- $\beta(pair(t_1,t_2)) := A_{pair}(\beta(t_1),\beta(t_2))$
- $\beta(string_0(t)) := A_{string_0}(\beta(t))$
- $\beta(string_1(t)) := A_{string_1}(\beta(t))$
- $\beta(\varepsilon) := A_\varepsilon()$
- $\beta(garb(N^c)) := c$
- $\beta(garbCom(t,N^c)) := c$
- $\beta(garbEnc(t,N^c)) := c$
- $\beta(garbSig(t,N^s)) := s$
- $\beta(t) := \bot$ if no case matches

The total function $\tau : \{0,1\}^* \to \mathbf{T}$. (by taking the first matching rule)

- $\tau(r) := N$ if $r = r_N$ for some $N \in \mathbf{N}_P$ and N occurred in a term sent from Π^C
- $\tau(r) := N^r$ if r is of type nonce
- $\tau(c) := enc(ek(M),t,N)$ if c has earlier been output by $\beta(enc(ek(M),t,N))$ for some $M \in \mathbf{N}, N \in \mathbf{N}_P$
- $\tau(c) := enc(ek(M),\tau(m),N^c)$ if c is of type ciphertext and $\tau(A_{ekof}(c)) = ek(M)$ for some $M \in \mathbf{N}_P$ and $m := A_{dec}(A_{dk}(r_N),c) \neq \bot$
- $\tau(c) := garbEnc(\tau(A_{ekof}(c)),N^c)$ if c is of type encryption
- $\tau(c) := ek(N)$ if c has earlier been output by $\beta(ek(N))$ for some $N \in \mathbf{N}_P$
- $\tau(c) := ek(N^c)$ if c is of type encryption key
- $\tau(c) := dk(N)$ if $c = A_{dk}(r_N)$ for some N that occurred in a subterm of the form $ek(N)$ or $dk(N)$ before
- $\tau(c) := dk(N^e)$ if c is of type decryption key and e is the encryption key corresponding to c
- $\tau(s) := sig(sk(M),t,N)$ if s has earlier been output by $\beta(sig(sk(M),t,N))$ for some $M,N \in \mathbf{N}_P$
- $\tau(s) := sig(sk(M),\tau(m),N^s)$ if s is of type signature and $\tau(A_{vkof}(s)) = vk(M)$ for some $M \in \mathbf{N}$ and $m := A_{ver_{sig}}(A_{vkof}(s),s) \neq \bot$
- $\tau(s) := garbSig(\tau(A_{vkof}(s)),N^s)$ if s is of type signature
- $\tau(s) := vk(N)$ if $s = A_{vk}$ for some N that occurred in a subterm of the form $vk(N)$ or $sk(N)$ before
- $\tau(s) := vk(N^s)$ if s is of type verification key
- $\tau(s) := sk(N)$ if $s = A_{sk}$ for some N that occurred in a subterm of the form $vk(N)$ or $sk(N)$ before
- $\tau(s) := sk(N^c)$ if s is of type signing key and c is the signing key corresponding to s
- $\tau(z) := crs(N)$ if $z = A_{crs}(r_N)$ for some N that occurred in a subterm of the form $crs(N)$ before
- $\tau(z) := crs(N^z)$ if z is of type common reference string
- $\tau(z) := com(c,m,N)$ if z is of type commitment and z has been output earlier by $\beta(com(c,m,N))$ for some $N \in \mathbf{N}_P$.
- $\tau(z) := com(\tau(c),x_z,N^z)$ if z is of type commitment and where $c := A_{crsof}(z)$ and $x_z \in X$ is a variable.
- $\tau(z) := \hat{f}(\overline{s})$ if z is of type zero-knowledge and z has been output earlier by $\beta(\hat{f}(\overline{s}))$ for some $\overline{s}$
- $\tau(z) := ZK(t_1,t_2,N_2)$ if z is of type zero-knowledge proof and z has been output earlier by $\beta(ZK(t_1,t_2,N_2))$ for some $N_2 \in \mathbf{N}_P$
- $\tau(z) := ZK(t_1,t_2,N^z)$ if z is of type zero-knowledge proof, $A_{ver_{zk}}(A_{crsof}(z),z) = 1$, $m_{t_1} := A_{getPub}(z) \neq \bot$, $t_1 := C[x_1,\ldots,x_n] := \tau(m_{t_1}) \neq \bot$, $crs(N) = \tau(A_{crsof}(m_{t_1}))$ for some $N \in \mathbf{N}_P$, and $\alpha' := \mathbf{SymbExtr}(S, C[\alpha(x_1),\ldots,\alpha(x_n)])$ where S is the set of terms sent to the adversary before and $t_2 := randomnessTree(t_1)$, where $randomnessTree(t_1)$ is defined by replacing $\langle com(crs,t,r),U\rangle$ by r (in the FMZK variant), or $\langle t,r\rangle$ (in the CMZK) variant. If $\bot \in \alpha'(extrWit(t_1))$, we say an *extraction-failure* on (N,z,m_{t_1}) occurred, see below for the

behavior of Sim in this case. Otherwise, set $\alpha := \alpha' \circ \alpha$.[8]

- $\tau(z) := and_{ZK}(\tau(z'), \tau(z''))$ if z is of type zero-knowledge proof and $A_{splitAnd}(z) = (z', z'') \neq \bot$, $A_{ver_{zk}}(A_{crsof}(z), z) = 0$.
- $\tau(z) := ZK(t_1, t_2, N^z)$ if z is of type zero-knowledge proof, and $t_1 := \tau(A_{getPub}(z))$, where $t_2 := randomnessTree(t_1)$. Since z is of type zero-knowledge, $A_{getPub}(z)$ is a valid statement and $randomnessTree(t_1)$ is defined. Moreover, set $\alpha'(x) := \bot$ for each $x \in extractWitness(t_1)$ and $\alpha := \alpha' \circ \alpha$.
- $\tau(m) := pair(\tau(A_{fst}(m)), \tau(A_{snd}(m)))$ if m is of type pair
- $\tau(m) := string_0(\tau(m'))$ if m is of type payload-string and $m' := A_{unstring_0}(m) \neq \bot$
- $\tau(m) := string_1(\tau(m'))$ if m is of type payload-string and $m' := A_{unstring_1}(m) \neq \bot$
- $\tau(m) := \varepsilon$ if m is of type payload-string and $m = A_{\varepsilon}()$
- $\tau(m) := garb(N^m)$ otherwise

When an extraction-failure on (N, z, m_{t_1}) occurs(i.e., when in the computation of τ, $\alpha' = \mathbf{SymbExtr}(z, t_1, \alpha)$, the simulator computes $(crs, simtd, extd) \leftarrow \mathbf{K}(1^\eta; r_N)$ to get the extraction trapdoor $extd$ corresponding to $crs = A_{crs}(r_N)$. Then the simulator invokes $m_w := \mathbf{E}(m_{t_1}, z, extd)$. If $(m_{t_1}, m_w) \notin \mathcal{R}$, we say a *ZK-break* occurred.

Upon a message (`update`) by the transparent hybrid execution, Sim responds with its current witness variable assignment $\alpha : X \to \mathbf{T}'$, and sets $\alpha := \emptyset$.

We define τ^* by the same case distinction as τ but remove the case in which an extraction failure may occur (i.e., the case where we invoke $\mathbf{SymbExtr}(S, x)$. Consequently, every adversary generated ZK-proof is by τ^* parsed as a proof that contains garbage commitments. Thus, by definition, there is no extraction failure during a computation of τ^*.

The faking simulator Sim_f. In the transformations from Sim over Sim_i ($i = 1$ to 7) to Sim_f all invocations of cryptographic algorithms (such as A_{ZK} and A_{enc}) are replaced by oracles and thereafter by faking oracles.

- We define Sim_1 like Sim but we change β to use zero-knowledge oracles instead of computing A_{crs}, A_{mkZK}, and A_{com}. More precisely, assume an oracle $\mathcal{O}_{ZK,com}$ that internally picks $(crs, simtd, extd) \leftarrow \mathbf{K}(1^\eta)$ and that responds to four kinds of queries: Upon a (`crs`)-query, it returns crs, and upon a (`prove`, x, w)-query, it returns $\mathbf{P}(x, w, crs)$ if $(x, w) \in R_{\text{hon}}^{\text{comp}}$ and $\bot$ otherwise. Upon a (`extd`)-query, it returns $extd$. For each $N \in \mathbf{N}_P$, Sim_1 maintains an instance $\mathcal{O}_{ZK,com}^N$ of $\mathcal{O}_{ZK,com}$. Upon a (`commit`, m)-query, it uniformly chooses a random bitstring r and returns $A_{com}(crs, m, r)$. Then Sim_1 computes $\beta(crs(N))$ with $N \in \mathbf{N}_P$ as $\beta(crs(N)) := \mathcal{O}_{ZK,com}^N(\texttt{crs})$, and Sim_1 computes $\beta(ZK(t_1, t_2, N_2))$ with $N_2 \in \mathbf{N}_P$ as $\beta(ZK(t_1, t_2, N_2)) := \mathcal{O}_{ZK,com}^N(\texttt{prove}, \beta(t_1), \beta(t_2))$. In case of an extraction-failure, Sim_1 performs a (`extd`)-query to get $extd$. (Here and in the descriptions of $Sim_2, \ldots, Sim_7, Sim_f$, we implicitly require that $\beta(t)$ caches the results of the oracle queries and does not repeat the oracle query when β is applied to the *same* term t again.)
  In the definition of $\tau(z) = crs(N)$ for $N \in \mathbf{N}_P$, instead of checking $z = A_{crs}(r_N)$, Sim_1 checks whether z is equal to the (`crs`)-query outcomes for all oracles $\mathcal{O}_{ZK,com}^N$ which have been used so far.
- We define Sim_2 like Sim_1, except that we replace the oracle $\mathcal{O}_{ZK,com}$ by an oracle $\mathcal{O}_{\text{sim}}$. That oracle behaves like $\mathcal{O}_{ZK,com}$, except that upon a (`prove`, x, w)-query, it returns $\mathbf{S}(x, crs, simtd)$ if $(x, w) \in R_{\text{hon}}^{\text{comp}}$ and $\bot$ otherwise.
- We define Sim_3 like Sim_2, except that we replace the oracle $\mathcal{O}_{\text{sim}}$ by an oracle $\mathcal{O}'_{\text{sim}}$.

[8] Formally, this is defined as $\{(x, C[C'_1[\underline{z}_1], \ldots, C'_n[\underline{z}_n]]) \mid C[\underline{y}] = \alpha(x) \wedge C'[\underline{z}] = \alpha'(y)\}$.

That oracle behaves like $\mathcal{O}_{\text{sim}}$, except that upon a $(\texttt{prove}, x, w)$-query, it returns $\mathbf{S}(x, crs, simtd)$ (even if $(x, w) \notin R_{\text{hon}}^{\text{comp}}$). Therefore the simulator only queries $(prove, x)$ and does not compute w any more.

- We define Sim_4 like Sim_3, but we change β to call the oracle $\mathcal{O}_0$ instead of computing A_f. For every $N \in \mathbf{N}_P$ the simulator Sim_4 maintains a distinct oracle $\mathcal{O}_0^N(\texttt{f}, \cdot, \cdot, simtd)$ for $crs(N)$. Moreover, $\mathcal{O}_0^N(\texttt{f}, \cdot, \cdot, simtd)$ gets a special command $(simtd)$ for $\mathcal{O}_{ZK,com}^N$ upon which the simulation trapdoor $simtd$ is output. Upon the first call $\mathcal{O}_0^N$ queries $\mathcal{O}_{ZK,com}^N$ with $(simtd)$.
- We define Sim_5 like Sim_6, but we change β and τ to call the simulating oracle $\mathcal{O}_1(\texttt{f}, \cdot, \cdot, simtd)$ instead of querying $\mathcal{O}_0(\texttt{f}, \cdot, \cdot, simtd)$.
- We define Sim_6 like Sim_5, but we change β and τ to use encryption oracles instead of computing A_{enc}, A_{dec}, A_{ek}, A_{dk}. More precisely, assume an oracle $\mathcal{O}_{enc}$ that internally picks $(ek, dk) \leftarrow KeyGen_{enc}(1^\eta)$ and that responds to three kinds of queries: Upon an $(\texttt{ek})$-query, it returns ek. Upon a $(\texttt{enc}, m)$-query, it returns $ENC(ek, m)$. Upon a $(\texttt{dec}, c)$-query, it returns $DEC(dk, c)$. Sim_6 maintains an instance $\mathcal{O}_{\text{enc}}^N$ for each $N \in \mathbf{N}_P$. Then Sim_6 computes $\beta(ek(N))$ with $N \in \mathbf{N}_P$ as $\beta(ek(N)) := \mathcal{O}_{\text{enc}}^N(\texttt{ek})$. And it computes $\beta(enc(ek(N), t, M))$ with $N, M \in \mathbf{N}_P$ as $\beta(enc(ek(N), t, M)) := \mathcal{O}_{\text{enc}}^N(\texttt{enc}, \beta(t))$. And it computes $\beta(dk(N)) := \bot$. And in the computation of $\tau(c)$ for c of type ciphertext, the computation of $A_{dec}(A_{dk}(r_N), c)$ is replaced by $\mathcal{O}_{\text{enc}}^N(\texttt{dec}, c)$.
  In the definition of $\tau(c) = ek(N)$ and $\tau(c) = dk(N)$ for $N \in \mathbf{N}_P$, instead of checking $c = A_{ek}(r_N)$ and $c = A_{dk}(r_N)$, Sim_6 checks whether c is equal to the corresponding query outcomes for all oracles $\mathcal{O}_{enc}^N$ which have been used so far.
- We define Sim_7 like Sim_6, except that we replace the oracle $\mathcal{O}_{enc}$ by an oracle $\mathcal{O}_{\text{fake}}$. That oracle behaves like $\mathcal{O}_{enc}$, except that upon an $(\texttt{enc}, x)$-query, it returns $ENC(ek, 0^{|x|})$.
- We define Sim_f like Sim_7, but we change β to use signing oracles instead of computing A_{vk}, A_{sk}, A_{sig}. More precisely, we assume an oracle $\mathcal{O}_{sig}$ that internally picks $(vk, sk) \leftarrow KeyGen_{sig}(1^\eta)$ and that responds to two kinds of queries: Upon a $(\texttt{vk})$-request, it returns vk, and upon a $(\texttt{sig}, m)$-request, it returns $SIG(sk, m)$. Sim_f maintains an instance $\mathcal{O}_{sig}^N$ for each $N \in \mathbf{N}_P$. Then Sim_f computes $\beta(vk(N))$ with $N \in \mathbf{N}_P$ as $\beta(vk(N)) := \mathcal{O}_{sig}^N(\texttt{vk})$. And $\beta(sk(N))$ with $N \in \mathbf{N}_P$ as $\beta(sk(N)) := \bot$. And $\beta(sig(sk(N), t, M))$ with $N, M \in \mathbf{N}_P$ as $\beta(sig(sk(N), t, M)) := \mathcal{O}_{sig}^N(\texttt{sig}, \beta(t))$. In the definition of $\tau(c) = vk(N)$ and $\tau(c) = sk(N)$ for $N \in \mathbf{N}_P$, instead of checking $c = A_{vk}(r_N)$ and $c = A_{sk}(r_N)$, Sim_f checks whether c is equal to the corresponding query outcomes for all oracles $\mathcal{O}_{sig}^N$ which have been used so far.

Large parts of the computational soundness proof follow the computational soundness proof for non-malleable ZK proofs from Backes, Bendun and Unruh [BBU13].

The following two lemmas are shown in the work of Backes, Bendun, and Unruh [BBU13].

3.6.5. Sim_f is Dolev-Yao

Lemma 5 (Underivable subterms)**.** *For any invocation of τ or τ^* in the hybrid execution of Sim_f, let m denote the input to τ or τ^*, let u' denote the output of τ or τ^*, and let S be the set of all messages sent from Π^C to Sim_f up to that invocation of τ or τ^*.*

Let C be a context and $u \in \boldsymbol{T}$ such that $u' = C[u]$ and $S \not\vdash u$.

Then there is a term t_{bad} and a context D such that D can be obtained by the following grammer:

$$\begin{aligned} D ::= \Box \mid & \mathit{pair}(t, D) \mid \mathit{pair}(D, t) \mid \mathit{enc}(\mathit{ek}(N), D, M) \\ \mid & \mathit{enc}(D, t, M) \mid \mathit{sig}(\mathit{sk}(M), D, M) \\ \mid & \mathit{com}(D, r, N) \mid \mathit{com}(t, D, M) \\ \mid & \mathit{com}(t, D, N) \text{ (for public commitments)} \\ \mid & \mathit{ZK}(D, r, M) \mid \mathit{garbEnc}(D, M) \\ \mid & \mathit{garbSig}(D, M) \mid \mathit{garbCom}(D, M) \\ & \text{with } N \in \boldsymbol{N}_P, M \in \boldsymbol{N}_E, t, t' \in \boldsymbol{T} \end{aligned}$$

with $u = D[t_{bad}]$ such that $S \not\vdash t_{bad}$ and such that one of the following holds:

1. $t_{bad} \in \boldsymbol{N}_P$
2. $t_{bad} = \mathit{enc}(p, m, N)$ *with* $N \in \boldsymbol{N}_P$
3. $t_{bad} = \mathit{sig}(k, m, N)$ *with* $N \in \boldsymbol{N}_P$
4. $t_{bad} = \mathit{ZK}(x, r, N)$ *with* $N \in \boldsymbol{N}_P$
5. $t_{bad} = \mathit{sig}(\mathit{sk}(N), m, M)$ *with* $N \in \boldsymbol{N}_P$, $M \in \boldsymbol{N}_E$
6. $t_{bad} = \mathit{com}(\mathit{crs}(N), m, M)$ *with* $N, M \in \boldsymbol{N}_P$ *and* t_{bad} *is a witness commitment*
7. $t_{bad} = \mathit{ZK}(t, r, N)$ *with* $N \in \boldsymbol{N}_P$
8. $t_{bad} = \mathit{crs}(N)$ *with* $N \in \boldsymbol{N}_P$
9. $t_{bad} = \mathit{ek}(N)$ *with* $N \in \boldsymbol{N}_P$
10. $t_{bad} = \mathit{vk}(N)$ *with* $N \in \boldsymbol{N}_P$
11. $t_{bad} = \mathit{sk}(N)$ *with* $N \in \boldsymbol{N}_P$
12. $t_{bad} = \mathit{dk}(N)$ *with* $N \in \boldsymbol{N}_P$

Proof. We prove the lemma by structural induction on u. We formulate the proof for an invocation of τ, for an invocation of τ^* the proof is identical. There are the following cases:

Case 1: " $u \in \{\mathit{ek}(N), \mathit{vk}(N), \mathit{crs}(N), \mathit{dk}(N), \mathit{sk}(N)\}$ with $N \notin \mathbf{N}_P$"

Let $u = C(N)$ for $C \in \{\mathit{ek}, \mathit{vk}, \mathit{crs}, \mathit{dk}, \mathit{sk}\}$. By protocol conditions 1 and 13 each C-node has as annotation a nonce from $\mathbf{N}_P$. Therefore u cannot be honestly generated, that means there is some $e \in \{0,1\}^*$ such that $\tau(e) = u$ and u has the form $C(N^e)$. But then $S \vdash u$ contradicting the premise of the lemma.

Case 2: " $u \in \{\mathit{ek}(N), \mathit{vk}(N), \mathit{crs}(N), \mathit{dk}(N), \mathit{sk}(N)\}$ with $N \in \mathbf{N}_P$"

Then the claim is fulfilled with $D := \Box$ and $t_{bad} = u$.

Case 3: "$u = \mathit{garb}(u_1)$"

By protocol condition 2 no garbage term is generated by the protocol. Therefore there is a $c \in \{0,1\}^*$ such that $\tau(c) = \mathit{garb}(N^c) = u$. But this means that $S \vdash u$, contradicting the premise of the lemma.

Case 4: "$u = \mathit{garbEnc}(u_1, u_2)$ or $u = \mathit{garbSig}(u_1, u_2)$"

By protocol condition 2 no garbage term is generated by the protocol. So there exists a $c \in \{0,1\}^*$ with $\tau(c) = \mathit{garbEnc}(u_1, N^c)$ or $\tau(c) = \mathit{garbSig}(u_1, N^c)$. Since $S \vdash N^c$ it follows that $S \not\vdash u_1$, because $S \not\vdash u$. Applying the induction hypothesis on u_1 leads to a context D' and a term t_{bad}. Using this term t_{bad} and the context $\mathit{garbEnc}(D', N^c)$, respectively $\mathit{garbSig}(D', N^c)$, shows the claim.

Case 5: "$u = pair(u_1, u_2)$"

Since $S \not\vdash u$ there is an $i \in \{1, 2\}$ such that $S \not\vdash u_i$. Let D be the context and t_{bad} the term given by applying the induction hypothesis to u_i. Then $D_1 := pair(D, M)$ or $D_2 := pair(M, D)$ is the context for the term u depending on i with the same term t_{bad}.

Case 6: "$u = \varepsilon$"

This case cannot happen because $S \vdash \varepsilon$, so the premise of the lemma is not fulfilled.

Case 7: "$u = string_0(u_1)$ or $u = string_1(u_1)$"

Again the premise is not fulfilled since inductively $S \vdash u_1$ with base case $u_1 = \varepsilon$ and therefore $S \vdash string_i(u_1)$ for $i \in \{0, 1\}$.

Case 8: "$u = N$ with $N \in \mathbf{N}_P \backslash \mathcal{N}$"

This case is impossible since u is not in the range of τ.

Case 9: "$u = N$ with $N \in \mathcal{N}$"

The context $D := \square$ and term $t_{bad} := u$ satisfy the lemma in this case.

Case 10: "$u = N$ with $N \in \mathbf{N}_E$"

In this case $S \vdash u$ by definition and therefore the lemma's premise does not hold.

Case 11: "$u = enc(u_1, u_2, N)$ with $N \in \mathbf{N}_P$"

The lemma is satisfied by $t_{bad} = u$ and $D = \square$.

Case 12: "$u = enc(u_1, u_2, u_3)$ with $u_3 \notin \mathbf{N}_P$ and $S \not\vdash u_1$"

Since $u_3 \notin \mathbf{N}_P$ it follows that u cannot be honestly generated because of protocol condition 7. Therefore there is a $c \in \{0, 1\}^*$ with $\tau(c) = enc(ek(M), u_2, N^c) = u$ for some $M \in \mathbf{N}_P$. Apply the induction hypothesis to u_1 getting t_{bad} and context D we can define $D' := enc(D, u_2, N^c)$ fulfilling the claim of the lemma with t_{bad}.

Case 13: "$u = enc(u_1, u_2, u_3)$ with $u_3 \notin \mathbf{N}_P$ and $S \vdash u_1$"

Since $u_3 \notin \mathbf{N}_P$ it follows that u cannot be honestly generated because of protocol condition 7. Therefore there is an $c \in \{0, 1\}^*$ with $\tau(c) = enc(ek(M), u_2, N^c) = u$ for some $M \in \mathbf{N}_P$. Since $S \vdash u_1$, $S \vdash N^c$, and $S \not\vdash u$, it follows that $S \not\vdash u_2$. Let D be the context and t_{bad} be the term resulting by the induction hypothesis applied to u_2. Then $D' := enc(ek(M), D, N^c)$ together with t_{bad} satisfies the lemma.

Case 14: "$u = com(u_1, u_2, N)$ with $N \in \mathbf{N}_P$ and u is a witness commitment"

By protocol condition 13, we know that there is a $c \in \{0, 1\}^*$ such that $\tau(c) = com(crs(N), t, N^c)$. Hence, the lemma is satisfied by $t_{bad} = com(crs(N), t, N^c)$ and $D = \square$.

Case 15: "$u = com(u_1, u_2, M)$ with $M \in \mathbf{N}_P$ and u is a public commitment"

As above, by protocol condition 13 $u_1 = crs(N)$ for some $N \in \mathbf{N}_P$. Since u is a public commitment, $S \vdash uv(u_2, M)$. Hence, by applying $open(com(u_1, u_2, M), uv(u_2, M)) = u_2$, we know that by induction hypothesis there is a D such that $S \vdash D[t_{bad}]$. Consequently, the lemma is satisfied by $D' = com(u_1, D, M)$.

Case 16: "$u = com(u_1, u_2, M)$ with $M \in \mathbf{N}_E$"

Since $u_3 \notin \mathbf{N}_P$ it follows that u cannot be honestly generated because of protocol

condition 7. Hence, there is a $c \in \{0,1\}^*$ such that $\tau(c) = com(u', x, N^c)$. Since $S \vdash u'$, $S \vdash N^c$, and $S \nvdash u$, it follows that $S \nvdash x$. Let D be the context and t_{bad} be the term resulting by the induction hypothesis applied to u_2. Then $D' := com(u', D, N^c)$ together with t_{bad} satisfies the lemma.

Case 17: "$u = sig(u_1, u_2, N)$ with $N \in \mathbf{N}_P$"

Use context $D := \Box$ and $t_{bad} = u$.

Case 18: "$u = sig(sk(N), u_1, u_3)$ with $u_3 \notin \mathbf{N}_P$ and $N \in \mathbf{N}_P$"

Since $u \in \mathbf{T}$ and $u_3 \notin \mathbf{N}_P$ follows that $u_3 \in \mathbf{N}_E$. Therefore the context $D := \Box$ and $t_{bad} = u$ proves the claim.

Case 19: "$u = sig(u_1, u_2, u_3)$ and $u_3 \notin \mathbf{N}_P$ and u_1 is not of the form $sk(N)$ with $N \in \mathbf{N}_P$"

Since $u_3 \notin \mathbf{N}_P$ we get by protocol condition 7 that u is not honestly generated, i.e., there is an $s \in \{0,1\}^*$ such that $\tau(s) = sig(sk(M), u_2, N^s) = u$ with $M \in \mathbf{N}$. Because u_1 has not the form $sk(N)$ for any $N \in \mathbf{N}_P$ follows that $M \in \mathbf{N}_E$, so $S \vdash M$ and therefore $S \vdash sk(M)$. In total we have $S \vdash u_1$, $S \vdash u_3$ but $S \nvdash u$ which implies that $S \nvdash u_2$. Applying the induction hypothesis to u_2 leads to a context D and a term t_{bad}. Defining $D' := sig(sk(M), D, N^s)$ completes the claim.

Case 20: "$u = ZK(u_1, u_2, N)$ with $N \in \mathbf{N}_P$"

Defining $t_{bad} = u$ and $D := \Box$ suffices.

Case 21: "$u = ZK(u_1, u_2, N)$ with $N \in \mathbf{N}_E$"

Since τ uses the function $randomnessTree(u_1)$ for u_2 and $N \in \mathbf{N}_E$, we know that $S \nvdash u_1$.

In this case we use the induction hypothesis on $getPub(u) = u_1$ to get the term t_{bad} and a context D. Then using t_{bad} and $D' := ZK(D, u_2, N)$ satisfies the lemma.

Case 22: "$u = ZK(u_1, u_2, N)$ with $N \in \mathbf{N}_E$"

This case cannot occur because u is not in the range of τ.

Case 23: "$u = \hat{f}(t_1, \ldots, t_n, x, w, N)$"

By the definition of τ, we know that $\tau(m) = u$ only if m was already sent by the protocol; hence already $S \vdash u$ holds true and this case cannot occur.

□

Lemma 6. *For any (direct or recursive) call of the function $\beta(t)$ performed by Sim_f, it holds that $S \vdash t$ where S is the set of all terms sent by Π^C to Sim_f up to that point.*

Proof. We prove it by induction on the recursion depth of the β-function. The base case is that $\beta(t)$ is directly invoked. But then t itself was received by the protocol, i.e., $t \in S$ and therefore $S \vdash t$.

So let $\beta(t)$ be called as subroutine of some t'. By induction hypothesis we have $S \vdash t'$. We need to show that $S \vdash t$. According to the definition of β there are the following possibilities for t':

1. $t' = sig(sk(N), t, M)$ with $N, M \in \mathbf{N}_P$
2. $t' = pair(t_1, t_2)$ with $t \in \{t_1, t_2\}$

3. $t' = string_0(t)$ or $t' = string_1(t)$
4. $t' = enc(ek(N^e), t, M)$ with $M \in \mathbf{N}_P$
5. $t' = ZK(t, t_2, N)$ with $N \in \mathbf{N}_P$

Note that the case $t' = enc(ek(N), t, M)$ with $N, M \in \mathbf{N}_P$ does not occur because—in contrast to Sim—the simulator Sim_f does not recursively invoke β on t but uses an oracle and produces $ENC(ek_N, 0^{\ell(t)})$. Analogously, in Sim_f $com(crs(N), t, M)$ does not recursively invoke β but uses an oracles that commits to $0^{\ell(t)}$. By protocol condition 13, we know that the protocol does not send the term $com(crs(N^e), t, M)$ for $N^e \in \mathbf{N}_E$. The case $t' = ZK(t_1, t, N)$ is not possible, either, because the simulator Sim_f calls the simulation oracle to construct the proof and therefore $\beta(\cdot)$ is not called on the witness t. Also the case $t' = \hat{f}(t_1, \ldots, t_n, t_{n+1}, t_{n+2}), N$ with $t \in \{t_1, \ldots, t_{n+2}\}$ and $N \in \mathbf{N}_P$ does not occur since all honest ZK-transformations are simulated.

Case 1: $S \vdash sig(sk(N), t, M) = t'$. Using $ver_{sig}(vkof(t'), t') = t$ we get $S \vdash t$.

Case 2: $S \vdash pair(t_1, t_2) = t'$. With $fst(t') = t_1$, $snd(t') = t_2$, and $t \in \{t_1, t_2\}$ we get $S \vdash t$.

Case 3: The cases $t' = string_0(t)$ and $t' = string_1(t)$ work as the two preceding using $unstring_0$ and $unstring_1$.

Case 4: $S \vdash enc(ek(N^e), t, M)$. Because $S \vdash N^e$ it follows that $S \vdash dk(N^e)$, so decryption can be applied resulting in t.

Case 5: $S \vdash ZK(t, t_2, N) = t'$. The lemma follows by applying the destructor $getPub$. □

□

Lemma 7. *For any invocation $\tau(m)$ of τ or $\tau^*(m)$ of τ^* in the hybrid execution of Sim_f, the following holds with overwhelming probability: Let S be the set of terms t that the protocol sent to the adversary up to the invocation $\tau(m)$ or $\tau^*(m)$. Then $S \vdash \tau(m)$ or $S \vdash \tau^*(m)$, respectively.*

In particular, Sim_f is DY for $\boldsymbol{M}$ and Π.

Proof. Assume there occurs an m as input of τ or τ^* such that $S \nvdash \tau(m)$ or $S \nvdash \tau^*(m)$, respectively. Consider the first such input m. Now we can use lemma 5 with context $C = \Box$ and $u' = u = t$ leading to a term t_{bad} and a context D such that t_{bad} is of the form 1-12 given by Lemma 5. Let m_{bad} be the corresponding bitstring, i.e. $\tau(m_{bad}) = t_{bad}$. For each of these cases we will derive that it can only happen with negligible probability. Note that τ^* only differ from τ in the case a ZK-proof $ZK(t_1, t_2, M)$ is output with $M \in \mathbf{N}_E$. We formulate the proof for an invocation of τ; the case of τ^* is identical.

Case 1: "$t_{bad} = N \in \mathbf{N}_P$".
By construction of β and Sim_f, it follows that Sim_f has only access to r_N if $\beta(N)$ is computed directly or in τ. Because $S \nvdash N$ we get by Lemma 6 that β was never invoked on N, i.e. Sim_f has only access to r_N via τ. Considering the definition of τ we see that r_N is used for comparisons. In particular if $\tau(r)$ is called for an r having type nonce then the simulator checks for all $N \in \mathbf{N}_P$ whether $r = r_N$. This check does not help guessing r_N because it only succeeds if r_N was guessed correctly and therefore the probability that $m_{bad} = r_N$ as input of τ is negligible.

Case 2: "$t_{bad} = enc(p, m, N)$ with $N \in \mathbf{N}_P$".

By definition τ only returns t_{bad} if $\beta(t_{bad})$ was called earlier. But since $S \not\vdash t_{bad}$ and Lemma 6 this case cannot occur.

Case 3: "$t_{bad} = sig(k, m, N)$ with $N \in \mathbf{N}_P$".
This case is completely analogue to the case that $t_{bad} = enc(p, m, N)$ with $N \in \mathbf{N}_P$.

Case 4: "$t_{bad} = com(crs(N), m, M)$ with $N, M \in \mathbf{N}_P$ and t_{bad} is witness commitment".
By definition τ only returns t_{bad} if $\beta(t_{bad})$ was called earlier. But since $S \not\vdash t_{bad}$ and Lemma 6 this case cannot occur.

Case 5: "$t_{bad} = ZK(x, r, N)$ with $N, M \in \mathbf{N}_P$".
By definition of τ, t_{bad} is only returned if it was a result of $\beta(t_{bad})$ earlier. But because $S \not\vdash t_{bad}$ and Lemma 6 this can not be the case.

Case 6: "$t_{bad} = crs(N)$ with $N \in \mathbf{N}_P$".
Since Lemma 6 β was not called on $crs(N)$, therefore $\tau(m) \neq crs(N)$ for all $m \in \{0,1\}^*$.

Case 7: "$t_{bad} = sig(sk(N), m', M)$ with $N \in \mathbf{N}_P$, $M \in \mathbf{N}_E$".
Because $S \not\vdash t_{bad}$ follows that β was not invoked on t_{bad}. Therefore m_{bad} is a signature that was not produced by the signing oracle, but it is valid with respect to verification key vk_N. Because of the strongly existential unforgeability this can only be the case with negligible probability.

Case 8: "$t_{bad} = ek(N)$ with $N \in \mathbf{N}_P$".
By Lemma 6 follows that the function β was not called on t_{bad}. So ek_N was not requested from the encryption oracle. Protocol condition 8 implies that the only term sent by Π^C containing $dk(N)$ is of the form $ZK(\cdot, \square, N)$. But since the zero-knowledge proofs are simulated in the latter case $dk(N)$ is not part of any computation, i.e. $\beta(dk(N))$ is not computed and dk_N is never requested from the encryption oracle. By Lemma 6 $S \not\vdash dk(N)$. Hence for all terms of the form $t = enc(ek(N), \cdot, \cdot)$ holds $S \not\vdash t$ and therefore no encryption having encryption key ek_N is requested from the oracle. Thus the only remaining possibly use of ek_N is requesting the decryption oracle. But by implementation condition 4 these requests will always fail unless $A_{ekof}(m_{bad}) = ek_N$, i.e. ek_N has already been guessed. This can only happen with negligible probability.

Case 9: "$t_{bad} = vk(N)$ with $N \in \mathbf{N}_P$".
By Lemma 6, $\beta(vk(N))$ is never computed and vk_N is never requested from the signing oracle. Assume $S \vdash sk(N)$. Then $S \vdash sig(sk(N), \varepsilon, N^e)$ for some $N^e \in \mathbf{N}_E$. But using the destructor $vkof$ follows that $S \vdash vk(N)$, a contradiction. So we also have $S \not\vdash sk(N)$ and $S \not\vdash sig(sk(N), \cdot, \cdot) = t$. Thus $\beta(sk(N))$ and $\beta(t)$ are never computed and therefore neither sk_N nor a signature with respect to sk_N is requested from the signing oracle. So the probability that $vk_N = m_{bad}$ occurs as input of τ is negligible.

Case 10: "$t_{bad} = sk(N)$ with $N \in \mathbf{N}_P$".
Because of protocol condition 9 t_{bad} can only occur in terms of the form $sig(\square, \cdot, \cdot, N)$ and $ZK(\cdot, \square, N)$. In the second case the construction of the proof is replaced by the simulation oracle and could be done by the adversary itself. So the adversary is be able to compute sk_N only having signatures. By the strongly existential unforgeability of the signature scheme, this can only happen with

negligible probability.

Case 11: "$t_{bad} = dk(N)$ with $N \in \mathbf{N}_P$".
Procotol condition 8 ensures that dk only occurs in terms of the form $dec(\square, \cdot)$ and $ZK(\cdot, \square, N)$. The latter case is replaced by the simulation oracle and can be computed by the adversary itself. Therefore the adversary is able to decrypt ciphertexts only knowing ciphertexts. By the IND-CCA property, this can only occur with negligible probability.

In total we get that if Sim_f is not DY, then with non-negligible probability Sim_f performs a computation of $\tau(m_{bad})$ but m_{bad} can only occur with negligible probability as an argument of τ, a contradiction. Therefore the assumption that Sim_f is not DY has to be false, i.e. Sim_f is DY. □

3.6.6. Sim_f is indistinguishable

Lemma 8 (Relating the relations)**.** *Let $R^{\text{comp}}_{\text{hon}}, R^{\text{comp}}_{\text{adv}}$ be relations implementing $R^{\text{sym}}_{\text{adv}}$ with usage restriction $R^{\text{sym}}_{\text{hon}}$.*

1. *In the hybrid execution of Sim and Sim_3 it holds with overwhelming probability: If $(x, w) \in R^{\text{sym}}_{\text{hon}}$ and x, w occur as node annotation of a ZK node in the execution, then it holds $(\beta(x), \beta(w)) \in R^{\text{comp}}_{\text{hon}}$.*
2. *In the hybrid execution of Sim_2 it holds with overwhelming probability: If $(m_x, m_w) \in R^{\text{comp}}_{\text{adv}}$ for some bitstrings m_x, m_w, then it holds $(\tau(m_x), \tau^*(m_w)) \in R^{\text{sym}}_{\text{adv}}$.*

Proof. We first define an environment η mapping terms to bitstrings. η depends on the current state of the execution. We will use η in both parts of the lemma. So let $t_1, \ldots, t_n$ be the terms sent by the protocol to the simulator so far.

For any term or subterm t that occurs as argument to β or output of τ, we define η as follows:

- For $t = N^m$ define $\eta(t) := m$.
- For $t = C(t_1, \ldots, t_n, N^m)$ define $\eta(t) := m$ for all C as stated in definition 34.[9]
- For $t = crs(N)$ with $N \in \mathbf{N}_P$ define $\eta(t)$ to be the crs produced by the oracle $\mathcal{O}^N_{ZK,com}$.
- For $t = ZK(x, r, M)$ with $M \in \mathbf{N}_P$ define $\eta(t)$ to be proof produced by $\mathcal{O}^N_{ZK,com}$ in the computation of $\beta(t)$.
- For $t = com(crs, m, N)$ with N $in\mathbf{N}_P$ define $\eta(t)$ to be the commitment produced by $\mathcal{O}^N_{ZK,com}$ in the computation of $\beta(t)$.
- For $t = N$ with $N \in \mathbf{N}_P$ we distinguish 2 cases. If t does neither occur in a term of the form $crs(t)$ nor in $ZK(x, r, t)$ for some x, r then define $\eta(t) := r_N$. Otherwise let $\eta(t)$ be undefined, i.e. $\eta(t) := \perp$.

Note that η is a consistent environment with overwhelming probability.

Most properties of consistency are satisfied by construction. The ZK case holds because of the indistinguishability of true proofs and their simulations. The only property that needs to be proven is the injectivity of η. We distinguish by the type of η's output.

[9] These are $\{ek, dk, vk, sk, enc, sig, com, crs, garbCom, garbSig, garbEnc, garb\}$

- Type nonce. A collision $r_M = r_N$ for $M, N \in \mathbf{N}_P$ occurs with negligible probability. For $N, M \in \mathbf{N}_P$, a collision occurs with negligible probability, because $r_N = r_M$ occurs with negligible probabilityand because a collision with the randomness of $\mathcal{O}^N_{ZK,com}$ has negligible probability (otherwise it would be possible to guess that randomness, compute the simulation trapdoor and fake proofs). The case $\eta(N^a) = \eta(N^b)$ for $a \neq b$ is even impossible. So consider the case $\eta(N) = \eta(M)$ for $N \in \mathbf{N}_P, M \in \mathbf{N}_E$. By protocol condition 2, it follows that M was output of τ, i.e. $M = N^n$ for some $n \in \{0,1\}^*$. First, let N be a nonce occurred inside $crs(N)$. Then it holds $\eta(N) = \bot \neq n = \eta(N^n)$.
  Otherwise, if N was used before n was received by the simulator, then n would have been parsed to N by construction of τ. So the first occurrence of N has to be after n was received. But then the adversary guessed a nonce. This can only happen with negligible probability.
- Type decryption key. For the same reasons as in the case of type nonce we only consider the case $\eta(dk(N)) = \eta(dk(M))$ for $N \in \mathbf{N}_P, M \in \mathbf{N}_E$. By protocol condition 2, it follows that $dk(M) = dk(N^d)$ for some $d \in \{0,1\}^*$, $dk(N^d)$ was subterm of an output of τ, and d was not output of β earlier (otherwise d would have been parsed to $dk(N)$). So the adversary used either no input or only encryptions plus the encryption key to compute $dk(N)$. By the CCA property, this can only be the case with negligible probability.
- Type signing key. This case is completely analogue to the decryption key type using the strongly existentially unforgeability instead of the CCA property.
- Type encryption key. As in the previous cases we can only need to consider $\eta(ek(N)) = \eta(ek(M))$ for $N \in \mathbf{N}_P, M \in \mathbf{N}_E$. By protocol condition 2, it follows that $ek(M) = ek(N^e)$ for some $e \in \{0,1\}^*$. But then τ parsed e to $ek(N^e)$, so neither $ek(N)$ nor $dk(N)$ was used. This means the adversary guessed an encryption key without having any information about it. This can only happen with negligible probability.
- Type verification key and common reference string. Analogue to the case of encryption key.
- Type zero-knowledge proof. Because τ is deterministic , the adversary can not generate two different zero-knowledge proofs which are mapped to the same bistring. So if there is a collision, then between a protocol generated proof and a adversary generated one.
- Type ciphertext, signature, and commitment. Analogue to the case of zero-knowledge proofs.
- Type pair. If there is a collision of two pairs, then there is a collision in the first argument and in the second. So by induction hypothesis this case occurs with negligible probability.
- Type payload-string. This type does not contain any nonces. So applying η to a term of this type leads to a unique bitstring which cannot be hit by any other term of this type (by implementation condition 17).
- No type. The only term which has no type is $garb(t)$ for $t \in \mathbf{T}$. By protocol condition 2 and construction of τ, it has to hold that $t = N^m$ for some $m \in \{0,1\}^*$.

Proof of part 1 *of the lemma.*
By Definition 35[10], it suffices to show that if $(x, w) \in R^{\text{sym}}_{\text{hon}}$ then there is a consistent $\eta \in \mathcal{E}$

[10]The part we will use here says $(x, w) \in R^{\text{sym}}_{\text{hon}}$ and $img_\eta(x) \neq \bot \neq img_\eta(w)$ implies $(img_\eta(x), img_\eta(w)) \in$

such that $(img_\eta(x), img_\eta(w)) = (\beta(x), \beta(w))$ since $\beta(x) \neq \bot \neq \beta(w)$. We show that the η defined above satisfies this criterium. Here, we prove the case for Sim_3. The proof for Sim is analoguous with the only difference in the cases of ZK and crs. Here, the definition of η is done as for enc and ek and the proof, as well.

For any term t that can occur in the execution of Sim_3 as annotation of a ZK node's statement or witness, we show that $img_\eta(t) = \beta(t)$. This will be done by structural induction on the term t:

- "$t = N$ with $N \in \mathbf{N}_P$". In this case $\beta(N) = r_N$. The nonce N may not occur as last argument of ZK or crs and inside x or w (protocol conditions 7 and 13). So N did not occur as last argument of ZK nor as argument of crs. Thus, it holds $img_\eta(N) = \eta(N) = r_N$ by definition of η.
- "$t = N$ with $N \in \mathbf{N}_E$". In this case $N = N^n$ for some $n \in \{0,1\}^*$. Thus $\eta(N^n) = n = \beta(N^n)$.
- "$t \in \{ek(u), dk(u), vk(u), sk(u)\}$ with $u \in \mathbf{T}$". In this case, it holds that $u \in \mathbf{N}$. If $u \in \mathbf{N}_E$, i.e. $u = N^c$ for some $c \in \{0,1\}^*$, then $\beta(t) = c = \eta(t) = img_\eta(t)$ by construction. So, consider $u \in \mathbf{N}_P$. Let $C \in \{ek, dk, vk, sk\}$ be the constructor such that $t = C(u)$. Then, it holds that $img_\eta(t) = A_C(img_\eta(u)) \overset{(*)}{=} A_C(\beta(u)) = \beta(t)$. Since $u \in \mathbf{N}_P$ and occurs in $C(u)$, it follows that u does neither occur in $crs(u)$ nor in $ZK(x, r, u)$ for $x, r \in \mathbf{T}'$ (protocol conditions forbid that these nonces are used more than once). Thus $img_\eta(u) = r_u = \beta(u)$. Hence equality $(*)$ holds.
- "$t = crs(N)$ with $N \in \mathbf{N}_P$". By definition $\beta(t)$ produces the crs using $\mathcal{O}^N_{ZK,com}$ and $img_\eta(crs(N)) = \eta(crs(N))$ which was defined as $\beta(t)$. Thus, it holds $img_\eta(t) = \beta(t)$.
- "$t = crs(N)$ with $N \in \mathbf{N}_E$". This case is analogue to the case $ek(N)$ with $N \in \mathbf{N}_E$.
- "$t = enc(u_1, u_2, u_3)$". If $u_3 \in \mathbf{N}_E$, then this case is analogue to the case $t = ek(u)$. So let $N := u_3 \in \mathbf{N}_P$. Then $\beta(t) = A_{enc}(\beta(u_1), \beta(u_2), r_N) = A_{enc}(img_\eta(u_1), img_\eta(u_2), r_N)$ by induction hypothesis. The nonce N may only occur inside this encryption and as witness of the ZK-proof (protocol condition 7). Thus, by $r_N = \eta(N) = img_\eta(N)$, it follows $\beta(t) = A_{enc}(img_\eta(u_1), img_\eta(u_2), img_\eta(N)) = img_\eta(t)$.
- "$t = sig(u_1, u_2, u_3)$". If $u_3 \in \mathbf{N}_E$, then this case is analogue to the case $t = ek(u)$. So let $N := u_3 \in \mathbf{N}_P$. By definition of τ, it follows that t was honestly generated. This means there was a sig-computation node that produced t. By protocol condition 11 this node is annotated by an sk-node. Since the protocol only uses its randomness (protocol condition 1), it follows that $u_1 = sk(M)$ for some $M \in \mathbf{N}_P$. Then, it holds $\beta(t) = A_{sig}(A_{sk}(r_M), \beta(u_2), r_N)$. Again, $r_N = img_\eta(N)$; the same holds for M. Since $\beta(sk(M)) = A_{sk}(M)$, it follows by induction hypothesis that $A_{sk}(r_M) = img_\eta(sk(M))$. In total, it holds $\beta(t) = A_{sig}(img_\eta(sk(M)), img_\eta(u_2), img_\eta(N)) = img_\eta(t)$.
- "$t = pair(u_1, u_2)$ where $u_1, u_2 \in \mathbf{T}$". By induction hypothesis, it follows $\beta(u_i) = img_\eta(u_i)$. Thus, it holds $\beta(pair(u_1, u_2)) = A_{pair}(\beta(u_1), \beta(u_2)) = A_{pair}(img_\eta(u_1), img_\eta(u_2)) = img_\eta(pair(u_1, u_2))$.
- "$t \in \{string_0(u), string_1(u), \varepsilon\}$ with $u \in \mathbf{T}$". These cases are analogous to the case $t = pair(u_1, u_2)$.
- "$t \in ZK(t_1 \wedge t_1', t_2 \wedge t_2', rand(N, N'))$ with $t_i, t_i' \in \mathbf{T}$". This case are analogous to the case $t = pair(u_1, u_2)$ using $splitAnd$.
- "$t \in \{garb(u_1), garbSig(u_1, u_2, u_3), garbEnc(u_1, u_2), garbCom(u_1, u_2), com(u_1, x, u_2)\}$ where $u_i \in \mathbf{T}$ and $\alpha(x) = \bot$." By protocol condition 2 follows

$R^{\text{comp}}_{\text{hon}}$.

that t was generated by τ, i.e. the last argument of t has the form N^m for some $m \in \{0,1\}^*$. By definition of β, it holds that $\beta(t) = m$. On the other hand, by definition of img_η, it holds $img_\eta(t) = \eta(t) = m$, as well.

- "$ZK(t_1, t_2, t_3)$ where $\alpha(x) = \perp$ for each $x \in extrWit(t_1)$". This case is analogous to the case above.

Proof of part 2 of the lemma.
It suffices to show that for each $m \in \{0,1\}^*$, that occurs with non-negligible probability in a hybrid execution of Sim_2, there is some η such that $m = img_\eta(\tau(m))$ holds. Then it follows by definition 35[11] $(m_x, m_w) \in R_{\text{adv}}^{\text{comp}} \implies (img_\eta(\tau(m_x)), img_\eta(\tau(m_w))) \in R_{\text{adv}}^{\text{comp}} \implies (\tau(m_x), \tau^*(m_w)) \in R_{\text{adv}}^{\text{sym}}$.

Take the same definition of η as in the case before. Note that this definition is canonical for an execution and does not depend on the term $\tau(m)$.

We will prove $m = img_\eta(\tau(m))$ by structural induction. Note that this suffices for τ^*, as well, since all cases for τ^* occur in τ.

- $\tau(m) = N$ for some $N \in \mathbf{N}_P$
  By construction of τ, it follows that $N \in \mathcal{N}$. Thus N was not argument of a *crs* or the last argument of a *ZK* node, by protocol conditions 1 and 7. Then $img_\eta(\tau(m)) = r_N = m$ where the last equality holds because of the definition of τ.
- $\tau(m) = N^m$
  Then by construction of η holds that $img_\eta(\tau(m)) = img_\eta(N^m) = \eta(N^m) = m$.
- $\tau(m) = enc(ek(M), t, N)$ for some $M \in \mathbf{N}, N \in \mathbf{N}_P$
  By definition of η holds that $img_\eta(\tau(m)) = img_\eta(enc(ek(M), t, N)) = A_{enc}(A_{ek}(\eta(M)), img_\eta(t), \eta(N))$. By definition of τ follows that m was earlier output by β and thus evaluating t again using img_η gives the same bitstring m_t, r_N is the same argument as in the earlier call and $ek(M)$ is the same, too. By determinism of the implementations (implementation condition 1) follows that the output is m.
- $\tau(s) \in \{sig(sk(M), t, N), com(crs(M), t_2, N)\}$ for some $M, N \in \mathbf{N}_P$
  This case is analogue to the one of $enc(ek(M), t, N)$ for some $M, N \in \mathbf{N}_P$.
- $\tau(m) = ek(N)$ for some $N \in \mathbf{N}_P$
  By definition of τ, it follows $m = A_{ek}(r_N)$. On the other hand, it holds that $img_\eta(ek(N)) = A_{ek}(\eta(N)) = A_{ek}(r_N) = m$ by construction.
- $\tau(m) \in \{ vk(N), sk(N), dk(N)\}$ for some $N \in \mathbf{N}_P$
  The same as the case of $ek(N)$.
- $\tau(m) = crs(N)$ for some $N \in \mathbf{N}_P$
  By definition of τ follows that m was output after a call of β on $crs(N)$. Thus m was output by the oracle and $\eta(N)$ is by definition the randomness used by the oracle to construct m. Thus $img_\eta(crs(N)) = A_{crs}(\eta(N)) = m$ where the last equality holds because of the definition of $\eta(N)$.
- $\tau(m) = ZK(t_1, t_2, N)$ for some $N \in \mathbf{N}_P$
  By definition of η follows that $img_\eta(ZK(t_1, t_2, N)) = \eta(ZK(t_1, t_2, N)) = m$.
- $\tau(m) = pair(t_1, t_2)$
  This case follows by the induction hypothesis and the determinism of the implementations.
- $\tau(m) \in \{string_0(t_1), string_1(t_1), \varepsilon\}$

[11] At this point we use $(img_\eta(x), img_\eta(w)) \in R_{\text{adv}}^{\text{comp}}$ implies $(x, w) \in R_{\text{adv}}^{\text{sym}}$.

The case of ε is trivial, since the implementation is deterministic. For the other cases holds that - by definition of τ - $t_1 = \tau(m')$ having $m' = A_{unstring_i}(m)$ where $i \in \{0,1\}$ and $\tau(m) = string_i(t_1)$. Applying the induction hypothesis to t_1 leads to $img_\eta(t_1) = m'$ and thus $img_\eta(\tau(m)) = A_{string_i}(img_\eta(\tau(m'))) = A_{string_i}(m')$ $= A_{string_i}(A_{unstring_i}(m)) = m$. Here the last equality holds by implementation condition 17.

- $\tau(m) \in \{enc(ek(M), t, N^m),\ ek(N^m),\ dk(N^m),\ garbEnc(t, N^m),\ com(t, t_2, N^m),$ $garbCom(t, N^m), sig(sk(M), t, N^m),\ garbSig(t, N^m),\ vk(N^m),\ sk(N^m),\ crs(N^m),$ $ZK(x, r, N^m),\ garb(N^m)\}$ for some $M \in \mathbf{N}_P$
  All of these cases follow immediately by definition of η and definition 34.

The proof for Sim_2 is the same. Recall, the only difference between Sim_3 and Sim_2 is that Sim_3 does not check if $(x, w) \in R^{\text{comp}}_{\text{hon}}$ any more.

□

Lemma 9 (No ZK-breaks)**.** *In the hybrid execution with Sim_1, ZK-breaks occur only with negligible probability.*

Proof. The simulator Sim_1 does not know the simulation trapdoor, does not have access to a simulation oracle, and only queries the extraction trapdoor once before it halts. We consider the machine M that behaves exactly as Sim_1 and stops before it queries the extraction trapdoor. This machine M serves as a valid adversary against the extractability game, because it never queries the extraction trapdoor. The last part of Sim_1 serves as a valid challenger in the extractability game. A ZK-break occurs if and only if M wins in the extractability game. But we assumed this probability to be negligible; consequently the probability that ZK-breaks occur is negligible. □

Lemma 10 (No invalid symbolic witnesses)**.** *Assume that Sim_3 is DY. Then, in the $\boldsymbol{D}_{\text{trans}}$-hybrid execution of Sim_3, for each ZK node with arguments t_1, t_2, t_3, it holds that $(t_2, extrWit(t_3)) \in R^{\text{sym}}_{\text{hon}}$ with overwhelming probability.*

The same holds for Sim if Sim is DY.

Proof. If Sim_3 is DY, then the hybrid execution of Sim_3 corresponds to a symbolic execution with overwhelming probability.

By definition of the hybrid execution, any hybrid execution is a valid symbolic execution, as long as the simulator does not send a term in the adversary's knowledge. Since Sim_3 is DY, this occurs only with negligible probability.

In the symbolic execution, the property $(t_2, t_3) \in R^{\text{sym}}_{\text{hon}}$ holds since Definition 6 by construction of $R^{\text{sym}}_{\text{hon}}$ and $R^{\text{comp}}_{\text{hon}}$. Thus in the case that the hybrid execution corresponds to a symbolic one, it follows that $(t_2, t_3) \in R^{\text{sym}}_{\text{hon}}$ with overwhelming probability.

The same proof shows the statement for Sim. □

Lemma 11 (Preservation of simulator-properties)**.**

(i) We have

$$TH\text{-}Nodes_{\boldsymbol{M},\Pi_{\mathrm{p}},Sim}(k) \overset{C}{\approx} TH\text{-}Nodes_{\boldsymbol{M},\Pi_{\mathrm{p}},Sim_f}(k)$$

(ii) We have that Sim is DY if and only if Sim_f is DY.

(iii) Let $\mathit{TH\text{-}Nodes}^{S}_{\boldsymbol{M},\Pi_{\mathrm{p}},\mathit{Sim}_i}(k)$ *be the distribution of traces of the knowledge set* S *from hybrid execution with* Sim_i*. Then, we have*

$$\mathit{TH\text{-}Nodes}^{S}_{\boldsymbol{M},\Pi_{\mathrm{p}},\mathit{Sim}}(k) \overset{C}{\approx} \mathit{TH\text{-}Nodes}^{S}_{\boldsymbol{M},\Pi_{\mathrm{p}},\mathit{Sim}_f}(k)$$

Proof of (i) *and* (ii). For $x \in \{1,\ldots,7,f\}$ or x being the empty word. The same way, we denote the event that a simulator Sim_x is DY in that execution by DY_x. We abbreviate $\mathit{TH\text{-}Nodes}_{\mathbf{M},\Pi_{\mathrm{p}},\mathit{Sim}_x}(k)$ as $\mathit{TH\text{-}Nodes}_x$.

To show the lemma, we will show that

$$(\mathrm{DY}, \mathit{TH\text{-}Nodes}) \overset{C}{\approx} (\mathrm{DY}_1, \mathit{TH\text{-}Nodes}_1)$$
$$(\mathrm{DY}_1, \mathit{TH\text{-}Nodes}_1) \overset{C}{\approx} (\mathrm{DY}_2, \mathit{TH\text{-}Nodes}_2)$$
$$(\mathrm{DY}_2, \mathit{TH\text{-}Nodes}_2) \overset{C}{\approx} (\mathrm{DY}_3, \mathit{TH\text{-}Nodes}_3)$$
$$(\mathrm{DY}_3, \mathit{TH\text{-}Nodes}_3) \overset{C}{\approx} (\mathrm{DY}_4, \mathit{TH\text{-}Nodes}_4)$$
$$(\mathrm{DY}_4, \mathit{TH\text{-}Nodes}_4) \overset{C}{\approx} (\mathrm{DY}_5, \mathit{TH\text{-}Nodes}_5)$$
$$(\mathrm{DY}_5, \mathit{TH\text{-}Nodes}_5) \overset{C}{\approx} (\mathrm{DY}_6, \mathit{TH\text{-}Nodes}_6)$$
$$(\mathrm{DY}_6, \mathit{TH\text{-}Nodes}_6) \overset{C}{\approx} (\mathrm{DY}_7, \mathit{TH\text{-}Nodes}_7)$$
$$(\mathrm{DY}_7, \mathit{TH\text{-}Nodes}_7) \overset{C}{\approx} (\mathrm{DY}_f, \mathit{TH\text{-}Nodes}_f)$$

It is obvious that Dolev-Yao-ness transfers as stated in the lemma by transitivity. But the extraction failures transfer because in the presence of an extraction failure each simulator immediately stops. Thus if the extraction failures would not transfer as stated above, it would be possible to differentiate the node traces by their length.

We will show $(\mathrm{DY}_2, \mathit{TH\text{-}Nodes}_2) \overset{C}{\approx} (\mathrm{DY}_3, \mathit{TH\text{-}Nodes}_3)$ at the end, because we need the intermediate result to prove it.

- $(\mathrm{DY}, \mathit{TH\text{-}Nodes}) \overset{C}{\approx} (\mathrm{DY}_1, \mathit{TH\text{-}Nodes}_1)$

  Transforming Sim to Sim_1 is done by replacing invocations of the ZK algorithms and commitment algorithms by oracle-queries. We can replace $A_{crs}(r_N)$ by a $(\texttt{crs})$-query to $\mathcal{O}^{N}_{ZK,com}$ because N is only used inside this crs (protocol condition 13) and the distributions of the implementation and the oracle are the same. Since $\tau(c)$ in Sim_1 not checks whether $c = A_{crs}(r_N)$ but whether c is the result of some $(\texttt{crs})$-query, the node traces have the same distribution.

  The same holds for the replacement of A_{mkZK} by the $(\texttt{prove}, x, w)$ oracle query to $\mathcal{O}^{N}_{ZK,com}$ and of A_{com} by the $(\texttt{commit}, m)$ oracle query to $\mathcal{O}^{N}_{ZK,com}$. The randomness—the third argument of the ZK proof—only occurs inside this proof and nowhere else (protocol condition 7), so we can replace it by the oracle's randomness as in the crs case.

  By implementation condition 26, it holds that if $(x,w) \notin R^{\mathrm{comp}}_{\mathrm{hon}}$ the implementation, as well as the oracle, output $\perp$. So A_{mkZK} and the $(\texttt{prove}, x, w)$-query return $\perp$ in the same cases. In the case that $(x,w) \in R^{\mathrm{comp}}_{\mathrm{hon}}$ both compute a proof of x using witness w. Thus, it holds $(\mathrm{DY}, \mathit{TH\text{-}Nodes}) \overset{C}{\approx} (\mathrm{DY}_1, \mathit{TH\text{-}Nodes}_1)$.

- $(\mathrm{DY}_1, \mathit{TH\text{-}Nodes}_1) \overset{C}{\approx} (\mathrm{DY}_2, \mathit{TH\text{-}Nodes}_2)$

  In this step we replace $\mathcal{O}_{ZK,com}$ by $\mathcal{O}_{Sim}$ which returns a simulated proof for x if for the input (x, w) it holds that $(x, w) \in R_{\mathrm{hon}}^{\mathrm{comp}}$, and correspondingly a faked commitment upon a `commit`-query.

  If we change both simulators to not extract the proof in case of an extraction failure, then the *TH-Nodes* does not change. The simulator stops after handling extraction failures in any case. By definition of zero-knowledge these two modified cases are indistinguishable (using the fact that the simulator and prover are only invoked if $(x, w) \in R_{\mathrm{hon}}^{\mathrm{comp}}$). Thus $(\mathrm{DY}_1, \mathit{TH\text{-}Nodes}_1)$ and $(\mathrm{DY}_2, \mathit{TH\text{-}Nodes}_2)$ are indistinguishable, too.

- $(\mathrm{DY}_3, \mathit{TH\text{-}Nodes}_3) \overset{C}{\approx} (\mathrm{DY}_4, \mathit{TH\text{-}Nodes}_4)$

  As the randomness can only occur at transformations, letting the oracles draw the randomness yields an indistinguishable execution. Moreover, adding an extra check for the validity of the witness is possible because all proofs have been checked anyway (protocol conditions 17). Hence, the traces are indistinguishable and the Dolev-Yaoness carries over.

- $(\mathrm{DY}_4, \mathit{TH\text{-}Nodes}_4) \overset{C}{\approx} (\mathrm{DY}_5, \mathit{TH\text{-}Nodes}_5)$

  We apply the sound transformation property and obtain the indistinguishability of the traces and hence the preservation of the DYness.

- $(\mathrm{DY}_5, \mathit{TH\text{-}Nodes}_5) \overset{C}{\approx} (\mathrm{DY}_6, \mathit{TH\text{-}Nodes}_6)$

  In this step we replace encryptions, decryptions and key-generation by an encryption-oracle as we did for the zero-knowledge proofs in the step from *Sim* to Sim_1. Because Sim_5 does not compute witnesses of zero-knowledge proofs anymore, nonces of encryptions are only used once (by protocol condition 7). Nonces of keys were already only used once (by protocol condition 1). So replacing the implementation of encryptions, decryptions and the public key does not change the distribution of the node trace or ZK-Breaks (since we adapted τ accordingly, cf. the replacement of A_{crs} in Sim_1). In addition we can define $\beta(\mathit{dk}(N)) := \bot$ because decryption keys are not used as input to β (by protocol condition 8 and the use of an oracle for decrypting).

  We did neither change the bitstrings that are sent to the adversary nor the way they are parsed. So the property of DY did not change either.

- $(\mathrm{DY}_6, \mathit{TH\text{-}Nodes}_6) \overset{C}{\approx} (\mathrm{DY}_7, \mathit{TH\text{-}Nodes}_7)$

  In the step from Sim_6 to Sim_7, the only change that is done is the replacement of the encryption oracle by a fake oracle that always encrypts $0^{|m|}$ instead of m. By construction of τ the protocol execution asks only for decryptions of ciphertexts which were not generated by the encryption oracle (since only β invokes the encryption oracle). So a run of the protocol is a valid adversary for the CCA property where the challenger is the encryption oracle. In order to get indistinguishability the adversary has to be able to use ZK-breaks, DYness and node traces to distinguish both executions. Obviously, it is possible to use DYness and the node traces. For the case of ZK-breaks, we have to require that $R_{\mathrm{adv}}^{\mathrm{sym}}$ is efficiently decidable.

  Thus replacing *ENC* by $\mathcal{O}_{\mathrm{fake}}$ leads to an indistinguishable execution and hence $(\mathrm{DY}_6, \mathit{TH\text{-}Nodes}_6)$ and $(\mathrm{DY}_7, \mathit{TH\text{-}Nodes}_7)$ are computationally indistinguishable.

- $(\mathrm{DY}_7, \textit{TH-Nodes}_7) \overset{C}{\approx} (\mathrm{DY}_f, \textit{TH-Nodes}_f)$

  As in the case for Sim_5 and Sim_6, we have the case that after removing the witnesses in Sim_5 the nonces, used as randomness for signatures, are only used (by protocol condition 7) once for signing a message.

  The same holds for verification and signing keys (by protocol condition 1). Thus we can replace signing and computation of verification/signing keys by invocations of $\mathcal{O}_{sig}$ without changing the distribution of (DY, *TH-Nodes*) (since we adopted τ accordingly, cf. the replacement of A_{crs} in Sim_1). In a run of the protocol β is never applied to $sk(N)$ (by protocol condition 9 and the use of an oracle for signing), so we can define $\beta(sk(N)) := \bot$ without changing the distribution of (DY, *TH-Nodes*).

- $(\mathrm{DY}_2, \textit{TH-Nodes}_2) \overset{C}{\approx} (\mathrm{DY}_3, \textit{TH-Nodes}_3)$

  We have already proven that $(\mathrm{DY}_\mathrm{f}) \overset{C}{\approx} (\mathrm{DY}_3)$. Together with the fact that Sim_f is DY (Lemma 7), it follows that Sim_3 is DY. By Lemma 10, it follows that $(t_2, t_3) \in R_{\mathrm{hon}}^{\mathrm{sym}}$ for all ZK-nodes with arguments $t_1, \ldots, t_6$ in a $\mathbf{D}_{\mathrm{trans}}$-hybrid execution of Sim_3 (with overwhelming probability). Applying Lemma 8 leads to $(\beta(t_2), \beta(t_3)) \in R_{\mathrm{hon}}^{\mathrm{comp}}$ for a $\mathbf{D}_{\mathrm{trans}}$-hybrid execution of Sim_3 with overwhelming probability. The only difference between Sim_2 and Sim_3 is that Sim_2 checks whether $(\beta(t_2), \beta(t_3)) \in R_{\mathrm{hon}}^{\mathrm{comp}}$ or not. Because this check would succeed with overwhelming probability in Sim_3, it actually succeeds in Sim_2.

  Thus the distribution of (DY, *TH-Nodes*) is the same in Sim_2 as in Sim_3.

  □

□

Proof of (iii). Let $\Omega_i := \textit{TH-Nodes}^{S}_{\mathbf{M},\Pi_\mathrm{p},Sim_i}(k)$. Ω and Ω_1 are indistinguishable as only a different randomness is used. For Ω_1 and Ω_2 we can reduce the indistinguishability to the zero-knowledge property. We construct a machine M^D that breaks the zero-knowledge property with non-negligible probability if there is a distinguisher D that distinguishes Ω_1 from Ω_2.

- Draw a random coin $b \leftarrow_R \{0,1\}$.
- Internally, run the hybrid execution with the simulator Sim_{b+1}.
- Forward every prover or simulator call of β to the zero-knowledge challenger.
- At the end of the execution, output the trace of the knowledge set S to the distinguisher D. If D guesses Ω_{b+1}, M^D guesses b; otherwise, M^D makes a random guess $b' \leftarrow_R \{0,1\}$.

Ω_2 and Ω_3 are indistinguishable because in (i) it has been shown that that omitting the additional check does not change the trace, hence also not the knowledge set trace length. Moreover, this additional check does not affect τ, and the result of β because β is only called on true proofs in Ω_2 and Ω_3.

Ω_3 and Ω_4 again are indistinguishable since only the randomness is now freshly chosen for every call. But the only place at which a nonce is reused is in the witness of a proof. Sim_3 and Sim_4, however, simulate all proofs and therefore do not use the witnesses at all.

The indistinguishability of Ω_4 and Ω_5 can be reduced against the IND-CCA property of the encryption scheme. We construct a machine M^D that breaks the IND-CCA property

with non-negligible probability if there is a distinguisher D that distinguishes Ω_4 from Ω_5.

- Draw a random coin $b \leftarrow_R \{0,1\}$.
- Internally, run the hybrid execution with the simulator Sim_{b+4}.
- Forward every encryption call of β to the IND-CCA challenger.
- Forward every decryption call of τ, i.e., excluding the cases in which τ only does a look-up, to the IND-CCA challenger.
- At the end of the execution, output the trace of the knowledge set S to the distinguisher D. If D guesses Ω_{b+4}, M^D guesses b; otherwise, M^D makes a random guess $b' \leftarrow_R \{0,1\}$.

The indistinguishability of Ω_5 and Ω_f again follows from the fact that only proofs reuse randomness in the witness. But all proofs are simulated; therefore, letting the signing oracle choose fresh randomness is indistinguishable from using the protocol nonce, which is also fresh by protocol condition 7. □

No quasi extraction-failure. We first show that no quasi extraction-failures happen, and then show that even extraction-failures do not happen with more than negligible probability. In order to be able to prove that extraction failures do not happen, we have to apply the extractability. However, the extractability property of the zero-knowledge proofs is not applicable to simulated proofs. Therefore, we have to prove extraction failures for Sim_1, which does not simulate any zero-knowledge proofs. However, without applying the zero-knowledge property we cannot show the symbolic zero-knowledge property, but we can still apply the IND-CCA property of the encryption scheme. Hence, we show a weaker property, namely that the only reason why extraction failures can occur is that the symbolic zero-knowledge property is violated.

Formally, we define in Figure 3.5 a *witness-knowledge relation* $\vdash_w$, which extends the knowledge relation $\vdash$ with all messages that already occurred in a zero-knowledge proof as a witness. Then, we define a relaxed symbolic extraction algorithm $\mathbf{SymbExtr'}$, which tries to find a valid symbolic witness w such that $S \vdash_w w$ and outputs $\bot$ if it fails. We say that a *quasi-extraction failure* happens if $\mathbf{SymbExtr'}(S,x) = \bot$.

The proof goes along the lines of the proof of extraction failures in the work of Backes, Bendun, and Unruh [BBU13]. We first show that in Sim_1 ZK-Breaks happen with negligible probability (Lemma 9). Then, we show that τ and τ^* with overwhelming probability output a term t such that $S \vdash_w t$ (Lemma 15). And finally we show that quasi-extraction failures only happen with negligible probability (Lemma 16).

The heart of this proof lies in the proof of the statement that τ and τ^* with overwhelming probability only produce terms t such that $S \vdash_w t$ (Lemma 15). Even though we cannot apply the zero-knowledge property, we can still show that encryptions hide their plaintexts, even if the attacker knows the extraction trapdoor. However, since the zero-knowledge proofs might contain in the witness the randomness with which a ciphertext has been constructed, we cannot naïvely apply the IND-CCA property for showing that encryptions hide their plaintexts. We construct for every protocol nonce n a simulator $Sim_{cf,n}$ that fakes all ciphertexts that observably contain this nonce, and show that this nonce is not guessable in the execution with $Sim_{cf,n}$ (Lemma 13). We also show that this simulator is indistinguishable from Sim_1 as long as no randomness of any faked ciphertext is sent as a witness of a proof (Lemma 14). Then, we show that whenever an underivable term is parsed by Sim_1 there is an underivable subterm t that uses a nonce n (Lemma 12) such

$$\frac{S \vdash m}{S \vdash_w m} \qquad \frac{S \vdash ZK(t_2, t_3, t_4)}{S \vdash_w extrWit(t_3)} \qquad \frac{S \vdash_w \bar{t} \quad \bar{t} \in \mathbf{T} \quad F \in \mathbf{C} \cup \mathbf{D}' \quad eval_F(\bar{t}) \neq \bot}{S \vdash_w eval_F(\bar{t})}$$

Figure 3.5.: Deduction rules for the witness knowledge relation: $\mathbf{D}' := \mathbf{D} \cup \{extract\}$

that t is either in $Sim_{cf,n}$ not guessable, hence also not in Sim_1, or t is already derivable, if the randomness of a faked ciphertext is already leaked (Lemma 15).

Witness-knowledge relation. The cryptographic definition of extractability, does not exclude that the extractor applies also extracts witnesses of ZKPs that have been used as witnesses. Therefore, we need to grant the attacker the oppurtunity to extract witnesses out of ZKPs that are used as a witness in another ZKP. We call this operation *extract*, and it is defined as follows:

$$extract(com(t_1, t_2, t_3)) \quad = \quad t_2$$

Notation. Let S be the set of messages that has been sent to the symbolic attacker. We write $M(n)$ for the following event: A bitstring z is sent in a round ℓ in which the execution is still alive every bitstring z such that $S_\ell \not\vdash_w \tau(z)$ and $\tau(z)$ contains an underivable subterm that uses the nonce n. Definition 12 defines the meaning of an underivable subterms t using a nonces n.

Chosen nonces and encryptions, observably contained terms, being caught, and dead tails. For the simulator $Sim_{cf,n}$, n is the *chosen nonce*. We recursively define *chosen encryptions*. An encryption $enc(t, m, N)$ for which m contains the chosen nonce n as a subterm, is called a *chosen encryption* unless n is in m already contained in a chosen encryptions. In this way, every term has for every occurrence of a chosen nonce at least one chosen encryption. More formally, for a chosen nonce n the term $enc(t, m, N)$ is a *chosen encryption* if there is a context C such that $C[n] = m$ and there is no pair of contexts D, D' such that $D[enc(t, D'[n], N)] = m$ and $enc(t, D'[n], N)$ is already a chosen encryption. Moreover, the randomness symbol N of a chosen encryption $enc(t, m, N)$ is called *dangerous* from that point on.

Loosely speaking, we say that a t term is (witness) *observably contained* in another term m if changing something in t is always observable in $\beta(t)$ for an attacker that only sees $\beta(m)$ and has access to the extraction trapdoor. Because $Sim_{cf,n}$ does not simulate the zero-knowledge proofs and only fakes chosen encryptions, it suffices to consider chosen encryptions. More formally, let $t, m \in \mathbf{T}$ be two terms. We say that t is (witness) *observably contained* in m if there are no two context C, D such that $C[enc(ek(N), D[t], N')] = m$, $S \not\vdash_w N$, $N, N' \in \mathbf{N}$, and $enc(ek(N), D[t], N')$ is a chosen encryption.

We stop the simulation before a treacherous zero-knowledge proof is sent to the attacker. More formally, an invocation of $\beta(t) =: q$ with a zero-knowledge proof $ZK(crs(N), t_1, t_2, N')$[12] is called a *catching call* of β if the witness t_1 observably contains a dangerous randomness symbol r and there is no context C such that $t_1 = C[enc(ek(N), m, r)]$. $Sim_{cf,n}$ halts at the latest point at which the following two conditions are satisfied: (i) a catching call of β

[12] We stress that t might also be a transformed proof.

is about to be performed for the dangerous randomness symbol r and (ii) β is called with a chosen encryption with r. In the original execution we call the part after a catching call the *dead tail* of the execution.

The chosen encryption faking simulator $Sim_{cf,n}$. For a chosen nonce n, we define a faking simulator $Sim_{cf,n}$ that fakes all chosen encryptions but leaves everything else in the simulation untouched except for halting before a catching call of β is computed. We show that the hybrid execution with $Sim_{cf,n}$ is indistinguishable from the hybrid execution with Sim_1 even if the distinguisher knows the extraction trapdoor; for proving this indistinguishability, we introduce the intermediate simulators Sim'_1, Sim'_2, and Sim'_3.

- We define Sim'_2 like Sim_1, but we change β and τ to use encryption oracles for all honestly generated encryptions instead of computing A_{enc}, A_{dec}, A_{ek}, A_{dk}. More precisely, assume an oracle $\mathcal{O}_{enc}$ that internally picks $(ek, dk) \leftarrow KeyGen_{enc}(1^\eta)$ and that responds to three kinds of queries: Upon an $(\mathtt{ek})$-query, it returns ek. Upon a $(\mathtt{enc}, m)$-query, it returns $ENC(ek, m)$. Upon a $(\mathtt{dec}, c)$-query, it returns $DEC(dk, c)$. Sim'_2 maintains an instance $\mathcal{O}^N_{\text{enc}}$ for each $N \in \mathbf{N}_P$. Then Sim'_2 computes $\beta(ek(N))$ with $N \in \mathbf{N}_P$ as $\beta(ek(N)) := \mathcal{O}^N_{\text{enc}}(\mathtt{ek})$. And it computes $\beta(enc(ek(N), t, M))$ with $N, M \in \mathbf{N}_P$ as $\beta(enc(ek(N), t, M)) := \mathcal{O}^N_{\text{enc}}(\mathtt{enc}, \beta(t))$. And it computes $\beta(dk(N)) := \bot$. And in the computation of $\tau(c)$ for c of type ciphertext, the computation of $A_{dec}(A_{dk}(r_N), c)$ is replaced by $\mathcal{O}^N_{\text{enc}}(\mathtt{dec}, c)$.
- We define Sim'_3 like Sim'_2, except that we replace the oracle $\mathcal{O}_{enc}$ by an oracle $\mathcal{O}_{\text{fake}}$. That oracle behaves like $\mathcal{O}_{enc}$, except that upon an $(\mathtt{enc}, x)$-query, it returns $ENC(ek, 0^{|x|})$.
- We define $Sim_{cf,n}$ like Sim'_3, but we change β to use signing oracles instead of computing A_{vk}, A_{sk}, A_{sig}. More precisely, we assume an oracle $\mathcal{O}_{sig}$ that internally picks $(vk, sk) \leftarrow KeyGen_{sig}(1^\eta)$ and that responds to two kinds of queries: Upon a $(\mathtt{vk})$-request, it returns vk, and upon a $(\mathtt{sig}, m)$-request, it returns $SIG(sk, m)$. $Sim_{cf,n}$ maintains an instance $\mathcal{O}^N_{sig}$ for each $N \in \mathbf{N}_P$. Then $Sim_{cf,n}$ computes $\beta(vk(N))$ with $N \in \mathbf{N}_P$ as $\beta(vk(N)) := \mathcal{O}^N_{sig}(\mathtt{vk})$. And $\beta(sk(N))$ with $N \in \mathbf{N}_P$ as $\beta(sk(N)) := \bot$. And $\beta(sig(sk(N), t, M))$ with $N, M \in \mathbf{N}_P$ as $\beta(sig(sk(N), t, M)) := \mathcal{O}^N_{sig}(\mathtt{sig}, \beta(t))$.

We need to further characterize the underivable witnesses, i.e., the subterms that are underivable, given a witness w such that $S \nvdash_w w$.

Lemma 12 (Witness-underivable subterms)**.** *In a given step of the $\boldsymbol{D}_{\text{trans}}$-hybrid execution of Sim_1, let S be the set of messages sent from Π^T to Sim_1 before.*

Let $u' \in \boldsymbol{T}$ be the message sent from Sim_1 to the protocol in that step. Let C be a context and $u \in \boldsymbol{T}$ such that $u' = C[u]$ and $S \nvdash_w u$.

Then there is a term t_{bad} and a context D such that D can be obtained by the following grammer:

$$\begin{aligned} D ::= \Box \mid pair(t, D) \mid pair(D, t) \mid enc(ek(N), D, t) \mid enc(D, t, M) \mid sig(sk(M), D, M') \\ \mid com(D, r, N) \mid com(t, D, M'') \mid ZK(t, t', M) \mid ZK(D, t', M) \mid ZK(t', D, M) \mid \\ garbEnc(D, M) \mid garbSig(D, M) \end{aligned}$$

with $N \in \boldsymbol{N}_P, M, M' \in \boldsymbol{N}_E, M'' \in \boldsymbol{N}_P \cup \boldsymbol{N}_E, t, t' \in \boldsymbol{T}$ and $u = D[t_{bad}]$ such that $S \nvdash_w t_{bad}$ and such that one of the following holds:

1. $t_{bad} \in \boldsymbol{N}_P$
2. $t_{bad} = enc(p, m, N)$ *with* $N \in \boldsymbol{N}_P$
3. $t_{bad} = sig(k, m, N)$ *with* $N \in \boldsymbol{N}_P$
4. $t_{bad} = ZK(x, r, N)$ *with* $N \in \boldsymbol{N}_P$ *and* $crsof(ZK(x, r, N)) = crs(M)$ *and* $M \in \boldsymbol{N}_P$
5. $t_{bad} = sig(sk(N), m, M)$ *with* $N \in \boldsymbol{N}_P$, $M \in \boldsymbol{N}_E$
6. $t_{bad} = crs(N)$ *with* $N \in \boldsymbol{N}_P$
7. $t_{bad} = ek(N)$ *with* $N \in \boldsymbol{N}_P$
8. $t_{bad} = vk(N)$ *with* $N \in \boldsymbol{N}_P$
9. $t_{bad} = sk(N)$ *with* $N \in \boldsymbol{N}_P$
10. $t_{bad} = dk(N)$ *with* $N \in \boldsymbol{N}_P$

We say that an underivable subterm t_{bad} uses the nonce N.

Proof. We prove the lemma by structural induction on u. Recall that only upon a message $m \in \{0,1\}^*$ from the attacker Sim_1 sends a message $\tau(m)$. There are the following cases:

Case 1: "$u \in \{ek(N), vk(N), crs(N), dk(N), sk(N)\}$ with $N \notin \mathbf{N}_P$"

Let $u = C(N)$ for $C \in \{ek, vk, crs, dk, sk\}$. By protocol conditions 1 and 13 each C-node has as annotation a nonce from $\mathbf{N}_P$. Therefore u cannot be honestly generated, that means there is some $e \in \{0,1\}^*$ such that $\tau(e) = u$ and u has the form $C(N^e)$. But then $S \vdash_w u$ contradicting the premise of the lemma.

Case 2: "$u \in \{ek(N), vk(N), crs(N), dk(N), sk(N)\}$ with $N \in \mathbf{N}_P$"

Then the claim is fulfilled with $D := \Box$ and $t_{bad} = u$.

Case 3: "$u = garb(u_1)$"

By protocol condition 2 no garbage term is generated by the protocol. Therefore there is a $c \in \{0,1\}^*$ such that $\tau(c) = garb(N^c) = u$. But this means that $S \vdash_w u$, contradicting the premise of the lemma.

Case 4: "$u = garbEnc(u_1, u_2)$ or $u = garbSig(u_1, u_2)$"

By protocol condition 2 no garbage term is generated by the protocol. So there exists a $c \in \{0,1\}^*$ with $\tau(c) = garbEnc(u_1, N^c)$ or $\tau(c) = garbSig(u_1, N^c)$. Since $S \vdash_w N^c$ it follows that $S \nvdash_w u_1$, because $S \nvdash_w u$. Applying the induction hypothesis on u_1 leads to a context D' and a term t_{bad}. Using this term t_{bad} and the context $garbEnc(D', N^c)$, respectively $garbSig(D', N^c)$, shows the claim.

Case 5: "$u = com(u_1, u_2, N)$"

By protocol condition 13, we know that there is a $c \in \{0,1\}^*$ such that $\tau(c) = com(crs(N), t, N^c)$. Hence, the lemma is satisfied by $t_{bad} = com(crs(N), t, N^c)$ and $D = \Box$.

By applying $extract(com(u_1, u_2, M)) = u_2$, we know that by induction hypothesis there is a D such that $S \vdash_w D[t_{bad}]$. Consequently, the lemma is satisfied by $D' = com(u_1, D, M)$.

Case 6: "$u = pair(u_1, u_2)$"

Since $S \nvdash_w u$ there is an $i \in \{1, 2\}$ such that $S \nvdash_w u_i$. Let D be the context and t_{bad} the term given by applying the induction hypothesis to u_i. Then $D_1 := pair(D, M)$ or $D_2 := pair(M, D)$ is the context for the term u depending on i with the same term t_{bad}.

Case 7: "$u = \varepsilon$"

This case cannot happen because $S \vdash_w \varepsilon$, so the premise of the lemma is not fulfilled.

Case 8: "$u = string_0(u_1)$ or $u = string_1(u_1)$"

Again the premise is not fulfilled since inductively $S \vdash_w u_1$ with base case $u_1 = \varepsilon$ and therefore $S \vdash_w string_i(u_1)$ for $i \in \{0, 1\}$.

Case 9: "$u = N$ with $N \in \mathbf{N}_P \backslash \mathcal{N}$"

This case is impossible since u is not in the range of τ.

Case 10: "$u = N$ with $N \in \mathcal{N}$"

The context $D := \Box$ and term $t_{bad} := u$ satisfy the lemma in this case.

Case 11: "$u = N$ with $N \in \mathbf{N}_E$"

In this case $S \vdash_w u$ by definition and therefore the lemma's premise does not hold.

Case 12: "$u = enc(u_1, u_2, N)$ with $N \in \mathbf{N}_P$"

The lemma is satisfied by $t_{bad} = u$ and $D = \Box$.

Case 13: "$u = enc(u_1, u_2, u_3)$ with $u_3 \notin \mathbf{N}_P$ and $S \not\vdash u_1$"

Since $u_3 \notin \mathbf{N}_P$ it follows that u cannot be honestly generated because of protocol condition 7. Therefore there is a $c \in \{0, 1\}^*$ with $\tau(c) = enc(ek(M), u_2, N^c) = u$ for some $M \in \mathbf{N}_P$. Apply the induction hypothesis to u_1 getting t_{bad} and context D we can define $D' := enc(D, u_2, N^c)$ fulfilling the claim of the lemma with t_{bad}.

Case 14: "$u = enc(u_1, u_2, u_3)$ with $u_3 \notin \mathbf{N}_P$ and $S \vdash u_1$"

Since $u_3 \notin \mathbf{N}_P$ it follows that u cannot be honestly generated because of protocol condition 7. Therefore there is an $c \in \{0, 1\}^*$ with $\tau(c) = enc(ek(M), u_2, N^c) = u$ for some $M \in \mathbf{N}_P$. Since $S \vdash_w u_1$, $S \vdash_w N^c$, and $S \not\vdash_w u$, it follows that $S \not\vdash_w u_2$. Let D be the context and t_{bad} be the term resulting by the induction hypothesis applied to u_2. Then $D' := enc(ek(M), D, N^c)$ together with t_{bad} satisfies the lemma.

Case 15: "$u = sig(u_1, u_2, N)$ with $N \in \mathbf{N}_P$"

Use context $D := \Box$ and $t_{bad} = u$.

Case 16: "$u = sig(sk(N), u_1, u_3)$ with $u_3 \notin \mathbf{N}_P$ and $N \in \mathbf{N}_P$"

Since $u \in \mathbf{T}$ and $u_3 \notin \mathbf{N}_P$ follows that $u_3 \in \mathbf{N}_E$. Therefore the context $D := \Box$ and $t_{bad} = u$ proves the claim.

Case 17: "$u = sig(u_1, u_2, u_3)$ and $u_3 \notin \mathbf{N}_P$ and u_1 is not of the form $sk(N)$ with $N \in \mathbf{N}_P$"

Since $u_3 \notin \mathbf{N}_P$ we get by protocol condition 7 that u is not honestly generated, i.e., there is an $s \in \{0, 1\}^*$ such that $\tau(s) = sig(sk(M), u_2, N^s) = u$ with $M \in \mathbf{N}$. Because u_1 has not the form $sk(N)$ for any $N \in \mathbf{N}_P$ follows that $M \in \mathbf{N}_E$, so $S \vdash_w M$ and therefore $S \vdash sk(M)$. In total we have $S \vdash_w u_1$, $S \vdash_w u_3$ but $S \not\vdash_w u$ which implies that $S \not\vdash_w u_2$. Applying the induction hypothesis to u_2 leads to a context D and a term t_{bad}. Defining $D' := sig(sk(M), D, N^s)$ completes the claim.

Case 18: "$u = ZK(u_1, u_2, N)$ with $N \in \mathbf{N}_P$"

Defining $t_{bad} = u$ and $D := \Box$ suffices.

Case 19: "$u = ZK(u_1, u_2, N)$ with $N \in \mathbf{N}_E$"

Since τ uses $randomnessTree(u_1)$ for u_2 and $N \in \mathbf{N}_E$, we know that $S \not\vdash_w u_1$.

In this case we use the induction hypothesis on $getPub(u) = u_1$ to get the term t_{bad} and a context D. Then using t_{bad} and $D' := ZK(D, u_2, N)$ satisfies the lemma.

Case 20: "$u = ZK(u_1, u_2, N)$ with $N \in \mathbf{N}_E$"

This case cannot occur because u is not in the range of τ.

Case 21: "$u = \hat{f}(t_1, \ldots, t_n, x, w, N)$"

By the definition of τ, we know that $\tau(m) = u$ only if m was already sent by the protocol; hence already $S \vdash_w u$ holds true and this case cannot occur.

□

We show that for any MZK-safe protocol Π and any ppt attacker A the nonce n is not guessable in the execution with the n-faking simulator $Sim_{cf,n}$.

Lemma 13. *Let Π be an arbitrary MZK-safe protocol and A be an arbitrary ppt machine. In the $\boldsymbol{D}_{\text{trans}}$-hybrid execution with $Sim_{cf,n}$, Π, A, the probability that $M(n)$ happens is negligible in the security parameter.*

Proof. Assume that the adversary sends with non-negligible a bitstring z in a round ℓ in which the execution is still alive, and $S_\ell \not\vdash_w \tau(z)$ and $\tau(z)$ contains a chosen underivable subterm t that uses the chosen nonce n. Therefore, only two cases can occur. First, t has not been sent so far, and second t has only been sent along with an encryption. In the second case, however, we know by the construction of $Sim_{cf,n}$ that the encryption has been faked; in particular, t has never been constructed, i.e., β has never been applied to t. We show that for every possible underivable subterm that uses n the assumption leads to a contradiction.

Case 1: $t_{bad} \in \{n, enc(p, m, n), sig(k, m, n), ZK(x, r, n)\}$
$crs(n), ek(n), vk(n)$ with $M, n \in \mathbf{N}_P$.
If β has never been applied on t_{bad}, we know by the definition of τ that $n \notin \mathbf{N}_P$.

Case 2: $t_{bad} = sig(sk(n), m, M)$ with $n \in \mathbf{N}_P$, $M \in \mathbf{N}_E$.
The same argumentation as for Case 1 holds for $t_{bad} = sig(sk(n), m, M)$ and $n \in \mathbf{N}_P$, $M \in \mathbf{N}_E$ with the only difference that the reason why t_{bad}, even though unused, is unguessable is that otherwise the attacker would induce a ppt machine that breaks the unforgeability property of the signature scheme.

Case 3: $t_{bad} = sk(n)$ with $n \in \mathbf{N}_P$.
The same argumentation as for Case 1 holds for $t_{bad} = sk(n)$ and $n \in \mathbf{N}_P$; however, we additionally have to consider the case that the attacker is able to compute the signature key from the verification key. In this case, however, we can use the attacker to break the unforgeability property, which is a contradiction.

Case 4: $t_{bad} = dk(n)$ with $n \in \mathbf{N}_P$.
The same argumentation as for Case 1 holds for $t_{bad} = dk(n)$ and $n \in \mathbf{N}_P$; however, we additionally have to consider the case that the attacker is able to compute the decryption key from the encryption key. In this case, however, we can use the attacker to break the IND-CCA property, which is a contradiction.

□

In the following proofs, we often consider the $\mathbf{D}_{\text{trans}}$-hybrid execution of a protocol Π^C with a simulator $Sim'_{i,n}$. We extend the $\mathbf{D}_{\text{trans}}$-hybrid execution by letting *H-Nodes'* output the extraction trapdoors of all CRS and stops before a treacherous zero-knowledge proof is sent. This scenario defines a probability space on traces, which we denote as

$$\Omega_{i,n} := (t, extd) \leftarrow \mathit{TH\text{-}Nodes}'_{Sim'_{i,n},\Pi^C}$$

where *extd* are the extraction trapdoors of all protocol CRS, $Sim'_{1,n} := Sim_1$, with the chosen nonce n, and $Sim'_{cf,n} := Sim_{cf,n}$. Moreover, we write S_ℓ for the set of messages that after the ℓth round has been sent to $Sim'_{i,n}$.

Lemma 14. *Recall that p is the polynomial of the attacker. Then, the following difference is negligible in the security parameter* $\left|\ \Pr[\Omega_{cf,n} : M(n)] - \Pr[\Omega_{1,n} : M(n)]\ \right|$

Proof. Let $\ell \in \mathbb{N}$ be arbitrary but fixed. By protocol condition 7 nonces that are used in $\Omega_{1,n}$ as randomness for encryptions calls can only be further used in the witness of a zero-knowledge proof. However, for chosen encryptions, we abort before the attacker learns both, the faked ciphertext and the proof about the faked ciphertext. Since only chosen encryptions are faked $\Omega_{1,n}$ is indistinguishable from $\Omega_{2,n}$ even for distinguishers that know the extraction trapdoors *extd*.

By the IND-CCA property of the encryption scheme (implementation condition 8) $\Omega_{2,n}$ is indistinguishable from $\Omega_{3,n}$ even for distinguishers that know the extraction trapdoors *extd*.

By protocol condition 7 nonces that are used in $\Omega_{3,n}$ as randomness for signing calls can not be used anywhere else, in particular not as a witness in a zero-knowledge proof. Therefore, $\Omega_{3,n}$ is indistinguishable from $\Omega_{cf,n}$ even for distinguishers that know the extraction trapdoors *extd*.

Since $M(n)$ is efficiently computable for a machine that has access to an extraction oracle, the probability that $M(n)$ holds can only differ by a negligible amount in $\Omega_{1,n}$ and $\Omega_{cf,n}$. □

Finally, we are able to show that all invocations of τ and τ^* in Sim_1 adhere the witness knowldge relation.

Lemma 15 (Sim_1 adheres $\vdash_w$)**.** *For any invocation $\tau(m)$ of τ or $\tau^*(m)$ of τ^* in the $\mathbf{D}_{\text{trans}}$-hybrid execution of Sim_1, the following holds with overwhelming probability: Let S be the set of terms t that the protocol sent to the adversary up to the invocation $\tau(m)$ or $\tau^*(m)$. Then $S \vdash_w \tau(m)$ or $S \vdash_w \tau^*(m)$, respectively.*

Proof. Assume that with non-negligible probability the attacker sends a bitstring z in a round ℓ such that $S_\ell \nvdash_w \tau(z)$. Applying Lemma 12, we know that $\tau(z)$ has the following underivable subterms:

Case 1: $t_{bad} \in \{N, enc(p, m, N), sig(k, m, N),$
$ZK(x, r, N), crs(N), ek(N), vk(N)\}$ with $M, N \in \mathbf{N}_P$.
If t_{bad} has not been sent to the attacker so far, $\beta(t_{bad})$ has not been called so far.
By definition of τ (resp., τ^*), however, we then have $N, M \notin \mathbf{N}_P$.

If t_{bad} is contained in a term that has already been sent to the attacker, t_{bad} can only have occurred below an encryption. Assuming that N is the chosen nonce, we are either in the living part of the execution or in the dead tail. If we are in the dead tail, because of the protocol condition 7 the attacker already witness-knows the plaintext of a chosen encryption. By the definition of chosen encryptions, we know that the attacker then also witness-knows t_{bad}, which is a contradiction.

If z has been sent in the living part of the execution, we know by Lemma 13 that in the execution with $Sim_{cf,N}$ the attacker cannot send a bitstring z that such that $\tau(z)$ contains an underivable subterm t_{bad}, which uses this chosen nonce N, with more than negligible probability. By Lemma 14, we know that the same property holds for Sim_1. This property, however, is a contradiction to the assumption that z has been sent with non-negligible probability.

Case 2: $t_{bad} = sig(sk(N), m, M)$ with $N \in \mathbf{N}_P$, $M \in \mathbf{N}_E$.
The same argumentation as for Case 1 holds for $t_{bad} = sig(sk(N), m, M)$ and $N \in \mathbf{N}_P$, $M \in \mathbf{N}_E$ with the only difference that the reason why t_{bad}, even though unused, is unguessable is that otherwise the attacker would induce a ppt machine that breaks the unforgeability property of the signature scheme.

Case 3: $t_{bad} = sk(N)$ with $N \in \mathbf{N}_P$.
The same argumentation as for Case 1 holds for $t_{bad} = sk(N)$ and $N \in \mathbf{N}_P$; however, we additionally have to consider the case that the attacker is able to compute the signature key from the verification key. In this case, however, we can use the attacker to break the unforgeability property, which is a contradiction.

Case 4: $t_{bad} = dk(N)$ with $N \in \mathbf{N}_P$.
The same argumentation as for Case 1 holds for $t_{bad} = dk(N)$ and $N \in \mathbf{N}_P$; however, we additionally have to consider the case that the attacker is able to compute the decryption key from the encryption key. In this case, however, we can use the attacker to break the IND-CCA property, which is a contradiction.

□

In order to be able to prove that ZK-failures cannot happen with more than negligible probability, we first need to show that the relaxed symbolic extraction **SymbExtr′** always proceeds. The relaxed symbolic extraction only checks whether the resulting candidate for a witness w is witness-derivable, i.e., $S \vdash_w w$.

Lemma 16 (No quasi-extraction failures)**.** *In a $\boldsymbol{D}_{\text{trans}}$-hybrid execution with the simulator Sim_1 the following holds: A quasi-extraction failure only happens with negligible probability.*

Proof. Assume a quasi-extraction failure occurs with non-negligible probability. Then $\tau(z)$ is called for a bitstring z of type zero-knowledge proof such that the symbolic extraction fails. Therefore, z was not generated by the protocol, i.e. it was not output of the simulation oracle, and the corresponding *crs* was generated by the protocol (otherwise τ would not invoke the symbolic extraction). Let $N \in \mathbf{N}_P$ be defined by $crs(N) = \tau(A_{crs}(z))$. Let $m_x := A_{getPub}(z)$, $x := \tau(m_x)$, $m_w := E(m_x, z, extd_N)$[13]. Let S denote the set that the protocol already sent to the simulator in this execution.

[13] Here, $extd_N$ is the extraction trapdoor that the simulator receives from the oracle $\mathcal{O}^N_{ZK,com}$ by querying $(extd)$.

We have $\mathbf{SymbExtr}'(S,x) = \bot$ by definition of quasi-extraction failures. Thus one of the following cases occurs with non-negligible probability.

1. $(x,w) \notin R_{\text{adv}}^{\text{sym}}$
2. $S \not\vdash_w w$
3. $(x,w) \in R_{\text{adv}}^{\text{sym}}$ and $S \vdash_w w$ but $\mathbf{SymbExtr}'(S,x) = \bot$

We prove for each case that it only occurs with negligible probability, which leads to a contradiction to the assumption that quasi-extraction failures occur with non-negligible probability.

Case 1: $(x,w) \notin R_{\text{adv}}^{\text{sym}}$.
This constitutes a ZK-Break, which is a contradiction to Lemma 9.

Case 2: $S \not\vdash_w w$.
By Lemma 15 this case only happens with negligible probability.

Case 3: $(x,w) \in R_{\text{adv}}^{\text{sym}}$ and $S \vdash_w w$ but $\mathbf{SymbExtr}'(S,x) = \bot$.
By definition of $\mathbf{SymbExtr}'$ the symbol $\bot$ is only returned if there is no w such that $(x,w) \in R_{\text{adv}}^{\text{comp}}$ and $S \vdash_w w$. So this case cannot occur.

□

No extraction-failure. In order to conclude that an extraction-failure does not happen in *Sim*, we first show that in Sim_f no extraction-failure happens. And then, we apply the indistinguishability of the knowledge traces (Lemma 11) of *Sim* and Sim_f.

Lemma 17 (No extraction failures - FMZK model)**.** *For the FMZK model in a hybrid execution of* Sim_f *holds: An extraction failure can only occur with negligible probability.*

Proof. Assume an extraction failure occurs with non-negligible probability. Then, a bitstring z has been sent by the attacker such that $x = \tau(A_{getPub}(z))$ and $\mathbf{SymbExtr}(S,x) = \bot$ but $\mathbf{V}(A_{getPub}(z), z, crs) = 1$. By Lemma 16 and Lemma 11, we know that in Sim_f quasi-extraction failures only happen with negligible probability. Consequently, there is a $w \in \mathbf{SymbExtr}'(x)$ such that $\bot \neq w$, and by definition of $\mathbf{SymbExtr}'$ also $S \vdash_w w$ holds true. We observe that $S \vdash_w w$ if and only if there is a transformation $C[\bullet_1, \ldots, \bullet_n]$ such that for some $t_1, \ldots, t_n \in S$ we have $C[t_1, \ldots, t_n] = w$, where C is a context using constructors and destructors from $\mathbf{C} \cup \mathbf{D} \cup \{extract\}$.

Let D be a context such that $D[ZK(t_1, r_1, N_1), \ldots, ZK(t_n, r_n, N_n)] = ZK(t, r, N^z) = \tau(z)$ and $ZK(t_1, r_1, N_1), \ldots, ZK(t_n, r_n, N_n) \in S$. We show that D is with overwhelming probability zero-knowledge preserving.

Next, we show the following: for every proof z such that $\mathbf{V}(A_{getPub}(z), z, crs) = 1$ that the attacker sends, there is a $w \in \mathbf{SymbExtr}'(S, \tau(A_{getPub}(z)))$ such that $w \neq \bot$, a ZK-transformation f_{f_1,f_2} over $\mathbf{D}, \mathbf{D} \cup \{extract\}$, and there are $ZK(t_2^{(i)}, t_3^{(i)}, t_4^{(i)}) \in S$, $i \in [n]$, such that $\tau(A_{getPub}(z)) = f_1(t_2^{(1)}, \ldots, t_2^{(n)})$ and $w = f_2(t_3^{(1)}, \ldots, t_3^{(n)})$.

By Lemma 11, we know that the attacker can also in the hybrid execution with *Sim* construct such a message with non-negligible probability. The soundness property of the zero-knowledge proof system implies that $(x,w) \in R_{\text{adv}}^{\text{comp}}$ holds true with overwhelming probability. Lemma 8 in turn implies that $(x,w) \in R_{\text{adv}}^{\text{sym}}$. Therefore, there are witnesses $w_1, \ldots, w_n$ such that the transformation that the attacker applied produces a valid proof for (x,w). We further know that for the statement $x_1, \ldots, x_n$ there is at least one tuple

of witnesses $w'_1, \ldots, w'_n$ such that the transformation of the attacker does not produce a valid witness $(x, w') \notin R^{\text{sym}}_{\text{adv}}$.[14] But then, we can construct a machine M that emulates the entire execution (with *Sim*) twice and in one case uses $(w_1, \ldots, w_n)$ for the proofs and in the other case uses $(w'_1, \ldots, w'_n)$. M outputs 1 if the transformed proof is true and 0 if the transformed proof is wrong. If the probability of the attacker to produce such an above-described transformation is non-negligible, the probability of M distinguishing honest proofs from simulated proofs is non-negligible as well. This, however, contradicts the zero-knowledge property.

Next, we show that every such context D that is zero-knowledge preserving does not use *extract*. We show by induction on n that for all contexts D of depth n there is a context $\tilde{D}$ such that $\mathit{eval}\ D[M] = \mathit{eval}\ \tilde{D}[M]$ for all $M \in T$. The lemma directly following from this statement.

For $n = 0$ the context cannot contain $\tilde{\mathit{extract}}$. For $n > 0$ assume that the statement holds for all $n' < n$. Then, we either have $D[\bullet] \neq \mathit{extract}[\tilde{D}_1[\bullet]]$ or $D[\bullet] = \mathit{extract}[\tilde{D}_1[\bullet]]$. In the first case, the statement for n directly follows from the induction hypothesis. In the second case, observe that the statement cannot imply that the witness is a commitment, hence $D[\mathit{com}(c, t, r)] \neq \bot$ (for any c, t, r) and $D[t'] = \bot$ for any t' that is not a commitment. Hence, D is not zero-knowledge preserving. □

Lemma 18 (No extraction failures - CMZK model)**.** *In the CMZK model in a hybrid execution of Sim_f holds: An extraction failure can only occur with negligible probability.*

Proof. By Lemma 17, we know that an extraction failure up to malleability can only occur with negligible probability. So, the only events that we did not exclude yet are the successful verification of a proof $\hat{z}'$ such that $\tau(\hat{z}') = \mathit{setPub}(\mathit{ZK}(t, r), t\,\{c'/c\})$, where $\mathit{ZK}(t, r) = \tau(\hat{z})$ and $c' = \mathit{com}(\mathit{com}(crs, f(m), r')) = \mathit{applyF}(c, x)$ (for some $r' \in \mathbf{N}$) and $c = \mathit{com}(crs, m, r)$. We call this event Q.

Let vk be the verification key from the CRS. Recall that the statement is

$$(x, w) \in R \vee (\hat{\mathit{ver}}_{sig}(\tau(vk), sig, x') = t \wedge x = T_x(x') \wedge T_x \in \mathcal{T}).$$

Since the proof is simulated by Sim_f and by the extractability property, we know that with overwhelming probability the extracted signature s successfully verifies with vk. However, this is a contradiction to $\mathit{applyF} \notin \mathcal{T}$and to the the universal unforgeability of the signature scheme. Hence, Q could not have happened with non-negligible probability.

This argument can directly be generalized for any context that visibly applies *applyF*. □

The indistinguishability of *Sim*. Finally, applying Lemma 17 and Lemma 8, we are able to conclude that *Sim* with the $\mathbf{D}_{\text{trans}}$-transparent hybrid execution is indistinguishable from the real computational execution.

We show that given an attacker $\mathcal{A}$, a protocol Π_{p}, a polynomial p the traces of the transparent hybrid execution with M and *Sim* are indistinguishable from the traces of the computational execution *Nodes* with the implementation A and the attacker $\mathcal{A}$.

[14] If there would not be such a witness, then there would be a transformation that behaves on all zero-knowledge proofs with the statements $x_1, \ldots, x_n$ as the transformation of the attacker and on all other statements simply outputs a trivial true proof.

Lemma 19. *Sim is indistinguishable for* $\boldsymbol{M}, \Pi, A, E$ *and for every polynomial* p.

Proof. With overwhelming probability it's the case that the messages r_N, $enc(\ldots, N)$, $sig(\ldots, N)$, $ZK(\ldots, N)$ are different for all $N \in \mathbf{N}_P$. Furthermore, by Lemma 17, we know that no extraction-failures happen with *Sim* with overwhelming probability; hence, *Sim* does not abort. So we can restrict to that case for indistinguishability.

For proving indistinguishability of *Sim* for $\mathbf{M}, \Pi, A, E$ we need to show the following claims.

Claim 1: In the $\hat{F}$-transparent hybrid execution of *Sim* holds $\forall m \in \{0,1\}^* : \beta(\tau(m)) = m$.

Claim 2: In the $\hat{F}$-transparent hybrid execution, for any term t stored at a node ν holds, $\beta(t) \neq \bot$.

Claim 3: For all terms t that occur in the $\hat{F}$-transparent hybrid execution holds $\tau(\beta(t)) = t$.

Claim 4: In the $\hat{F}$-transparent hybrid execution, at any computation node $\nu = \nu_i$ with constructor or destructor F and arguments $\tilde{\nu_1}, \ldots, \tilde{\nu_n}$. Let $t_j = f_i^C(\tilde{\nu_j})$. Then holds: $\beta(eval_F(\underline{t})) = A_F(\beta(t_1), \ldots, \beta(t_n))$.

In the proof we will only use the first and the last claim. Claim 1 and Claim 3 follow from the definition of τ and β since τ first checks whether a bitstring is the result of a previous β call on some term t. If so, τ outputs this term t. The argumentation can be carried out via an induction over the sum of the depths of the recursion calls to β and τ and by performing an extensive case distinction over the definition of β and τ.

Claim 2 can be shown via an induction over the recursion depth of $\beta(t)$. In the base case, t can only be a nonce. Since A_t implements by implementation condition 3 a uniform distribution on $\{0,1\}^k$ we have $\Pr[m \leftarrow A_t : m = \bot] = 0$. In the induction step, first observe that $\underline{t}$ cannot contain $\bot$ by the definition of a $\hat{F}$-transparent hybrid execution. Consequently, either $t = f(\underline{t})$ for a constructor $f \in \mathbf{C}$ or $t = \hat{f}(\underline{t})$ for a protocol ZK-transformation $f \in \hat{F}$. We stress that f cannot be an attacker ZK-transformation, i.e., $f \in \mathbf{D}_{\text{trans}} \setminus \hat{F}$, since transformed proofs are parsed as attacker-generated ZK proofs.

If $t = f(\underline{t})$ and f is a constructor, we know by the induction hypothesis that all possible $\beta(\underline{t})$ calls do not return $\bot$. By the implementation conditions and the protocol conditions, we therefore conclude that in this case the probability that $\beta(f(\underline{t})) = \bot]$ is negligible.[15] If $t = \hat{f}(\underline{t})$ and $f \in \hat{F}$ is a protocol ZK-transformation, then we know by the strong derivation privacy [CKLM12] and the correctness and zero-knowledge property of the implementation of ZK proofs (Definition 33) that with overwhelming probability $A_{\hat{f}}$ does not fail if applied to verifiable proofs. Furthermore, we know by protocol conditions 17 that ZK-transformations are only applied to verifiable proofs.

The last Claim 4 follows from the definition of β since for all calls $\beta(F(\underline{t}))$ in which the arguments are the result of a node $\beta(F(\underline{t}))$ is defined as $A_F(\underline{\beta(t_1)})$.[16] The interesting case is $F = ver_{zk}$. At that point we need to show that for an honestly generated ZK proof with a statement x and the witness w we have that $(x, w) \in R_{\text{hon}}^{\text{sym}}$ if and only if its implementation is valid, i.e., $(\beta(x), \beta(w)) \in R_{\text{hon}}^{\text{comp}}$. By the Dolev-Yaoness of *Sim*, we know

[15] This can be shown by an extensive case distinction on f.

[16] Again, this claim can be shown by an induction over the recursion depth of the β call and by an extensive case distinction.

that $(x, w) \in R_{\text{hon}}^{\text{sym}}$ with overwhelming probability. By Lemma 8, we know that then also $(\beta(x), \beta(w)) \in R_{\text{hon}}^{\text{comp}}$.

Fix the randomness of the protocols nondeterministic nodes, the nonces, and the adversaries random tape.

Let ν_i, f_i be the nodes and functions as in the computational trace and ν_i^C, f_i^C be the ones in the hybrid trace. Let S_i be the state of the adversary before execution of the $i-th$ node and S_i^C the corresponding state of the adversary in the hybrid execution. We show by induction in i that $\nu_i = \nu_i^C$, $f_i = \beta \circ f_i^C$, and $S_i = S_i^C$.

Base case $i = 0$. The adversary E is in its starting configuration,i.e. $s_0 = s_0^C$, the node mapping function f is totally undefined, $f_0 = f_0^C =$ and the current node is the root of the protocol, $\nu_0 = \nu_0^C$.

Induction hypothesis: For all $j \leq i$ holds $\nu_j = \nu_j^C$, $f_j = \beta \circ f_j^C$ and $s_j = s_j^C$.

Induction step: $i \to i + 1$

Distinguish by the type of the nodes.

1. If $\nu_i = \nu_i^C$ is a computation node annotated with constructor or destructor.

   Let F/n be the annotated constructor or destructor and $\tilde{\nu_1}, \ldots, \tilde{\nu_n}$ be the annotations of ν_i and $\tilde{\nu_1^C}, \ldots, \tilde{\nu_n^C}$ the of ν_i^C. By induction hypothesis follows that $\tilde{\nu_k} = \tilde{\nu_k^C}$ for $1 \leq k \leq n$. So let $\tilde{m_i} = f_i(\tilde{\nu_i})$, denote the vector $(\tilde{m_1}, \ldots, \tilde{m_n})$ by $\underline{\tilde{m}}$. Analogue are $\tilde{t_i}$, $\underline{\tilde{t}}$ defined for $\tilde{\nu_i^C}$ and f_i^C.

   In the hybrid execution we compute $eval_F(\underline{\tilde{t}})$. By Claim 4 holds $\beta(eval_F(\underline{\tilde{t}})) = A_F(\beta(t_1), \ldots, \beta(t_n))$. Using the definition of t_j we get $\beta(t_j) = (\beta \circ f_i^C)(\nu_j^C)$. Now we can apply the induction hypothesis and replace $(\beta \circ f_i^C)$ by f_i and ν_j^C by ν_j, i.e. $(\beta \circ f_i^C)(\nu_j^C) = f_i(\nu_j)$. All together leads to the equality $\beta(eval_F(\underline{\tilde{t}})) = A_F(\underline{\tilde{m}})$. The right hand side is the outcome of the computational execution of the node ν_i. So the left side is defined if and only if the right side is, and we get ν_{i+1}, ν_{i+1}^C is the yes-successor of ν_i if the term is defined and the no-successor otherwise, i.e. $\nu_{i+1} = \nu_{i+1}^C$. Since $f_{i+1}(\nu_i) = A_F(\underline{\tilde{m}})$ and $f_{i+1}^C(\nu_i^C) = eval_F(\underline{\tilde{t}})$ we have seen that $f_{i+1} = (\beta \circ f_{i+1}^C)$ still holds. In both executions the adversary does not learn anything new, i.e. $S_{i+1} = S_i = S_i^C = S_{i+1}^C$.

2. If $\nu_i = \nu_i^C$ is a computation node annotated with nonce.

   Let N be the annotated nonce. Since $N \in \mathbf{T}$ we get $eval_N() = N \neq \bot$, so ν_{i+1}^C is the yes-successor of ν_i^C. In the computational case always the yes-successor is taken, therefore $\nu_{i+1} = \nu_{i+1}^C$. The adversary is not activated, so $S_{i+1} = S_i = S_i^C = S_{i+1}^C$. For the functions f_{i+1} and f_{i+1}^C we get $f_{i+1}(\nu_i) = r_N = \beta(N) = \beta(f_{i+1}^C(\nu_i^C))$. So the claim is true for $i + 1$.

3. If $\nu_i = \nu_i^C$ is a input node.

   Input nodes have a unique successor, so $\nu_{i+1} = \nu_{i+1}^C$ is this successor. In the computational as in the hybrid case the adversary is asked for input. Since $S_i = S_i^C$, i.e. in both cases the adversary has the same state, both computations are equal, so the resulting state S_{i+1} and S_{i+1}^C, too, and the adversary responds with m in the computational and m^C in the symbolic case, such that $m = m^C$. In the hybrid case the simulator forwards $t^C := \tau(m^C)$ to the protocol execution. $f_{i+1}(\nu_i) := m$ and $f_{i+1}(\nu) := f_i(\nu)$ for $\nu \neq \nu_i$ by definition in the computational case and $f_{i+1}^C(\nu_i) := t^C$

and $f^C_{i+1}(\nu) = f^C_i(\nu)$ for $\nu \neq \nu_i$ in the hybrid case. So by induction hypothesis $f_{i+1}(\nu) = (\beta \circ f^C_{i+1})(\nu)$ for $\nu \neq \nu_i$ and $(\beta \circ f^C_{i+1})(\nu_i) = \beta(t^C) = \beta(\tau(m^C)) \overset{(*)}{=} m^C = m = f_{i+1}(\nu_i)$, where $(*)$ holds because of claim 1. Therefore the invariant $f_{i+1} = (\beta \circ f^C_{i+1})$ holds, too.

4. If $\nu_i = \nu^C_i$ is a output node.

   An output node has a unique successor, so $\nu_{i+1} = \nu^C_{i+1}$ is the unique successor of ν_i. The functions f_i and f^C_i doesn't change in this step, so by induction hypothesis holds $f_{i+1} = f^C_{i+1}$, too. Let $\tilde{\nu}$ be the node in the annotation of ν_i. In the computational case $m := f_i(\tilde{\nu})$ is sent to the adversary E. In the hybrid case the simulator receives $t^C := f^C_i(\tilde{\nu})$ and forwards $m^C := \beta(t^C)$ to the adversary. By induction hypothesis ($f_i = (\beta \circ f^C_i)$ follows $m^C = m$, and therefore is the state of the adversary the same afterwards, i.e. $S_{i+1} = S^C_{i+1}$.

5. If $\nu_i = \nu^C_i$ is a control node.

   Let l be the annotated out-metadata of node ν_i. In the computational case l is sent to the adversary E and in the hybrid case it's sent to *Sim* which forwards it to E. Since $S_i = S^C_i$ the computation of the adversary is the same and therefore is $S_{i+1} = S^C_{i+1}$ and E returns l' in both cases. So the chosen successor node is the same in both cases, i.e. $\nu_{i+1} = \nu^C_{i+1}$. Since the functions f_i and f^C_i stay untouched, follows $f_{i+1} = f^C_{i+1}$.

6. If $\nu_i = \nu^C_i$ is a nondeterministic node.

   The adversary is in both cases not invoked, so $S_{i+1} = S_i = S^C_i = S^C_{i+1}$. The same holds for the mappings f_{i+1} and f^C_{i+1}. Since we fixed the random of nodes and $\nu_i = \nu^C_i$ both choose the same successor and therefore $\nu_{i+1} = \nu^C_{i+1}$.

So $Nodes^p_{\mathbf{M},A,\Pi_\mathrm{p},E} = TH\text{-}Nodes_{\mathbf{M},\Pi_\mathrm{p},Sim}(k)$ if r_N, $enc(\ldots,N)$, $sig(\ldots,N)$, $ZK(\ldots,N)$ are different for all $N \in \mathbf{N}_P$. Since this is the case with overwhelming probability they are indistinguishable. □

Theorem 3. (Computational soundness of MZK-safe protocols) *Any computational implementation of the MZK-safe model* ***M*** *(see Section 3.4) that satisfies the implementation conditions for MZK-safe protocols (see Section 3.5) is computationally sound for the class of MZK-safe protocols (see Section 3.4).*

Proof. By Lemma 7 and Lemma 11 we get by transitivity of indistinguishability that *Sim* is DY. By Lemma 19 we conclude that *Sim* is indistinguishable for $\mathbf{M}, \Pi, A, E$ and for every polynomial p and every MZK-safe protocol, where A satisfies the implementation conditions for MZK-safe protocols (see Section 3.5). Consequently, *Sim* is a good simulator with respect to the $\hat{F}$-transparent hybrid execution, and Lemma 4 implies that any implementation A of the MZK-safe model satisfying the implementation conditions is computationally sound for the class of MZK-safe protocols. □

3.7. Conclusion

We have presented a computationally sound, symbolic abstraction of malleable ZK proofs by means of an equational theory that is accessible to existing tools for automated verification

of security protocols. We have proved the computational soundness of our abstraction with respect to trace properties and using weak cryptographic assumptions.The abstraction and the computational soundness result are presented in CoSP, a framework for symbolic protocol analyses and conceptually modular computational soundness proofs.

3.8. Postponed definitions

$$
\begin{aligned}
dec(dk(t_1), enc(ek(t_1), m, t_2)) &= m \\
ver_{sig}(vk(t_1), sig(sk(t_1), t_2, t_3)) &= t_2 \\
ekof(enc(ek(t_1), t_2, t_3)) &= ek(t_1) \\
ekof(garbEnc(t_1, t_2)) &= t_1 \\
vkof(sig(sk(t_1), t_2, t_3)) &= vk(t_1) \\
vkof(garbSig(t_1, t_2)) &= t_1 \\
fst(pair(t_1, t_2))) &= t_1 \\
snd(pair(t_1, t_2)) &= t_2 \\
unstring_1(string_1(s)) &= s \\
unstring_0(string_0(s)) &= s \\
equals(x, x) &= x
\end{aligned}
\qquad
\begin{aligned}
isenc(enc(ek(t_1), t_2, t_3)) &= enc(ek(t_1), t_2, t_3) \\
isenc(garbEnc(t_1, t_2)) &= garbEnc(t_1, t_2) \\
issig(sig(sk(t_1), t_2, t_3)) &= sig(sk(t_1), t_2, t_3) \\
issig(garbSig(t_1, t_2)) &= garbSig(t_1, t_2) \\
iscom(com(t_1, t_2, t_3)) &= com(t_1, t_2, t_3) \\
iscom(garbCom(t_1, t_2)) &= garbCom(t_1, t_2) \\
iszk(ZK(t_1, t_2, t_3)) &= ZK(t_1, t_2, t_3) \\
isek(ek(t)) &= ek(t) \\
isvk(vk(t)) &= vk(t) \\
iscrs(crs(t_1)) &= crs(t_1)
\end{aligned}
$$

$$
\begin{aligned}
mkZK(A = B, r, N) &= ZK(A = B, r, N) \\
mkZK(A \neq B, r, N) &= ZK(A \neq B, r, N) \\
ver_{zk}(crs, ZK(t, r, N)) &= check_{zk}(crs, t) \\
getPub(ZK(t_1, r_1, N)) &= t_1 \\
setPub(ZK(t_1, r_1, N), t_2) &= ZK(t_2, r_1, N) \\
and_{ZK}(ZK(t_1, r_1, N), ZK(t_2, r_2, N')) &= ZK(t_1 \wedge t_2, r_1 \wedge r_2, rand(N, N')) \\
splitAnd(ZK(t_1 \wedge t_2, r_1 \wedge r_2, rand(N, N'))) &= pair(ZK(t_1, r_1, N), ZK(t_2, r_2, N')) \\
or_{ZK}(ZK(t_1, r_1, N), ZK(t_2, r_2, N')) &= ZK(t_1 \vee t_2, r_1 \vee r_2, rand(N, N')) \\
commute(ZK(t_1 \vee t_2, r_1 \vee r_2, N)) &= ZK(t_2 \vee t_1, r_2 \vee r_1, N) \\
rer_{zk}(ZK(t_1, r_1, N), r_2, N') &= ZK(rerPr(t_1, r_2, N'), rerPr(r_1, r_2, N'), \\
&\qquad rand(N, N')) \\
open(crs(t_1), com(crs(t_1), t_2, t_3), uv(t_2, t_3)) &= t_2 \\
crsof(com(crs(t_1), t_2, t_3)) &= crs(t_1) \\
crsof(garbCom(t_1, t_2)) &= t_1 \\
rer_{com}(com(crs(t_1), t_2, t_3), t_4) &= com(crs(t_1), t_2, rand(t_3, t_4)) \\
rer_{com}(garbCom(t_1, t_2), t_3) &= garbCom(t_1, rand(t_2, t_3)) \\
applyF(com(crs(t_1), t_2, t_3), t_4) &= com(crs(t_1), f(t_2, x), t_3) \text{ for } f \in \mathbf{D} \cup \mathbf{C}
\end{aligned}
$$

Figure 3.6.: Definition of the set **D** of destructors.

The full syntax of our symbolic model (for both, the FMZK and the CMZK model):

$$
\begin{aligned}
T ::= \;& enc(ek(N), T, N) \mid ek(N) \mid dk(N) \mid pair(T, T) \mid \\
& sig(sk(N), T, N) \mid vk(N) \mid sk(N) \mid S \mid N \mid crs(N) \\
& ZK(S, R, M) \mid com(crs(N), T, M) \mid uv(T, M) \\
& garb(N) \mid garbageEnc(ek(N), N) \mid garbageSig(sk(N), N) \mid garbCom(crs(N), N) \\
Q ::= \;& \varepsilon \mid string_0(Q) \mid string_1(Q)
\end{aligned}
$$

$$
\begin{aligned}
S ::= \;& S \wedge S \mid S \vee S \mid TX = TX \mid TX \neq TX \\
TX ::= \;& \langle com(crs(n), TY, M), U \rangle \mid F\langle \underbrace{TX, \ldots, TX}_{m-times} \rangle \\
R ::= \;& R \wedge R \mid R \vee R \mid RX = RX \mid RX \neq RX \\
RX ::= \;& M \mid F\langle \underbrace{RX, \ldots, RX}_{m-times} \rangle \qquad U ::= uv(TY, M) \mid \varepsilon \\
TY ::= \;& T \mid N \qquad M ::= N \mid rand(M, M) \qquad N ::= n
\end{aligned}
$$

where $F \in StF$, $n \in \mathbf{N}$ and m is the arity of F.

Figure 3.7.: Definition of the grammar of all terms T

If no rules applies, $rerPr$ outputs $\perp$. In the following m is the arity of the constructor or destructor F.

$$
\begin{aligned}
rerPr(S \wedge S', Q \wedge Q', N) &= rerPr(S, Q, N) \wedge rerPr(S', Q', N) \\
rerPr(S \vee S', Q \vee Q', N) &= rerPr(S, Q, N) \vee rerPr(S', Q', N) \\
rerPr(A = A', Q = Q', N) &= rerPr(A, Q, N) = rerPr(A', Q', N) \\
rerPr(A \neq A', Q \neq Q', N) &= rerPr(A, Q, N) \neq rerPr(A', Q', N) \\
rerPr(F\langle t_1, \ldots, t_M \rangle, F\langle u_1, \ldots, u_m \rangle, N) & \\
&= F\langle rerPr(t_1, u_1, N), \ldots, rerPr(t_m, u_m, N) \rangle
\end{aligned}
$$

$$
\begin{aligned}
rerPr(\langle C, U \rangle, \varepsilon, N) &= \langle C, U \rangle \\
rerPr(\langle com(crs(n), t, M), uv(M') \rangle, string_1(\varepsilon), N) & \\
&= \langle com(crs(n), t, rand(M, N), uv(rand(M', N)) \rangle
\end{aligned}
$$

$$
\begin{aligned}
rerPr(n, \varepsilon, N) &= n \\
rerPr(rand(t_1, t_2), \varepsilon, N) &= rand(t_1, t_2) \\
rerPr(n, string_1(\varepsilon), N) &= rand(n, N) \\
rerPr(rand(t_1, t_2), string_1(\varepsilon), N) &= rand(rand(t_1, t_2), N)
\end{aligned}
$$

Figure 3.8.: The definition of the re-randomization $rerPr$

$$
\begin{aligned}
L'(S_1 \vee S_2) &:= L'(S_1) \vee L'(S_2) \\
L'(S_1 \wedge S_2) &:= L'(S_1) \wedge L'(S_2) \\
L'(A = B) &:= L'(A) = L'(B) \\
L'(A \neq B) &:= L'(A) \neq L'(B) \\
L'(F\langle X_1, \ldots, X_n\rangle) &:= F(L'(X_1), \ldots, L'(X_n)) \\
n &= \mathit{arity}(F) \\
L'(\langle \mathit{com}(\mathit{crs}, m, r), \varepsilon\rangle) &:= x_m \text{ , for } x_m \in V \\
L'(\langle \mathit{com}(\mathit{crs}, m, r), \mathit{uv}(m, r)\rangle) &:= m
\end{aligned}
$$

Figure 3.9.: The mapping L' from statement terms to logical formulas.

$$
\begin{aligned}
\mathit{crsof}(S_1 \vee S_2) &:= \mathit{crsof}(S_1) \quad \text{if } \mathit{crsof}(A) = \mathit{crsof}(B) \\
\mathit{crsof}(S_1 \wedge S_2) &:= \mathit{crsof}(S_1) \quad \text{if } \mathit{crsof}(A) = \mathit{crsof}(B) \\
\mathit{crsof}(A = B) &:= (\mathit{crsof}(A) \quad \text{if } \mathit{crsof}(A) = \mathit{crsof}(B) \\
\mathit{crsof}(A \neq B) &:= \mathit{crsof}(A) \quad \text{if } \mathit{crsof}(A) = \mathit{crsof}(B) \\
\mathit{crsof}(F\langle X_1, \ldots, X_n\rangle) &:= \mathit{crsof}(X_1) \quad \text{if } \forall i, j. \mathit{crsof}(X_i) = \mathit{crsof}(X_j) \\
n &= \mathit{arity}(F) \\
\mathit{crsof}(\langle \mathit{com}(\mathit{crs}, m, r), \varepsilon\rangle) &:= \mathit{crs} \\
\mathit{crsof}(\langle \mathit{com}(\mathit{crs}, m, r), \mathit{uv}(r)\rangle) &:= \mathit{crs}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{extrSta}(S_1 \vee S_2) &:= (\mathit{extrSta}(S_1), \mathit{extrSta}(S_2)) \\
\mathit{extrSta}(S_1 \wedge S_2) &:= (\mathit{extrSta}(S_1), \mathit{extrSta}(S_2)) \\
\mathit{extrSta}(A = B) &:= (\mathit{extrSta}(A), \mathit{extrSta}(B)) \\
\mathit{extrSta}(A \neq B) &:= (\mathit{extrSta}(A), \mathit{extrSta}(B)) \\
\mathit{extrSta}(F\langle X_1, \ldots, X_n\rangle) &:= F\langle \mathit{extrSta}(X_1), \ldots, \mathit{extrSta}(X_n)\rangle \\
n &= \mathit{arity}(F) \\
\mathit{extrSta}(\langle \mathit{com}(\mathit{crs}, m, r), \varepsilon\rangle) &:= \varepsilon \\
\mathit{extrSta}(\langle \mathit{com}(\mathit{crs}, m, r), \mathit{uv}(m, r)\rangle) &:= m
\end{aligned}
$$

$$
\begin{aligned}
\mathit{extrNon}(S_1 \vee S_2) &:= (\mathit{extrNon}(S_1), \mathit{extrNon}(S_2)) \\
\mathit{extrNon}(S_1 \wedge S_2) &:= (\mathit{extrNon}(S_1), \mathit{extrNon}(S_2)) \\
\mathit{extrNon}(A = B) &:= (\mathit{extrNon}(A), \mathit{extrNon}(B)) \\
\mathit{extrNon}(A \neq B) &:= (\mathit{extrNon}(A), \mathit{extrNon}(B)) \\
\mathit{extrNon}(F\langle X_1, \ldots, X_n\rangle) &:= (\mathit{extrNon}(X_1), \ldots, \mathit{extrNon}(X_n)) \\
n &= \mathit{arity}(F) \\
\mathit{extrNon}(\langle \mathit{com}(\mathit{crs}, m, r), \varepsilon\rangle) &:= r \\
\mathit{extrNon}(\langle \mathit{com}(\mathit{crs}, m, r), \mathit{uv}(m, r)\rangle) &:= r
\end{aligned}
$$

$$
\begin{aligned}
\mathit{extrWit}(\langle \mathit{com}(\mathit{crs}, m, r), \varepsilon\rangle) &:= m \\
\mathit{extrWit}(\langle \mathit{com}(\mathit{crs}, m, r), \mathit{uv}(m, r)\rangle) &:= \varepsilon
\end{aligned}
$$

extrWit is defined just like *extrNon* except for the base cases.
The algorithms A_{extrSta} and A_{crsof} are defined just like the symbolic counterpart except that the case distinction is performed using the decoding functions $d_\wedge, d_\vee, d_=, d_\neq, d_F$.

Figure 3.10.: The symbolic extraction destructors.

Chapter 4.

Abstracting Secure Multi-Party Computation

[This chapter is based on a work with Michael Backes and Matteo Maffei [BMM10]. I am the main contributor to all parts that occur in this chapter.]

4.1. Motivation

General-purpose secure multi-party computation (SMPC) is one of the most impressive achievements of modern cryptography [GMW87]. On a high level, in an SMPC a number of parties $P_1, \ldots, P_n$ wish to securely compute the value $F(d_1, \ldots, d_n)$, for some well-known public function F, where each party P_i holds a private input d_i. This multi-party computation is considered secure if it does not divulge any information about the private inputs to other parties; more precisely, no party can learn more from the participation in the SMPC than she could learn purely from the result of the computation already.

SMPC provides solutions to various real-life problems such as e-voting, private bidding and auctions, secret sharing etc. The recent advent of efficient general-purpose implementations (e.g., FairplayMP [BNP08]) paves the way for the deployment of SMPC into modern cryptographic protocols. Recently, the effectiveness of SMPC as a building block of large-scale and practical applications has been demonstrated by the sugar-beet double auction that took place in Denmark: The underlying cryptographic protocol [Bog+09], developed within the Secure Information Management and Processing project, is based on SMPC.

Dolev-Yao models are highly amenable to automated verification techniques, but before this work no computationally sound symbolic abstraction of SMPC was known.

Our Contribution. We present an abstraction of SMPC within the applied π-calculus [AF01]. This abstraction consists of a process that receives the inputs from the parties involved in the protocol over private channels, computes the result, and sends it to the parties again over private channels, however augmented with certain details to enable computational soundness results, see below. This abstraction can be used to model and reason about larger cryptographic protocols that employ SMPC as a building block.

Building upon an existing type-checker [BHM08b], we propose an automated verification technique for protocols based on our SMPC abstraction. We exemplify the applicability of our framework by analyzing the sugar-beet double auction protocol proposed in [Bog+09].

We establish computational soundness results (in the sense of preservation of trace properties) for protocols built upon our abstraction of SMPC. This result is obtained in essentially two steps: We first establish a connection between our symbolic abstraction of SMPC in the applied π-calculus (symbolic setting) and the notion of an ideal functionality for SMPC in the UC framework [CLOS02], which constitutes a low-level abstraction of

SMPC that is defined based on bitstrings, Turing machines, etc. (cryptographic setting) Second, we build upon existing results on the secure realization of this functionality in the UC framework in order to obtain a secure cryptographic realization of our symbolic abstraction of SMPC. This computational soundness result holds for SMPC that involve arbitrary arithmetic operations; moreover, it is compositional, since the proof is parametric over the other (non-interactive) cryptographic primitives used in the symbolic protocol and within the SMPC itself. Computational soundness holds as long as these primitives are shown to be computationally sound (e.g., in the CoSP framework [BHU09]). We prove in particular the computational soundness of a Dolev-Yao model with public-key encryption, signatures, and the aforementioned arithmetic operations, leveraging and extending prior work in CoSP. Such a result allows for soundly modelling and verifying many applications employing SMPC as a building block, including the case studies considered in this paper.

Related work. Computational soundness was first shown by Abadi and Rogaway [AR02], against passive adversaries and for symmetric encryption. Subsequent work extended the protocol language and extended the results for stronger security properties [ABW06; AJ01; BCK05; HLM03; Lau01] and considered active adversaries [BHU09; BP04; BPW03a; BPW03b; BPW07; CH06; CW05; JLM05; Lau04; MW04; SBBPW06]. Recent work even proved computational soundness for observational equivalence [AF06; CC08; Com08]. All these results, however, only consider non-interactive cryptographic primitives, such as encryptions, signatures, and non-interactive zero-knowledge proofs. Up to this work, no general computational soundness proof for an interactive primitive in combination with Dolev-Yao style non-interactive abstractions was known.

Non-interactive primitives only get one input and output one result. Hence, all computation steps that are done between the input and the output can be abstracted way. Interactive primitives, on the other hand, are reactive and may involve several communication steps, such as committing to an input. The abstraction of these intermediate communication steps is desirable in order to make the automated verification feasible. But such an abstraction significantly complicates the computational soundness proofs.

A salient approach for the abstraction of interactive cryptographic primitives is the Universal Composability framework [Can01]. The central idea is to define and prove the security of a protocol by comparison with an ideal trusted machine, called the ideal functionality. Although this framework has proven convenient for modularly proving security of cryptographic protocols by hand, it is not suited to automation of security proofs, given the intricate operational semantics of the UC framework. Dolev-Yao models (e.g., the applied π-calculus) offer a higher level of abstraction compared to ideal functionalities in the UC framework; in particular the adversary is abstracted from an arbitrary interactive ppt machine to a set of rules. Most importantly, Dolev-Yao models typically enjoy a simpler operational semantics than the UC framework (e.g., Dolev-Yao models typically do not comprise randomness), which enables automation of security proofs. The different degree of abstraction in these models is best understood by considering digital signatures: While computational soundness proofs for Dolev-Yao abstractions of digital signatures use standard techniques [BHU09; BPW03a; CW05] finding a sound ideal functionality for digital signatures has proven to be quite intricate [BH04; Can04; DDMRS06]. Yet, securely realizable ideal functionalities constitute a useful tool for proving computational soundness of a Dolev-Yao model. In our proof, we establish a connection between our symbolic abstraction of SMPC and such an ideal functionality; subsequently, we exploit

that this ideal functionality is securely realizable. A similar proof technique has already been applied by Canetti and Herzog [CH06] for showing computational soundness of the symbolic abstraction of public-key encryption. For the formal verification of our symbolic abstraction, we use a type-system for authorization policies (c.f. [BHM08b; FGM07]). Specifically, we use the type system of Backes, Hritcu, and Maffei.[BHM08b]

A generic symbolic abstraction of ideal functionalities has been proposed in [DKP09]. In that work it is shown that the different notions of simulatability, known in the literature, collapse in the symbolic abstraction. In contrast to our approach, that work does not address computational soundness guarantees and does not explicitly consider SMPC.

The concept of a general secure two-party computation has been introduced by Yao [Yao82]. This work has been extended to general secure multi-party computations in [GMW87]. General secure multi-party computation in the presence of arbitrary surrounding protocols, in the above-mentioned UC framework, has been studied in [CLOS02].

There are domain-specific languages for specifying functionalities for secure multi-party computations and for deriving secure cryptographic realizations. FairplayMP [BNP08; MNPS04], e.g., automatically generates a constant-round secure multi-party computation protocol given a program in the language SFDL. Another domain-specific language SMCL is presented in [Nie09]. SMCL supports reactive functionalities as well. Moreover, this work introduces a type system for checking non-interference properties of the programmed functionalities. These approaches only focus on a single instance of a secure multi-party computation. Our framework, in contrary, can be used to reason about an unbounded number of SMPC sessions and, most importantly, about a large class of cryptographic protocols employing SMPC as a building block. Moreover, our symbolic abstraction is amenable to automated verification techniques and our computational soundness result ensures that such verification techniques provide security guarantees for the actual computational implementation.

Outline of this chapter. Section 4.2 presents our SMPC abstraction. Section 4.3 explains the technique used to statically analyze SMPC-based protocols and applies it to our case study. Section 4.4 presents the computational implementation of a process. Section 4.5 studies the computational soundness of our abstraction. Section 4.5.2 shows that computational soundness results shown in the CoSP framework [BHU09] carry over to our framework, and Section 4.6 concludes.

4.2. The symbolic abstraction of SMPC

In this section, we adopt a variant of the applied π-calculus with destructors [BHM08b]. After that, we present the symbolic abstraction of secure multi-party computation within this calculus.

4.2.1. Abstracting SMPC in the Applied π-calculus

We recall that a secure multi-party computation is a protocol among parties $P_1, \ldots, P_n$ to jointly compute the result of a function $\mathcal{F}$ applied to arguments $m_1, \ldots, m_n$, where m_i is a private input provided by party P_i. More generally, not only a function but a reactive, stateful computation is performed, which requires the participants to maintain a

$$
\begin{aligned}
\mathbf{inout}_i &:= !\big(inloop_i(z).in_i(x_i, sid').\overline{adv}\langle sid'\rangle.\texttt{if } sid = sid' \\
&\quad \texttt{then } \overline{lin_i}\langle x_i\rangle \texttt{ else } \overline{inloop_i}\langle\texttt{sync()}\rangle\big) \mid \overline{inloop_i}\langle\texttt{sync()}\rangle \\
\mathbf{deliver}_i &:= \overline{in_i}\langle y_i, sid\rangle.\overline{inloop_i}\langle\texttt{sync()}\rangle \\
\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F}) &:= sidc(sid).\ \nu\underline{lin}.\ \nu\underline{inloop}. \\
&\quad \big(\mathbf{input}_1 \mid \cdots \mid\ \mathbf{input}_n \mid \mathcal{F}[\mathbf{deliver}_1 \mid \ldots \mid \mathbf{deliver}_n]\big)
\end{aligned}
$$

Figure 4.1.: The process **SMPC** as the symbolic abstraction of secure multi-party computation

(synchronized) state. At the end of the computation, each party should not learn more than the result (or, more generally, a local view r_i of the result). Since the overall protocol may involve several secure multi-party computations, a session identifier *sid* is often used to link the private inputs to the intended session. Coming up with an abstraction of SMPC that is amenable to automated verification and that can be proven computationally sound is technically challenging, and it required us to refine the abstraction based on insights that were gained in the soundness proof. For the sake of exposition, we thus present simple, intuitive abstraction attempts of SMPC in the applied π-calculus first, explain why these attempts do not allow for a computational soundness result, and then successively refine them until we reach the final abstraction.

First attempt. A first, naive attempt to symbolically abstract SMPC in the applied π-calculus is to let parties send to each other the public information along with the enveloped private input on a private channel. This message can be represented by a term $\texttt{smpc}(F, i, m, sid)$, where i is the principal's identifier. The abstraction then consists of a destructor `result` whose semantics is defined by a rule like

$$\texttt{result}(m_i, i, \texttt{smpc}(\mathcal{F}, 1, m_1, sid), \ldots, \texttt{smpc}(\mathcal{F}, n, m_n, sid)) = \ \pi(i, \mathcal{F}(m_1, \ldots, m_n))$$

where $\pi(i, \cdot)$ denotes the projection on the i-th element. We are facing the common symbolic adversary model of an active, but static adversary, i.e., the adversary controls the (asynchronous) network, but only compromises parties before the protocol execution starts. In this setting, computational soundness of this abstraction cannot be shown: a computational attacker (i) is capable of altering the delivery of messages, and (ii) learns the session identifiers that occur in the header of each individual message. Our abstraction attempt does not grant the adversary any such capabilities; hence a symbolic adversary is constrained much stronger than a computational adversary, thus preventing computational soundness results. These problems could be tackled by modifying the abstraction such that all messages are sent and received over a public channel. The adversary would then decide which parties receive which messages. The resulting abstraction, however, would not be tight enough anymore: Corrupted symbolic parties could send different messages to the participants, and hence cause them to compute the function $\mathcal{F}$ on different inputs. Such an attack, however, is computationally excluded by a secure multi-party computation protocol.

Second attempt. We solved the aforementioned problems by introducing an explicit trusted party to whom every participant i sends its private input and receives its own result in return over a private channel in_i. This follows the general ideal-functionality paradigm for defining security that was already used in [GMW87] for defining SMPC in the cryptographic setting. Such a trusted party can be nicely abstracted as a distinguished process in the applied π-calculus. A first attempt, called **SMPC_temp**, could look as follows:

$$\begin{aligned}\textbf{SMPC\_temp} := {} & sidc(sid).in_1(x_1, = sid) \dots in_n(x_n, = sid). \\ & \texttt{let } (y_1, \dots, y_n) = \texttt{result}(\mathcal{F}, x_1, \dots, x_n) \texttt{ in } \overline{in_1}\langle y_1, sid\rangle \dots \overline{in_n}\langle y_n, sid\rangle\end{aligned}$$

There still is discrepancy between this abstraction and the computational model that invalidates computational soundness: a computational adversary learns the session identifier, which is instead concealed by **SMPC_temp**. In addition, the computation of $\mathcal{F}$ is shifted to the evaluation of the destructor `result`. Such a complicated destructor would make mechanized verification extremely difficult.

Final abstraction. To make the abstraction amenable to automated verification, in particular type-checking, we represent $\mathcal{F}$ as a context that explicitly performs the computation. The resulting abstraction of secure multi-party computation is depicted in Figure 4.1 as the process **SMPC**. This process is parametrized by an adversary channel adv, a session identifier channel $sidc$, n private channels in_i for each of the n participants, and a context $\mathcal{F}$. We implicitly assume that private channels are authenticated such that only the ith participant can send messages on channel in_i. The computational implementation of SMPC implements this authentication requirement. Furthermore, **SMPC** contains two restricted channels for every party i: an internal loop channel $inloop_i$ and an internal input channel lin_i.

SMPC receives a session identifier over the channel $sidc$. Then $n + 1$ subprocesses are spawned: a process $\textbf{input}_i$ for every participant i that is responsible for collecting the ith input and for divulging public information, such as the session identifier, to the adversary, and a process that performs the actual multi-party computation. Here $\textbf{input}_i$ waits (under a replication) on the loop channel $inloop_i$ for the trigger message `sync`() of the next round, and expects the private input x_i and a session identifier sid' over in_i. It then sends the session identifier sid' to the adversary, checks whether the session identifier sid' equals sid, and finally sends the private input x_i on the internal input channel lin_i. The actual multi-party computation is performed in the last subprocess: after the private inputs of the individual parties are collected from the internal input channels lin_i, the actual program $\mathcal{F}$ is executed.

After each computation round, the subprocesses $\textbf{deliver}_i$ send the individual outputs over the private channels in_i to every participant i along with the session identifier sid. In order to trigger the next round, `sync`() is sent over the internal loop channels $inloop_i$.

The abstraction allows for a large class of functionalities $\mathcal{F}$ as described below.

Definition 37. *[SMPC-suited context] An* SMPC-suited context *is a context $\mathcal{F}$ such that:*

1. $fv(\mathcal{F}) = \{sid\}$ *and* $fn(\mathcal{F}[\mathbf{0}]) = \{lin_1, \dots, lin_n\}$.
2. *Bound names and variables are distinct and different from free names and free variables.*

Although one might expect additional constraints on the context $\mathcal{F}$ (e.g., it is terminating, it does not contain replications, etc.), it turns out that such constraints are not necessary as, intuitively, having more traces in the symbolic setting does not break computational soundness. Such constraints would probably simplify the proof of computational soundness, but they would make our abstraction less intuitive and less general.

Finally, we briefly describe how we model the corruption of the parties involved in the secure multi-party computation. In this paper we consider static corruption scenarios, in which the parties to be corrupted are selected before the computation starts. As usual in the applied π-calculus, we model corrupted parties by letting the adversary know the secret inputs of the corrupted parties. This is achieved by letting the channel in_i occur free in the process if party i is corrupted. As we only consider static corruption, we restrict our attention to processes that do not send these channels in_i over a public channel (for a formal definition see Definition 43 of well-formed processes).

Arithmetics in the Applied π-calculus. One of the most common applications of SMPC is the evaluation of arithmetic operations on secret inputs. Modelling arithmetic operations in the applied π-calculus is straightforward. We encode numbers in binary form via the $string_0(M)$, $string_1(M)$, and $empty()$ constructor applications. Arithmetic operations are modelled as destructors. For instance, the greater-equal relation is defined by the destructor $\mathtt{ge}(M_1, M_2)$, which returns M_1 if M_1 is greater equal then M_2, M_2 otherwise. With this encoding, numbers and cryptographic messages are disjoint sets of values, which is crucial for the soundness of our analysis and the computational soundness results.

The Millionaires problem. For the sake of illustration, we show how to express the Millionaires problem in our formalism (two parties wish to determine who is richer, i.e., whose input is bigger, without divulging their inputs to each other.) The protocol is parametrized by two numbers x_1 and x_2:

$$\begin{aligned}\nu sid.\ \nu sidc.\ \nu c_1.\ \nu c_2.\ \overline{sidc}\langle sid\rangle \mid \overline{adv}\langle sid\rangle \mid \mathbf{SMPC}(adv, sidc, c_1, c_2, \mathbf{compare})\\ \mid c_1(sid).\overline{c_1}\langle (x_1, id_1), sid\rangle.c_1(y_1, sid_1)\\ \mid c_2(sid).\overline{c_2}\langle (x_2, id_2), sid\rangle.c_2(y_2, sid_2)\end{aligned}$$

$$\begin{aligned}\mathbf{compare}[\bullet] \ :=\ & lin_1((x_1, z_1)).lin_2((x_2, z_2)).\mathtt{let}\ x_1 = \mathtt{ge}(x_1, x_2)\\ & \mathtt{then\ let}\ y_1 = y_2 = z_1\ \mathtt{in}\ \bullet\ \mathtt{else\ let}\ y_1 = y_2 = z_2\ \mathtt{in}\ \bullet\end{aligned}$$

where $[y_1 := z, y_2 := z]$ denotes the instantiation of variables y_1 and y_2 with variable z, which can be defined by encoding, and $\bullet$ is the hole of the context.

4.3. Formal Verification

We propose a technique for formally verifying processes that use the symbolic abstraction **SMPC**. In principle, our abstraction is amenable to several verification techniques for the applied π-calculus such as [BAF05; Bla01; FGM07]. In this paper, we rely on the type-checker for security policies presented in [BHM08b], which we extend to support arithmetic operations. This type-checker enforces the robust safety property (cf. Definition 30).

We decorate the protocol linked to our SMPC abstraction with assumptions and assertions, as depicted in Figure 4.2. The assumptions are used to mark the inputs of the secure

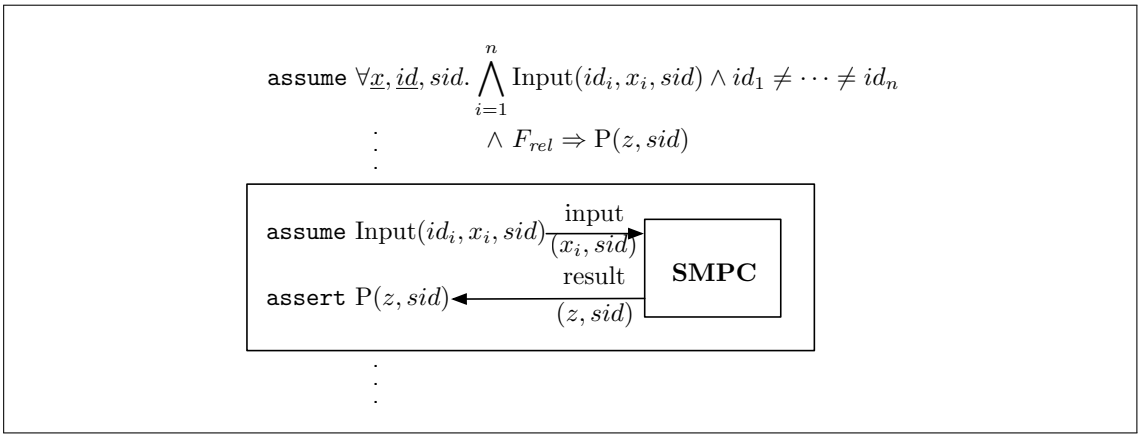

Figure 4.2.: Assumptions and assertions for a process linked to our **SMPC** abstraction.

multi-party computation and to specify the correctness property for the secure multi-party computation. The assertion is used to check that the result of the SMPC fulfills the correctness property.

Specifically, for every party i an assumption `assume` $\mathrm{Input}(id_i, x_i, sid)$ is placed upon sending a private input x_i with session identifier sid, where id_i is the (publicly known) identity of party i. To check the correctness of the secure multi-party computation, `assert` $\mathrm{P}(z, sid)$ is placed immediately after the reception of the result (z, sid). We also assume a security policy, which takes in general the following form:

$$\forall \underline{id}, \underline{x}, sid, z.\big(\wedge_{i=1}^{n} \mathrm{Input}(id_i, x_i, sid) \wedge id_1 \neq \ldots \neq id_n \wedge F_{rel}\big) \Rightarrow \mathrm{P}(z, sid)$$

The formula F_{rel} characterizes the expected relation between the inputs and the output. As an example, the policy and party annotations for the Millionaires problem would be as follows:

$$\forall id_1, id_2, x_1, x_2, sid.\big(\mathrm{Input}(id_1, x_1, sid) \wedge \mathrm{Input}(id_2, x_2, sid) \wedge id_1 \neq id_2 \wedge x_1 \geq x_2\big) \Rightarrow \mathrm{Richer}(id_1, sid).$$

Each party would be annotated as follows:

$$P_i \ := \ \texttt{assume}\ \mathrm{Input}(id_i, x_i, sid) \mid \overline{c_i}\langle (x_i, id_i), sid\rangle.c_i(y_i, sid_i).\texttt{assert}\ \mathrm{Richer}(y_i, sid).$$

Arithmetics in the analysis. We extended the type-checker to support arithmetic operations. Specifically, we modelled arithmetic operations as predicates in the logic, defined their semantics following the semantics given in the calculus, and added a few general properties, such as the transitivity of the greater-equal relation. The type theory is extended to track the arithmetic properties of terms (e.g., while typing `let` $z = \texttt{ge}(x_1, x_2)$ `then` P, the type-checker tracks that z is the greatest value between x_1 and x_2 and uses this information to type P). The type theory supports this kind of extensions as long as the set of added values is disjoint from the set of cryptographic messages and the added destructors do not operate on cryptographic terms, which holds true for our encoding of arithmetics.

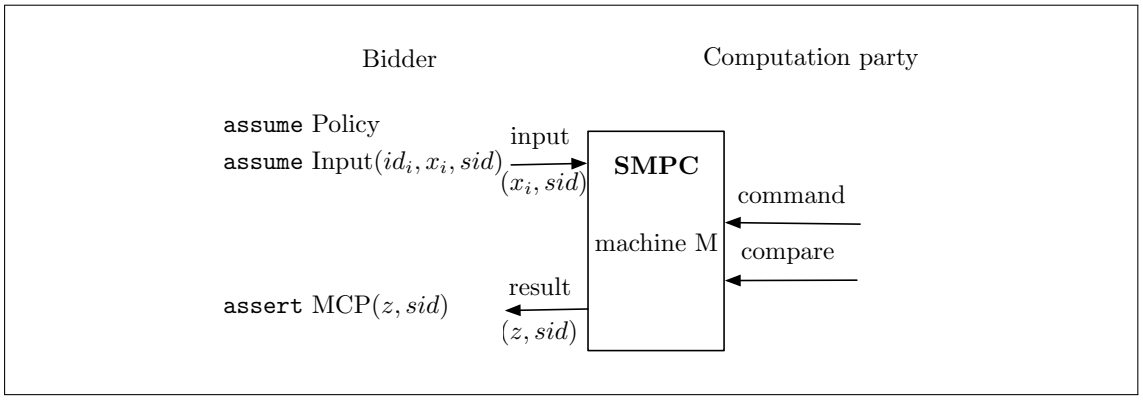

Figure 4.3.: Our model of the sugar-beet double auction via the abstraction **SMPC**.

Case study: sugar-beet double auction

As a case study for our symbolic abstraction, we formalized and analyzed the sugar-beet double auction that has been realized within the SIMAP project by using an SMPC [Bog+09]. This protocol constitutes the first large scale application of an SMPC. The double auction protocol determines a market clearing price for sugar-beets. More specifically, first, a set of prices is fixed; then, for every price each producer commits itself to an amount of sugar-beets that it is willing to sell, and each buyer commits itself to the amounts of sugar-beets that it is willing to buy. The market clearing price is the maximal market clearing price for which the supply did not yet exceed the demand.

The protocol assumes that for every producer the list with the amounts of sugar-beets that this producer is willing to sell for each price is monotonically increasing. Analogously, for every buyer the list with the amount of sugar-beets that a buyer is willing to buy for each price is monotonically decreasing for this buyer. Beginning from the lowest price, the protocol computes for every price i the demand, i.e., the sum of the amounts that the buyers are willing to buy, and the supply, i.e., the sum of the amounts that the sellers are willing to sell. Then, the protocol compares the supply and the demand. If the supply is greater than the demand, the protocol outputs the price $i - 1$ as the market clearing price. In the other case, the protocol compares the next higher price $i + 1$ until the protocol exceeds the maximal price p, in which case the maximal price is output. [1]

Both the producers and the buyers might want to keep their bids secret. Hence, the private input of every party has to be kept private. In the sugar-beet double auction developed by the SIMAP project, the sellers and buyers perform a joint secure multi-party computation. The producer and buyer parties send initially an input and receive at the end of the computation the result, i.e., the market clearing price. The protocol comprises three computation parties that execute a subprotocol, which constitutes a synchronous SMPC. In addition, before the start of the protocol, producers and buyers receive a signature on the list of participants, the session id, and the set of prices from a trusted party. This signature is verified by each participant and sent as an input to the SMPC, which verifies and checks the equality of such signatures. This ensures that all participants agree on the

[1] We only consider the maximal market clearing price. The actual sugar-beet double auction protocol has a more elaborate procedure for choosing one market clearing price.

common public inputs. Notice that this SMPC performs arithmetic operations as well as cryptographic operations, yet on distinct values.

We modelled the SMPC with a context $\mathcal{F}$ that expects inputs from the producers and buyers and commands from the computation parties (see Figure 4.3). Upon sending the input x, the producer and buyer parties assume $\text{Input}(id, x, sid)$, where id is the identifier for that party and sid is the session identifier. Upon receiving the final result z, sid every party asserts $\text{MCP}(z, sid)$ (this predicate is defined below). Assume there are m different prices; then, we represent the inputs of the producers and the buyer, i.e., the list with the amounts, as a tuple $bids := (x_1, \ldots, x_m)$. Additionally, every producer uses a flag $b := 1$ and every buyer a flag $b := 0$. Every producer and buyer sends the pair $(b, bids)$ to the secure multi-party computation.

Finally, we are ready to state the policy characterizing the result of the computation that is performed: the predicate $\text{MCP}(z, sid)$ holds true if there are appropriate input predicates $\text{Input}(id_i, x_i, sid)$ and z is the maximal price for which demand is greater than or equal to the supply, which we characterize by the predicate $\text{Is\_max}(x_1, \ldots, x_n, z)$ (the semantics of this predicate is defined in terms of basic arithmetic operations supported by our type-checker), where $\pi_i(t)$ denotes the i-th element of the tuple t.

$$\forall x_1, \ldots, x_n, z, i.\ Q(x_1, \ldots, x_n, z) \wedge Q(x_1, \ldots, x_n, i) \Rightarrow z \geq i \Rightarrow \text{Is\_max}(x_1, \ldots, x_n, z) \quad \text{and}$$

$$\forall b_1, \ldots, b_n, bids_1, \ldots, bids_n, l.\ \underbrace{\sum_{j=1}^{n} \pi_l(bids_j) \cdot (1 - b_j)}_{\text{demand}} \geq \underbrace{\sum_{j=1}^{n} \pi_l(bids_j) \cdot b_j}_{\text{supply}} \Rightarrow Q((b_1, bids_1), \ldots, (b_n, bids_n), l)$$

The policy for the correctness of the computation (called Policy in Figure 4.3) is then defined as follows:

$$\forall z, sid, x_1, \ldots, x_m, id_1, \ldots, id_n.\ \text{Input}(id_1, x_1, sid) \wedge \cdots \wedge \text{Input}(id_n, x_n, sid) \wedge id_1 \neq \ldots \neq id_n \wedge \text{Is\_max}(x_1, \ldots, x_m, z) \Rightarrow \text{MCP}(z, sid)$$

Verification. The verification of this policy is challenging in that our abstraction comprises about 1400 lines of code and it relies on complex functions. The type-checker succeeds in 2 minutes and 22 seconds on a MacBook Pro (Intel 2GHz Dual Code, 4GB RAM). The source code is depicted in Appendix A. We type checked the protocol with 2 prices, 3 computation parties, and 2 input parties.

4.4. Computational Execution

In this section, we present a computational soundness result for our abstraction of secure multi-party computations. Our result builds on the universal composability (UC) framework [Can01; CLOS02], where the security of a protocol ρ is defined by comparison with an ideal functionality $\mathcal{I}$. The proof proceeds in three steps, as depicted in Figure 4.4.

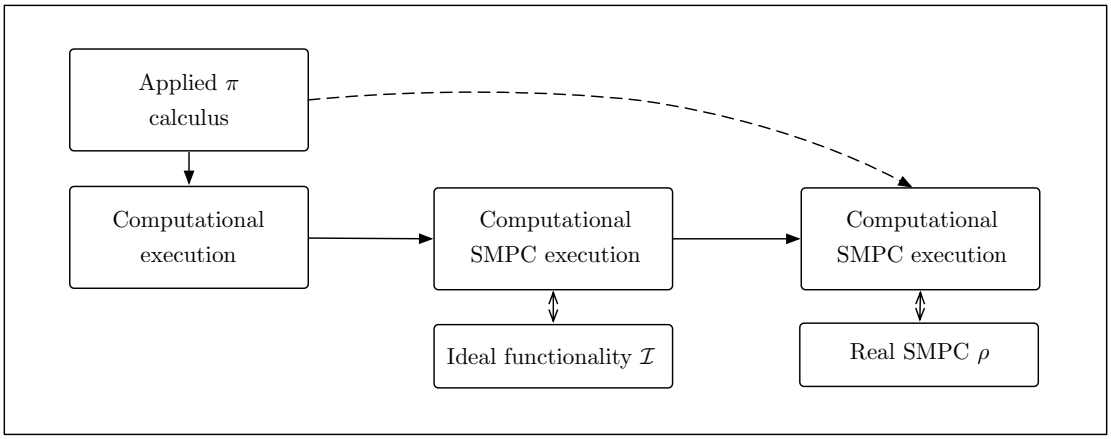

Figure 4.4.: Overview of the computational soundness proof

In the first step, we prove that the robust safety of an applied π-calculus process carries over to the computational setting, where the protocol is executed by interactive Turing machines operating on bitstrings instead of symbolic terms and using cryptographic algorithms instead of constructors and destructors. This part of the proof solely depends on *additional* non-interactive cryptographic primitives that might be used in the protocol (such as encryption and digital signatures). It does not depend on the secure multi-party computations, which are still abstracted symbolically by processes of the form $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$. Since the remaining steps of the proof are parametric over these non-interactive primitives, we decided to make our result general by assuming the first step of the proof, which is fairly standard and follows arguments similar to previous computational soundness results for non-interactive cryptographic primitives. In Section 4.5.2, we prove this step for processes using asymmetric encryption, digital signatures, and arithmetic operations, basically, applying verbatim the computational soundness proof presented in [BHU09].

The first part of the proof entails the computational soundness of a process executing the abstraction $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$. A computational implementation of the protocol, instead should execute an actual SMPC protocol ρ. In the second step of the proof, we show that for each SMPC-suited context $\mathcal{F}$, the computational execution of our abstraction $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ is indistinguishable from the execution of a protocol $\mathcal{I}$ (see Construction 1) that solely comprises a single incorruptible machine.

The third step of the proof builds on a result from [CLOS02], which ensures that for any well-formed ideal functionality $\mathcal{I}$ there is a protocol ρ that securely realizes $\mathcal{I}$ in the UC framework, which ensures in particular that the trace properties of $\mathcal{I}$ carry over to the actual implementation ρ.

These three steps allow us to conclude that for each abstraction $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ there exists an implementation ρ such that the trace properties of any well-formed process P containing SMPC occurrences $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$, and in particular the security policies enforced by the type system, carry over from the execution that merely executes a process P and executes $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ as a regular subprocess to the execution that communicates with ρ upon each call of a subprocess $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$. By leveraging the composability of the UC framework and the realization result for SMPC in

the UC framework, we finally conclude that if a protocol based on our SMPC abstraction is robustly safe, then there exists an implementation of that protocol that is computationally safe.

We review the characterization of secure multi-party computations in the UC framework in Section 4.4.1. We present the computational execution for our SMPC abstraction in Section 4.4.2. In Section 4.5.1 we introduce the notion of computational safety. Finally, we state our computational soundness result in Section 4.5.7. Throughout the entire section, we use the notation that n is the number of parties in the current secure multi-party computation, i.e., for $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ we have that $n := |\underline{in}|$.

4.4.1. SMPC in the UC framework

In this section, we discuss how to construct an interactive sub-protocol in the UC framework out of an SMPC process. We first review a general framework for universally composable interactive cryptographic protocols: the UC framework (Section 4.4.1.1), and then construct the ideal functionality from the SMPC process (Section 4.4.1.2).

4.4.1.1. The UC Framework

The UC framework defines the security of a protocol ρ by comparison with an ideal functionality $\mathcal{I}$. The ideal functionality for secure multi-party computations resembles our SMPC abstraction in that it essentially consists of a single machine performing an ideal computation. In the UC framework, a protocol ρ is called a secure realization of an ideal functionality $\mathcal{I}$ if for every attack on ρ there is an attack on $\mathcal{I}$. More precisely, for any adversary that interacts with ρ, there is a simulator that interacts with $\mathcal{I}$ such that no protocol environment interacting with both the protocol and the adversary can tell an execution of ρ from an execution of $\mathcal{I}$. In this work, all these machines are polynomial-time.

As an example consider an authenticated channel between Alice and Bob with a passive attacker. In the real world Alice would call a protocol that signs the message m to be communicated and sends the signed message over the network such that Bob would verify the signature. In the setting with a trusted machine T, however, Alice sends the message m directly to T[2]; T notifies the attacker about m, and T directly sends m to Bob. This trusted machine is called the *ideal functionality*.

Another example would be a protocol that establishes a secret channel: an attacker controlling the network can see the length of the message and prevent the message from reaching its destination. The ideal functionality for such a secret channel protocol leaks nothing more than the length of the message to the attacker and gives the attacker no more interaction capabilities than control over the delivery of messages.

In the UC framework, a *network* is modelled as a set of interactive Turing machines (ITMs). Every ITM has a unique identity. The recipient of a message is addressed by the identity of that ITM. We say a network S is executable if it contains an ITM $\mathcal{Z}$ with distinguished input and output tape and with the special identity `env`. An execution of S with input $z \in \{0,1\}*$ and security parameter $k \in \mathbb{N}$ is the following execution of the network: Initially, $\mathcal{Z}$ is activated with the message z on its input tape. Whenever an ITM

[2]Recall that T and Alice are directly connected, as well as T and Bob.

$M_1 \in S$ finishes an activation with an outgoing message m (tagged with the identity of the receiver $M_2 \in S$) on its outgoing communication tape, M_2 is invoked with m (tagged with the identity of the sender M_1) on its incoming communication tape. If an ITM terminates its activation without an outgoing message or sends a message to a non-existing ITM, the distinguished ITM $\mathcal{Z}$ is activated. The execution of the network terminates whenever $\mathcal{Z}$ writes a message on its distinguished output tape.

A network S may also contain a distinguished ITM $\mathcal{A}$, called the adversary. Sending a message over a public channel is modelled as sending a message to the adversary. A *UC protocol* is a network without an adversary and an environment. Corrupted parties are modelled as ITMs that directly forward all messages from and to the adversary. Moreover, the adversary can read all tapes of a corrupted party.

In the UC framework, it is possible to model a port-based communication, i.e., each party has finitely many ports on which it can send (outgoing ports $!p$) or receive (incoming ports $?p$) messages. In particular, in a session r every party i has an environment port $?\text{in}^e_{i,r}$, an environment port $!\text{out}^e_{i,r}$, an adversary port $?\text{in}^a_{i,r}$, and an adversary port $!\text{out}^a_{i,r}$. In all cases, port $?p$ is connected to some port $!p$ and vice-versa. Hence, the incoming environment port $?\text{in}^e_{i,r}$ and the outgoing environment port $!\text{out}^e_{i,r}$ are connected to an outgoing port $!\text{in}^e_{i,r}$ and an incoming port $?\text{out}^e_{i,r}$ of the environment, respectively. Similar reasoning holds for the adversary ports $?\text{in}^a_{i,r}$, $!\text{in}^a_{i,r}$, $?\text{out}^a_{i,r}$, and $?\text{out}^a_{i,r}$.

As previously mentioned, in the UC framework the security of a protocol is determined by comparison with the ideal functionality. Intuitively, we say that a protocol ρ UC-realizes τ if there exists a simulator such that the interaction of ρ with the adversary and the interaction of τ with the simulator are indistinguishable.

Definition 38 (Secure realization (UC) [Can01])**.** *Let ρ and τ be protocols. We say that ρ UC-realizes τ if for any polynomial-time adversary $\mathcal{A}$ there exists a polynomial-time adversary $\mathcal{S}$ such that for any polynomial-time environment $\mathcal{Z}$ the networks $\rho \cup \{\mathcal{A}, \mathcal{Z}\}$ (called the real model) and $\tau \cup \{\mathcal{S}, \mathcal{Z}\}$ (called the ideal model) are indistinguishable.*[3]

4.4.1.2. From an SMPC Process to an Ideal Functionality

For constructing an ideal functionality for an SMPC process, we first construct an algorithm $F_\mathcal{F}$ (where $\mathcal{F}$ denotes the SMPC process) that internally runs the computational π-execution, and then define an ideal functionality that internally runs this algorithm $F_\mathcal{F}$.

Constructing the algorithm $F_\mathcal{F}$. We construct for each $\mathcal{F}$ an algorithm $F_\mathcal{F}$ that emulates the computational π execution, i.e., $F_\mathcal{F}$ additionally expects a state as input and outputs an updated state. For the construction of $F_\mathcal{F}$, we assume a fixed and efficiently computable reduction strategy S. This reduction strategy acts as the adversary, i.e., it constitutes an interactive machine (though in this construction the interaction as well as the interactive machines are emulated by $F_\mathcal{F}$).

Recall that the **SMPC** process contains internal loop channels $inloop_i$ for ensuring that every party only accepts a new input if an output has been delivered. The ideal functionality performs these checks directly; hence, we can omit these loop channels and we get (for

[3] Recall that the execution terminates whenever the environment writes on its output tape. We say that two executions E, E' are indistinguishable if $|\Pr[E = 1] - Pr[E' = 1]|$ is negligible.

$n := |\underline{in}|$)

$$\mathcal{F}[\overline{in_1}\langle y_1\rangle \mid \ldots \mid \overline{in_n}\langle y_n\rangle].$$

As $\mathcal{F}$ expects the inputs over the local input channels lin_i, we place processes $\overline{lin_i}\langle x_i\rangle$ in parallel and set $\eta(x_i)$ to be the ith input. Finally, in order to determine which message S schedules to be sent as an output, we place processes $in_i(y'_i, sid)$ in parallel and set the ith output to be $\hat{\eta}(y'_i)$, where $\hat{\eta}$ is the variable mapping of the emulated execution after S finished.

We assume an efficient encoding for processes and the name and variable mappings, μ and η. Let $n := |\underline{in}|$; then, $F_{\mathcal{F}}$ performs on input $(m_1, \ldots, m_n, state)$ the following steps:

1. If *state* is empty, set

   $$\textbf{compute\_state}_0 := \mathcal{F}[\overline{in_1}\langle y_1\rangle \mid \ldots \mid \overline{in_n}\langle y_n\rangle]$$

   and $\mu := \eta := \emptyset$, where $\underline{y'}$ are fresh variables. Otherwise, try to extract a process P, a name mapping μ', and a variable mapping η' out of *state*. If this extraction fails abort; otherwise, set $\textbf{compute\_state}_0 := P$, $\mu := \mu'$, and $\eta := \eta'$. In both cases, with an empty state and a non-empty state, extend the variable mapping $\eta \cup \{x_1 := m_1, \ldots, x_n := m_n\}$ and set

   $$\begin{aligned}\textbf{compute\_state} := \textbf{compute\_state}_0 \mid \overline{lin_1}\langle x_1\rangle \mid \ldots \mid \overline{lin_n}\langle x_n\rangle \\ \mid in_1(y'_1) \mid \ldots \mid in_n(y'_n).\end{aligned}$$

2. Run the *Main loop* of the computational execution $\textsc{Exec}^{\pi}_{\textbf{compute\_state},S}$ with the name mapping μ and the variable mapping η.
3. Let $\textbf{compute\_state}'$ be the process to which $\textbf{compute\_state}_0$ has been reduced after the reduction strategy S terminated. Let Let and $\hat{\eta}$ and $\hat{\mu}$ be the variable and name mappings at that point. Let $state'$ be the encoding of $\hat{\eta}$, $\hat{\mu}$ and $\textbf{compute\_state}'$, and output $(\hat{\eta}(y'_1), \ldots, \hat{\eta}(y'_n), state')$. If $\eta(y'_i)$ is undefined, output a distinguished error symbol.

The Ideal Functionality for SMPC in the UC framework. In Construction 1 (and depicted in Figure 4.5), we construct a generic ideal functionality $\mathcal{I}_{sid,F,c}$ that serves as an abstraction of secure multi-party computations. This construction is parametric over the session identifier *sid*, the function F to be computed, and the corruption scenario c[4]. This function is stateful, it expects the SMPC inputs and a state, and it outputs the result and an updated state.

The ideal functionality receives the (secret) input message of party i from port $?\text{in}^{e}_{i,sid}$ along with a session identifier. The input message is stored in the variable x_i and the state $state_i$ of i is set to `input`. Both the session identifier and the length of the received message are leaked to the adversary, since in the actual implementation each party announces the session identifier and commits to its own private input, which leaks the length of that input (see part (a) of Construction 1). In addition, the (ideal) adversary is allowed to schedule the delivery of the results to party i, which is achieved by letting the adversary interact with the ideal functionality on port $?\text{in}^{a}_{i,sid}$ (see part (b) of Construction 1). Since the ideal functionality might be reactive (i.e., the computation might involve different execution rounds), our construction additionally keeps a round counter for each participant i, denoted by r_i.

[4] $c = (c_1, \ldots, c_n) \in \{0,1\}^*$ where party i is corrupted if $c_i = 1$ and uncorrupted if $c_i = 0$.

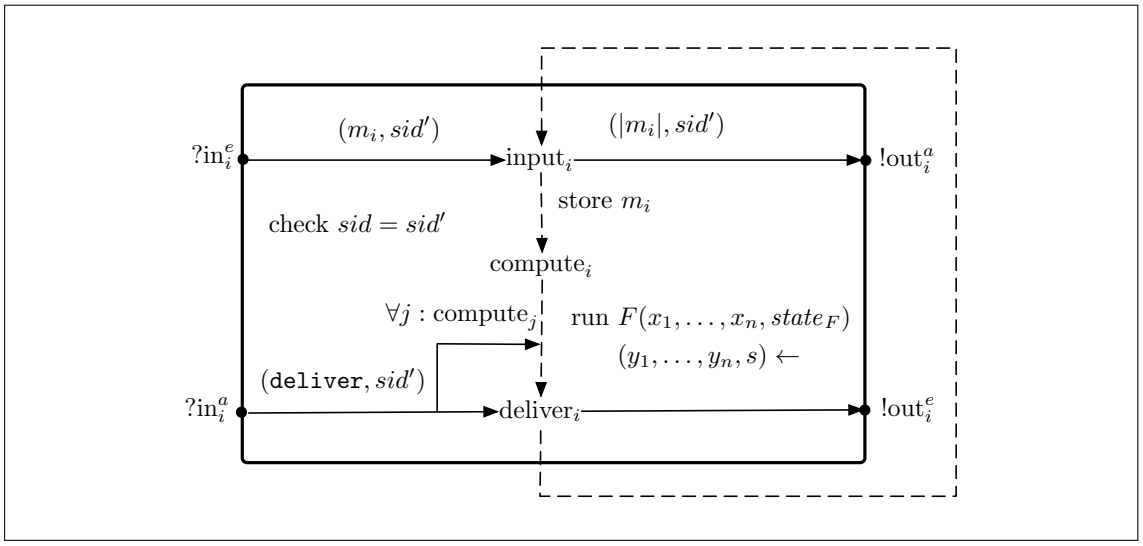

Figure 4.5.: The ideal functionality of SMPC

Construction 1 (SMPC Ideal functionality)**.** *We construct an interactive polynomial-time machine $\mathcal{I}_{sid,F}$, called* the SMPC ideal functionality*, which is parametric over a session identifier sid and a poly-time algorithm F. Initially, the variables of $\mathcal{I}_{sid,F}$ are instantiated as follows: $\forall i \in [1,n].state_i := \texttt{input}$, $r_i := 1$, and $state_F := \emptyset$. Upon an activation with message m on port p, $\mathcal{I}_{sid,F}$ behaves as follows.*

(*a*) Upon $(?\text{in}^e_{i,sid}, (m, sid'))$ *If $sid = sid'$ and $state_i = \texttt{input}$, then set $state_i := \texttt{compute}$ and $x_i := m$. If $state_i = \texttt{input}$, then send $(sid', |m|)$ on port $?\text{in}^a_{i,sid}$.*

(*b*) Upon $(?\text{in}^a_{i,sid}, (\texttt{deliver}, sid'))$*, if $\forall j \in [1,n].state_j = \texttt{compute}$ and $r_i = r_j$, then compute $(y_1, \ldots, y_n, s) \leftarrow F(x_1, \ldots, x_n, state_F)$ and set $state_F := s$ and $\forall j \in [1,n].state_j := \texttt{deliver}$.*
If $state_i = \texttt{deliver}$, set $r_i := r_i + 1$, $state_i := \texttt{input}$ and send y_i on port $!\text{out}^e_{i,sid}$.

Since we consider static corruptions in this paper, the ports $!\text{out}^e_{i,sid}$ and $?\text{in}^e_{i,sid}$ of the ideal functionality are redirected to the adversary whenever a party i is corrupted, as this party is under the control of the attacker. Such a port redirection is technically achieved by using dummy parties that act as message forwarders. We model the corruption scenario by a tuple $c_1, \ldots, c_n$ of bits, where $c_i = 1$ if and only if party i is corrupted. In the following, we denote by $\mathcal{I}_{sid,F,c}$ the UC protocol composed of the ideal functionality $\mathcal{I}_{sid,F}$ and the dummy parties required to connect the adversary to ports $!\text{out}^e_{i,sid}$ and $?\text{in}^e_{i,sid}$ if $c_i = 1$.

The function F essentially plays the same role in the ideal functionality as the SMPC-suited context $\mathcal{F}$ in our symbolic abstraction of SMPC.

4.4.2. Computational execution of a process

Since the applied π-calculus only has semantics in the symbolic model (without probabilities and without the notion of a computational adversary), we need to introduce a notion of computational execution for applied π-calculus processes. Our computational implementation of a symbolic protocol P builds on the computational execution of the applied π-calculus

that has been proposed in [BHU09]. This is a probabilistic polynomial-time algorithm that expects as input the symbolic protocol Q, a set of deterministic polynomial-time algorithms $\underline{A}$ for the constructors and destructors in Q, and a security parameter k. This algorithm executes the protocol by interacting with a computational adversary. In the operational semantics of the applied π calculus, the reduction order is non-deterministic. This non-determinism is resolved by letting the adversary determine the order of the reduction steps. The computational execution sends the process to the adversary and expects a selection for the next reduction step. In the following, we follow the convention that "fresh variable" or "fresh name" means a variable or name that does not occur in any of the variables maintained by the algorithm. The computational execution tightly follows the semantics of the applied π-calculus, with the exception that it operates on bitstrings and does not instantiate names and variables in the process but rather maintains an environment η that stores the bitstrings assigned to the free variables in P, and an interpretation μ of the free names in P as bitstrings. Given η and μ, we can computationally evaluate a term or a destructor application D to a bitstring $\mathrm{ceval}_{\eta,\mu,k,\underline{A}} D$ by using the algorithms $\underline{A}$ for the constructors and destructors in D. (We will often omit k and $\underline{A}$ for readability if these are clear from the context.) We set $\mathrm{ceval}_{\eta,\mu} D := \bot$ if the application of one of the algorithms in $\underline{A}$ fails. In abuse of notation, in the following we will use evaluation contexts with distinguished multiple holes.

The computational execution together with the adversary $\mathcal{A}$ constitutes a network. The adversary, however, can be seen as the environment; if the execution $\textsc{Exec}^{\pi}$ stops, $\mathcal{A}$ is activated.

Definition 39 (Computational π-execution)**.** *Let Q be a closed process. Let Adv be an interactive machine called the adversary. We define the computational π-execution as an interactive machine $\textsc{Exec}^{\pi}_{Q,\underline{A}}(1^k)$ that takes a security parameter k as argument and interacts with Adv:*

- Start*: Let P be obtained from Q by deterministic α-renaming so that all bound variables and names in Q are distinct. Let η and μ be a totally undefined partial functions from variables and names, respectively, to bitstrings. Let $a_1, \ldots, a_n$ denote the free names in P. For each i, pick $r_i \in \mathrm{Nonces}_k$ at random. Set $\mu := \mu \cup \{a_1 := r_1, \ldots, a_n := r_n\}$. Send $(r_1, \ldots, r_n)$ to Adv.*[5]
- Main loop: *Send P to the adversary and expect an evaluation context E from the adversary. Distinguish the following cases:*
  - *$P = E[M(x).P_1]$: Request two bitstrings c, m from the adversary. If $c = \mathrm{ceval}_{\eta,\mu} M$, set $\eta := \eta \cup \{x := m\}$ and $P := E[P_1]$.*
  - *$P = E[\nu a.P_1]$: Pick $r \in \mathrm{Nonces}_k$ at random, set $P := E[P_1]$ and $\mu := \mu(a := r)$.*
  - *$P = E[\overline{M_1}\langle N\rangle.P_1][M_2(x).P_2]$: If $\mathrm{ceval}_{\eta,\mu} M_1 = \mathrm{ceval}_{\eta,\mu} M_2$, then set $P := E[P_1][P_2]$ and $\eta := \eta \cup \{x := \mathrm{ceval}_{\eta,\mu} N\}$.*
  - *$P = E[\mathit{let}\ x = D\ \mathit{in}\ P_1\ \mathit{else}\ P_2]$: If $m := \mathrm{ceval}_{\eta,\mu} D \neq \bot$, set $\eta := \eta \cup \{x := m\}$ and $P := E[P_1]$; Otherwise set $P := E[P_2]$.*
  - *$P = E[$`assert` $F.P_1]$: Let $P := E[P_1]$ and raise the tuple $((F_1, \ldots, F_n), F, \eta, \mu, P)$ where* `assume` *F_1, ...,* `assume` *F_n are the top level assumptions*[6] *in P.*
  - *$\tilde{P} = E[!Q]$: Let Q' be obtained from Q by deterministic α-renaming so that all*

[5] In the applied π-calculus, free names occurring in the initial process represent nonces that are honestly chosen but known to the attacker.

[6] `assume` F is top-level in P if there exists a context E such that $P = E[$`assume` $F]$.

bound variables and names in Q' are fresh. Set $P := E[Q'|!Q]$.
- *$P = E[\overline{M}\langle N\rangle.P_1]$: Request a bitstring c from the adversary. If $c = \mathrm{ceval}_{\eta,\mu} M$, set $P := E[P_1]$ and send $\mathrm{ceval}_{\eta,\mu} N$ to the adversary.*
- *In all other cases, do nothing.*

For any interactive machine $\mathcal{A}$, we define $\text{EXEC}^{\pi}_{Q,\underline{A},\mathcal{A}}(1^k)$ as the interaction between $\text{EXEC}^{\pi}_{Q,\underline{A}}(1^k)$ and $\mathcal{A}$; the output of $\text{EXEC}^{\pi}_{Q,\underline{A},\mathcal{A}}(1^k)$ is the output of $\mathcal{A}$.

We let $Assertions^{\pi}_{P,\underline{A},p,\mathcal{A}}(k)$ denote the distribution of sequences of assertion tuples of the form $((F_1,\ldots,F_n),F,\eta,\mu,P)$ raised in an interaction of $\text{EXEC}^{\pi}_{P,\underline{A},\mathcal{A}}(1^k)$ within the first $p(k)$ computation steps (jointly counted for $\mathcal{A}(1^k)$ and $\text{EXEC}^{\pi}_{P,\underline{A},\underline{\tau}}(1^k)$).

When applied to a protocol built on our abstraction of SMPC, the execution EXEC^{π} executes the abstraction $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$, which corresponds to a trusted host performing an ideal computation. Our computational soundness result, however, has to hold for an arbitrary protocol that incorporates an actual secure multi-party computation protocol.

We thus introduce the notion of SMPC implementation *Exec*, which differs from EXEC^{π} in that the SMPC protocol is executed instead of the abstraction $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$. *Exec* takes as input the security parameter, a process Q, the algorithms $\underline{A}$ for the constructors and destructors in Q, and a family τ of UC protocols, one for each of the SMPC in Q. Intuitively, *Exec* is meant to act as an interface between the adversary and the UC protocol, which can be either the ideal functionality or the actual SMPC protocol (i.e., τ will be either a family of ideal protocols or a family of real protocols, respectively).

The behavior of *Exec* is depicted in Figure 4.6, where $sidc(sid).\mathbf{SMPC}'(adv, \underline{in}, \mathcal{F}) := \mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$. The adversary may (i) initialize the secure multi-party computation, in which case a session identifier r is sent over the channel $sidc$; (ii) schedule the delivery of the output to some honest party, in which case the process and the environment η are updated accordingly; (iii) schedule the input of some honest party i, in which case this input is sent to the UC protocol over the port $!\mathrm{in}^e_{i,r}$; (iv) schedule the input of some dishonest party i, in which case this input is sent to the UC protocol over the port connected to $!\mathrm{in}^e_{i,r}$; and (v) send a message to party i, in which case this message is forwarded to port $!\mathrm{in}^a_{i,r}$. In all these cases, except for (i) and (ii), the computational execution interacts with the protocol and the protocol answers with a message m. If m is sent over the outgoing port of some honest party i (i.e., it is received from port $?\mathrm{out}^e_{i,r}$), then m is stored in a buffer waiting for delivery, otherwise m is directly forwarded to the attacker.

The execution *Exec* together with the adversary $\mathcal{A}$ and all the protocol parties for any session constitute a network. Each session r induces a subnetwork comprising the computational execution *Exec* and the protocol parties for the session r. In this session r the execution *Exec* plays the role of both the environment and the adversary, respectively, listening on the ports $?\mathrm{out}^e_{i,r}$ and $?\mathrm{out}^a_{i,r}$ $(i \in [1,n])$. In particular, *Exec* is activated if no other machine in the subnetwork, composed of the is active anymore. If the execution *Exec* terminates without having sent a message to a party, the adversary is activated.

Definition 40 (Computational SMPC execution)**.** *Let Q and $\underline{A}$ be as in Definition 39. Let τ denote a family of UC protocols (intuitively, this family is composed of the implementations of the SMPC protocols for each of the SMPC-suited contexts $\mathcal{F}$, the session identifiers r, and the corruption scenarios c, which we denote by $\tau_{r,\mathcal{F},c}$). We define the interactive*

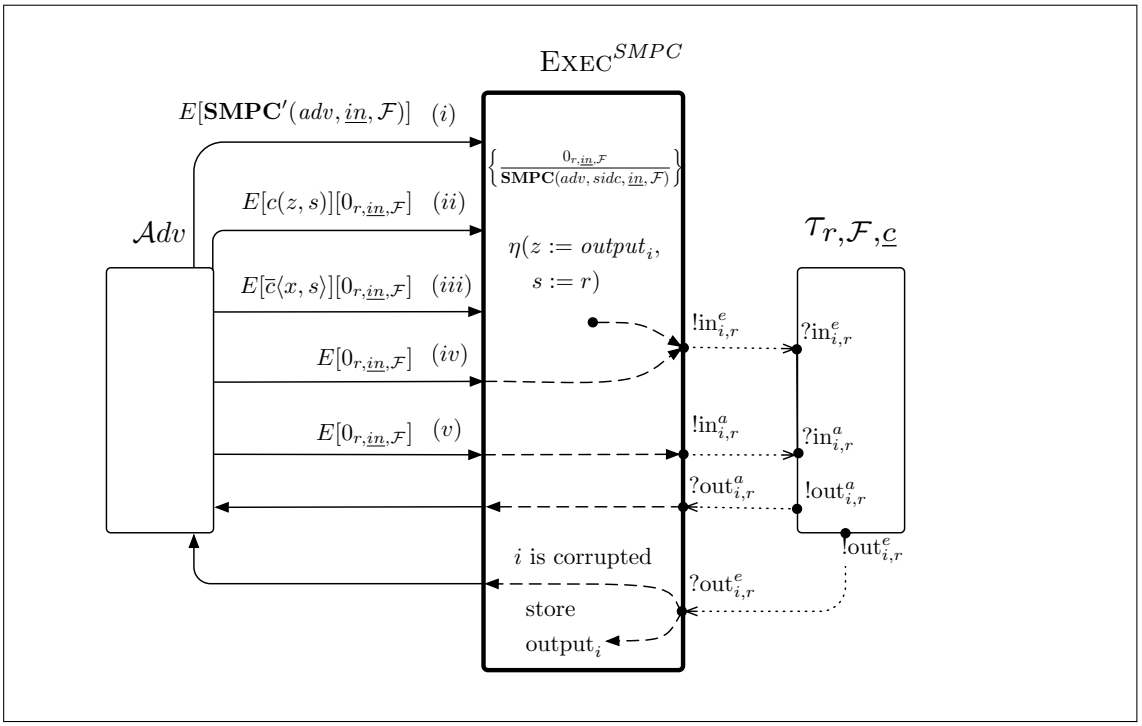

Figure 4.6.: The computational SMPC execution, where $r := \mu(sid)$ and $sidc(sid).\mathbf{SMPC}'(adv, \underline{in}, \mathcal{F}) := \mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$.

machine $Exec_{Q,\underline{A},\tau}(1^k)$ *by modifying the main loop of* $\text{EXEC}^{\pi}_{Q,\underline{A}}(1^k)$ *(see Definition 39) as follows:*

- (*i*) *(the secure multi-party computation is initialized)* $P = E[\mathbf{SMPC}'(adv, \underline{in}, \mathcal{F})]$. *Set* $r := \text{ceval}_{\eta,\mu}(M)$, $\eta := \eta \cup \{sid := r\}$, *and* $P := E[0_{r,\underline{in},\mathcal{F}}]$.[7]
- (*ii*) *(the output of some honest party is delivered)* $P = E[c(z,s).Q][0_{r,\underline{in},\mathcal{F}}]$ and $output^r_i = m \neq \bot$, and $\exists i \in [1,n] : \text{ceval}_{\eta,\mu}\, in_i = \text{ceval}_{\eta,\mu}\, c$:
  Set $\eta := \eta \cup \{z := m, s := r\}$, *set* $output^r_i := \bot$, *and* $P := E[Q][0_{r,\underline{in},\mathcal{F}}]$.
- (*iii*) *(the input of some honest party is scheduled)* $P = E[\overline{c}\langle x,s\rangle.Q][0_{r,\underline{in},\mathcal{F}}]$ and $\exists i \in [1,n] : \text{ceval}_{\eta,\mu}\, in_i = \text{ceval}_{\eta,\mu}\, c$:
  Let $m := \text{ceval}_{\eta,\mu}(x,s)$ *and send* m *to* $\tau_{r,\mathcal{F},c}$ *over the port* $!\text{in}^e_{i,r}$.[8]
- (*iv*) *(the input of some dishonest party is scheduled)* $P = E[0_{r,\underline{in},\mathcal{F}}]$ *Request a bitstring* m *from the adversary. If* $m = (ch, r, m')$ *and* $\eta(in_i) = ch$ *for some* $i \in [1,n]$, *send* m' *over the port* $!\text{in}^e_{i,r}$.
- (*v*) *(the adversary communicates with the protocol)* $P = E[0_{r,\underline{in},\mathcal{F}}]$ *Request a bitstring* m *from the adversary. If* $m = (i, r, m')$[9] *then send* m' *over the port* $!\text{in}^a_{i,r}$.

In addition, in all cases but (*i*) *and* (*ii*), *whenever a message* m' *is received over a port* $?\text{out}^e_{i,r}$ *with* i *not being corrupted, set* $output^r_i := m'$; *whenever a message* m' *is received over* $?\text{out}^a_{i,r}$ *or over* $?\text{out}^e_{i,r}$ *for a corrupted* i, (m', i) *is forwarded to the adversary. Moreover,*

[7] The SMPC abstraction is replaced by the dummy process $0_{r,\underline{in},\mathcal{F}}$, which is tagged with the session identifier r, the input channels $\underline{in}$, and the function to be computed $\mathcal{F}$.
[8] $!\text{in}^e_{i,r}$ is connected to the incoming port $?\text{in}^e_{i,r}$ of party i in $\tau_{r,\mathcal{F},c}$
[9] We implicitly assume that $\forall i \in [1,n].\mu(in_i) \notin [1,n]$.

we do not send P to the adversary but the erasure of P in which all $0_{r,\underline{in},\mathcal{F}}$ are replaced by 0.

For any interactive machine $\mathcal{A}$, we define $Exec_{Q,\underline{A},\tau,\mathcal{A}}(1^k)$ as the interaction between $Exec_{Q,\underline{A},\tau}(1^k)$ and $\mathcal{A}$; the output of $Exec_{Q,\underline{A},\tau,\mathcal{A}}(1^k)$ is the output of $\mathcal{A}$.

We let $Assertions^{\mathrm{SMPC}}_{P,\underline{A},\tau,p,\mathcal{A}}(k)$ denote the distribution of sequences of assertion tuples of the form $((F_1,\ldots,F_n),F,\eta,\mu,P)$ raised in an interaction of $Exec_{P,\underline{A},\tau,\mathcal{A}}(1^k)$ within the first $p(k)$ computation steps (jointly counted for $\mathcal{A}(1^k)$ and $Exec_{P,\underline{A},\tau}(1^k)$).

4.5. Computational soundness

Finally, in Section 4.5.6, we extend the embedding of the applied π-calculus into CoSP in order to make our result compatible with all known CoSP results.

4.5.1. Computational safety

For defining computational safety, we first need to recall the logic for the security policies that we are considering. The logic has to fulfill some standard properties such as monotonicity, closure under substitution, and it should allow the replacement of equals by equals.

We assume a set $\mathcal{D}_s$ of deduction rules in the sequent calculus that define a deduction relation $\vdash_s$.[10] Let Γ be a set of formulas in the logic. We say that a formula F is entailed by a premise Γ, denoted as $\Gamma \models_s F$, if there is a proof tree (using $\mathcal{D}_s$) such that the conclusion of the deduction rule at the root of the tree is $\Gamma \vdash_s F$.The finite depth of a proof tree is exploited in the proof of Lemma 24.

Following the approach of Backes, Hritcu and Maffei [BHM08b], we characterize destructor application tests by introducing for each destructor an uninterpreted function symbol $d^{\#}$ and a predicate Red such that $\mathrm{Red}(d^{\#}(M_1,\ldots,M_n),M)$ holds true only if $d(M_1,\ldots,M_n)=M$ holds true, where M_i, M are terms.[11] We could assume $\mathcal{D}_s$ to contain the following deduction rule:

$$\frac{d(M_1,\ldots,M_n)=M}{\vdash_s \mathrm{Red}(d^{\#}(M_1,\ldots,M_n)=M}.$$

For the sake of a better presentation of the computational entailment relation, however, we want to keep the names of the variables that have been replaced by terms. Therefore, we introduce a mapping *eval* from variables to terms that just stores which variable has been replaced by which term in the current process. Then, we represent the rule as

$$\frac{d(eval(v_1),\ldots,eval(v_n))=M}{\vdash_s \mathrm{Red}(d^{\#}(v_1,\ldots,v_n)=v},$$

with v_i, v being variables only.

[10] We refer the interested reader to [GTL89]. The sequence calculus has been introduced by Gentzen in 1934 [Gen34].

[11] One minor difference is that in our setting destructors are partial functions and in [BHM08b] destructors are defined via a reduction relation; in particular, destructors might be non-deterministic, i.e., relations.

Moreover, we assume $\mathcal{D}_s$ to contain rule for universal quantification:

$$\frac{\Gamma, C\left\{\frac{x'}{x}\right\} \vdash_s \underline{B}}{\Gamma, \forall x.C \vdash_s \underline{B}}\ \mathrm{L}\forall \qquad \frac{\Gamma \vdash_s C\left\{\frac{x'}{x}\right\}, \underline{B}}{\Gamma \vdash_s \forall x.C, \underline{B}}\ \mathrm{R}\forall,$$

where x' is a fresh variable. And, we assume analogous rules for existential quantification.

The computational entailment relation $\models_{\eta,\mu,\underline{A}}$ is based on the symbolic entailment relation $\models_s$. We define the computational entailment relation by the same inference rules with the only difference that universal quantification indeed quantifies over all bitstrings and all destructor application tests $\mathrm{Red}(d^{\#}(v_1,\ldots,v_n), v)$ correspond to the check $A_d(\mathrm{ceval}_{\eta,\mu}\, v_1, \ldots, \mathrm{ceval}_{\eta,\mu}\, v_n) = \mathrm{ceval}_{\eta,\mu}\, v$, where A_d is the computational implementation of d.

Definition 41 (Computational entailment $\models_{\eta,\mu,\underline{A}}$)**.** *Let $\vdash_s$ be a symbolic entailment relation that is inductively defined by a set of inference rules $\mathcal{D}_s$. Let η and μ be variable and name mappings, and let $\underline{A}$ be implementations. We define a set of inference rules $D_{\eta,\mu,\underline{A}}$ as D_s with the following modifications:*

(a) *Replace* $\vdash_s$ *by* $\vdash_{\eta,\mu,\underline{A}}$,

(b) *replace* $\dfrac{d(eval(v_1),\ldots,eval(v_n)) = eval(v)}{\vdash_s d(v_1,\ldots,v_n) = v}$

by $\dfrac{A_d(\mathrm{ceval}_{\eta,\mu}\, v_1,\ldots,\mathrm{ceval}_{\eta,\mu}\, v_n) = \mathrm{ceval}_{\eta,\mu}\, v}{\vdash_{\eta,\mu,\underline{A}} d(v_1,\ldots,v_n) = v}$,

(c) *replace* $\dfrac{\Gamma, C\left\{\frac{x'}{x}\right\} \vdash_s \underline{B}}{\Gamma, \forall x.C \vdash_s \underline{B}}$ *by* $\dfrac{\forall b \in \{0,1\}^* : \Gamma, C\left\{\frac{x'}{x}\right\} \vdash_{\eta\cup\{x':=b\},\mu,\underline{A}} \underline{B}}{\Gamma, \forall x.C \vdash_{\eta,\mu,\underline{A}} \underline{B}}$, *and*

(d) *replace* $\dfrac{\Gamma \vdash_s C\left\{\frac{x'}{x}\right\}, \underline{B}}{\Gamma \vdash_s \forall x.C, \underline{B}}$ *by* $\dfrac{\forall b \in \{0,1\}^* : \Gamma \vdash_{\eta\cup\{x':=b\},\mu,\underline{A}} C\left\{\frac{x'}{x}\right\}, \underline{B}}{\Gamma \vdash_{\eta,\mu,\underline{A}} \forall x.C, \underline{B}}$.

We say that a formula F is entailed *by a premise Γ, denoted as $\Gamma \models_{\eta,\mu,\underline{A}} F$, if there is a proof tree of finite depth (using $\mathcal{D}_{\eta,\mu,\underline{A}}$) such that the conclusion of the deduction rule at the root of the tree is $\Gamma \vdash_{\eta,\mu,\underline{A}} F$.*

For our computational soundness result it is important that an asserted or assumed formula does quantify over arbitrary messages but only over protocol messages. This is formalized by requiring that for any quantified variable x there is a predicate P that holds x as an argument. We call such formulas well-formed.

Definition 42 (Well-formed formula)**.** *Let $\models_s$ be the entailment relation and $\mathcal{D}_s$ the set of deduction rules from above. Let x, x', x_i denote variables, let p denote a predicate, and d a destructor. Let F be formula over predicates $p(x_1,\ldots,x_m)$ and destructor application tests $d(x_1,\ldots,x_m) = x'$. We say that F is* well-formed *iff F has a proof tree of finite depth (using $\mathcal{D}_s$) and for all subformulas F' of F we have that*

- *if $F' = \forall x.F''$, we have $F'' = \forall x.p(\ldots,x,\ldots) \Rightarrow F'''$, and*
- *if $F' = \exists x.F''$, we have $F'' = \exists x.p(\ldots,x,\ldots) \wedge F'''$.*

A note on well-formed formulas. This well-formedness condition might seem heavily restrictive, but almost every formula can be easily converted into a well-formed formula. For example, the general security policy

$$\forall \underline{id}, \underline{x}, sid, z.\big(\wedge_{i=1}^{n} \mathrm{Input}(id_i, x_i, sid) \wedge id_1 \neq \ldots \neq id_n \wedge F_{rel}\big) \Rightarrow \mathrm{P}(z, sid)$$

presented in Section 4.3 can be easily converted into a well-formed formula (given a free predicate p):

$$\begin{aligned}\forall \underline{id}, \underline{x}, sid, z.\big(&\wedge_{i=1}^{n} p_i^{id}(id) \wedge_{i=1}^{n} p_i(x) \wedge p^{sid}(sid) \wedge p^{mcp}(z) \\ &\wedge_{i=1}^{n} \mathrm{Input}(id_i, x_i, sid) \wedge id_1 \neq \ldots \neq id_n \wedge F_{rel}\big) \Rightarrow \mathrm{P}(z, sid).\end{aligned}$$

This formula additionally requires that in the process at an appropriate place contains `assume` $p^a(m)$ for each $p^a(m)$ that occurs in the formula.

For the computational soundness proof, we require all formulas to be well-formed. Moreover, in tight correspondence to the UC model, we require session identifiers to be unique. In addition, as we only consider static corruption, we need to require that the private channels in_i (occurring in an SMPC occurrence $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$) are never sent over a public channel. Hence, we require that a process only contains well-formed formulas and for all subprocess $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ that the context $\mathcal{F}$ is SMPC-suited (see Definition 37), the channels in_i are never sent over a public channel, every session identifier is sent at most once over a channel $sidc$.

For the next definition, we need the following notation. We call a channel $sidc$ that occurs in an SMPC subprocess $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ process a session identifier channel and a channel in_i that occurs in such an SMPC subprocess a party channel. A *public channel* is a channel that is either free or a channel that has been sent over a public channel.

Definition 43 (Well-formed processes)**.** *A process P is* well-formed *if the following conditions hold:*

1. *For all asserts* `assert` F *in P the formula F is a predicate and for all assumes* `assume` F *the formula F is a well-formed formula.*
2. *For all subprocesses $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ of P, F is an SMPC-suited context.*
3. *Every session identifier is only sent once over a session identifier channel sidc.*
4. *A non-free party channel in_i (i.e., a term that contains in_i) is never sent over a public channel.*

We call a process an atomic process *if it does not contain any assumptions and only assertions* `assert` *true and* `assert` *false.*

The computational notion of robust safety depends on the computational notion of logical entailment. We now introduce two definitions of robust computational safety, with respect to $\textsc{Exec}^{\pi}$ and *Exec*, respectively.

Definition 44 (Robust computational safety)**.** *Let P be a process, $\underline{A}$ an implementation of the destructors in P, and τ a family of secure multi-party computations. We say that P is π-(resp. SMPC-)robustly computationally safe using $\underline{A}$ (resp. $\underline{A}, \tau$) iff for all*

polynomial-time interactive machines $\mathcal{A}$ *and all polynomials* p,

$\Pr[$*for all* $((F_1,\ldots,F_n),F,\eta,\mu,Q)\in a, \{F_1,\ldots,F_n\} \models_{\eta,\mu,\underline{A}} F : a \leftarrow Assertions^{\pi}_{P,\underline{A},p,\mathcal{A}}(k)]$

(resp.

$\Pr[$*for all* $((F_1,\ldots,F_n),F,\eta,\mu,Q)\in a, \{F_1,\ldots,F_n\} \models_{\eta,\mu,\underline{A}} F : a \leftarrow Assertions^{\mathrm{SMPC}}_{P,\underline{A},\tau,p,\mathcal{A}}(k)]$

)

is overwhelming in k.

A symbolic model is computationally sound if robust safety carries over to the computational setting. This definition is used in our first theorem, which is parameterized over the non-interactive primitives used in the protocol.

Definition 45 (Computationally sound model)**.** *Let* $\underline{A}$ *be a set of constructor and destructor implementations. We say that a symbolic model* $(\mathcal{D},\mathcal{P})$ *is computationally sound using* $\underline{A}$ *iff for all* $P\in\mathcal{P}$ *such that* P *is robustly safe,* P *is* π*-robustly computationally safe using* $\underline{A}$.

Definition 46 (Well-formed symbolic models)**.** *A symbolic model* $(\mathcal{D},\mathcal{P})$ *is called* well-formed *if all processes* $P\in\mathcal{P}$ *are well-formed and* $\mathcal{P}$ *fulfills the following closure properties:*

(*i*) *If* $P\in\mathcal{P}$ *and* P' *is a subprocesses of* P*, then* $P'\in\mathcal{P}$.

(*ii*) *If* $P\in\mathcal{P}$*, then* $\nu n.P\in\mathcal{P}$ *for any name* n.

(*iii*) *If* $P\in\mathcal{P}$ *and* P' *is statically equivalent to* P*, then* $P'\in\mathcal{P}$.

(*iv*) *If* $P\in\mathcal{P}$ *and* P' *is an erasure of* P *obtained by replacing each assumption* `assume` F *with* $\overline{c_{p_1}}\langle v_1\rangle \mid \ldots \overline{c_{p_n}}\langle v_n\rangle$*, where* v_i *is a quantified variable with guards* p_i *in* F*, and replacing each* `assert` F' *by*

$$c_{p'_1}(w_1).\ldots.c_{p'_m}(w_m).\texttt{if } d(v_1,\ldots,v_s)=v \texttt{ then } 0 \texttt{ else assert } \mathit{false}$$

where $d\in\mathcal{D}$ *and* $w_1,\ldots,w_m$ *are quantifies variables with guards* $p'_1,\ldots,p'_m$ *(respectively) that occur in an* `assume` F'' *in* P*, then for this erasure* P' *we have* $\nu\underline{c}.\ \mathcal{P}$*, where* $\underline{c}$ *denotes all freshly introduced channels.*

(*v*) *If* $P\in\mathcal{P}$ *and* $P=P_1|P_2$*, then for* $Q :=$ `if` $d(M)=N$ `then assert` *false* `else` $0|P_2$ *we have* $Q\in\mathcal{P}$ *for any term* M*,* N *in the symbolic model and any destructor* $d\in\mathcal{D}$.

(*vi*) *If* $P\in\mathcal{P}$ *and* $P=P_1|P_2$*, then for* $Q :=$ `if` $d(M)=N$ `then` 0 `else assert` *false*$|P_2$ *we have* $Q\in\mathcal{P}$ *for any term* M*,* N *in the symbolic model and any destructor* $d\in\mathcal{D}$.

4.5.2. Computational soundness for non-interactive primitives

In this section, show that computational soundness for trace properties in CoSP (as defined in [BHU09]) implies computational soundness in our model. Leveraging the embedding of the applied π calculus into CoSP [BHU09], we conclude that there is an implementation $\underline{A}$ using which there is a computational sound abstraction for signatures and encryptions in our model. We extend prior work [BHU09] in the CoSP framework with numbers and arithmetics.

We model that numbers as symbolic bitstrings and arithmetic operations as destructors on these bitstrings. In this way, we enforce a strict separation between numbers and

cryptographic terms, such as signatures and keys. Thus, it is not possible to apply an arithmetic operation on nonces, signatures, or encrypted messages. Due to this separation, the impossibility result for computational soundness of XOR [BP05] does not apply. We review the extended symbolic model in Section 4.5.2.1. We review the corresponding implementation conditions in Section 4.5.2.2. As in [BHU09], we impose some standard conditions on protocols to ensure that all encryptions and signatures are produced by using fresh randomness and that secret keys are not sent around. A protocol satisfying these conditions is called *key-safe*. We review these protocol conditions in Section 4.5.2.3. We denote the resulting symbolic model as $(\mathcal{D}_{\mathrm{ESA}}, \mathcal{P}_{\mathrm{ESA}})$.

After reviewing and extending symbolic model, the implementation conditions, and the protocol conditions, we show in Section 4.5.2.4 how to extend the computational soundness proof to get the following result.

Theorem 4 (Computational soundness of $(\mathcal{D}_{\mathrm{ESA}}, \mathcal{P}_{\mathrm{ESA}})$)**.** *If enhanced trapdoor permutations exist, there is an length-regular implementation $\underline{A}$ such that $(\mathcal{D}_{\mathrm{ESA}}, \mathcal{P}_{\mathrm{ESA}})$ is a computationally sound, well-formed model.*

4.5.2.1. The symbolic model

We first specify the symbolic model $\mathsf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D}, \vdash)$:

- Constructors and nonces: Let $\mathbf{C} := \{enc/3, ek/1, dk/1, sig/3, vk/1, sk/1, pair/2, string_0/1, string_1/1, empty/0, garbageSig/2, garb/1, garbageEnc/2\}$ and $\mathbf{N} := \mathbf{N}_P \cup \mathbf{N}_E$. Here N_P and N_E are countably infinite sets representing protocol and adversary nonces, respectively. Intuitively, encryption, decryption, verification, and signing keys are represented as $ek(r), dk(r), vk(r), sk(r)$ with a nonce r (the randomness used when generating the keys). $enc(ek(r'), m, r)$ encrypts m using the encryption key $ek(r')$ and randomness r. $sig(sk(r'), m, r)$ is a signature of m using the signing key $sk(r')$ and randomness r. The constructors $string_0$, $string_1$, and $empty$ are used to model arbitrary strings used as payload in a protocol (e.g., a bitstring 010 would be encoded as $string_0(string_1(string_0(empty)))$). $garb$, $garbageEnc$, and $garbageSig$ are constructors necessary to express certain invalid terms the adversary may send, these constructors are not used by the protocol.
- Message type: We define $\mathbf{T}$ as the set of all terms M matching the following grammar:

$$\begin{aligned} M ::= {} & enc(ek(N), M, N) \mid ek(N) \mid dk(N) \mid sig(sk(N), M, N) \mid vk(N) \mid sk(N) \mid \\ & pair(M, M) \mid S \mid N \mid garb(N) \mid garbageEnc(M, N) \mid garbageSig(M, N) \\ S ::= {} & empty \mid string_0(S) \mid string_1(S) \end{aligned}$$

  where the nonterminal N stands for nonces.
- Destructors: $\mathbf{D} := \{dec/2, isenc/1, isek/1, ekof/1, verify/2, issig/1, isvk/1, vkof/2, fst/1, snd/1, unstring_0/1, unstring_1/1, equals/2, add/2, sub/2, mult/2, ge/2, xor\}$. The destructors $isek$, $isvk$, $isenc$, and $issig$ realize predicates to test whether a term is an encryption key, verification key, ciphertext, or signature, respectively. $ekof$ extracts the encryption key from a ciphertext, $vkof$ extracts the verification key from a signature. $dec(dk(r), c)$ decrypts the ciphertext c. $verify(vk(r), s)$ verifies the signature s with respect to the verification key $vk(r)$ and returns the signed message if successful. The destructors fst and snd are used to destruct pairs, and

$$
\begin{aligned}
unstring_0(string_0(s)) &= s \\
unstring_1(string_1(s)) &= s \\
normalize(s) &= \bar{\iota}(\iota(s)) \\
add(string_i(s), string_j(s')) &= \bar{\iota}(\iota(string_i(s)) + \iota(string_j(s'))) \\
sub(string_i(s), string_j(s')) &= \bar{\iota}(\iota(string_i(s)) - \iota(string_j(s'))) \\
mult(string_i(s), string_j(s')) &= \bar{\iota}(\iota(string_i(s)) \cdot \iota(string_j(s'))) \\
ge(string_i(s), string_j(s')) &= string_i(s) \text{ if } \iota(string_i(s)) \geq \iota(string_j(s')) \\
xor(string_i(s), empty) &= string_i(s) \\
xor(empty, string_i(s)) &= string_i(s) \\
xor(string_i(s), string_i(s')) &= string_0(xor(s, s')) \\
xor(string_i(s), string_{1-i}(s')) &= string_1(xor(s, s')) \\
&\qquad \text{where } i, j \in \{0, 1\}
\end{aligned}
$$

$$
\begin{aligned}
dec(dk(t_1), enc(ek(t_1), t_2, t_3)) &= t_2 \\
isenc(enc(ek(t_1), t_2, t_3)) &= enc(ek(t_1), t_2, t_3) \\
isenc(garbageEnc(t_1, t_2)) &= garbageEnc(t_1, t_2) \\
isek(ek(t)) &= ek(t) \\
ekof(enc(ek(t_1), m, t_2)) &= ek(t_1) \\
ekof(garbageEnc(t_1, t_2)) &= t_1 \\
verify(vk(t_1), sig(sk(t_1), t_2, t_3)) &= t_2 \\
issig(sig(sk(t_1), t_2, t_3)) &= sig(sk(t_1), t_2, t_3) \\
issig(garbageSig(t_1, t_2)) &= garbageSig(t_1, t_2)
\end{aligned}
\qquad
\begin{aligned}
isvk(vk(t_1)) &= vk(t_1) \\
vkof(sig(sk(t_1), t_2, t_3)) &= vk(t_1) \\
vkof(garbageSig(t_1, t_2)) &= t_1 \\
fst(pair(t_1, t_2)) &= t_1 \\
snd(pair(t_1, t_2)) &= t_2 \\
equals(t_1, t_1) &= t_1
\end{aligned}
$$

Figure 4.7.: Destructor Rules

$$
\frac{m \in S}{S \vdash m} \qquad \frac{N \in \mathbf{N}_E}{S \vdash N} \qquad \frac{S \vdash \underline{t} \qquad \underline{t} \in \mathbf{T} \qquad F \in \mathbf{C} \cup \mathbf{D} \qquad eval_F(\underline{t}) \neq \bot}{S \vdash eval_F(\underline{t})}
$$

Figure 4.8.: Deduction rules for the symbolic model

the destructors *unstring*$_0$ and *unstring*$_1$ allow to parse payload-strings. (Destructors *ispair* and *isstring* are not necessary, they can be emulated using *fst*, *unstring*$_i$, and *equals*$(\cdot, empty)$.)
The behavior of the destructors is given by the following rules; an application matching none of these rules evaluates to $\bot$. We define the functions ι from symbolic strings, denoted as $\mathcal{S}tring$ to natural numbers, denoted as $\mathbb{N}$. We use a standard binary encoding of numbers: For $s \in \mathcal{S}tring$, $\iota(s) := \sum_{i=1}^{|m|} 2^{i-1} \cdot s_i$, where $s_i = 1$ if the ith string constructor is $string_1$ and $s_i = 0$ if the ith string constructor is $string_0$, and, for $n \in \mathbb{N}$, $\bar{\iota}(n)$ denotes the be the corresponding encoding of the number n as symbolic bitstring $s \in \mathcal{S}tring$.

- Deduction relation: $\vdash$ is the smallest relation satisfying the rules in Figure 4.8.

4.5.2.2. Implementation conditions

Obtaining a computational soundness result for the symbolic model $\mathbf{M}$ requires its implementation to use an IND-CCA2 secure encryption scheme and a strongly existentially unforgeable signature scheme. More precisely, we require that (A_{ek}, A_{dk}), A_{enc}, and A_{dec} form the key generation, encryption and decryption algorithm of an IND-CCA2-secure scheme; and that (A_{vk}, A_{sk}), A_{sig}, and A_{verify} form the key generation, signing, and verification algorithm of a strongly existentially unforgeable signature scheme. Let $A_{isenc}(m) = m$ iff m is a ciphertext. (Only a syntactic check is performed; it is not necessary to check whether m was correctly generated.) A_{issig}, A_{isek}, and A_{isvk} are defined analogously. A_{ekof} extracts the encryption key from a ciphertext, i.e., we assume that ciphertexts are tagged with their encryption key. Similarly A_{vkof} extracts the verification key from a signature, and A_{verify} can be used to extract the signed message from a signature, i.e., we assume that signatures are tagged with their verification key and the signed message. Nonces are implemented as (suitably tagged) random k-bit strings. A_{pair}, A_{fst}, and A_{snd} construct and destruct pairs. We require that the implementation of the constructors are length regular, i.e., the length of the result of applying a constructor depends only on the lengths of the arguments. No restrictions are put on A_{garb}, $A_{garbageEnc}$, and $A_{garbageSig}$ as these are never actually used by the protocol. (The implementation of these functions need not even fulfill equations like $A_{isenc}(A_{garbageEnc}(x)) = A_{garbageEnc}(x)$.)

We require that the implementation A of the symbolic model $\mathbf{M}$ has the following properties. We use a standard binary encoding of numbers, $\langle m\rangle_2 := \sum_{i=1}^{|m|} 2^{i-1} \cdot m_i$, where m_i is the ith bit of the string m, and let $[n]_2$ be the corresponding encoding of the number n as a bitstring.

1. A is an computational implementation of $\mathbf{M}$ in the sense of Definition 6.
2. There are disjoint and efficiently recognizable sets of bitstrings representing the types nonces, ciphertexts, encryption keys, decryption keys, signatures, verification keys, signing keys, pairs, and payload-strings. The set of all bitstrings of type nonce we denote Nonces_k.[12] (Here and in the following, k denotes the security parameter.)
3. The functions A_{enc}, A_{ek}, A_{dk}, A_{sig}, A_{vk}, A_{sk}, A_{pair}, A_{string_0}, and A_{string_1} are length-regular. We call an n-ary function f *length-regular* if $|m_i| = |m_i'|$ for $i = 1, \ldots, n$ implies $|f(\underline{m})| = |f(\underline{m'})|$. All $m \in \text{Nonces}_k$ have the same length.
4. A_N for $N \in \mathbf{N}$ returns a uniformly random $r \in \text{Nonces}_k$.
5. Every image of A_{enc} is of type ciphertext, every image of A_{ek} and A_{ekof} is of type encryption key, every image of A_{dk} is of type decryption key, every image of A_{sig} is of type signature, every image of A_{vk} and A_{vkof} is of type verification key, every image of A_{empty}, A_{string_0}, and A_{string_1} is of type payload-string.
6. For all $m_1, m_2 \in \{0,1\}^*$ we have $A_{fst}(A_{pair}(m_1, m_2)) = m_1$ and $A_{snd}(A_{pair}(m_1, m_2)) = m_2$. Every m of type pair is in the range of A_{pair}. If m is not of type pair, $A_{fst}(m) = A_{snd}(m) = \bot$.
7. For all m of type payload-string we have that $A_{unstring_i}(A_{string_i}(m)) = m$ and $A_{unstring_i}(A_{string_j}(m)) = \bot$ for $i, j \in \{0,1\}$, $i \neq j$. For $m = empty$ or m not of type payload-string, $A_{unstring_0}(m) = A_{unstring_1}(m) = \bot$. Every m of type payload-string is of the form $m = A_{unstring_0}(m')$ or $m = A_{unstring_1}(m')$ or $m = empty$ for some m' of type payload-string. Moreover, for $\langle m\rangle_2 \geq \langle m'\rangle_2$, we have $A_{sub}(m, m') :=$

[12] This would typically be the set of all k-bit strings with a tag denoting nonces.

$[\langle m\rangle_2 - \langle m'\rangle_2]_2$ and $A_{ge}(m,m') = m$, and for $\langle m\rangle_2 < \langle m'\rangle_2$ we have $A_{sub}(m,m') := \bot$ and $A_{ge}(m,m') := \bot$. Furthermore, we have $A_{add}(m,m') := [\langle m\rangle_2 + \langle m'\rangle_2]_2$.

8. $A_{ekof}(A_{enc}(p,x,y)) = p$ for all p of type encryption key, $x \in \{0,1\}^*$, $y \in \text{Nonces}_k$. $A_{ekof}(e) \neq \bot$ for any e of type ciphertext and $A_{ekof}(e) = \bot$ for any e that is not of type ciphertext.
9. $A_{vkof}(A_{sig}(A_{sk}(x),y,z)) = A_{vk}(x)$ for all $y \in \{0,1\}^*$, $x,z \in \text{Nonces}_k$. $A_{vkof}(e) \neq \bot$ for any e of type signature and $A_{vkof}(e) = \bot$ for any e that is not of type signature.
10. $A_{enc}(p,m,y) = \bot$ if p is not of type encryption key.
11. $A_{dec}(A_{dk}(r),m) = \bot$ if $r \in \text{Nonces}_k$ and $A_{ekof}(m) \neq A_{ek}(r)$. (This implies that the encryption key is uniquely determined by the decryption key.)
12. $A_{dec}(A_{dk}(r), A_{enc}(A_{ek}(r),m,r')) = m$ for all $r,r' \in \text{Nonces}_k$.
13. $A_{verify}(A_{vk}(r), A_{sig}(A_{sk}(r),m,r')) = m$ for all $r,r' \in \text{Nonces}_k$.
14. For all $p,s \in \{0,1\}^*$ we have that $A_{verify}(p,s) \neq \bot$ implies $A_{vkof}(s) = p$.
15. $A_{isek}(x) = x$ for any x of type encryption key. $A_{isek}(x) = \bot$ for any x not of type encryption key.
16. $A_{isvk}(x) = x$ for any x of type verification key. $A_{isvk}(x) = \bot$ for any x not of type verification key.
17. $A_{isenc}(x) = x$ for any x of type ciphertext. $A_{isenc}(x) = \bot$ for any x not of type ciphertext.
18. $A_{issig}(x) = x$ for any x of type signature. $A_{issig}(x) = \bot$ for any x not of type signature.
19. We define an encryption scheme $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ as follows: KeyGen picks a random $r \leftarrow \text{Nonces}_k$ and returns $(A_{ek}(r), A_{dk}(r))$. $\mathsf{Enc}(p,m)$ picks a random $r \leftarrow \text{Nonces}_k$ and returns $A_{enc}(p,m,r)$. $\mathsf{Dec}(k,c)$ returns $A_{dec}(k,c)$. We require that then $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is IND-CCA secure.
20. We define a signature scheme $(\mathsf{SKeyGen}, \mathsf{Sig}, \mathsf{Verify})$ as follows: $\mathsf{SKeyGen}$ picks a random $r \leftarrow \text{Nonces}_k$ and returns $(A_{vk}(r), A_{sk}(r))$. $\mathsf{Sig}(p,m)$ picks a random $r \leftarrow \text{Nonces}_k$ and returns $A_{sig}(p,m,r)$. $\mathsf{Verify}(p,s,m)$ returns 1 iff $A_{verify}(p,s) = m$. We require that then $(\mathsf{SKeyGen}, \mathsf{Sig}, \mathsf{Verify})$ is strongly existentially unforgeable.
21. For all e of type encryption key and all $m \in \{0,1\}^*$, the probability that $A_{enc}(e,m,r) = A_{enc}(e,m,r')$ for uniformly chosen $r,r' \in \text{Nonces}_k$ is negligible.
22. For all $r_s \in \text{Nonces}_k$ and all $m \in \{0,1\}^*$, the probability that $A_{sig}(A_{sk}(r_s),m,r) = A_{sig}(A_{sk}(r_s),m,r')$ for uniformly chosen $r,r' \in \text{Nonces}_k$ is negligible.

Note that any IND-CCA secure encryption scheme and strongly existentially unforgeable signature scheme can be transformed into an implementation satisfying the above conditions by suitably tagging and padding the ciphertexts, signatures, and keys.

4.5.2.3. The Class of Key-safe Protocols

The computational soundness result we derive in this section requires that the CoSP protocol satisfies certain constraints. In a nutshell, these constraints require that encryption, signing, and key generation always use fresh randomness, that decryption only uses honestly generated (i.e., through key generation) decryption keys, that only honestly generated keys are used for signing, and that the protocol does not produce garbage terms. Decryption and signing keys may not be sent around. (In particular, this avoids the so-called key-cycle and key-commitment problems.) We call protocols satisfying these conditions *key-safe.* We stress that key-safe protocols are not a requirement induced by our framework as such. In

fact, requirements similar to key-safeness are standard and state-of-the art assumptions for soundness results (either explicit or implicitly enforced by the modeling, see, e.g., [AR02; BPW03a; MW04]).

A CoSP protocol is *key-safe* if it satisfies the following conditions:

1. The argument of every *ek*-, *dk*-, *vk*-, and *sk*-computation node and the third argument of every *enc*- and *sig*-computation node is an N-computation node with $N \in \mathsf{N}_P$. (Here and in the following, we call the nodes referenced by a protocol node its arguments.) We call these N-computation nodes *randomness nodes*. Any two randomness nodes on the same path are annotated with different nonces.
2. The argument of every $string_i$-node ($i \in \{0,1\}$) is either a $string_j$-node ($j \in \{0,1\}$) or an *empty*-node.
3. Every computation node that is the argument of an *ek*-computation node or of a *dk*-computation node on some path p occurs only as argument to *ek*- and *dk*-computation nodes on that path p.
4. Every computation node that is the argument of a *vk*-computation node or of an *sk*-computation node on some path p occurs only as argument to *vk*- and *sk*-computation nodes on that path p.
5. Every computation node that is the third argument of an *enc*-computation node or of a *sig*-computation node on some path p occurs exactly once as an argument in that path p.
6. Every *dk*-computation node occurs only as the first argument of a *dec*-destructor node.
7. The first argument of a *dec*-destructor node is a *dk*-computation node.
8. Every *sk*-computation node occurs only as the first argument of a *sig*-computation node.
9. The first argument of a *sig*-computation node is an *sk*-computation node.
10. There are no computation nodes with the constructors *garb*, *garbageEnc*, *garbageSig*, or $N \in \mathsf{N}_E$.

4.5.2.4. The computational soundness proof

With the definition of the symbolic model, the implementation conditions, and the protocol conditions at hand, we are finally in a position to discuss the slight modifications to the original computational soundness proof [BHU09]. We do not need to change much from the original proof since we add transparent terms that are completely independent of the cryptographic operations.

Theorem 4. (Computational soundness of $(\mathcal{D}_{\mathrm{ESA}}, \mathcal{P}_{\mathrm{ESA}})$) *If enhanced trapdoor permutations exist, there is an length-regular implementation $\underline{A}$ such that $(\mathcal{D}_{\mathrm{ESA}}, \mathcal{P}_{\mathrm{ESA}})$ is a computationally sound, well-formed model.*

Proof. Recall that by Theorem 1, it suffices to construct a simulator *Sim* that is Dolev-Yao and indistinguishable. Since we only add destructors on payload strings (e.g., $string_1(string_0(empty))$) and a simulator only operates on messages, i.e., constructors and nonces, the simulator does not need to be changed.

Since payload string do not hide information, i.e., they are transparent, the proof for Dolev-Yaoness applies verbatim to our extended model. For the proof for indistinguishability

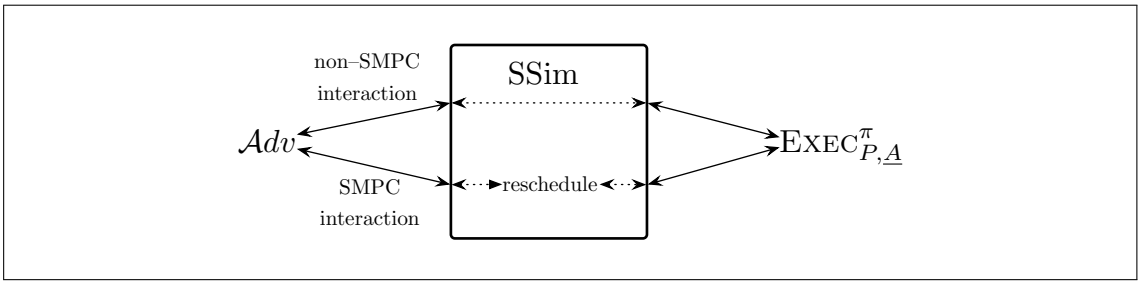

Figure 4.9.: The scheduling simulator SSim in interaction with the execution EXEC^{π} and the adversary.

only Case 4 in Claim 4 in Lemma 4 has to be extended by the following line:

For $F \in \{add, sub, mult, ge\}$ and m of type payload-string we have $\beta(\iota(m)) = \langle m \rangle_2 \in \mathbb{N}$.

The rest of the proof applies verbatim to our extended model, which concludes the proof. □

4.5.3. From the π-execution to the SMPC-execution

In this section, we prove that all computational robustness, i.e., trace properties, are preserved from the computational π-execution (see Definition 39) to the computational SMPC-execution (see Definition 40).

Let an occurrence of a subprocess $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ in a process P be called an SMPC session. Recall that the SMPC execution *Exec* for every SMPC session $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ does not reduce the subprocess $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ itself but interacts with a UC protocol, which we call the implementation of **SMPC**. The lemma states that there is a family of ideal functionalities in interaction such that for all well-formed processes P all valid assertions in the computational execution EXEC^{π} with an implementation $\underline{A}$ also hold in the computational execution *Exec* with an implementation $\underline{A}$ and $\mathcal{I}$ (as an implementation of **SMPC**). The family of ideal functionalities $\mathcal{I}$ is constructed in Construction 1.

The proof compares the computational execution EXEC^{π} with the computational execution *Exec* that uses the ideal functionality $\mathcal{I}$ as the implementation of **SMPC**. More precisely, let k be the security parameter and p be an arbitrary polynomial; we show that there is a polynomially bounded function l and a machine SSim, called the *scheduling simulator*, such that for all well-formed processes P, all implementations $\underline{A}$, and all adversaries $\mathcal{A}$ we have that the distribution $Assertions^{\pi}_{P,\underline{A},(p+l),\langle \text{SSim},\mathcal{A}\rangle}(k)$ is statistically indistinguishable from the distribution $Assertions^{\text{SMPC}}_{P,\underline{A},\mathcal{I},p,\mathcal{A}}(k)$ for all k.

Lemma 20. *For every well-formed atomic P, there exists a family $\mathcal{I}$ of SMPC ideal functionalities such that if P is π-robustly computationally safe using $\underline{A}$ and $\underline{A}$ is length-regular (see Implementation Condition 3), then P is SMPC-robustly computationally safe using $\underline{A}$, $\mathcal{I}$.*

Proof Outline. Before we present the full proof, we first give a proof outline. We show that there is a ppt simulator SSim that we can plug in between EXEC^{π} and the adversary.

Such that for any process well-formed process P, for any adversary $\mathcal{A}$ the interaction with the execution $\text{EXEC}^{\pi}_{P,\underline{A}}$ together with SSim is indistinguishable from the interaction with the execution $Exec_{P,\underline{A},\mathcal{I}}$ (using the ideal functionality $\mathcal{I}$) (see Figure 4.9). This simulator SSim simply forwards the messages as long as the messages do not belong to the interaction with an **SMPC** occurrence, i.e., as long as no subprocess $\mathbf{SMPC}(sidc, \underline{in}, \mathcal{F})$ (for some $sidc$, $\underline{in}$, and $\mathcal{F}$) is scheduled by the adversary. For all messages in an interaction of an **SMPC** occurrence, SSim basically lets $\text{EXEC}^{\pi}_{P,\underline{A}}$ behave like $F_{\mathcal{F}}$. In particular, SSim schedules the same reduction strategy S as $F_{\mathcal{F}}$ (see Section 4.4.1.2). Moreover, in order to output the same leakage as $\mathcal{I}$ the scheduling simulator SSim needs to be able to compute the length of a message given the appropriate term. As all implementation algorithms are length-regular and the length of nonces is fixed, the scheduling simulator can efficiently compute the length of a message given the corresponding term.

We show that the two interactions, between $\text{EXEC}^{\pi}_{P,\underline{A}}$, SSim and $\mathcal{A}$ and between $Exec_{P.\underline{A},\mathcal{I}}$ and $\mathcal{A}$, are indistinguishable for any $\mathcal{A}$.

4.5.4. The construction of the scheduling simulator

The scheduling simulator SSim basically schedules the reduction steps that correspond to the action of $\mathcal{I}$. SSim simulates against an adversary $\mathcal{A}$ the execution *Exec* while interacting withEXEC^{π}. We stress that SSim does not have additional capabilities compared to a usual adversary against EXEC^{π}.

As SSim does not know the bitstring $\eta(sid)$ corresponding to the symbolic session identifier sid from the beginning of the execution, SSim internally assigns an identifier γ to every **SMPC** occurrence $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ and tags in an internal copy of the current process $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ (and later on $0_{r,\underline{in},\mathcal{F}}$) with γ, denoted as $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})_{\gamma}$ and $0_{r,\underline{in},\mathcal{F}}$ as $0_{r,\underline{in},\mathcal{F},\gamma}$.[13]

There is a discrepancy between the process that is executed by EXEC^{π} and the process that the adversary expects from the simulation of *Exec*: In *Exec* every SMPC session is replaced by $0_{r,\underline{in},\mathcal{F}}$ after initialization; in EXEC^{π}, on the other hand, every SMPC session is treated as a regular subprocess. Therefore, for each $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ occurrence γ, the scheduling simulator stores the state of this occurrence in a process $\mathbf{smpc\_state}(\gamma)$. Moreover, SSim stores the current process for the adversary as $\tilde{P}$ and keeps track of the modifications as follows. SSim translates the evaluation context from the adversary by replacing every $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})_{\gamma}$ (or $0_{r,\underline{in},\mathcal{F},\gamma}$, respectively) by $\mathbf{smpc\_state}(\gamma)$. Thereafter, SSim uses the translated evaluation context $\hat{E}$ to address the subprocess to be reduced in the execution EXEC^{π} and sends appropriate evaluation contexts until the process in EXEC^{π} is a desired state. At this point SSim updates $\tilde{P}$ such that it corresponds to a reduction step in the execution *Exec*.

The process $\mathbf{smpc\_state}(\gamma)$ consists of a subprocess $\mathbf{compute\_state}(\gamma)$ for the current state of $\mathcal{F}$ and a context $\mathbf{smpc\_frame}(\gamma)[\bullet]$, i.e.,

$$\mathbf{smpc\_state}(\gamma) := \mathbf{smpc\_frame}(\gamma)[\mathbf{compute\_state}(\gamma)].$$

$\mathbf{smpc\_frame}(\gamma)[\bullet]$ can be characterized by the state of every party. Therefore, SSim maintains a function $state_{\pi}$ with which SSim keeps track of the state $state_{\pi}(\gamma, i)$ of a party i in an SMPC session γ. In Definition 47, we formally define the context $\mathbf{smpc\_frame}(\gamma)[\bullet]$.

[13] Similar to the computational execution, SSim does not send $0_{r,\underline{in},\mathcal{F},\gamma}$ to the adversary but only 0.

There is a yet another gap between $\mathcal{I}$ and **SMPC**: The ideal functionality $\mathcal{I}$ expects an explicit command $(i, s, \texttt{deliver})$ for outputting the result; on the other hand, the process $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ expects an appropriate sequence of evaluation contexts. Moreover, we need to maintain two partial mappings *sessionid* and $\tilde{\mu}$. $sessionid(\gamma)$ is in the course of the execution assigned the real session identifier as soon as the upon receiving the first accepted input, the real session identifier is leaked to the adversary. This mapping is used to later on check, whether the adversary addresses the commands to the correct session. $\tilde{\mu}$ stores the all channel bitstrings that the execution $\textsc{Exec}^{\pi}$ sends to the adversary, such as the bitstrings for the private channels of the corrupted parties.

In some cases, SSim checks whether for a given term c there is an i such that $\mathrm{ceval}_{\eta,\mu}\, in_i = \mathrm{ceval}_{\eta,\mu}\, c$. Although SSim does not know η and μ, the simulator can efficiently perform this check as it knows all intermediate processes and, hence, all reduction steps.

Characterizing the state of an SMPC session. For the construction of SSim we need to characterize the different states of the distinguished process $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$. This is done by a context, called $\mathbf{smpc\_frame}(\gamma)[\bullet]$, that is defined in Definition 47. In our abstraction, every party of a secure multi-party computation can be in the following states: `init`, `input`, `compute`, and `deliver`. In the state `init`, the entire session is not initialized yet; in the state `input`, the party expects an input; in the state `compute`, the party is ready to start the main computation; and, in the state `deliver`, the party is ready to deliver a message. These states are stored in a mapping $state_\pi$ (which is maintained by SSim) such that $state_\pi(\gamma, i)$ is the state of party i in the session γ.

Definition 47 (**smpc_frame**)**.** *Let a mapping $state_\pi$ from internal session identifiers and party identifiers to states given. We assign a process to each state $state_\pi(\gamma, i)$. Let $\alpha := (state_\pi, \gamma, (adv,\ sidc, \underline{in}, \mathcal{F}))$.*

$$
\begin{array}{rrcl}
state_\pi(\gamma,i) = \texttt{input}: & inp_i(\alpha) & := & in_i(x_i, sid').\overline{adv}\langle sid'\rangle. \\
 & & & \texttt{if}\ \ sid = sid'\ \ \texttt{then}\ \ \overline{lin_i}\langle x_i\rangle \\
 & & & \quad \texttt{else}\ \ \overline{inloop_i}\langle \texttt{sync()}\rangle \mid \boldsymbol{input}_i \\
state_\pi(\gamma,i) = \texttt{compute}: & inp_i(\alpha) & := & \overline{lin_1}\langle x_i\rangle \mid \boldsymbol{input}_i \\
state_\pi(\gamma,i) \in \{\texttt{deliver}, \texttt{init}\}: & inp_i(\alpha) & := & \boldsymbol{input}_i
\end{array}
$$

We define the context $initp(\alpha)$ as $initp(\alpha)[\bullet] := sidc(sid).\nu\underline{lin}.\ \nu\underline{inloop}.(\bullet)$ if for all $i \in [1, |\underline{in}|]$ $state_\pi(\gamma, i) = init$ and $initp(\alpha)[\bullet] := \bullet$ otherwise.

We define the context $\boldsymbol{smpc\_frame}(\gamma)[\alpha]$ as follows.

$$\boldsymbol{smpc\_frame}(\alpha)[\bullet] \ := \ initp(\alpha)\big[inp_1(\alpha) \mid \ldots \mid inp_n(\alpha) \mid \bullet\big]$$

In the construction of SSim, $state_\pi$ and $(adv, sidc, \underline{in}, \mathcal{F})$ are mostly clear from the context; more precisely, in the construction $state_\pi$ is globally defined and for every γ there is only one tuple $(adv, sidc, \underline{in}, \mathcal{F})$. Hence, if there is no ambiguity we also write $\mathbf{smpc\_frame}(\gamma)[\bullet]$ for $\mathbf{smpc\_frame}(\alpha)[\bullet]$.

The scheduling simulator keeps track of subprocesses Q in a process P; this is done via a context E such that $E[Q] = P$. As the process P might be reduced, we need for every reduction step of the process an appropriate reduction step of the context E.

Definition 48 (Reduction of a context)**.** *Let P be a process, Q be a subprocess, and E be the evaluation context such that $E[Q] = P$. We say that P* is reduced to *P' if there is*

a subprocess R of P and a corresponding evaluation context L such that $L[R] = P$ and there is a process R' such that $P' = L[R']$. We say that the evaluation context E is reduced *to E' according to the reduction from P to P' if there is a context L' with two distinct holes such that $L'[R][Q] = P$, $L'[R][\bullet] = E[\bullet]$, and $L'[\bullet][Q] = L[\bullet]$ and $L'[R'][Q] = P'$ and $L'[R'][\bullet] = E'[\bullet]$.*

Construction of SSim**.** Before we present the actual construction of the scheduling simulator, we introduce some notation for convenience. Throughout this section we use n for the amount of parties in an **SMPC** occurrence, i.e., for an occurrence $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ we consider n to be $|\underline{in}|$.

With abuse of notation, we denote the replacement of every occurrence of $0_{r,\underline{in},\mathcal{F},\gamma}$ in a process P by $\mathbf{smpc\_state}(\gamma)$ as

$$P\left\{\frac{\mathbf{smpc\_state}(\gamma)}{0_{r,\underline{in},\mathcal{F},\gamma}}\right\}.$$

Analogously, we write for an evaluation context E for the context in which each occurrence of $0_{r,\underline{in},\mathcal{F},\gamma}$ is replaced by $\mathbf{smpc\_state}(\gamma)$ as

$$E\left\{\frac{\mathbf{smpc\_state}(\gamma)}{0_{r,\underline{in},\mathcal{F},\gamma}}\right\}.$$

We use for a context $\tilde{E}$ and a process $\tilde{P}$ the abbreviations $\hat{E} := \tilde{E}\left\{\frac{\mathbf{smpc\_state}(\gamma)}{0_{r,\underline{in},\mathcal{F},\gamma}}\right\}$, $\hat{P} := \tilde{P}\left\{\frac{\mathbf{smpc\_state}(\gamma)}{0_{r,\underline{in},\mathcal{F},\gamma}}\right\}$, and $sidc(sid).\mathbf{SMPC}'(adv, \underline{in}, \mathcal{F}) := \mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$.

1. *Upon receiving the initial process P_0,* set $\tilde{P} := P_0$. Enumerate every occurrence $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ with an internal session identifier γ, and tag this occurrence $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ in $\tilde{P}$ with γ. Let initially for each γ $\mathbf{compute\_state}(\gamma) := \mathcal{F}$, $delivery(\gamma, i) := false$, let $sessionid(\gamma)$ be completely undefined, and let $state_\pi(\gamma, i) :=$ `init` for all $i \in [1, n]$. For any γ let $corrupt(\gamma, i) := true$, if the corresponding in_i in $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ is free; otherwise let $corrupt(\gamma, i) := false$.

   Forward all bitstrings that are initially sent by the execution EXEC^π to the adversary. In addition store all channel names in the partial mapping $\tilde{\mu}$.

   Then proceed as in 2.

2. *Main loop:* Let P be the last process that has been sent by the execution EXEC^π. Check whether $\hat{P} = P$. If the check fails, stop the entire simulation. Otherwise, construct the erasure of $\tilde{P}$, i.e., remove in $\tilde{P}$ the tag γ from each $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ and remove the tag $(r, \underline{in}, \mathcal{F}, \gamma)$ from each $0_{r,\underline{in},\mathcal{F},\gamma}$. Send the erasure of $\tilde{P}$ to the adversary $\mathcal{A}$. Then, expect an evaluation context $\tilde{E}$. We distinguish the following cases for $\tilde{E}$.

   a) $\tilde{E}$ **schedules the initialization**

   - **Evaluation context:** $\tilde{P} = \tilde{E}[\mathbf{SMPC}'(adv, \underline{in}, \mathcal{F})_\gamma]$
   - **State check:** Check whether $state_\pi(\gamma, i) := init$ for all $i \in [1, n]$.

   Schedule appropriate reduction steps until the current process in EXEC^π is

   $$\hat{E}[\mathbf{smpc\_frame}(state'_\pi, \gamma, \alpha)[\mathbf{compute\_state}(\gamma)]],$$

where $\alpha := (adv, sidc, \underline{in}, \mathcal{F})$ and $state'_\pi(\gamma, i) := \texttt{input}$ for all $i \in [1, n]$ and $state'_\pi$ coincides with $state_\pi$ on all other arguments.[14]

Set $\tilde{P} := \tilde{E}[0_{r,\underline{in},\mathcal{F},\gamma}]$, $state_\pi(\gamma, i) := \texttt{input}$ for all $i \in [1, n]$.

b) $\tilde{E}$ **schedules an input to a corrupted party** i**.**

- **Evaluation context:** $\tilde{P} = \tilde{E}[0_{r,\underline{in},\mathcal{F},\gamma}]$
- **State check:** Request a bitstring m from the adversary. Check whether $m = (c, s, \texttt{input}, m')$ and $state_\pi(\gamma, i) = \texttt{input}$.

Schedule appropriate reduction steps until the current process in Exec^π is

$$\hat{E}[\textbf{smpc\_frame}(state'_\pi, \gamma, \alpha)[\textbf{compute\_state}(\gamma)]],$$

where $\alpha := (adv, sidc, \underline{in}, \mathcal{F})$ and $state'_\pi(\gamma, i) := \texttt{compute}$ and $state'_\pi$ coincides with $state_\pi$ on all other arguments. In the first reduction step, the execution Exec^π requests a message and the bitstring for the channel in_i. Upon such a request of the execution, send $(c, (m', s))$ in response. If the execution does not accept the channel name c, i.e., if $c \neq \mu(in_i)$, we schedule appropriate reduction steps until the current process of Exec^π is again

$$\hat{E}[\textbf{smpc\_frame}(state_\pi, \gamma, \alpha)[\textbf{compute\_state}(\gamma)]].$$

Moreover, in that case we leave $\tilde{P}$ and $state_\pi(\gamma, i)$ unchanged.

If the execution accepts the channel name c, we proceed and check in case $\tilde{\mu}(in_i)$ is defined whether $c = \tilde{\mu}(in_i)$; if $\tilde{\mu}(in_i)$ is not defined set $\tilde{\mu}(in_i) := c$. If the check fails, abort the entire simulation. Set $channel(\gamma, i) := c$. Exec^π requests in a later step the bitstring $\mu(adv)$ for the channel adv. Upon such a request, send $cadv$ and expect a bitstring s' in response. Forward $(s', |m'|, i)$ to the adversary and store the session identifier s' that is sent to the adversary upon an accepted input, i.e., set $sessionid(\gamma) := s'$. Reduce for all $j \neq i$ for which $context(\gamma, i)$ is defined $context(\gamma, j)$ according to the reduction steps that have just been scheduled.

Set $state_\pi(\gamma, i) := \texttt{compute}$, and let $\tilde{P}$ remain unchanged.

c) $\tilde{E}$ **schedules an input to an honest party** i**.**

- **Evaluation context:** $\tilde{P} = \tilde{E}[\overline{c}\langle x, s\rangle.Q][0_{r,\underline{in},\mathcal{F},\gamma}]$
- **State check:** Check whether there is an $i \in [1, n]$ such that $\text{ceval}_{\eta,\mu}\, c = \text{ceval}_{\eta,\mu}\, in_i$, $state_\pi(\gamma, i) = \texttt{input}$, and $delivery(\gamma, i) = false$.

Let $Q' := Q\left\{\frac{\textbf{smpc\_state}(\gamma')}{0_{r,\underline{in},\mathcal{F},\gamma}}\right\}$. Schedule appropriate reduction steps until the current process of Exec^π is

$$\hat{E}[\textbf{smpc\_frame}(state'_\pi, \gamma, \alpha)[\textbf{compute\_state}(\gamma)]],$$

where $\alpha := (adv, sidc, \underline{in}, \mathcal{F})$ and $state'_\pi(\gamma, i) := \texttt{compute}$ and $state'_\pi$ coincides with $state_\pi$ on all other arguments.

Exec^π requests in a later step the bitstring $\mu(adv)$ for the channel adv. Upon such a request, send $cadv$ and expect a bitstring s' in response. Recall that all

[14] Recall that we have at this point $\textbf{compute\_state}(\gamma) = \mathcal{F}[\textbf{deliver}_1 \mid \ldots \mid \textbf{deliver}_n]$, as the session has just been initialized.

implementations are length-regular; as we know the length of the nonces, we can compute the message length of any message. Let x_i be the input term for party i. We compute the length of the bitstring $\eta(x_i)$ corresponding to the input of party i. Then, we send (s', l, i) to the adversary and store the session identifier s' that is sent to the adversary upon an accepted input, i.e., set $\mathit{sessionid}(\gamma) := s'$. Reduce for all $j \neq i$ for which $\mathit{context}(\gamma, i)$ is defined $\mathit{context}(\gamma, j)$ according to the reduction steps that have just been scheduled.

Set $\mathit{state}_\pi(\gamma, i) := \texttt{compute}$, and $\tilde{P} := \tilde{E}[Q][0_{r,\underline{in},\mathcal{F},\gamma}]$.

d) **Start the main computation upon the first delivery command for a party i.**

- **Evaluation context:** $\tilde{P} = \tilde{E}[0_{r,\underline{in},\mathcal{F},\gamma}]$
- **State check:** Request a bitstring m from the adversary. Check whether $s = \mathit{sessionid}(\gamma)$ and $\mathit{state}_\pi(\gamma, i) = \texttt{compute}$ for all $i \in [1, n]$. Moreover, check whether (i) $m = (i, s, \texttt{deliver})$ or (ii) $m = (c, s, \texttt{deliver})$. In case (ii), check furthermore whether there is an $i \in [1, n]$ such that $c = \tilde{\mu}(in_i)$.

Simulate the execution $\textsc{Exec}^\pi$ on the process

$$Q := \mathbf{compute\_state}(\gamma) \mid \underbrace{\overline{lin_1}\langle x_1\rangle \mid \ldots \mid \overline{lin_n}\langle x_n\rangle}_{=:Q_{lin}} \mid \underbrace{in_1(y_1', sid) \mid \ldots \mid in_n(y_n', sid)}_{=:Q_{in}}$$

against the fixed and efficiently computable reduction strategy S that is used by $F_\mathcal{F}$ (see Section 4.4.1.2) as follows:

– (*an input is read*) For every context $E[\bullet_1][\bullet_2] = E'[\bullet_1] \mid E''[\bullet_2] \mid Q_{in}$ that is sent by S such that $E''[P'][\overline{lin_i}\langle x_i\rangle] = Q_{lin}$ for some P', send

$$\begin{aligned} E'''[\bullet_1][\bullet_2] \;:=\; & \hat{E}[\mathit{inp}_1(\alpha) \mid \ldots \mid \mathit{inp}_{i-1}(\alpha) \mid \bullet_2 \\ & \mid \mathbf{input}_i \mid \mathit{inp}_{i+1}(\alpha) \mid \ldots \mid \mathit{inp}_n(\alpha) \mid E'[\bullet_1]] \end{aligned}$$

to the execution $\textsc{Exec}^\pi$, where $\alpha := (\mathit{state}_\pi, \gamma, (\mathit{adv}, \mathit{sidc}, \underline{in}, \mathcal{F}))$ and $\mathit{inp}_j(\alpha)$ is defined as in Definition 47. Receive an updated process

$$P' =: \hat{E}[\mathbf{smpc\_frame}(\gamma)[\mathbf{compute\_state}(\gamma)]]$$

from the execution. If P' is unmodified (i.e., $P' = E'''[P'][\overline{lin_i}\langle x_i\rangle]]$), send $\mathbf{compute\_state}(\gamma) \mid Q_{lin} \mid Q_{in}$ as a response to E to the reduction strategy S. Otherwise if $P' = E'''[P''][0]]$, send $E'[P''] \mid E''[0] \mid Q_{in}$ as a response to E to the reduction strategy S. Update $Q_{lin} := E''[0]$ and reduce for all i for which $\mathit{context}(\gamma, i)$ is defined the context $\mathit{context}(\gamma, i)$ according to the reduction of $\mathbf{compute\_state}(\gamma)$ (see Definition 48). In all other cases, abort the entire simulation of SSim.

– (*a step inside $\mathcal{F}$ is performed*) For every context $E[\bullet] = E'[\bullet] \mid Q_{lin} \mid Q_{in}$ that is sent by S, send $\hat{E}[\mathbf{smpc\_frame}(\gamma)[E[\bullet]]]$ to the execution. Receive the updated process $P' =: \hat{E}[\mathbf{smpc\_frame}(\gamma)[\mathbf{compute\_state}(\gamma)]]$ from the execution and send $\mathbf{compute\_state}(\gamma) \mid Q_{lin} \mid Q_{in}$ as a response to E to the reduction strategy S. Reduce for all i for which $\mathit{context}(\gamma, i)$ is defined the context $\mathit{context}(\gamma, i)$ according to the reduction of $\mathbf{compute\_state}(\gamma)$ (see Definition 48). Do the same for an evaluation context E with two distinct holes, i.e., for $E[\bullet_1][\bullet_2] = E'[\bullet_1][\bullet_2] \mid Q_{lin} \mid Q_{in}$.

– (*an output is sent*) For every context $E[\bullet_1][\bullet_2] = E'[\bullet_1] \mid Q_{lin} \mid E''[\bullet_2]$ that is sent by S such that $E''[in_i(x_i)] = Q_{in}$ and $\mathbf{compute\_state}(\gamma) = E'[\mathbf{deliver}_i]$, store $context(\gamma, i) := E'[\bullet]$. Send $E'[0] \mid Q_{lin} \mid E''[0]$ as a response to E to the reduction strategy S. Update $Q_{in} := E''[0]$ and reduce for all i for which $context(\gamma, i)$ is defined the context $context(\gamma, i)$ according to the reduction of $\mathbf{compute\_state}(\gamma)$ (see Definition 48).

Let P'' be the last process that has been sent by Exec^{π}. Set $state_{\pi}(\gamma, i) :=$ `deliver` for all $i \in [1, n]$. Update $\mathbf{compute\_state}(\gamma)$, i.e., set

$$\hat{E}[\mathbf{smpc\_frame}(\gamma)[\mathbf{compute\_state}(\gamma)]] := P''.$$

Let $\tilde{P}$ remain unchanged.

Thereafter, proceed as in the case of a delivery command for corrupted parties (case 2e).

e) **The delivery command for a party i is sent.**

- **Evaluation context:** $\tilde{P} = \tilde{E}[0_{r,\underline{in},\mathcal{F},\gamma}]$
- **State check:** Request a bitstring m from the adversary. Check whether $s = sessionid(\gamma)$, and $state_{\pi}(\gamma, i) =$ `deliver` and either (i) $m = (i, s,$ `deliver`$)$ or (ii) $m = (c, s,$ `deliver`$)$. In case (ii) additionally check whether there is an $i \in [1, n]$ such that $\tilde{\mu}(in_i) = c$.

In case (i), set $delivery(\gamma, i) := true$, $state_{\pi}(\gamma, i) =$ `input`, and let $\tilde{P}$ remain unchanged.

In case (ii), schedule appropriate reduction steps until the current process of Exec^{π} is

$$\hat{E}[\mathbf{smpc\_frame}(state'_{\pi}, \gamma, \alpha)[context(\gamma, i)[0]]],$$

where $\alpha := (adv, sidc, \underline{in}, \mathcal{F})$ and $state'_{\pi}(\gamma, i) := input$ and $state'_{\pi}(\gamma', j) := state_{\pi}(\gamma', j)$ for all $(\gamma', j) \neq (\gamma, i)$.

Upon a request from the execution Exec^{π} for a bitstring, send $\tilde{\mu}(in_i)$ to Exec^{π}. Expect a pair (m', s') and forward (m', i) to the adversary. Reduce for all $j \neq i$ for which $context(\gamma, i)$ is defined $context(\gamma, j)$ according to the reduction steps induced by $\hat{E}[\mathbf{smpc\_frame}(\gamma)[context(\gamma, i)[\bullet]]]$.

Let P' be the last process that has been sent by Exec^{π}. Set $\hat{E}[\mathbf{smpc\_state}(\gamma)] := P'$. Let $\tilde{P}$ remain unchanged. Set $state_{\pi}(\gamma, i) :=$ `input`.

f) **The output of party i is delivered to an honest party.**

- **Evaluation context:** $\tilde{P} = \tilde{E}[c(x).Q][0_{r,\underline{in},\mathcal{F},\gamma}]$
- **State check:** Check whether there is an i such that $\text{ceval}_{\eta,\mu}\, c = \text{ceval}_{\eta,\mu}\, out_i$. Check whether $state_{\pi}(\gamma, i) =$ `input`, $delivery(\gamma, i) = true$.

Let $Q' := Q\left\{\frac{\mathbf{smpc\_state}(\gamma')}{0_{r,\underline{in},\mathcal{F},\gamma}}\right\}$. In case (ii), schedule appropriate reduction steps until the current process of Exec^{π} is

$$\hat{E}[Q'][\mathbf{smpc\_frame}(state'_{\pi}, \gamma, \alpha)[context(\gamma, i)[0]]],$$

where $\alpha := (adv, sidc, \underline{in}, \mathcal{F})$ and $state'_{\pi}(\gamma, i) := input$ and $state'_{\pi}(\gamma', j) := state_{\pi}(\gamma', j)$ for all $(\gamma', j) \neq (\gamma, i)$.

Reduce for all $j \neq i$ for which $context(\gamma, i)$ is defined $context(\gamma, j)$ according to the reduction steps induced by $\hat{E}[\mathbf{smpc\_frame}(\gamma)[context(\gamma, i)[\bullet]]]$.

Let P' be the last process that has been sent by EXEC^{π}. Set $\tilde{P} := \tilde{E}[Q][0_{r,\underline{in},\mathcal{F},\gamma}]$ and $delivery(\gamma, i) := false$.

g) $\tilde{P} = E[!Q]$: For each bound variable and name in Q, choose fresh variables, or name, from the same equivalence class, according to the same enumerations as in EXEC^{π}, yielding a process Q'. For any $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ occurrence in Q, assign a fresh internal identifier γ and set for all $i \in [1, n]$ $state_{\pi}(\gamma, i) = \texttt{init}$, $\mathbf{compute\_state}(\gamma) := \mathcal{F}$, $delivery(\gamma, i) := false$, $corrupt(\gamma, i) := true$ if in_i is free in the initial process P_0, and $corrupt(\gamma, i) := false$ otherwise. Check whether the process in response is equal to $\tilde{E}[Q]\{^{\mathbf{smpc\_state}(\gamma)}/_{0_{r,\underline{in},\mathcal{F},\gamma}}\}$. If the check fails, abort; otherwise, set $\tilde{P} := E[E[Q' \mid !Q]$.

h) $\tilde{P} = \tilde{E}[M(x).Q]$: Request two bitstrings (c, m) from the adversary. Send to the execution $\hat{E}$. If EXEC^{π} requests two bitstrings, we send (c, m). Check whether the process in response is equal to $\tilde{E}[Q]\{^{\mathbf{smpc\_state}(\gamma)}/_{0_{r,\underline{in},\mathcal{F},\gamma}}\}$. If the check fails, abort; otherwise, set $\tilde{P} := E[Q]$.

i) $\tilde{P} = \tilde{E}[\overline{M}\langle N\rangle.Q]$: Request a bitstring c from the adversary. Send to the execution $\hat{E}$. If the execution requests a bitstrings send c and expect a bitstring m. Check whether the term N is actually a channel c. If the check succeeds, set $\tilde{\mu}(c) := m$. Send m to the adversary $\mathcal{A}$.

Moreover, check whether the process in response is equal to $\tilde{E}[Q]\{^{\mathbf{smpc\_state}(\gamma)}/_{0_{r,\underline{in},\mathcal{F},\gamma}}\}$. If the check fails, abort; otherwise, set $\tilde{P} := E[Q]$.

j) $\tilde{P} = \tilde{E}[\nu a.Q]$: Send to the execution $\hat{E}$. and check whether the process in response is equal to $\tilde{E}[Q]\{^{\mathbf{smpc\_state}(\gamma)}/_{0_{r,\underline{in},\mathcal{F},\gamma}}\}$. If the check fails, abort; otherwise, set $\tilde{P} := E[Q]$.

k) $\tilde{P} = \tilde{E}[\overline{M_1}\langle N\rangle.P_1][M_2(x).P_2]$: Send to the execution $\hat{E}$. and check whether the process in response is equal to $\tilde{E}[Q]\{^{\mathbf{smpc\_state}(\gamma)}/_{0_{r,\underline{in},\mathcal{F},\gamma}}\}$. If the check fails, abort; otherwise, set $\tilde{P} := \tilde{E}[P_1][P_2]$.

l) $\tilde{P} = \tilde{E}[let\ x = D\ in\ P_1\ else\ P_2]$: Send to the execution $\hat{E}$ and check whether the process in response is equal to $\tilde{E}[P'']\{^{\mathbf{smpc\_state}(\gamma)}/_{0_{r,\underline{in},\mathcal{F},\gamma}}\}$ with $P'' \in \{P_1, P_2\}$. If the check fails, abort; otherwise, set $\tilde{P} := E[P'']$ with the matching P''.

m) $\tilde{P} = \tilde{E}[\texttt{assert}(F).Q]$: Send to the execution $\hat{E}$ and check whether the process in response is equal to $\tilde{E}[Q]\{^{\mathbf{smpc\_state}(\gamma)}/_{0_{r,\underline{in},\mathcal{F},\gamma}}\}$. If the check fails, abort; otherwise, set $\tilde{P} := E[Q]$.

n) In all other cases do nothing.

A note on the network model. Recall that every time a party A sends a message to a party B, the machine B is activated. Hence, upon activation by the execution, i.e., when SSim receives a process or a message from EXEC^{π}, or by the adversary, i.e., when SSim receives an evaluation context or a message from $\mathcal{A}$, it might happen that SSim first finishes his actions from the previous round and only thereafter reads the new input from the execution.

4.5.5. The proof of the soundness of SSim

The proof compares the parts of the internal state of $\text{EXEC}^{\pi}_{P,\underline{A},\langle \text{SSim},\mathcal{A}\rangle}$ with parts of the internal state of $Exec_{P,\underline{A},\mathcal{I},Adv}$, which is shown in Lemma 21. The proof proceeds by induction over the evaluation contexts that $\mathcal{A}$ has sent. The sequence of internal states that is being compared is defined in Definition 49.

We perform an induction over the internal states of the two settings. However, there are slight technical mismatches between the two settings. These mismatches are not essential to the proof and are removed in Definition 49where we define the two distributions over the (adjusted) internal states. In Lemma 21, we show that these two distributions are statistically indistinguishable.

Notation. For an SMPC occurrence $\mathbf{smpc\_state}(\gamma)$, we need for each $i \in n$ the notion of the input and the output variable of party i. Recall the process inp_i from the smpc frame (see Definition 47): $inp_i = Q|\mathbf{input}_i$. If there is a subprocess $\overline{lin_i}\langle x_i\rangle$ in Q, then x_i is called the *input variable of* $\boldsymbol{smpc\_state}(\gamma)$. The output variable of party i in $\mathbf{smpc\_state}(\gamma)$ is defined via $context(\gamma, i)$. If $context(\gamma, i)$ is defined and

$$\mathbf{smpc\_frame}(\gamma)[context(\gamma, i)[\mathbf{deliver}_i]] = \mathbf{smpc\_state}(\gamma),$$

then y_i in $\overline{in_i}\langle y_i, sid\rangle.\overline{inloop_i}\langle \texttt{sync}()\rangle$ is called the *output variable of party i in* $\boldsymbol{smpc\_state}(\gamma)$.

Let $\perp$ be a distinguished error symbol. Moreover, for the sequence of internal states, we enumerate all SMPC occurrences in $Exec_{P,\underline{A},\mathcal{I},\mathcal{A}}$ with internal session identifiers γ in the same way as the scheduling simulator SSim.

Definition 49 (Internal states)**.** *Let P be a process. Let $\mathcal{A}$ be a machine, called the adversary. Let Exec be as in Definition 40, and* EXEC^{π} *be as in Definition 39. The rth round of $Exec_{P,\underline{A},\mathcal{I},\mathcal{A}}$ is the rth round of the main loop of Exec. The rth round of* $\text{EXEC}^{\pi}_{P,\underline{A},\langle \text{SSim},\mathcal{A}\rangle}$ *is the rth round of the main loop of* SSim*.*

- *(m'_r, n'_r, m_r, n_r) Let m'_r be the messages that are sent in* $\text{EXEC}^{\pi}_{P,\underline{A},\langle \text{SSim},\mathcal{A}\rangle}$ *to the adversary $\mathcal{A}$ in the rth round and n'_r be the messages that are sent by the adversary $\mathcal{A}$ in the rth round; analogous with m_r and n_r for $Exec_{P,\underline{A},\mathcal{I},\mathcal{A}}$.*
- *$(result_i^{\gamma,r}, input_i^{\gamma,r})$ Let $input_i^{\gamma,r} := x_i$ from the ideal functionality $\mathcal{I}_{u,\underline{in},\mathcal{F},\gamma}$ (see Construction 1).*[15] *Analogously, let $result_i^{\gamma,r} := y_i$ from the ideal functionality $\mathcal{I}_{u,\underline{in},\mathcal{F},\gamma}$. On the other hand, for* $\text{EXEC}^{\pi}_{P,\underline{A},\langle \text{SSim},\mathcal{A}\rangle}$ *let $x_i^{\gamma,r}$ and $y_i^{\gamma,r}$ be the input and output variable, respectively, of party i in $\boldsymbol{smpc\_state}(\gamma)$ in round r.*
- *(**compute_state**) Let $\boldsymbol{compute\_state}_r^{\gamma}$ be the process that $F_{\mathcal{F}}$ would extract from $state_F$. If $state_F = \emptyset$ set $\boldsymbol{compute\_state}_0^{\gamma} := \mathcal{F}^{\gamma}[0]$. On the other hand, let $\boldsymbol{compute\_state}_r'^{\gamma}$ be the process $\boldsymbol{compute\_state}(\gamma)\left\{\frac{\boldsymbol{deliver}_i}{0}\right\}$ of* SSim*, i.e., for every i every occurrence of a process $\boldsymbol{deliver}_i$ is replaced by the empty process 0.*
- *(η'', μ'') η'' and μ'' are the variable and name mapping of* EXEC^{π} *restricted on all variables and names, respectively, that are not locally bounded in a subprocess $\boldsymbol{smpc\_state}$. η and μ are the variable and name mapping, respectively, of Exec.*
- *$(\eta_\gamma, \mu_\gamma)$ Let $\mathcal{I}_\gamma$ denote the session in Exec that belongs to the internal session identifier γ. The variable and name mappings η_γ and μ_γ are the mappings that $F_{\mathcal{F}}$ in $\mathcal{I}_\gamma$ would*

[15] In particular, $input_i^{\gamma,r} := \perp$, if x_i is undefined.

extract from $state_F$. *The mappings* η'_γ *and* μ'_γ *are obtained by restricting* η' *and* μ' *to the variables and names, respectively, in* ***smpc_state***(γ).

- *(*$state'_\pi$*) Let* $state''_\pi$ *be the state mapping of* SSim *in round* r. *Then, let* $state'_\pi(\gamma, i) := \perp$ *if* $state''_\pi(\gamma, i) = init$ *and* $state'_\pi(\gamma, i) := state''_\pi(\gamma, i)$ *otherwise. On the other hand, in Exec, let* $\mathcal{I}_\gamma$ *denote the session that belongs to the internal session identifier* γ. *Let* $state_\pi(\gamma, \cdot)$ *be the state mapping of* $\mathcal{I}_\gamma$. *If* $\mathcal{I}$ *has not been initialized yet, set* $state_\pi(\gamma, i) := \perp$ *for all* $i \in [1, n]$.

Let $\Gamma(r)$ *be the set of all internal session identifiers in round* r. *Let*

$$((m_s, n_s), (\textbf{\textit{compute\_state}}_s^\gamma)_{\gamma\in\Gamma(r)}, state_\pi^s, (\eta^s, \mu^s), (\underline{input}^{\gamma,s})_{\gamma\in\Gamma(r)}, (\underline{result}^{\gamma,s})_{\gamma\in\Gamma(r)})_{s=0}^r$$

be the sequence of internal states of the execution $Exec_{P,\underline{A},\mathcal{I},\mathcal{A}}$ up to the rth round. *Let*

$$((m'_s, n'_s), (\textbf{\textit{compute\_state}}'^\gamma_s)_{\gamma\in\Gamma(r)}, state'^{\,s}_\pi, (\eta''^s, \mu''^s), (\eta'(\underline{x}^{\gamma,s}))_{\gamma\in\Gamma(r)}, (\eta'(\underline{y}^{\gamma,s}))_{\gamma\in\Gamma(r)})_{s=0}^r$$

be the sequence of internal states of the execution $\textsc{Exec}^\pi_{P,\underline{A},\mathcal{I},\langle \mathrm{SSim},\mathcal{A}\rangle}$ up to the rth round.

Let $T^{\mathcal{A}}_{P,r}$ *be the distribution of the* r*th extended prefix of the execution* $Exec_{P,\underline{A},\mathcal{I},\mathcal{A}}$ *and, analogously,* $T'^{\mathcal{A}}_{P,r}$ *be the distribution for* $\textsc{Exec}^\pi_{P,\underline{A},\langle \mathrm{SSim},\mathcal{A}\rangle}$

For proving Lemma 20, we show that for all security parameters k and for all polynomials p there is a polynomially bounded function l and a machine SSim such that for all well-formed processes P, all implementations $\underline{A}$, and all adversaries $\mathcal{A}$ we have that the distribution $Assertions^\pi_{P,\underline{A},(p+l),\langle \mathrm{SSim},\mathcal{A}\rangle}(k)$ is statistically indistinguishable from the distribution $Assertions^{\mathrm{SMPC}}_{P,\underline{A},\mathcal{I},p,\mathcal{A}}(k)$. For proving the indistinguishability of these two distributions over assertion tuples, we show an even stronger property: We show that even the distributions over the sequences of internal states of both settings are statistically indistinguishable. Lemma 21 is shown by an induction over the internal states of the two settings.

Lemma 21. *Let* $T^D_{P,r}$ *and* $T'^D_{P,r}$ *be defined as in Definition 49. Then, for all machines* D *and for all* $r \in \mathbb{N}$ *the two distributions* $T^D_{P,r}$ *and* $T'^D_{P,r}$ *are statistically indistinguishable, i.e., for all (possibly unbounded) interactive machines* D, *all polynomials* p, *and all* $r \in \mathbb{N}$ *there is a* $k_0 \in \mathbb{N}$ *such that for all* $k > k_0$

$$\left| \Pr b = 1 : b \leftarrow D(1^k, t), t \leftarrow T^D_{P,r} - \Pr b = 1 : b \leftarrow D(1^k, t), t \leftarrow T'^D_{P,r} \right| \leq p(k)$$

holds true.

Proof. We prove the statement by induction on the round r of *Main loop*, of either the execution *Exec* or the scheduling simulator SSim. More precisely, our induction hypothesis is that the two distributions $T^D_{P,r-1}$ and $T'^D_{P,r-1}$ are indistinguishable for any distinguisher D. The processes that are sent to the adversary are indistinguishable as long as all checks on *expect* in SSim succeed.

For $r = 0$, we only need to compare *Start*. In *Start* both *Exec* and $\textsc{Exec}^\pi$ only send the free names and the initial process to the adversary. Moreover, the initially the current process is the same in both settings, and we have for all γ

$$\textbf{compute\_state}^\gamma_0 = \textbf{compute\_state}'^\gamma_0 = \mathcal{F}^\gamma[0].$$

Hence, T_0 and T'_0 are indistinguishable.

For $r > 0$, we assume that T_{r-1} and T'_{r-1} are statistically indistinguishable. Hence, all previously sent subprocesses are indistinguishable in both settings. As we can see by the construction of SSim, *Exec*, and EXEC^{π}, in all cases the check $P = \tilde{P}$ succeeds. Let P be the process that has been sent to the adversary at the beginning of round r.

We distinguish the following cases for the evaluation context $\tilde{E}$ that is sent by the adversary in response to the last process $\tilde{P}$ that has been sent by SSim or *Exec*, respectively. Recall that $state_{\pi}$ is maintained by $\mathcal{I}$ as well as by SSim. With abuse of notation, we denote both $state_{\pi}$ and $state'_{\pi}$ from T_r and T'_r, respectively, with the mapping $state_{\pi}$ in the following case distinction.

(a) $\tilde{E}$ **schedules the initialization.**

- **Evaluation context:** $\tilde{P} = \tilde{E}[\mathbf{SMPC}'(adv, \underline{in}, \mathcal{F})_{\gamma}]$
- **State:** $state_{\pi}(\gamma, i) := \perp$ for all $i \in [1, n]$

The scheduling simulator SSim sets $state'_{\pi}(\gamma, i) := input$ for all $i \in [1, n]$. On the other hand, initially, the internal state of $\mathcal{I}_{r,\mathcal{F}}$ is set to `input`. No other values in T_r and T'_r change compared to the round $r - 1$; hence and by induction hypothesis, T_r and T'_r are indistinguishable.

(b) $\tilde{E}$ **schedules an input of a corrupted party** i**.**

- **Evaluation context:** $\tilde{P} = \tilde{E}[0_{r,\underline{in},\mathcal{F},\gamma}]$
- **State:** A bitstring m has been requested from the adversary. Check whether $m = (c, s, \texttt{input}, m')$, $s = sessionid(\gamma)$, and $state_{\pi}(\gamma, i) = \texttt{input}$.

The scheduling simulator first sends evaluation contexts such that the execution asks for a channel name c, a message m', and a session id s for $in_i(x_i^{\gamma,r}, sid)$, and, then, the simulator sends $(c, (s, m'))$. In the two settings the probability that $c = \text{ceval}_{\eta,\mu}\, in_i$ and $sid = \text{ceval}_{\eta,\mu}\, sid^{\gamma}$ is the same, where η and μ denote the variable and the name mapping of *Exec* or EXEC^{π}, respectively. If the two bitstrings are accepted, we have $\eta(x_i^{\gamma,p}) = m'$. On the other hand, we know by the construction of $\mathcal{I}$ that $input_i^p = m'$ holds true as well. By the induction hypothesis we know that T_{r-1} and T'_{r-1} are computationally indistinguishable. As only the list of inputs differs $r - 1$ and r and we have seen that $Exec_{P,\underline{A}}$ accepts the inputs with the same probability as $\text{EXEC}^{\pi}_{P,\underline{A},\text{SSim}}$ accepts the input, T_r and T'_r are computationally indistinguishable.

(c) $\tilde{E}$ **schedules an input for an honest party** i**.**

- **Evaluation context:** $\tilde{P} = \tilde{E}[\overline{c}\langle x, s\rangle.Q][0_{r,\underline{in},\mathcal{F},\gamma}]$
- **State:** There is an $i \in [1, n]$ such that $\text{ceval}_{\eta,\mu}\, c = \text{ceval}_{\eta,\mu}\, in_i$ and $state_{\pi}(\gamma, i) = \texttt{input}$.

By construction of the computational ideal functionality, the computational ideal functionality accepts if and only if **SMPC** accepts the input. Moreover, we know that $input_i^{\gamma,r} = \eta(x_i^{\gamma,r})$ and $state_{\pi}(\gamma, i) = \texttt{compute}$. By induction hypothesis, we conclude that T_r and T'_r are indistinguishable.

(d) **Start the main computation before delivering the first result for party** i**.**

- **Evaluation context:** (i) $\tilde{P} = \tilde{E}[0_{r,\underline{in},\mathcal{F},\gamma}]$ or (ii) $\tilde{P} = \tilde{E}[c(x).Q][0_{r,\underline{in},\mathcal{F},\gamma}]$
- **State:** A bitstring m has been requested from the adversary. $state_{\pi}(\gamma, i) = \texttt{compute}$ for all $i \in [1, n]$. Moreover, either (i) $m = (c, s, \texttt{deliver})$ and $s =$

sessionid(γ) or (ii) $m = (i, s, \texttt{deliver}')$ and $s = sessionid(\gamma)$. In case (ii), check additionally whether there is an $i \in [1, |\underline{in_i}|]$ such that $\text{ceval}_{\eta,\mu}\, c = \text{ceval}_{\eta,\mu}\, in_i$.

At this point there is a mismatch between the two settings: If in $Exec_{P,\underline{A},\mathcal{I},\mathcal{A}}$ the adversary sends a delivery request for a party that is uncorrupted party and for which the channel name has not been already sent to the adversary, there is an exponentially small chance of the adversary succeeding; in the communication with the scheduling simulator, on the other hand, there is no chance of succeeding. However, this situation only happens with exponentially small probability.

As a first step, let us note that by Definition 37 $fv(\mathcal{F}[\mathbf{0}]) \subseteq \emptyset$. $\eta(fv(\mathcal{F}[\mathbf{0}]))$ is contained in T_{r-1} and $\eta'(fv(\mathcal{F}[\mathbf{0}]))$ is contained in T'_{r-1}. Hence, by induction hypothesis, $\eta(fv(\textbf{compute\_state}(\gamma)))$ and $\eta'(fv(\textbf{compute\_state}'(\gamma)))$ are indistinguishable.

$\mathcal{I}$ as well as SSim runs the main computation whenever `deliver` is scheduled and $state_\pi(\gamma, i) = compute$ for all $i \in [1, n]$ where $\underline{in}$ are the channels involved in the session γ. Recall that $\mathcal{I}$ and SSim use the same reduction strategy and that this reduction strategy schedules the same reduction steps. By induction hypothesis, we know that $state_\pi^{r-1}$, η_γ^{r-1}, μ_γ^{r-1}, $\textbf{compute\_state}_{r-1}^\gamma$ and ${state'}_\pi^{r-1}$, ${\eta'}_\gamma^{r-1}$, ${\mu'}_\gamma^{r-1}$, $\textbf{compute\_state}'^\gamma_{r-1}$ are indistinguishable.[16]

Since SSim as well as $F_\mathcal{F}$ run $\text{Exec}^\pi_{\textbf{compute\_state}(\gamma),\underline{A}}$, $F_\mathcal{F}(input_1^{\gamma,r}, \ldots, input_n^{\gamma,r}, state_F)$ =: ($result_1^r$, …, $result_n^r$) and $\eta'(y_1^{\gamma,r}), \ldots, \eta'(y_n^{\gamma,r})$ are indistinguishable for a distinguisher that is given T_{r-1} or T'_{r-1}. Moreover, as $\textbf{compute\_state}_{r-1} = \textbf{compute\_state}'_{r-1}$ holds by induction hypothesis, $\textbf{compute\_state}_r = \textbf{compute\_state}'_r$.

Thereafter, we are in the case for delivery with corrupted parties or honest parties, respectively.

(e) **The delivery command for a party i is sent.**

- **Evaluation context:** $\tilde{P} = \tilde{E}[0_{r,\underline{in},\mathcal{F},\gamma}]$
- **State:** A bitstrings m has been requested from the adversary, $state_\pi(\gamma, i) =$ `deliver`, and (i) $m = (i, s, \texttt{deliver})$ and $s = sessionid(\gamma)$ or (ii) $m = (c, s,$ `deliver`) and $s = sessionid(\gamma)$.

In case (i), we set in both settings $state_\pi(\gamma, i)$ to `input`. Hence, T_r and T'_r remain indistinguishable.

In case (ii), in both settings the bitstring c is sent to the execution. If $c = \eta(in_i)$ the result is sent to the adversary in both settings. Moreover, both settings set $state_\pi(\gamma, i) :=$ `input`.

By induction hypothesis, or as we have already seen in the case (d), we know that these two results are computationally indistinguishable. Hence, we conclude that T_r and T'_r are indistinguishable.

(f) **The output of party i is delivered to an honest party.**

- **Evaluation context:** $\tilde{P} = \tilde{E}[c(x).Q][0_{r,\underline{in},\mathcal{F},\gamma}]$
- **State:** There is an i such that $\text{ceval}_{\eta,\mu}\, c = \text{ceval}_{\eta,\mu}\, in_i$ and $state_\pi(\gamma, i) =$ `input`.

We stress that the scheduling simulator can compute whether there is an i such that $\text{ceval}_{\eta,\mu}\, c = \text{ceval}_{\eta,\mu}\, in_i$.

[16] Formally, we consider the distribution obtained by projecting $T^D_{P,r-1}$ and ${T'}^D_{P,r-1}$ to ($state_\pi^{r-1}$, η_γ^{r-1}, μ_γ^{r-1}, $\textbf{compute\_state}_{r-1}^\gamma$) and (${state'}_\pi^{r-1}$, ${\eta'}_\gamma^{r-1}$, ${\mu'}_\gamma^{r-1}$, $\textbf{compute\_state}'^\gamma_{r-1}$), respectively.

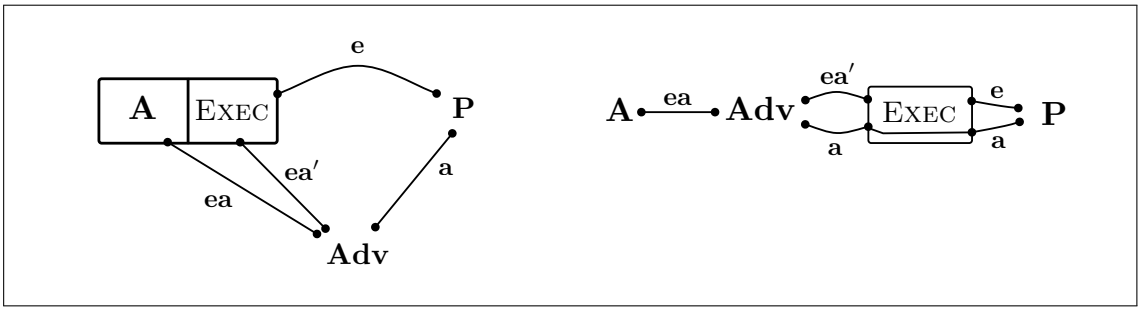

Figure 4.10.: The instantiation

Figure 4.11.: The transformed setup

We have the invariant that $delivery(\gamma, i) = true$ if and only if $output_i$ is defined. Hence, both $\text{EXEC}^{\pi}_{P,\underline{A},\langle \text{SSim},D\rangle}$ and $Exec_{P,\underline{A},\mathcal{I},D}$ perform a message synchronization, i.e., the message is output to the honest party.

By induction hypothesis $result_i^{\gamma,r-1}$ and $\mu(y_i^{\gamma,r})$ are indistinguishable; hence, η^r, μ^r and η''^r, μ''^r are indistinguishable as well. Therefore, T_r and T'_r are indistinguishable as well.

(g) In all other cases, either the scheduling simulator only replaces the unreduced **SMPC** with the current state **smpc_state** of EXEC^{π} in the evaluation context $\tilde{E}$ and forwards the modified evaluation context to EXEC^{π}, or both SSim and either *Exec* or $\mathcal{I}$ ignore the evaluation context. Since EXEC^{π} coincides on these cases with *Exec* and the scheduling simulator only forwards messages, T_r and T'_r are indistinguishable as well.

□

Finally, we prove the lemma. As a reminder, we restate the lemma:

Lemma 20. *For every well-formed P, there exists a family $\mathcal{I}$ of SMPC ideal functionalities such that if P is π-robustly computationally safe using $\underline{A}$, then P is SMPC-robustly computationally safe using $\underline{A}$, $\mathcal{I}$.*

Proof of Lemma 20. In Lemma 21 we show that the distributions over the sequence of processes that are sent to the adversary are indistinguishable. Hence, it suffices to solely show that the messages inside the events are indistinguishable. But, as the two executions coincide on all processes that do not contain **SMPC**, assertion tuples that contain message that are distinguishable can be used to distinguish $T^D_{P,r}$ from $T'^D_{P,r}$ which, however, is a contradiction to Lemma 21. □

4.5.6. Leveraging UC-realizability

As a next step, we show that if a protocol ρ UC realizes our ideal functionality $\mathcal{I}$, then, ρ is a sound implementation of **SMPC** as well.

Lemma 22 (Leveraging UC-realizability)**.** *Let ρ and $\mathcal{I}$ be two family of SMPC protocols and ideal functionalities such that ρ UC-realizes $\mathcal{I}$ (i.e., $\rho_{sid,\mathcal{F},\underline{c}} \in \rho$ iff $\mathcal{I}_{sid,\mathcal{F},\underline{c}} \in \mathcal{I}$ and $\rho_{sid,\mathcal{F},\underline{c}}$ UC-realizes $\mathcal{I}_{sid,\mathcal{F},\underline{c}}$). For every well-formed P, if P is π-robustly computationally safe using $\underline{A}$, $\mathcal{I}$ then P is SMPC-robustly computationally safe using $\underline{A}$, ρ.*

Proof outline. Before, we present the full proof, we first present a proof overview. By assumption we know that ρ UC realizes $\mathcal{I}$; hence, there is for any ppt adversary a ppt simulator such that for all environments the interaction between ρ and the adversary is indistinguishable from the interaction between $\mathcal{I}$ and the simulator. We only consider a dummy adversary that basically executes the commands of the environment. Moreover, for our setting, we instantiate the environments as (the emulation of) a pair of machines: the execution **SMPC** and a ppt machine that constitutes the adversary against **SMPC**, called hereafter the distinguisher.

In Figure 4.10, **P** denotes the protocol, i.e., either ρ or $\mathcal{F}$, respectively, **Adv** denotes the adversary, i.e., the ppt adversary $\mathcal{A}$ from the real setting or the simulator Sim, respectively. Moreover, the figures show the following connections (for a session r):

- **e** denotes the connections between the environment output ports $!\text{out}^e_{i,r}$ and incoming ports of the environment $?\text{out}^e_{i,r}$ and the environment input port $!\text{in}^e_i$ and outgoing ports $?\text{in}^e_{i,r}$ of the protocol (for any $i \in [1, |\underline{in}|)$,
- **a** denotes the connections between the adversary output ports $!\text{out}^a_{i,r}$ and incoming port $?\text{out}^a_{i,r}$ of the adversary and the adversary input $?\text{in}^a_{i,r}$ and outgoing ports $!\text{in}^a_{i,r}$ of the adversary,
- **ea** denotes the connections between the output ports of the distinguisher **A** and input ports of the adversary for the distinguisher and the distinguisher **A** for the adversary, respectively, and finally
- **ea**' denotes the connections between the output ports of the execution and input ports of the adversary for the execution and the execution for the adversary, respectively.

In the setting depicted in Figure 4.10, the UC protocol P directly communicates with the adversary **Adv**. The distinguisher can only communicate via the dummy adversary with the execution. The dummy adversary, forwards all messages of the form (i, s, m), where $i \in [1, n]$ to the i party of the protocol session s; all other message are redirected to the execution. By assumption, the indistinguishability of the two interactions holds, i.e., between ρ and the dummy adversary and $\mathcal{I}$ and the simulator for the dummy adversary.

As a next step, we transform the setting such that we are in the final setting. We modify the dummy adversary such that it redirects all messages to the execution (see Figure 4.11). We can show that in this setting the two interactions are indistinguishable as well i.e., the interaction between ρ and the dummy adversary and $\mathcal{I}$ and between the simulator for the dummy adversary.

The full proof follows.

Proof. Let $\underline{A}$ be a set of constructor and destructor implementations. Then, we show that for all processes Q in the applied π calculus and for all ppt machines $\mathcal{A}$, hereafter called the distinguisher, there is a ppt machine Sim such that for all ppt machines A the following indistinguishability holds true for A the two distributions are indistinguishable $Exec_{Q,\underline{I},\rho,\langle\mathcal{A},A\rangle}$ and $Exec_{Q,\underline{I},\mathcal{I},\langle Sim,A\rangle}$. This indistinguishability, in turn, implies that

$$\Big|\Pr\left((F_1,\ldots,F_n),F,\eta,\mu\right) \in Assertions^{\text{SMPC}}_{Q,\underline{A},\mathcal{I},p,\langle Sim,A\rangle}(k), \{F_1,\ldots,F_n\} \models_{\eta,\mu,\underline{A}} F$$

$$-\Pr\left((F_1,\ldots,F_n),F,\eta,\mu\right) \in Assertions^{\text{SMPC}}_{Q,\underline{A},\rho,p,\langle\mathcal{A},A\rangle}(k), \{F_1,\ldots,F_n\} \models_{\eta,\mu,\underline{A}} F\Big|$$

is negligible. Hence, the statement follows.

The proof begins with the UC setting. As ρ UC realizes $\mathcal{I}$, we know that for all ppt adversaries M there is a simulator *Sim* such that for all ppt environments E the following two settings are indistinguishable: first, $\mathbf{P} := \rho_{sid,\underline{in},\mathcal{F}}$ and $\mathbf{Adv} := M$ and, second, $\mathbf{P} := \mathcal{I}_{sid,\underline{in},\mathcal{F}}$ and $\mathbf{Adv} := Sim$. By the universal composability [Can01] and by the assumption $\rho \overset{UC}{\geq} \mathcal{F}$, there is a simulator for the concurrent execution of all sessions. Throughout the proof, we denote by *Sim* that simulator. The figures show, for clarity, only the transformations on one protocol session, but we perform the transformations on all sessions simultaneously.

As the UC realization property ensures that for all adversaries there is a valid simulator for all environment, we can concentrate on one particular adversary and a subset of all possible environments. In the course of the proof, we basically consider the dummy adversary, called M_D. Moreover, we split the environment E into two parts. The first part constitutes a ppt distinguisher **A**, and the second part constitutes the computational execution $Exec_{Q,\underline{A}}$. Then, we transform the environment E in several steps in which the messages from the distinguisher **A** and the UC adversary **Adv** are redirected such that after each transformation the two settings remain indistinguishable for **A**.

In Figure 4.10 and Figure 4.11, **P** denotes the protocol, i.e., either ρ or $\mathcal{F}$, respectively, **Adv** denotes the adversary, i.e., the ppt adversary $\mathcal{A}$ from the real setting or the simulator *Sim*, respectively. Moreover, the figures show the following connections (for a session r):

- **e** denotes the connections between the environment output ports $!\text{out}^e_{i,r}$ and incoming ports of the environment $?\text{out}^e_{i,r}$ and the environment input port $!\text{in}^e_{i,r}$ and outgoing ports $?\text{in}^e_{i,r}$ of the protocol (for any $i \in [1, |\underline{in}|)$,
- **a** denotes the connections between the adversary output ports $!\text{out}^a_{i,r}$ and incoming port $?\text{out}^a_{i,r}$ of the adversary and the adversary input $?\text{in}^a_{i,r}$ and outgoing ports $!\text{in}^a_{i,r}$ of the adversary,
- **ea** denotes the connections between the output ports of the distinguisher **A** and input ports of the adversary for the distinguisher and the distinguisher **A** for the adversary, respectively, and finally
- **ea'** denotes the connections between the output ports of the execution and input ports of the adversary for the execution and the execution for the adversary, respectively.

The instantiation (Figure 4.10):. We consider a family of environments that consist of two parts: a ppt distinguisher **A** and the computational SMPC execution $Exec_{Q,\underline{A}}$. The distinguisher **A** can only communicate with the execution $Exec_{Q,\underline{A}}$ and the UC adversary **Adv**. The execution is modified in that all messages (i, s, m), which the execution would usually send over an adversary port to the protocol, are ignored. Recall that we have a nested scheduling order: Whenever a protocol party of the UC attacker is stops without sending a message, the execution is activated. Whenever the execution stops the distinguisher is activated.[17]

We restrict our attention to the UC adversary, hereafter called M_D, that basically executes the commands of the adversary. As we consider static corruption, the dummy adversary only needs to forward messages. Upon a message m, M_D checks whether $m = (i, sid, m')$ and $i \in [1, |\underline{in}|]$, where *sid* is the session identifier of every party.[18] If the check succeeds,

[17] Technically, these two machines are emulated by one master-machine that respects the constraints mentioned.

[18] Recall that we assumed that the channel names are distinct from $[1, n]$ for the maximal number of parties

M_D sends m' over the port $!\text{in}_i^a$. Otherwise, send m over $\mathbf{ea}'$.

As ρ UC-realizes $\mathcal{F}$, there is a simulator Sim for each ppt adversary such that the two settings are indistinguishable for all environments $E(\mathbf{A}, Exec_{Q,\underline{A}})$ where $\mathbf{A}$ is an arbitrary ppt machine.

The transformed setup (Figure 4.11):. We change the setting such that all ports from the protocol that have been connected to the adversary ports ($\mathbf{a}$) are here connected to the environment. Analogously, all ports from the adversary ($\mathbf{a}$) that have been connected to the protocol are here connected to the environment. In particular, all corrupted parties do not redirect the messages to the UC adversary **Adv** anymore but send the messages to the execution. Here, we consider the machine M'_D that internally runs M_D and forwards everything to M_D. Whenever M_D wants to send a message m' over $!\text{in}_{i,r}^a$, M'_D sends $m := (i, sid, m')$ to the execution. As M'_D reverts the effect of M_D, effectively M'_D only forwards every message from the distinguisher $\mathbf{A}$. We do the same transformation for the simulator Sim_{M_D} obtaining a simulator Sim'_{M_D}.

Moreover, we consider usual execution $Exec_{Q,\underline{A}}$. In particular, the execution forwards all messages that it receives over a port $?\text{out}_{i,r}^a$ to the UC adversary M'_D. And, for every message (i, s, m) that is received by the adversary, the execution forwards m over $!\text{in}_{i,r}^a$. We stress that the execution does not pay attention over which port a message has been sent by the adversary; the execution processes all of them in the same way.

After these modification we effectively are in the situation in which $\mathbf{A}$ only communicates with the adversary, the adversary only communicates with the execution, and the execution communicates with the adversary and the protocol. The execution checks upon a message over a channel $?\text{out}_i^e$ whether i is corrupted. If the check succeeds, the execution forwards the message to the adversary. Hence, the execution only forwards the messages that are addressed to the adversary. Two settings, consequently, remain indistinguishable for $\mathbf{A}$. More precisely, given a successful distinguisher for the two settings in this transformed setup, we can construct a successful distinguisher for the initial setup, which contradicts assumption that ρ_i UC realizes $\mathcal{I}_i$.

As M'_D merely forwards every message, we have shown that for any ppt distinguisher $\mathbf{A}$ the following two settings are indistinguishable: $Exec_{Q,\underline{A},A_\rho,\langle M'_D,A\rangle}$ and $Exec_{Q,\underline{A},A_\mathcal{I},\langle Sim'_{M_D},A\rangle}$.

It remains to show that the following difference is negligible:

$$\Big|\Pr\left((F_1,\ldots,F_n),F,\eta,\mu\right) \in Assertions^{\text{SMPC}}_{Q,\underline{A},\mathcal{I},p,\langle Sim,A\rangle}(k), \{F_1,\ldots,F_n\} \models_{\eta,\mu,\underline{A}} F$$

$$-\Pr\left((F_1,\ldots,F_n),F,\eta,\mu\right) \in Assertions^{\text{SMPC}}_{Q,\underline{A},\rho,p,\langle\mathcal{A},A\rangle}(k), \{F_1,\ldots,F_n\} \models_{\eta,\mu,\underline{A}} F\Big|.$$

The assumption tuples that are raised can only differ in the messages that they contain, since the process is observable to the distinguisher. The two executions, however, only differ in the implementation of **SMPC**; hence, it remains to show that the messages sent over $\mathbf{e}$ are indistinguishable. Above, we showed that by the UC security, we can conclude that the messages that are sent over $\mathbf{e}$ are indistinguishable in the two settings (see Figure 4.10). As the execution is a polynomial-time machine, everything it computes out of indistinguishable inputs remains indistinguishable. Hence, the statement follows. □

in Q that perform a joint secure multi-party computation.

Extending the CoSP-embedding of the Applied π-calculus. The applied π calculus that is considered in the CoSP-embedding is a variant of the calculus that we consider in the current work: instead of assumes and asserts they only provide atomic events, i.e., events that do not carry a message. Security properties are expressed as properties over traces of atomic events. The operational semantics of their variant of the applied π calculus comprises an additional rule for reducing events, and each reduction step may be labelled by an event. With this notation, a process is said to *satisfy* a property if for every reduction sequence the sequence of raised events is in the trace property. The computational execution of the applied π calculus with events, presented in [BHU09], is basically identical to Exec^{π} with the difference that a sequence of events is output instead of a sequence of assertion tuples. A process is said to *computationally satisfy* a trace property if for any adversary the probability that the event trace is in the trace property (in a polynomially long interaction) is overwhelming.

Throughout this section, we consider two versions of the applied π calculus: the *applied π calculus with atomic events*, which is used in [BHU09], and the *applied π calculus with assumes and asserts*, which is the calculus that is used in the present work.

We simply erase in a process P all occurrences of `assert` *true* and replace all occurrences of `assert` *false* with *event bad*, obtaining a process $\mathcal{T}(P)$. The trace property $\mathcal{T}_p(P)$ then simply requires that the event *bad* has not been raised. Then, the following corollary follows immediately.

Corollary 1. *There is a translation $\mathcal{T}$ from processes to processes and a translation $\mathcal{T}_p$ from processes to traces such that the following holds true: For any well-formed process P (in the applied π calculus with assumes and asserts) if P is robustly safe, then, $\mathcal{T}(P)$ satisifies $\mathcal{T}_p(P)$. Moreover, whenever $\mathcal{T}(P)$ computationally satisfies $\mathcal{T}_p(P)$, P is robustly computationally safe.*

4.5.7. Plugging the results together

We now state the main computational soundness result of this work: the robust safety of a process using non-interactive primitives and our SMPC abstraction carries over to the computational setting, as long as the non-interactive primitives are computationally sound. This result ensures that the verification technique from Section 4.3 provides computational safety guarantees. We stress that the non-interactive primitives can be used both within the SMPC abstractions and within the surrounding protocol.

In Section 4.5.2, we review the computational soundness result for encryption and signatures using CoSP [BHU09]. This soundness result only holds for the class $\mathcal{P}_k$ of key-safe protocols (e.g., secret keys are not leaked, there no key cycles, only fresh randomness is used).

Given such a class of protocols $\mathcal{P}$ for which there is an implementation such that $\underline{A}$ is a computationally sound abstraction using $\underline{A}$, we show that subclass of well-formed processes in $\mathcal{P}$ is a computationally sound abstraction using encryptions and signatures (see Section 4.5.2). This result builds on and extends prior work on computational soundness of the applied π-calculus [BHU09].

In [CLOS02] a general construction is given for realizing any ideal functionality, which satisfies a particular well-formedness condition. Unfortunately, it turns out that their

proof has a flaw. The problem with the construction is, basically, that their construction transforms any ideal functionality into a circuit, and all parties jointly compute this circuit. However, as they consider an attacker that controls the network, any potentially realizable ideal functionality has to hand the message delivery over to the ideal attacker. In particular the ideal functionality expects some message from the attacker for the message delivery. In the circuit that they construct, however, they close all input ports from the attacker. Hence, the circuit, which expects an order from the attacker for the message delivery, will not output anything. Therefore, as the ideal setting sends an output but the realization does not, an environment can simply distinguish the two settings. We propose a simple fix: A well-formed ideal functionality, additionally, does not expect any message from the adversary. And we construct a wrapping machine $\mathcal{W}(\tau)$ that for a given well-formed ideal functionality τ and executes τ, leak the message lengths of the inputs, and sends the outputs upon a command from the ideal functionality. Then, we can apply the construction of [CLOS02] on the ideal functionality τ, yielding a protocol ρ_τ, and prove, along the lines of [CLOS02], that there is for any well-formed ideal functionality τ a protocol ρ_τ such that ρ UC realizes $\mathcal{W}(\tau)$.

Our ideal functionality $I_{sid,\mathcal{F},\underline{c}}$ constitutes such an ideal functionality $\mathcal{W}(\tau)$, where τ is the submachine that executes F and stores the state of F. With these modifications, we are able to apply the result presented in [CLOS02].

Lemma 23. *[CLOS02] Assume that enhanced trapdoor permutations exists. Then, for all $\mathcal{I}_{sid,\mathcal{F},\underline{c}}$ there exists a non-trivial protocol in the CRS-model that UC realizes $\mathcal{I}_{sid,\mathcal{F},\underline{c}}$ in the presence of malicious, static adversaries.*

We stress that Lemma 22, and therefore also Theorem 5, holds for any secure multi-party computation (SMPC) protocols that UC realizes our ideal functionality $\mathcal{I}$. Other SMPC protocols might be attractive for efficiency reasons. However, other SMPC protocols might not realize the ideal functionality for any corruption scenario. Hence, they impose additional protocol constraints in that they require that for all occurrences $\mathbf{SMPC}(adv, sidc, \underline{in}, \mathcal{F})$ only certain subsets of $\{in_1, \ldots, in_n\}$ are free.

As a next step, we show that computational soundness for atomic processes already suffices for guaranteeing computational soundness for the large class of well-formed processes.

Lemma 24. *Let $\mathcal{P}$ be a class of protocols. Assume there is an implementation $\underline{A}, \rho$ such that for all atomic processes $P \in \mathcal{P}$ the robust safety of P implies the SMPC-robust computational safety of P using $\underline{A}, \rho$. Then also for all well-formed $P \in \mathcal{P}$ the robust safety of P implies the SMPC-robust computational safety of P using $\underline{A}, \rho$.*

Proof. Let a be a sequence of assertion tuples such that $\Pr[a = a' : a' \leftarrow Assertions^{\text{SMPC}}_{P,\underline{A},\tau,p,\mathcal{A}})]$ is non-negligible. Then, for every assertion tuple $t := ((F_1, \ldots . F_n), F, \eta, \mu, Q)$ in a let R_t be a corresponding reduction sequence from the initial process P to (Q, η, μ).

In the applied π-calculus, we apply the reduction sequence R_t to the initial process P resulting in the same process Q (up to the renaming σ of the variables that has been performed by the computational execution). We know by assumption that there is a proof tree T of finite depth for the statement $\{F_1, \ldots, F_n\} \models_s F$ over symbolic terms (see Definition 41). We define a proof tree $\hat{T}$ based on T by applying the following modifications recursively from the root to the leafs and by considering the fixpoint of this recursively defined set:

(*a*) Replace $\dfrac{\underline{T}}{\dfrac{\Gamma, C\left\{\dfrac{x'}{x}\right\} \vdash_s \underline{B}}{\Gamma, \forall x.C \vdash_s \underline{B}}}$ by $\dfrac{\forall b \in \{0,1\}^* : \dfrac{\underline{T}\left\{\dfrac{\eta' \cup \{x' := b\}}{\eta'}\right\}}{\Gamma, C\left\{\dfrac{x'}{x}\right\} \vdash_{\eta' \cup \{x' := b\}, \mu', \underline{A}} \underline{B}}}{\Gamma, \forall x.C \vdash_{\eta', \mu', \underline{A}} \underline{B}}$,

(*b*) replace $\dfrac{\dfrac{\underline{T}}{\Gamma \vdash_{\eta', \mu', \underline{A}} C\left\{\dfrac{x'}{x}\right\}, \underline{A}}}{\Gamma \vdash_s \forall x.C, \underline{A}}$ by $\dfrac{\forall b \in \{0,1\}^* : \dfrac{\underline{T}\left\{\dfrac{\eta' \cup \{x' := b\}}{\eta'}\right\}}{\Gamma \vdash_{\eta' \cup \{x' := b\}, \mu', \underline{A}} C\left\{\dfrac{x'}{x}\right\}, \underline{A}}}{\Gamma \vdash_{\eta', \mu', \underline{A}} \forall x.C, \underline{A}}$,

(*c*) replace $\dfrac{d(eval(v_1), \ldots, eval(v_n)) = v}{\vdash_s \mathrm{Red}(d^{\#}(v_1, \ldots, v_n), v)}$ by

$\dfrac{A_d(\mathrm{ceval}_{\eta', \mu'}\, \sigma(v_1), \ldots, \mathrm{ceval}_{\eta', \mu'}\, \sigma(v_n)) = \mathrm{ceval}_{\eta', \mu'}\, \sigma(v)}{\vdash_{\eta', \mu', \underline{A}} \mathrm{Red}(d^{\#}(v_1, \ldots, v_n), v)}$, and finally

(*d*) replace all remaining $\vdash_s$ by $\vdash_{\eta, \mu, \underline{A}}$.

We claim that this (possibly infinite) tree $\hat{T}$ constitutes a proof tree for $\{F_1, \ldots, F_n\} \models_{\eta, \mu, \underline{A}} F$. Assume that there is a deduction step s that does not constitute a valid deduction step. As T is of finite depth, $\hat{T}$ is also of finite depth, and we can perform an induction over the depth of the proof tree. In the induction proof, we distinguish the following cases:

(*i*) (*s is a $\forall$-rule*) By induction $T\left\{\frac{b}{a}\right\}$, is a proof tree and by construction s is valid.

(*ii*) ($s \neq \dfrac{A_d(\mathrm{ceval}_{\eta', \mu'}\, \sigma(v_1), \ldots, \mathrm{ceval}_{\eta', \mu'}\, \sigma(v_n)) = v}{\vdash_{\eta', \mu', \underline{A}} \mathrm{Red}(d^{\#}(v_1, \ldots, v_n), v)}$ *and s is not a $\forall$-rule*) In this case, there is a already in T a corresponding step s' that is invalid, which is a contradiction to the assumption that T is a valid proof tree.

(*iii*) ($s = \dfrac{A_d(\mathrm{ceval}_{\eta', \mu'}\, v_1, \ldots, \mathrm{ceval}_{\eta', \mu'}\, v_n) = \mathrm{ceval}_{\eta', \mu'}\, v}{\vdash_{\eta, \mu, \underline{A}} \mathrm{Red}(d^{\#}(v_1, \ldots, v_n), v)}$) As all formulas are well-formed, we know that all quantified variables are messages that actually occur in the protocol. We construct a process $\hat{P}$ out of the process P that is robust safe but not robust computationally safe. We introduce in $\hat{P}$ the restricted channel names $c_v, c_{v_1}, \ldots, c_{v_n}$. As all formulas in the initial process P are well-formed, there is for each v_i an assumption `assume` F_j in Q that contains a predicate p that has v_i as an argument, i.e., $p(\ldots, v_i, \ldots)$. This assumption `assume` F_j also exists in the initial process P. Replace in P this `assume` F_j with $\overline{c_{v_i}}\langle v_i \rangle$ (analogously, for v). Then, we replace `assert` F with the following process:

$c_v(v).c_{v_1}(v_1). \ldots .c_{v_n}(v_n).$`if` $d(v_1, \ldots, v_n) = v$ `then assert` *true* `else assert` *false*

Erase all other `assert` and `assume` statements. We observe that the resulting process $\hat{P}$ is robustly safe as we assumed that P is robustly safe. By assumption, however, $\hat{P}$ is not SMPC-robustly computationally safe, which is a contradiction to the computational soundness of atomic processes.

□

We finally state our main computational soundness result. We apply a result of [CLOS02]

for UC-realizable multi-party computation protocols for virtually all ideal functionalities. This result only holds under standard cryptographic assumptions. First, we assume that for each session all parties share a commonly known random bitstring, called the common reference string (CRS). Such a setup is called the *CRS-model*. Second, we assume that *enhanced trapdoor permutations* exist, which roughly are permutations that are easy to compute and hard to invert unless a secret trapdoor is known. Moreover, as also required in other computational soundness results [BHU09; BU10], we require all implementations $\underline{A}$ to be *length-regular* (see Section 4.5.3), i.e., the length of the output is easily computable by the length of the input (see Section 4.5.3).

Theorem 5 (Computational soundness of symbolic SMPC)**.** *Let $(\mathcal{D}, \mathcal{P})$ be computationally sound, well-formed model using $\underline{A}$, where $\underline{A}$ is length-regular. If enhanced trapdoor permutations exist, then there is a family ρ of SMPC implementations in the CRS-model such that for each well-formed $P \in \mathcal{P}$, the robust safety of P implies the SMPC-robust computational safety of P using $\underline{A}, \rho$.*

Proof. The proof consists of four steps. Let atomic proceses be processes that only contain assertions of the form `assert` *false* and do not contain assumption. We show in Lemma 20 that for each atomic process P the π-robust computational safety using $\underline{A}$ implies SMPC-robust computational safety using $\underline{A}$ and $\mathcal{I}$. In Lemma 22, we show for any two UC protocols τ and ρ such that τ UC-realizes τ' the SMPC-robust computational safety of P using $\underline{A}$ and τ' implies SMPC-robust computational safety using $\underline{A}$ and τ. In [CLOS02] it has been shown that if enhanced trapdoor permutations exist there is a protocol ρ that UC-realizes $\mathcal{I}$ in the CRS-model; hence, by applying Lemma 22 to ρ and $\mathcal{I}$ we obtain SMPC-robust computational safety using $\underline{A}$ and ρ for all atomic processes P. In Lemma 24 we show for each class of protocols $\mathcal{P}$ if the robust safety of each atomic $P \in calP$ implies the SMPC-robust computational safety of P using $\underline{A}$ and ρ, then the robust safety of each well-formed $P \in \mathcal{P}$ also implies the SMPC-robust computational safety of P using $\underline{A}$ and ρ. □

This theorem entails a computational soundness result for protocols based on encryptions, signatures, and abstractions of secure multi-party computations

As a corollary of Theorem 5 and Theorem 4, we get the computational soundness of protocols using SMPC, public-key encryption, signatures, and arithmetics, with such non-interactive cryptographic primitives possibly used within both SMPC and the surrounding protocol.

Corollary 2 (SMPC computational soundness with $(\mathcal{D}_{\mathrm{ESA}}, \mathcal{P}_{\mathrm{ESA}})$)**.** *There is an implementation $\underline{A}$ and a family of UC-protocols τ such that for each well-formed $P \in \mathcal{P}_{\mathrm{ESA}}$, the robust safety of P implies the SMPC-robust computational safety of P using $\underline{A}$ and τ.*

4.6. Conclusion

We have presented an abstraction of SMPC in the applied π-calculus. We have shown that the security of protocols based on this abstraction can be automatically verified using a type system, and we have established computational soundness results for this abstraction including SMPC that involve arbirtrary arithmetic operations. This is the first work to

tackle the abstraction, verification, and computational soundness of protocols based on an interactive cryptographic primitive.

Our framework allows for the verification of protocols incorporating SMPC as a building block. In particular, the type-checker ensures that the inputs provided by the participants to the SMPC are well-formed and, after verifying the correctness of SMPC, the type-checker obtains a characterization of the outputs (e.g., in the Millionaires problem, the output is the identity of the participant providing the greatest input) that can be used to establish global properties of the overall protocol. Our framework is general and covers SMPC based on arithmetic operations as well as on cryptographic primitives.

This work focuses on trace properties, which include authenticity, integrity, authorization policies, and weak secrecy (i.e., the attacker cannot compute a certain value)[19]. In the next chapter, we extend our framework to uniformity, i.e. equivalence of protocols with the same control flow.

[19]To symbolically model weak secrecy, one can add the process c(x).if x=n then assert false (where c is a public channel) to check the secrecy of n: The protocol is robustly safe only if the attacker does not learn n.

Chapter 5.

Computational Soundness for General Interactive Primitives w.r.t. Equivalence Properties

[This chapter is based on a work with Michael Backes and Tim Ruffing [BMR14]. I mainly contributed to the computational soundness proof itself (Section 5.3 & Section 5.4). For the sake of comprehensiveness, I also review the other parts of our work (Section 5.2).]

5.1. Motivation

Interactive cryptographic primitives, such as interactive zero-knowledge proofs [GMR89], verifiable computation [VSBW13] or blind signatures [Cha82], are able to ensure sophisticated security properties that are impossible to achieve for non-interactive primitives. An example for a unique property in the interactive case is that for interactive zero-knowledge proofs the verifier can not prove to a third party that the prover issued a convincing proof. For such interactive primitives, Canetti, Backes, Pfitzmann, and Waidner introduced frameworks for proving strong, composable security guarantees of interactive primitives [BPW07; Can01]. Many cryptographic primitives were successfully proven to have this strong universal composability guarantee [CGS08; CLOS02; Fis06; KO12].

There is a successful line of research that proves computational soundness for many symbolic models, i.e., the absence of attacks against the symbolic abstraction implies the absence of attacks in suitable cryptographic model. Most of these computational soundness (CS) results against active attacks, however, have been specific to the class of trace properties [BBU13; BCW13; BHU09; BMM10; BU10; CKKW06; CW05; CW11; GGV08; JLM05; MW04], which is only sufficient as long as strong notions of privacy are not considered, e.g., in particular for establishing various authentication properties. Only few CS results are known for the class of equivalence properties against active attackers, and these results either do not cover interactive primitives [BL06; BP04; BPW03a; CC08; CCS12; CH11; CHKS12; SBBPW06] or do not allow to combine the DY model with non-interactive primitives [KTG12].

Contribution. We prove for interactive primitives computational soundness for uniformity, as long as these interactive primitive satisfy the cryptographic definition of universal composability [Can01]. We compare symbolic bi-protocols that use ideal functionalities with their computational counterparts that use the corresponding UC-secure realizations. Given a Dolev-Yao model (for non-interactive primitives) that is computationally sound for uniformity and given UC-secure interactive primitives, we show that the uniformity of

the symbolic bi-protocol implies the indistinguishability of the computational counterpart. We stress that this result is parametric in the Dolev-Yao model, i.e., in the non-interactive primitives.

5.2. Interactive Primitives in CoSP

An interactive primitives is a protocol for several parties that needs communication (i.e., interaction) among these parties to produce a result. Examples for widely deployed interactive primitives, are key-agreement protocols (for establishing a symmetric session key), secure channel protocols, (e.g., TLS), or consensus protocols (e.g., as in Bitcoin). Less widely deployed but more sophisticated examples are ORAM protocols (for storing data on a server while hiding the access patterns to the data), or SMPC protocols (for jointly computing any function without revealing the private inputs).

Along the lines of CoSP, a straight-forward way of abstracting interactive primitives (e.g., an SMPC scheme) would be to abstract each single message with Dolev-Yao style abstractions, i.e., abstracting every cryptographic primitive (e.g., the secret sharing scheme or the zero-knowledge proof scheme) as a Dolev-Yao style abstraction. While such a piece-wise abstraction is computationally sound, it requires a computational soundness result for all cryptographic primitives (i.e., all pieces) used in the interactive primitives.

Goldreich, Micali, and Widgerson introduced a abstraction for secure multi-party computation schemes [GMW87] that has been generalized to any interactive cryptographic protocol by Canetti [Can01] and Pfitzmann and Waidner [PW01]. This kind of abstraction models an interactive primitive as a single machine, called an ideal functionality, that has a direct (i.e., unobservable) connection to all participating parties.

In this section, we show how to model ideal functionality-style abstractions of interactive primitives in the CoSP framework. Our proof strategy is to represent the ideal functionality as a CoSP subprotocol that only contains computation nodes and then to derive a single destructor for this CoSP subprotocol.

5.2.1. Ideal Functionalities

Before we are able to introduce ideal functionalities as CoSP subprotocols, we formulate basic syntactic constraints on symbolic models that are essentially used to define both the ideal functionalities and the protocols that call these functionalities. First, it should be possible to construct pairs in the symbolic model. Second, we require that there is a distinguished dummy term *null*() that can be tested to be equal to other terms. Note that we only require the existence of certain constructors and destructors, but we do not impose explicit semantic restrictions symbolically. However, Definition 51 formulates criteria for the computational implementation of the required constructors and destructors.

Definition 50 (Symbolic conditions)**.** *A CoSP symbolic model* $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D})$ *is* standard *if it fulfills the following conditions.*

1. *There is a constructor* $pair/2 \in \mathbf{C}$ *as well as destructors* $fst/1, snd/2 \in \mathbf{D}$*.*
2. *There is a constructor* $null/0 \in \mathbf{C}$ *as well as a destructor* $equals/2 \in \mathbf{D}$*.*
3. $\mathbf{N} = \mathbf{N}' \uplus \mathbf{N}_{\text{INT}}$*. We call* $\mathbf{N}_{\text{INT}}$ the set of nonces for interactive primitives*.*

In the following definition, we require the computational implementations of *standard* symbolic models to fulfill natural semantics for the mentioned constructors and destructors. We will need the implementation A_{equals} of the destructor *equals*/2 only to express a unary predicate $P(x) := (A_{equals}(x, A_{null}()) \neq \bot)$ on bitstrings x. Intuitively, $A_{null}()$ is used in certain contexts to indicate that a message belongs to communication with the the attacker. As P provides only an interface to specify this decision, additional constraints on A_{equals} are not necessary.[1]

Definition 51 (Implementation conditions)**.** *A CoSP computational implementation* $\mathbf{A}$ *for the standard symbolic model* $\mathbf{M}$ *is* standard *if it fulfills the following conditions.*

1. *For all* $x, y \in \{0,1\}^*$*, we have* $A_{fst}(A_{pair}(x, y)) = x$ *and* $A_{snd}(A_{pair}(x, y)) = y$*.*
2. $A_{null}()$ *is of a unique type. That is, no algorithm* A_C *in* $\mathbf{A}$ *with* $C \in \mathbf{C} \setminus \{null\}$ *produces* $A_{null}()$ *on any input.*[2]
3. *For all* $N \in \mathbf{N}_{\text{INT}}$*, the same algorithm* A_N *is used.*

We define ideal functionalities as parameterized CoSP protocols that can be formulated as a single destructor, whose arguments correspond to the parameters of the protocol. Loosely speaking, an application of the destructor corresponds to a message sent to the UC machine implementing the ideal functionality. This allows another CoSP protocol to use the ideal functionality like a subprotocol (similar to the UC framework) by applying the destructor.

5.2.1.1. Communication

Technically, a CoSP ideal functionality $\mathcal{F}$ is a parameterized CoSP protocol that excepts five parameters as inputs, denoted by *state*, *sid*, *sender*, *input*, and *rand*. We shall briefly explain their meaning. Since algorithms in CoSP are stateless as opposed to machines in the UC, we model the state explicitly by the first parameter. The second parameter *sid* is interpreted as a session id. We stress that the functionality itself does not manage sessions (this is done by the CoSP protocols that use $\mathcal{F}$) and that there is no joint state between different sessions. The only purpose of the *sid* parameter is to provide $\mathcal{F}$ access to its session id.[3] The *sid* parameter give $\mathcal{F}$ access to its session id. A message sent to $\mathcal{F}$ is modeled by the parameters *sender* and *input*, where *sender* represents an identifier of the sending party and *input* are the contents. If the message comes from the attacker, *sender* is *null*(). Thus only one message from one party can be sent to $\mathcal{F}$ per invocation. This form of communication is closely related to the sequential execution model in UC: Whenever the execution is handed over to a machine M, e.g., an ideal functionality, only one other machine M' may have written a message to a tape of M. Finally, *rand* should be fresh randomness that $\mathcal{F}$ can use.

For the output, $\mathcal{F}$ contains *result nodes*. They indicate the end of an invocation of $\mathcal{F}$ and encode its output. Note that there may be (infinite) paths through the protocol tree of $\mathcal{F}$,

[1] We have chosen this encoding of P because it fits existing computational soundness results [BHU09; BMU12; BU10] that often have a destructor *equals*/2. In principle, our result works also with different encodings of P.

[2] This can be achieved by a suitable tagging.

[3] In contrast to the UC framework [Can01], we need not require that $\mathcal{F}$ ignores invocations with a wrong session id.

which do not contain any result nodes, however we will require that a symbolic execution of $\mathcal{F}$ reaches a result after finitely many steps.

Every result node μ_r and its second argument node μ'_r are computational nodes that are both annotated with the *pair* constructor. The term (or bitstring) constructed by the result node is to be interpreted as a triple, encoded using two pairs.[4]

5.2.1.2. Formal Definition

We want to define a destructor whose application basically corresponds to a symbolic execution of $\mathcal{F}$. The result of the destructor should be the same as the term produced by the reached result node in $\mathcal{F}$. To be able to define such a destructor, we require that in a symbolic execution of $\mathcal{F}$, a result node is reached for all possible terms that can be passed as parameter, i.e., for all terms in the considered message type $\mathbf{T}$. An alternative approach would not consider all possible input terms but only a set of reasonable and allowed parameter values. For instance, one could require that only nonces are valid values for the *rand* parameter. However, the present formulation simplifies the formal treatment significantly. Technically, a destructor is necessary because destructors (as defined in Definition 1) are arbitrary partial functions that map terms to terms, whereas constructors are formalized in CoSP only as symbols with an arity.

Recall that parameterized CoSP protocols are protocols that contain, in addition to references to other nodes, references to parameters. These protocols can further contain nodes without successors. Given terms $\underline{t}$ that instantiate the parameters of a parameterized protocol Π, the symbolic execution of Π is defined canonically: Whenever a parameter reference to parameter i is resolved, the parameter t_i is used. This allows us to define an ideal functionality in CoSP.

Definition 52 (CoSP ideal functionalities and ideal models)**.** *Suppose the symbolic model* $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D})$ *is standard.*[5] *A* CoSP ideal functionality *is an efficient probabilistic parameterized CoSP protocol on the symbolic model* $\mathbf{M}$ *that adheres to the following conditions:*

1. $\mathcal{F}$ references *parameters state, sid, sender, input and rand.*
2. $\mathcal{F}$ *contains no other nodes than computation nodes that are not be annotated by a nonce.*
3. *There is a subset result*$(\mathcal{F})$ *of the nodes in* $\mathcal{F}$*, such that* $\mu \in$ *result*$(\mathcal{F})$ *implies that there is no* $\nu' \in$ *result*$(\mathcal{F})$ *on the path from* μ *to the root. The nodes in result*$(\mathcal{F})$ *are called* result nodes of $\mathcal{F}$*. A result node has no successor.*
4. *Each result node* μ *is annotated with the constructor pair. The second referenced node of* μ *is another computation node* μ' *with the constructor pair.*
5. *The symbolic execution of* $\mathcal{F}$ *reaches a result nodes with all parameters* $t_{state}, t_{sid}, t_{sender}, t_{input}, t_{rand} \in \mathbf{T}$.

An ideal model $\mathbf{F}$ *on* $\mathbf{M}$ *is a countable set such that each* $\mathcal{F} \in \mathbf{F}$ *is a CoSP ideal functionality.*

[4] A different encoding of triples, i.e., without pairs, would in principle also be possible. Again, we have chosen the present formulation because existing computational soundness results typically include pairs.

[5] $\mathbf{M}$ is *standard* if it has a *pair* constructor as well as destructors *fst* and *snd* with the usual semantics; and there is a distinguished dummy term *null*() that can be tested to be equal to other terms, using a destructor *equals*.

We denote an $\mathcal{F} \in \mathbf{F}$ by *ideal functionality* if there is no danger of confusion with an ideal functionality in UC sense.

5.2.1.3. Ideal Functionalities in the Symbolic Setting

Given the formalization of an ideal functionality $\mathcal{F}$, defining the destructor that executes $\mathcal{F}$ is straightforward.

Definition 53 (Ideal Destructor)**.** *Let an ideal model* $\mathbf{F}$ *based on the symbolic model* $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D})$ *be given. Let* $\mathcal{F} \in \mathbf{F}$ *and* $\mathcal{F}$ *be the corresponding symbolic protocol of* $\mathcal{F}$*. The* ideal destructor of $\mathcal{F}$ *is defined as*

$$D_{\mathcal{F}} : \mathbf{T}^5 \to \mathbf{T} \text{ with } (t_{state}, t_{sid}, t_{sender}, t_{input}, t_{rand}) \mapsto t_{res}.$$

Here t_{res} *is the term produced by the reached result node in the symbolic execution of* $\mathcal{F}$ *with parameters* $t_{state}, t_{sid}, t_{sender}, t_{input}, t_{rand}$*.*

We need to show that $D_{\mathcal{F}}$ is a deterministic as required by Definition 1: Definition 52 ensures that a result node ν_r is reached for all input terms. A step of the symbolic CoSP execution is deterministic if the node processed in this step is a computation node or if it is a non-deterministic node with exactly one successor. The body of an ideal functionality contains only such nodes. Hence, until ν_r is reached, all full traces of $\mathcal{F}$ are identical up to (and including) this node if we fix a parameter function f_{par}. In particular the term computed by ν_r is uniquely determined.

If $\mathbf{F}$ is an ideal model based on $\mathbf{M}$ and an ideal destructor $D_{\mathcal{F}}$ for every $\mathcal{F}$ in $\mathbf{F}$, we extend $\mathbf{M}$ by these destructors.

Definition 54 (Extended Symbolic Model)**.** *Let an ideal model* $\mathbf{F}$ *based on a symbolic model* $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D})$ *be given. We define the* extended symbolic model with respect to $\mathbf{F}$ *as the symbolic model* $\mathbf{M_F} := (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D_F})$ *where* $\mathbf{D_F} := \mathbf{D} \cup \{D_{\mathcal{F}}/5 \mid \mathcal{F} \in \mathbf{F}\}$*.*

5.2.1.4. Ideal Functionalities in the Computational Setting

The extended symbolic model $\mathbf{M_F}$ can be implemented by different computational implementations, in particular by a computational variant of an ideal destructor $D_{\mathcal{F}}$ with $\mathcal{F} \in \mathbf{F}$, and an implementation that is constructed using a real protocol which UC-realizes this functionality. The former is formally specified by the following definition. It makes use of the computational execution of parameterized protocols, which resolves references to parameters by bitstrings passed at invocation time.

Definition 55 (Canonical Algorithm)**.** *Let an extended symbolic model* $\mathbf{M_F}$ *based on* $\mathbf{M}$ *and a computational implementation* $\mathbf{A}$ *of* $\mathbf{M}$ *be given. Furthermore, let* E *be a useless attacker machine, i.e.,* E *halts on the first activation. The* canonical algorithm of $\mathcal{F}$ *is the algorithm*

$$A_{\mathcal{F}} : \mathbb{N} \times (\{0,1\}^*)^5 \to \{0,1\}^*$$
$$\text{with } (b_{state}, b_{sid}, b_{sender}, b_{input}, b_{rand}) \mapsto b_{res},$$

that runs the an unbounded variant of the computational execution of $\mathcal{F}$ *and stops if the first reached result node is reached. The output* b_{res} *is the bitstring computed by the*

that node. The first argument of $A_{\mathcal{F}}$ represents the security parameter and the other arguments determine the parameters. If the result node does not produce a bitstring, i.e. the computation outputs $\perp$, then the output of $A_{\mathcal{F}}$ is also $\perp$. If the result node produces $\perp$ as output, then the output of $A_{\mathcal{F}}$ is also $\perp$.

By definition, $\mathcal{F}$ is efficient (in the sense of Definition 26) and the algorithms in $\mathbf{A}$ are computable in polynomial time. Though, this does not imply that $A_{\mathcal{F}}$ is polynomial-time. In fact, it is possible that a result node is not reached and the execution may not terminate at all.[6] The following Lemma addresses these issues and formulates sufficient criteria that may help to establish that a canonical algorithm $A_{\mathcal{F}}$ is computable in deterministic polynomial time. First, it suffices that the body of $\mathcal{F}$ is finitely high, which implies that the length of the trace up to a result node does not depend on the parameters. Second, it is possible to address the mentioned problem by putting a constraint on the output size of the algorithms in $\mathbf{A}$.

Lemma 25. *Let a canonical algorithm $A_{\mathcal{F}}$ of an ideal functionality $\mathcal{F}$ be given such that $A_{\mathcal{F}}$ uses the computational implementation $\mathbf{A}$. Let $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D})$ the symbolic model that $\mathbf{A}$ is based on. Moreover, let $n := k + l$ where k is the security parameter and l the sum of the lengths of the other inputs of $A_{\mathcal{F}}$.*

The algorithm $A_{\mathcal{F}}$ is computable in deterministic polynomial time, if one of the following conditions holds:

1. *The body of $\mathcal{F}$ is finitely high, i.e. there is $c \in \mathbb{N}$ such that each result node is on level c or above.*
2. *There is a polynomial p such that $A_{\mathcal{F}}$ produces a trace whose number of entries is bounded by $p(n)$ and the output length of each algorithm $(A_x)_{x \in \mathbf{C} \cup \mathbf{D}}$ is bounded by $n + O(1)$.*

Proof. A CoSP protocol $\mathcal{F}$ is efficient (in the sense of Definition 26), i.e. its node identifiers are computable in polynomial time by definition. Since $\mathbf{A}$ is a computational implementation in the sense of definition Definition 6, all algorithms of $\mathbf{A}$ are computable in polynomial time in k plus the length of their inputs. These inputs are taken from the range of the functions f and $f_{\mathbf{N}}$ mapping node identifiers to bitstrings in an execution of $Nodes^{\star}_{\mathbf{M},\mathbf{A},\mathcal{F},E_d}$. Thus, it suffices to show that the lengths of these are polynomially bounded (in k). This is clear for the bitstrings in the range of $f_{\mathbf{N}}$ as each of them is computed by a polynomial-time algorithm A_N with $N \in \mathbf{N}$, whose running time does only depend on k. Regarding the bitstrings in the range of f, we distinguish cases on the possible conditions in the Lemma.

1. Each algorithm A_x with $x \in \mathbf{C} \cup \mathbf{D}$ produces output that is polynomial-sized in its input size. As the algorithms are applied for at most c times to the inputs of length l, the length of each bitstring in the range of f is bounded by a polynomial in n.
2. The condition immediately implies that the length of each bitstring in the range of f is bounded by $n + O(p(n))$.

□

[6] Even if the number of processed nodes is polynomially bounded, the runtime can be super-polynomial, if the functions in $\mathbf{A}$ produce too large outputs. For instance, consider the function F which accepts an input of the form 1^x and outputs 1^{2^x}. Clearly, it is computable in deterministic polynomial time. A trace of length $O(k)$ can implement a k-fold application of F on the input 1, which yields the exponentially long bitstring 1^{2^k}.

Recall that Definition 54 extends a symbolic model $\mathbf{M}$ by ideal destructors $D_{\mathcal{F}}$, resulting in a extended symbolic model $\mathbf{M}_{\mathcal{F}}$. Analogously, we extend a computational implementation $\mathbf{A}$ for $\mathbf{M}$ by the canonical algorithms $A_{\mathcal{F}}$, given that each $A_{\mathcal{F}}$ is computable in polynomial-time.

Definition 56 (Ideal Implementation)**.** *Let $\mathbf{M_F} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D_F})$ be an extended symbolic model based on $\mathbf{M}$. Let a computational implementation $\mathbf{A}$ of $\mathbf{M}$ be given. Assume that each canonical algorithms $A_{\mathcal{F}}$ for $\mathcal{F} \in \mathbf{F}$ is computable in deterministic polynomial time in $k+l$, where k is the security parameter and l the sum of the lengths of the other inputs. Let $A_{D_{\mathcal{F}}} := A_{\mathcal{F}}$ for every $\mathcal{F} \in \mathbf{F}$. The computational implementation $\mathbf{A_F} := (A_x)_{x \in \mathbf{C} \cup \mathbf{D_F} \cup \mathbf{N}}$ of $\mathbf{M_F}$ is called* ideal implementation*.*

The ideal implementation fulfills the definition of computational implementation of $\mathbf{M_F}$ because the algorithms of $\mathbf{A}$ and $\mathbf{A_F}$ are required to be computable in polynomial time.

This is stated in the following lemma. In the proof, we construct a *full* protocol $\hat{\Pi}$ from Π as follows: Remove each computation nodes ν with destructor $D_{\mathcal{F}}$ and replace it by the tree of the ideal functionality $\mathcal{F}$. The parameter references in $\mathcal{F}$ are changed to references to the nodes referenced by ν and the subtree rooted at the yes-successor of ν is appended to every result node of $\mathcal{F}$. $\hat{\Pi} \in \mathrm{P}$ is a protocol on $\mathbf{M}$, in particular it does not use $D_{\mathcal{F}}$. This enables us to make use of a computationally sound implementation $\mathbf{A}$ of $\mathbf{M}$.

Lemma 26 (Soundness of Ideal Implementations)**.** *Let $\mathbf{M_F}$ be an extended symbolic model based on $\mathbf{M}$, and let $\mathbf{A}$ be a computationally sound implementation of $\mathbf{M}$ for protocols $\Pi \in \mathrm{P}$. Suppose that $\mathbf{M_F}$ has the ideal implementation $\mathbf{A_F}$. Suppose that for every $\Pi \in P$, we have that the full protocol $\hat{\Pi}$ is in P.*

Then the ideal implementation $\mathbf{A_F}$ is computationally sound for $\mathbf{M_F}$ and P.

Proof. Let Π be an efficient CoSP bi-protocol in P that symbolically satisfies indistinguishability. By definition, every $\mathcal{F} \in \mathbf{F}$ contains no input and output nodes, i.e., no communication with the attacker is carried out. Thus the views of Π and the full protocol $\hat{\Pi}$ do not differ. This holds for the symbolic views as well as for the computational views. We conclude that the symbolic indistinguishability of Π implies the symbolic indistinguishability of $\hat{\Pi}$. Since $\mathbf{A}$ is a computationally sound implementation of the symbolic model $\mathbf{M}$ and $\hat{\Pi} \in \mathrm{P}$ is a protocol on $\mathbf{M}$ (in particular it does not use $D_{\mathcal{F}}$) $\hat{\Pi}$ with $\mathbf{A}$ is computationally indistinguishable. As the computational views of $\hat{\Pi}$ and Π are identical, the computational indistinguishability of Π follows. □

5.2.2. Realization of Implementations

Similarly to an algorithm $A_{\mathcal{F}}$, which executes an ideal functionality, we consider a polynomial-time algorithm $A_\phi : \mathbb{N} \times (\{0,1\}^*)^5 \to \{0,1\}^*$ that is a cryptographic realization of the functionality. It is called *real algorithm* and provides the same interface as an algorithm $A_{\mathcal{F}}$, i.e., it takes bitstrings $b_{state}, b_{sid}, b_{sender}, b_{input}, b_{rand}$ as input and produces a triple $(b'_{state}, (b_{receiver}, b_{output}))$, encoded as nested pair, as output. Note that, since the algorithms can output a state, each UC-protocol can be formulated as a real algorithm. If we have a cryptographic realization for every $\mathcal{F}$ in an ideal model $\mathbf{F}$, we can extend an computational implementation $\mathbf{A}$ to a *real implementation* $\mathbf{A}_\Phi$. $\mathbf{A_F}$ and $\mathbf{A}_\Phi$

allow us to compare an ideal implementation of the interactive primitives with a real one, like in the UC framework.

We write A_θ to denote an algorithm that is either the canonical algorithm for an ideal functionality θ or the algorithm for a real protocol θ. To make use of the UC framework, we first bring interactive algorithms to the UC setting by constructing machines in the UC sense from them. We write $\mu(\theta)$ for the machine that runs A_θ internally. It basically provides an interface to a computational CoSP execution that activates $\mu(\theta)$ whenever $\mathbf{A}_\theta$ should be executed. In case that θ is a real algorithm, we require that it ensures that $\mu(\theta)$ separates the state of distinct protocol parties. This models de facto a real protocol execution as the parties can only communicate via the attacker.

As we consider only UC protocol machines $\mu(\theta)$ as well as variants thereof, we leave the involved dummy parties in the definition of the ITM and in the remainder of the thesis implicit, i.e. we face the protocol machine in an UC execution directly with the environment and annotate each message with an explicit party identifier. Recall that the dummy parties just relay messages from the environment to the UC ideal functionality and vice-versa. We stress that this treatment is only a presentational decision; in fact it is straightforward to reintroduce the dummy parties.

Definition 57 (Standard UC machine for a CoSP algorithm)**.** *Let A_θ be an algorithm that uses the standard computational implementation $\mathbf{A}$ for the symbolic model $\mathbf{M}$. Let A_N be the algorithm for the set of nonces for interactive primitives (as defined in Definition 51). The interactive Turing machine (in the UC sense) $\mu(\theta)$ runs the following algorithm:*

- *At the beginning of the first activation, initialize the variable state $:= A_{null}()$.*
- *Whenever $\mu(A_\theta)$ is activated with a message input, let sender be the party identifier of the invoking party, or $A_{null}()$ if the message comes from the adversary. Let $rand := A_N(k)$ and $res := A_\mathcal{F}(k, (state, sid, sender, input, rand))$, where sid is the session ID of $\mu(\theta)$.*
  - *If $res = \bot$, send* `no` *to the environment and block all further activations.*[7]
  - *Otherwise continue: Let $state' := A_{fst}(res)$, $receiver := A_{snd}(A_{fst}(res))$ and $output := A_{snd}(A_{snd}(res))$. Set $state := state'$.*
    - *If $A_{equals}(receiver, A_{null}()) = \bot$, pass* `yes` *receiver, output to the environment.*
    - *Else pass output to the adversary.*

With the protocol machines at hand, we are able to inherit the notion of protocol realization of the UC framework. We define that a real protocol ϕ *realize* an ideal functionality $\mathcal{F}$ in terms of , if $\mu(\phi)$ UC realizes $\mu(\mathcal{F})$.

Definition 58 (Realization)**.** *Let an extended symbolic model $\mathbf{M_F}$ with a ideal implementation $\mathbf{A_F}$ and a real implementation $\mathbf{A}_\Phi$ be given. For $\phi \in \Phi$ and $\mathcal{F} \in \mathbf{F}$, the real protocol ϕ* realizes *an ideal functionality $\mathcal{F}$, if $\mu(\phi)$ UC realizes $\mu(\mathcal{F})$. Furthermore, we say $\mathbf{A}_\Phi$* realizes *$\mathbf{A_F}$ if for every real algorithm $\phi \in \Phi$ for an ideal functionality $\mathcal{F} \in \mathbf{F}$, we have that ϕ realizes $\mathcal{F}$.*

[7] Blocking can generally be realized by handing over the execution back to the activating party immediately.

5.2.3. Good Ideal Functionalities and Real Protocols

Our goal is to consider a UC environment that runs a computational CoSP execution but does not handle interactive nodes. Instead, this task should be delegated to a UC protocol machine. For a interactive algorithm A_θ however, the standard machine $\mu(\theta)$ does not suffice for this purpose:

The CoSP execution only communicates with the attacker when an input or an output node is reached in the protocol tree. Thus we have to ensure that not only the environment, which is supposed to simulate the CoSP execution, but also the protocol machine $\mu(\mathcal{F})$ (or $\mu(\phi)$) may not freely exchange messages with the attacker machine in the UC execution. Moreover, in the ideal setting, $\mu(\mathcal{F})$ alone does not provide enough information to the UC environment to simulate a CoSP execution, which outputs a computational view of a CoSP bi-protocol. This view must contain the communication between $\mathcal{F}$ and the simulator, which is not visible to the environment in the UC setting.

More precisely, a simulator can lie to the environment about the communication towards the ideal functionality. In other words, the ideal functionality can exchange messages with the simulator such that the state of other protocol entities is not touched. In a CoSP protocol tree however, there is no formal notion of protocol entities. In fact, CoSP abstracts away from them completely as only one protocol global state is encoded in the node identifier and this encoding is left to the embedding of the considered calculus. Thus it is not clear how to state formally that only the states of some protocol entities have changed.

Definition 59 (honest machine & CoSP compatible machine)**.** *Given a machine $\mu(\rho)$ for an interactive algorithm A_ρ, the corresponding* honest machine $\tilde{\mu}(\rho)$ *internally runs $\mu(\rho)$ and relays the communication with the following exception: If $\mu(\rho)$ generates output for the adversary, it is not forwarded, but stored. Instead, a subroutine output* `output ready` *is passed to the environment and all messages from the environment or the adversary are blocked*[8] *until the environment sends a subroutine input* `deliver`*. Then the stored message is passed to the adversary.*

Moreover, we define for a given $\mu(\rho)$ for an interactive algorithm A_ρ, the corresponding CoSP compatible machine $\hat{\mu}(\rho)$ *that internally runs $\mu(\rho)$ and relays the communication with the following two exceptions. 1) If $\mu(\rho)$ generates output m for the adversary, it is not forwarded, but stored. Instead, a subroutine output* (`output ready, m`) *is passed to the environment. 2) If $\mu(\rho)$ receives a message m from the adversary, it stores this messages, informs the environment with* `input ready,m`*, waits for a* `deliver` *messages from the environment (and ignores all other messages), and only then forwards m to $\mu(\rho)$.*

If the honest machine is used, the environment is informed before giving output to the adversary. Then the environment is forced to let $\tilde{\mu}(\rho)$ deliver the output to the adversary explicitly. This is similar to a computational CoSP execution with A_ρ where communication with the adversary can be observed in the views and the sent message is not available to the adversary until a special output node is reached.

Definition 60 (Condition for the ideal functionality)**.** *Let $\mathcal{F}$ be an ideal functionality using the computational implementation $\mathbf{A}$, and let A_N be the algorithm used for the nonces*

[8] Blocking can generally be realized by handing over the execution back to the activating party immediately.

for interactive primitives (see Definition 51). Consider an execution of $A_{\mathcal{F}}$ such that $res := A_{\mathcal{F}}(state, sid, sender, input, rand)$ with the following properties:

- $state, sid, input \in \{0,1\}^*$
- $sender = A_{null}()$, *i.e. the execution is initiated by a message from the adversary machine*
- $rand := A_N()$, *i.e. rand is drawn according to* A_N

$\mathcal{F}$ *is* good for $\mathbf{A}$ *if the following condition holds for each such execution of* $A_{\mathcal{F}}$*:*

- *If and only if* $input = A_{null}()$, *we have* $res \neq \perp$ *and* $A_{equals}(receiver, A_{null}())$. *In that case, we say that* $A_{\mathcal{F}}$ *has received a* dummy message *from the adversary machine.*
- *For invocations by dummy messages, we additionally require* $state' = state$ *and* $output = A_{null}()$, *where* $state' := A_{fst}(res)$ *and* $receiver := A_{snd}(A_{fst}(res))$. *That is,* $\mathbf{A}_{\mathcal{F}}$ *does not fail but ignores the invocation completely and sends* $A_{null}()$ *to the adversary machine.*

Another problem is a converse situation: Suppose that during an execution in the real setting, the honest machine $\tilde{\mu}(\phi)$ reports `output ready` to the environment, because $\mu(\phi)$ has generated a message for the adversary, whereas $\mu(\mathcal{F})$ in the ideal setting generates true subroutine output s. If $\tilde{\mu}(\phi)$ has been activated by a message from the dummy adversary $\mathcal{A}_d$, then the simulator $\mathcal{S}$ has been instructed by the environment to relay this message to the protocol machine. Thus $\mathcal{S}$ has been activated and is able to send a dummy message to $\tilde{\mu}(\mathcal{F})$, which delays the subroutine output such that in both settings, `output ready` is reported to the environment. However, this is not possible in the case that $\tilde{\mu}(\phi)$ has not been invoked by the adversary. Consequently, Definition 61 excludes this case.

Definition 61 (Condition for the real protocol)**.** *Let ϕ be a real protocol, and let $\mathbf{A}$ be a standard computational implementation. Moreover, let A_N be the algorithm used for the nonces for interactive primitives (see Definition 51). Consider an execution of ϕ such that $res := \phi(state_1, sid, sender, input, rand)$ with the following properties:*

- $state_1, sid, input \in \{0,1\}^*$
- $sender \neq A_{null}()$, *i.e. the execution is not initiated by a message from the adversary*
- $rand := A_N()$, *i.e. rand is drawn according to* A_N

ϕ *is* good for $\mathbf{A}$ *if for each such execution of ϕ, it holds that*

- $res \neq \perp$ *and*
- $destination =$ `network`.

The condition in Definition 61 for a real protocol ϕ can also be expressed as a condition on the corresponding real algorithm A_ϕ.

Lemma 27. *Let ϕ be a real protocol, and let $\mathbf{A}$ be a standard computational implementation. Moreover, let A_N be the algorithm used for the nonces for interactive primitives (see Definition 51). Consider an execution of the real algorithm A_ϕ such that $res := A_\phi(state, sid, sender, input, rand)$ with the following properties:*

- $state, sid, input \in \{0,1\}^*$
- $sender \neq A_{null}()$, *i.e. the execution is not initiated by a message from the adversary*
- $rand := A_N()$, *i.e. rand is drawn according to* A_N

If ϕ is good for $\mathbf{A}$, then for each such execution of A_ϕ, it holds that

- $res \neq \perp$ *and*
- $A_{equals}(receiver, A_{null}()) \neq \perp$ *with* $receiver := A_{snd}(A_{fst}(res))$*, i.e. the receiver is the adversary.*

Proof. The claim follows immediately by inspection of Definition 61. Particularly, the real algorithm A_ϕ sets $receiver := A_{null}()$ if $destination = \texttt{network}$. □

We stress that both conditions are rather technical requirements instead of severe restrictions. The conditions are fulfilled by a wide range of primitives, or can often be achieved by a trivial reformulation of the ideal functionality or the real protocol. The following paragraphs discuss these reformulations in more detail.

Immediate outputs. Let ϕ be real protocol. If ϕ gets inputs from a protocol party P_{in}, it is not able to pass a output immediately to a different party $P_{out} \neq P_{in}$ without having to communicate via the network (the adversary) in between. Thus a so-called *immediate output* is only possible back to P_{in}. That is, Definition 61 basically imposes the restriction that subroutine output cannot be given immediately back to P_{in} when ϕ received subroutine input from ϕ. This essentially means that ϕ (and also $\mathcal{F}$ if ϕ realizes $\mathcal{F}$) must be formulated such that the results of the protocol are output whenever they are locally determined for a party. That is, the outputs of ϕ need not be requested through an interface; the reply to such a request would be an immediate output. This is a natural assumption for interactive primitives, where cryptographic operations do not take place only locally as it is the case for encryptions or digital signatures for instance. Indeed, the ideal functionalities for public key encryptions and signatures proposed by Canetti use immediate outputs, see [Can01] for a general discussion of immediate outputs.

Corruption. Our approach can be used with different corruption models. However, to be compatible with the conditions in Definitions 60 and 61, we treat adaptive corruption formally slightly different from the original UC framework. In particular, the real protocol and the ideal functionality have to be formulated using the following conventions (for the byzantine corruption model): Whenever a party in the real protocol is corrupted, it reports that facts to the environment. Only if the environment acknowledges immediately back to the party, i.e. from he view of the party only if it is activated by the environment the next time, the party sends its entire internal state to the adversary. This requirement ensures that the simulator in the ideal world is not activated immediately after it has sent the corruption request to the ideal functionality, which would be excluded by the condition for the ideal functionality. Note that this is only a reformulation and does not affect the original UC model at all: w.l.o.g. consider the case that we are dealing with the dummy adversary. Then the environment will know exactly when a party will report that it has just been corrupted because the environment has sent the corresponding corruption request through the dummy adversary. Moreover, we can assume that the environment always acknowledges the corruption, because the environment does only gain additional information by this step and the rest of the protocol does not make progress meanwhile. Ideal functionalities usually require that inputs from a corrupted protocol party P can still be modified by the adversary even if they are already sent to the ideal functionality. This is the case whenever there is nothing written onto tapes of other machines that binds P to the input. Typically, the functionalities are formulated in a way that allows the adversary to send a message to the ideal functionality that contains modified inputs of P at any point

after the corruption and before the value is committed to. Such a message would typically not generate subroutine output and thus violate the condition for the ideal functionality. However, it is possible to formalize meaningful ideal functionalities in a way that requires the simulator to provide the modified inputs exactly when the subroutine output (that is delayed in the case of interactive protocols) is created.

Definition 62 (UC Adversary for CoSP Adversary)**.** *Given a CoSP adversary machine E, we define the corresponding UC adversary machine UC-Adv(E) as follows:*

- *Run E internally and forward the communication as described in the following items.*
- *If E sends an input for an interactive input node,[9] as defined by Item 4 in Definition 64, deliver it to the machine $\tilde{\mu}^{sid}(\phi)$. Otherwise, relay the message to the environment.*
- *Write all received message on the input tape of E, regardless of their origin.*

Recall that the overall goal is as follows: First, we implement a CoSP computational execution with a real protocol ϕ in the UC framework. If we assume that ϕ realizes $\mathcal{F}$, there is a simulator for the UC execution with $\mathcal{F}$. The next step is to interpret this UC execution again in the CoSP setting. Thus we also have to transform the UC simulator to the CoSP adversary. This is done by the next definition.

Definition 63 (CoSP Adversary for UC Adversary)**.** *Given a UC adversary machine $\mathcal{A}$, we define the corresponding CoSP adversary machine CoSP-Adv($\mathcal{A}$) as follows:*

- *Run $\mathcal{A}$ internally and forward the communication as described in the following items.*
- *Relay all messages generated by $\mathcal{A}$ to the computational CoSP execution, regardless of the tape they have been written on.*
- *Whenever a message m is received from the computational CoSP execution because an interactive output has been reached,[9] write m to the communication tape of $\mathcal{A}$, where $\mathcal{A}$ expects messages from a protocol machine. Other messages m' from the computational CoSP execution are written to the input tape of $\mathcal{A}$, where $\mathcal{A}$ expects messages from an environment.*

Lemma 28. *Suppose that the real protocol ϕ is good and the ideal functionality $\mathcal{F}$ is good. Further suppose that $\mu(\phi)$ UC-realizes $\mu(\mathcal{F})$. Then the honest machine $\tilde{\mu}(\phi)$ UC-realizes the honest machine $\tilde{\mu}(\mathcal{F})$.*

Proof. Let $\tilde{\mathcal{Z}}$ an arbitrary polynomial-time UC environment. As $\mu(\phi)$ realizes $\mu(\mathcal{F})$, there is a valid simulator $\mathcal{S}$ for the dummy adversary $\mathcal{A}_d$. The main part of the proof compares the executions of $Exec_{\tilde{\mu}(\mathcal{F}),\mathcal{S},\tilde{\mathcal{Z}}}$, $Exec_{\mu(\mathcal{F}),\mathcal{S},\mathcal{Z}}$, $Exec_{\mu(\phi),\mathcal{A}_d,\mathcal{Z}}$ and $Exec_{\tilde{\mu}(\phi),\mathcal{A}_d,\tilde{\mathcal{Z}}}$, where $\mathcal{Z}$ is an environment which internally runs $\tilde{\mathcal{Z}}$. $\mathcal{Z}$ hides the syntactic differences between the honest machines $\tilde{\mu}$ and the standard machines μ, i.e. the messages `output ready` and `deliver`, by acting as a wrapper for $\tilde{\mathcal{Z}}$. The considered executions are depicted in Figure 5.1. We prove that $\tilde{\mathcal{Z}}$ cannot distinguish these four executions. In particular, this environment cannot distinguish $Exec_{\tilde{\mu}(\mathcal{F}),\mathcal{S},\tilde{\mathcal{Z}}}$ and $Exec_{\tilde{\mu}(\phi),\mathcal{A}_d,\tilde{\mathcal{Z}}}$. In other words, $\mathcal{S}$ is also valid simulator for the executions with the honest machines, and $\tilde{\mu}(\phi)$ UC realizes $\tilde{\mu}(\mathcal{F})$.

The conditions for the ideal functionality $\mathcal{F}$ (Definition 60) ensure that $\mathcal{F}$, and thus $\mu(\mathcal{F})$, produces subroutine output if and only it receives a dummy message. Dummy messages

[9] This case can detected, and *sid* as well as ϕ can be computed because of Item 6 in Definition 64.

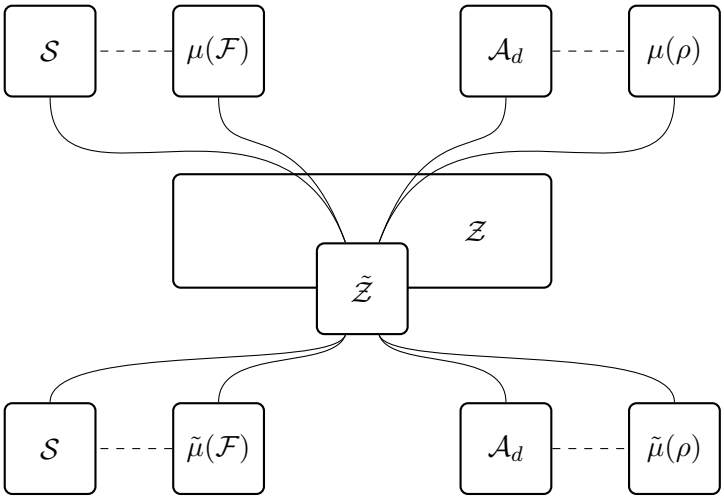

Figure 5.1.: The considered executions in the proof of Lemma 28

do not have any effect on the state of the machines in the network, since the activated machine immediately hands back to $\mathcal{S}$ without sending any information. That is, the validity of $\mathcal{S}$ does not depend on the dummy messages it sends to $\tilde{\mu}(\mathcal{F})$. Thus, without loss of generality, we may assume that $\mathcal{S}$ sends a dummy message to $\mu(\mathcal{F})$ if and only if $\mathcal{S}$ is instructed to send a message to the protocol but instead would send a message to the environment $\mathcal{Z}$ directly. This implies that $\mathcal{S}$ activates $\mu(\mathcal{F})$ whenever it is instructed by $\mathcal{Z}$ to relay a message to the protocol, i.e. whenever the dummy adversary $\mathcal{A}_d$ activates $\mu(\phi)$.

We distinguish cases on the possible actions of $\tilde{\mathcal{Z}}$ and the reaction of the machine activated after $\tilde{\mathcal{Z}}$. First, consider the case that $\tilde{\mathcal{Z}}$ instructs the respective adversary machine to deliver a message to the protocol, and that $\tilde{\mu}(\phi)$ as well as $\mu(\phi)$, respectively, generate subroutine output. In the ideal settings, assume for contradiction that $\mathcal{S}$ generates a message for the environment. Then the environment $\mathcal{Z}$ is able to distinguish $Exec_{\mu(\phi),\mathcal{A}_d,\mathcal{Z}}$ and $Exec_{\mu(\mathcal{F}),\mathcal{S},\mathcal{Z}}$; in the former setting $\mathcal{Z}$ has received a message from $\mu(\rho)$ whereas in the latter, the origin of the received the message is $\mathcal{S}$. This contradicts the validity of $\mathcal{S}$, which hence sends a message to $\mu(\mathcal{F})$. By construction, this is no dummy message. Thus Definition 60 guarantees that the internal instance of $\mu(\mathcal{F})$ does not directly reply to $\mathcal{S}$. Such a reply would be observable since $\tilde{\mu}(\mathcal{F})$ would report `output ready`. Instead, $\mu(\mathcal{F})$ generates subroutine output, which is relayed to $\tilde{\mathcal{Z}}$ by $\tilde{\mu}(\mathcal{F})$. As $\mathcal{S}$ is a valid simulator for $\mu(\mathcal{F})$, this output is indistinguishable from the output given in the real settings.

Second, assume that we are in the case that $\mu(\phi)$, internally run by $\tilde{\mu}(\phi)$, generates a message back to $\mathcal{A}_d$, after this adversary has forwarded m to $\tilde{\mu}(\phi)$. The honest machine $\tilde{\mu}(\mathcal{F})$ informs the environment with a message containing `output ready`. In this case, a standard message, i.e. not a dummy message, from $\mathcal{S}$ to $\mu(\mathcal{F})$ would lead to subroutine output. Again, $\mathcal{Z}$ could distinguish $Exec_{\mu(\phi),\mathcal{A}_d,\mathcal{Z}}$ and $Exec_{\mu(\mathcal{F}),\mathcal{S},\mathcal{Z}}$. Hence by construction, $\mathcal{S}$ generates a dummy message for $\mu(\mathcal{F})$ and $\tilde{\mu}(\mathcal{F})$ reports `output ready` to $\tilde{\mathcal{Z}}$.

Third, it remains to consider the case that $\tilde{\mathcal{Z}}$ sends subroutine input to the protocol machines. By Definition 61 and Lemma 27, we know that the protocol machine $\mu(\rho)$ in the real setting does not immediately reply to $\tilde{\mathcal{Z}}$; it sends a message to the adversary instead. Hence $\mu(\mathcal{F})$ does the same, otherwise $\mathcal{Z}$ could distinguish $Exec_{\mu(\phi),\mathcal{A}_d,\mathcal{Z}}$ and $Exec_{\mu(\mathcal{F}),\mathcal{S},\mathcal{Z}}$

trivially. The adversary machines are finally activated in all four settings. The rest of this case is analogous to the previous case.

Altogether, $\tilde{\mathcal{Z}}$ is informed about the communication between the protocol and the respective adversary in all four executions and especially cannot distinguish $Exec_{\tilde{\mu}(\mathcal{F}),\mathcal{S},\tilde{\mathcal{Z}}}$ and $Exec_{\tilde{\mu}(\phi),\mathcal{A}_d,\tilde{\mathcal{Z}}}$. That is, $\tilde{\mu}(\phi)$ UC realizes $\tilde{\mu}(\mathcal{F})$. □

5.3. Protocol Conditions for Interactive Primitives

In order to extend CoSP protocols by interactive primitives, we have to restrict the class of allowed CoSP protocols. The additional protocol conditions mainly plug the inputs and outputs of interactive primitives to the right nodes in the outer CoSP protocol. For instance, we require that a message produced by an interactive algorithm for the attacker, is really sent to the attacker via an output node and that the protocol cannot access it. In this section, we present these protocol conditions.

Condition 1 deals with the handling of different sessions and ensures that they will be kept separate. Conditions 2 and 3 specify how to handle the outputs of an interactive primitive. They require that there are nodes that compute the projections of the triple outputted by an interactive algorithm. Additionally, if the output is intended to be sent to the adversary, the two conditions ensure that there is an output node that does this task. Furthermore, the conditions 2 and 3 bound the information flow out of the interactive primitive: If the output is to be sent to the adversary, it may not be used anywhere else. Moreover, the UC environment that executes a CoSP protocol will delegate the handling of interactive primitives, i.e. the environment does not know about the internal state of the interactive primitives and the randomness they have used. Thus the control flow of the CoSP protocol may not depend on these pieces of information directly but only on the outputs generated by an interactive primitive.

Condition 4 restricts the argument nodes of an interactive node. In particular, it deals with the monadic state passing from one interactive node to the next interactive node that belongs to the same session. If the session is just initialized, i.e. there is no previous state, a computation node with constructor $A_{null}()$ must be referenced, which creates a new initial state.

Furthermore, conditions 4 and 5 together ensure that the randomness used by the interactive algorithms is not used twice. Condition 6 guarantees that intuitively, the adversary is able to know when the protocol expects a message to be passed to an ideal functionality, or that a message sent to the adversary has been generated by an ideal functionality. This attributes to the fact that a UC adversary machine has to specify the identity of another machine M explicitly if it wants to send a message to M, whereas a CoSP adversary machine is technically only faced with one single protocol machine. Condition 6 is necessary because we construct a UC adversary from a CoSP adversary In a typical embedding of a calculus into CoSP, the adversary is regularly presented a status of the execution at a control node and then may schedule the protocol, i.e. the adversary may decide that an ideal functionality is scheduled. Thus condition 6 is not unrealistic.

Definition 64 (Protocol conditions)**.** *Let P be a class of CoSP protocols that use an extended symbolic model $\mathbf{M_F} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D_F}, \vdash)$, and let $\mathbf{N}_{\text{INT}}$ be the corresponding set of nodes for interactive primitives. The* corresponding class of protocols for *P is the subset*

$P_{\mathbf{F}} \subseteq P$ such that the following criteria are additionally met:

1. *The sid argument node ν_{sid} of an interactive node must be a computation that produces a nonce $N \in \mathbf{N}$. On every path through ν_{sid}, there is no other computation node with nonce N. We say that two interactive nodes* belong to the same session *if and only their sid argument node is the same.*
2. *The following holds for every path to the protocol tree that contains an interactive node ν.*
   - *The only nodes on this path that may reference ν are two computation nodes $\nu_{state'}$ and ν_s with destructors fst and snd, respectively.*
   - *The only nodes on this path that may reference ν_s are two computation nodes $\nu_{receiver}$ and ν_{output} with destructors fst and snd, respectively.*
   - *The only node on this path that may reference $\nu_{receiver}$ is a computation node ν_{branch} with destructor equals. The first argument of ν_{branch} is $\nu_{receiver}$, the second is any computation node with constructor null.*
   - *The only node on this path that may reference ν_{output} is an output node ν_o in the yes-subtree of ν_{branch}. We denote ν_o by* interactive output node.
   - *No nodes reference ν_{branch}.*
   - *Only nodes in the no-subtree of ν_{branch} reference $\nu_{receiver}$.*
3. *Let ν' be an interactive node with destructor $\text{INT}_{\mathcal{F}}$, where $\mathcal{F} \in \mathbf{F}$. Moreover, let ν be the bottom-most predecessor[10] of ν' that is an interactive node with $\text{INT}_{\mathcal{F}}$ and belongs to the same session as ν'. Let $\nu_{state'}$, ν_{branch} and ν_o be defined as in the previous item; they belong to ν. The following folds:*
   - *The node ν_{branch} lies on the path from ν' to ν.*
   - *If ν' is in the yes-subtree of ν_{branch}, then ν_o lies on the path from ν' to ν.*
   - *If the input argument node ν'_{input} of ν' is an input node, it lies on the path from ν' to ν. If additionally, ν_o lies on the path from ν' to ν, then ν_o is a predecessor of ν'_{input}.*
   - *$\nu_{state'}$ is the state argument node of ν.*
4. *For each interactive node ν, let ν_{state},ν_{sender}, ν_{input} and ν_{rand} the respective nodes referenced by ν.*
   - *If there is not any predecessor of ν that is an interactive node and belongs to the same session, then ν_{state} is a computation node with constructor null.*
   - *ν_{state} is not referenced by other nodes than ν and ν references ν_{state} only as state argument.*
   - *ν_{sender} is a computation node annotated with a constructor $C \in \mathbf{C}$. C is the constructor null if and only if ν_{input} is an input node. If that is the case, ν_{input} is called* interactive input node.
   - *ν_{rand} is a computation node with nonce $N \in \mathbf{N}_{\text{INT}}$. On every path through ν_{rand}, there is no other computation node with nonce N. The interactive node ν is the only node that references ν_{rand} and ν references ν_{rand} only as rand argument.*
5. *Only computation nodes that are the rand argument node of an interactive node may*

[10] Predecessor of ν' is any node on the path from ν' to the root, excluding ν' itself.

be annotated with a nonce $N \in \mathbf{N}_{\text{INT}}$.

6. *Given all bitstrings sent to the adversary in a computational CoSP execution at a given point, it can be decided in deterministic polynomial time whether the next bitstring expected from the adversary (if any) is the input for an interactive input node* ν_{input}. *If that is the case, it is additionally possible to compute the session id belonging to* ν_{input} *efficiently.*

   Furthermore, given all bitstrings sent to the adversary in a computational CoSP execution at a given point, it can be decided in deterministic polynomial time whether the last bitstring sent to the adversary (if any) is the output sent by an interactive output node ν_o. *If that is the case, it is additionally possible to compute the session id belonging to* ν_o *efficiently.*

5.4. Computational Soundness

The main theorem of this chapter states that we can extend a computational soundness result for equivalence properties to a computational soundness result for interactive primitives that are soundly abstracted by ideal functionalities.

Theorem 6 (Computational Soundness for Interactive Primitives)**.** *Let* $\mathbf{M_F}$ *be an extended symbolic model (based on* $\mathbf{M}$*) and* $\mathbf{A}_\Phi$ *be a computational implementation of it, based on* $\mathbf{A}$. *Let* P *be a class of CoSP bi-protocols such that every bi-protocol in* P *fulfills the conditions for interactive primitives. Suppose that every* $\mathcal{F} \in \mathbf{F}$ *is a good ideal functionality and every* $\phi \in \Phi$ *is a good real protocol (see Definitions 60 and 61). Suppose that for every ideal functionality* $\mathcal{F} \in \mathbf{F}$ *and the corresponding real protocol* $\phi \in \Phi$*, we have that* $\mu(\phi)$ *UC-realizes* $\mu(\mathcal{F})$. *If* $\mathbf{A}$ *is a computationally sound implementation of* $\mathbf{M}$ *for* P *with respect to equivalence properties, then* $\mathbf{A}_\Phi$ *is computationally sound for* $\mathbf{M_F}$ *with respect to equivalence properties.*

Proof. For simplicity, we consider only one ideal functionality $\mathcal{F}$ and its implementation ϕ. Let $\Pi \in$ P be a symbolically indistinguishable bi-protocol using the destructor $D_\mathcal{F}$. Lemma 26 entails that the computational execution of Π with the canonical algorithm $A_\mathcal{F}$ is indistinguishable.

Recall that for an interactive primitive ρ (be it $\mathcal{F}$ or ϕ) with a computational implementation A_ρ, there is a CoSP compatible UC machine $\hat{\mu}(\rho)$ (see Definition 59). Since Π fulfills the protocol conditions, the CoSP computational execution of Π can be formulated as a UC machine CoSPUC that calls $\hat{\mu}(\mathcal{F})$ (or $\hat{\mu}(\phi)$, respectively) instead of executing $\mathcal{F}$ (or ϕ) on its own.[11] Because the computational execution with $A_\mathcal{F}$ is indistinguishable, we know that for all protocols $\Pi \in$ P CoSPUC with $left(\Pi)$ is indistinguishable from CoSPUC with $right(\Pi)$ when using $\hat{\mu}(\mathcal{F})$. Since $\tilde{\mu}(\mathcal{F})$ leaks less information than $\hat{\mu}(\mathcal{F})$, we can conclude that for all protocols $\Pi \in$ P CoSPUC with $left(\Pi)$ is indistinguishable from CoSPUC with $right(\Pi)$ when using $\tilde{\mu}(\mathcal{F})$.

[11] The UC machine CoSPUC is a formulation of the CoSP computational execution in the UC framework. In contrast to the CoSP computational execution, however, CoSPUC does not compute an interactive primitive ρ itself but calls $\hat{\mu}(\rho)$ instead. In order to produce the same output as the CoSP computational execution, CoSPUC constructs the CoSP view accordingly. In particular CoSPUC maps the messages (`output ready, m`) and (`input ready,m`) to view entries that correspond to adversary communication.

Suppose that $\mu(\phi)$ UC-realizes $\mu(\mathcal{F})$, and let $\tilde{\mu}(\phi)$ and $\tilde{\mu}(\mathcal{F})$ be its corresponding honest machines, as described in Definition 59. Given that ϕ and $\mathcal{F}$ are good (see Definitions 60 and 61), Lemma 28 shows that $\tilde{\mu}(\phi)$ UC-realizes $\tilde{\mu}(\mathcal{F})$. Since $\tilde{\mu}(\mathcal{F})$ UC-realizes $\tilde{\mu}(\phi)$, it follows that for all protocols $\Pi \in$ P CoSPUC with $\mathit{left}(\Pi)$ is indistinguishable from CoSPUC with $\mathit{right}(\Pi)$ when using $\tilde{\mu}(\phi)$. By the completeness of the dummy adversary, we can w.l.o.g. assume that the network adversary is the dummy adversary. Consequently, the environment learns the communication from the protocol to the network adversary. Recall that the only difference between $\hat{\mu}$ and $\tilde{\mu}$ is that $\hat{\mu}$ additionally leaks this communication from the protocol to the network adversary. Thus, it follows that for all protocols $\Pi \in$ P CoSPUC with $\mathit{left}(\Pi)$ is indistinguishable from CoSPUC with $\mathit{right}(\Pi)$ when using $\hat{\mu}(\phi)$.

□

5.5. Conclusion

This chapter presented the first computational soundness result that enables the extension of a computationally sound symbolic models with a UC-secure ideal functionality.

Chapter 6.

Bridging the Gap: From Trace Properties to Equivalence Properties

[This chapter is based on a paper with Michael Backes and Tim Ruffing [BMR14]. I contributed the idea for this work, and I am the main contributor of the part of the work that occurs in this chapter.]

6.1. Motivation

Most of the previous computational soundness (CS) results against active attacks have been specific to the class of trace properties [BBU13; BCW13; BHU09; BMM10; BU10; CKKW06; CW05; CW11; GGV08; JLM05; MW04], which is only sufficient as long as strong notions of privacy are not considered, e.g., in particular for establishing various authentication properties. Only few CS results are known for the class of equivalence properties against active attackers, which are restricted in of the following three ways: either they are restricted to a small class of simple processes, e.g., processes that do not contain private channels and abort if a conditional fails [CC08; CCS12; CH11], or they rely on non-standard abstractions for which it is not clear how to formalize any equivalence property beyond the secrecy of payloads [BL06; BP04; BPW03a], such as anonymity properties in protocols that encrypt different signatures, or existing automated tool support is not applicable [CHKS12; SBBPW06]. We are thus facing a situation where CS results, despite tremendous progress in the last decade, still fall short in comprehensively addressing the class of equivalence properties and protocols that state-of-the-art verification tools are capable to deal with. Moreover, it is unknown to which extent existing results on CS for trace properties can be extended to achieve more comprehensive CS results for equivalence properties.

Our Contribution. We close this gap by providing the first result that allows to leverage existing CS results for trace properties to CS results for an expressive class of equivalence properties: the uniformity of bi-processes in the applied π-calculus. Bi-processes are pairs of processes that differ only in the messages they operate on but not in their structure; a bi-process is uniform if for all surrounding contexts, i.e., all interacting attackers, both processes take the same branches. Blanchet, Abadi, and Fournet [BAF05] have shown that uniformity already implies observational equivalence. Moreover, uniformity of bi-processes corresponds precisely to the class of properties that the state-of-the-art verification tool ProVerif [BAF05] is capable to analyze, based on a symbolic model in the applied π-calculus. In contrast to previous work dealing with equivalence properties, we consider bi-protocols that use the fully fledged applied π-calculus, in particular including private channels and

non-determinate processes.

To establish this main result of our paper, we first identify the following general condition for symbolic models: "whenever a computational attacker can distinguish a bi-process, there is a test in the symbolic model that allows to successfully distinguish the bi-process." We say that symbolic models with this property *allow for self-monitoring*. We show that if a specific symbolic model fulfills this property, then there is for every bi-process a so-called *self-monitor*, i.e., a process that performs all relevant tests that the attacker could perform on the two processes of the bi-process, and that raises an exception if of these tests in the symbolic model distinguishes the bi-process. We finally show that whenever a symbolic model allows for self-monitoring, CS for uniformity of bi-processes automatically holds whenever CS for trace properties has already been established. This result in particular allows for leveraging existing CS results for trace properties to more comprehensive CS results for uniformity of bi-processes, provided that the symbolic model can be proven to allow for self-monitoring.

We exemplarily show how to construct a self-monitor for a symbolic model that has been recently introduced and proven to be computationally sound for trace properties by Backes, Malik, and Unruh [BMU12]. This symbolic model contains signatures and public-key encryption and allows to freely send and receive decryption keys. To establish that the model allows for self-monitoring, we first extend it using the common concept of a length function (without a length function, CS for uniformity of bi-processes and hence the existence of self-monitors trivially cannot hold, since encryptions of different lengths are distinguishable in general), and we show that this extension preserves the existing proof of CS for trace properties. Our main result in this paper then immediately implies that this extended model satisfies CS for uniformity of bi-processes.

6.2. Self-monitoring

In this section, we identify a sufficient condition for symbolic models under which CS for trace properties implies CS for equivalence properties for a class of *uniformity-enforcing protocols*, which correspond to uniform bi-processes in the applied π-calculus. We say that a symbolic model that satisfies this condition *allows for self-monitoring*. The main idea behind self-monitoring is that a symbolic model is sufficiently expressive (and its implementation is sufficiently strong) such that the following holds: whenever a computational attacker can distinguish a bi-process, there is a test in the symbolic model that allows to successfully distinguish the bi-process.

6.2.1. CS for Trace Properties

A trace property that captures that a certain bad state (a certain class of bad node identifiers) is not reachable, would be formalized as the set of all traces that do not contain bad nodes.

Formally, a trace property is a prefix-closed set of node-traces. We say that a protocol Π *symbolically satisfies* a trace property $\wp$ if for all traces t resulting from a symbolic execution, $\wp$ holds for t, i.e., $t \in \wp$. Correspondingly, we state that a protocol Π *computationally satisfies* $\wp$ if for any ppt attacker $\mathcal{A}$ and polynomial p the probability is overwhelming that

$\wp$ holds for t, i.e., $t \in \wp$, for any trace t resulting from a computational execution with $\mathcal{A}$ and p.

We first review the relevant definitions from the original CoSP framework. Instead of a view that contains the communication with the attacker, we are interested in a trace of the execution of a protocol.

Definition 65 (Symbolic and Computational Traces)**.** *Let (V_i, ν_i, f_i) be a (finite) list of triples produced internally by the symbolic execution (Definition 16) of a CoSP protocol Π. Then a list ν_i is a* symbolic trace *of Π.*

Analogously, given a polynomial p, the probability distribution on the list ν_i computed by the computational execution (Definition 21) of Π with polynomial p is called computational trace *of Π.*

CS for trace properties states that all attacks (against trace properties) that can be excluded for the symbolic abstraction can be excluded for the computational implementation as well. Hence, if all the symbolic traces satisfy a certain trace property, then all computational traces satisfy this property as well.

Definition 66 (Computational Soundness for Trace Properties [BHU09])**.** *A symbolic model $(\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D})$ is* computationally sound for trace properties *with respect to an implementation $\mathbf{A}$ for a class P of efficient protocols if the following holds: for each protocol $I \in \mathrm{P}$ and each trace property $\wp$, I computationally satisfies $\wp$ whenever I symbolically satisfies $\wp$.*

Uniformity-enforcing. For the connection to trace properties, we consider only *uniform* bi-protocols. A bi-protocol is uniform if for each symbolic attacker strategy, both its variants reach the same nodes in the CoSP tree, i.e., they never branch differently.[1] Formally, we require that the bi-protocols are *uniformity-enforcing*, i.e., when the left and the right protocol of the bi-protocol Π take different branches, the attacker is informed. Since taking different branches is only visible after a control node is reached, we additionally require that computation nodes are immediately followed by control nodes.

Definition 67 (Uniformity-enforcing)**.** *A class P of CoSP bi-protocols is* uniformity-enforcing *if for all bi-protocols $\Pi \in \mathrm{P}$:*

1. *Every control node in Π has unique out-metadata.*
2. *For every computation node ν in Π and for every path rooted at ν, a control node is reached before an output node.*

All embeddings of calculi the CoSP framework described so far, namely those of the applied π-calculus [BHU09] and RCF [BMU10], are formalized such that protocols written in these calculi fulfill the second property: these embeddings give the attacker a scheduling decision, using a control node, basically after every execution step. We stress that it is straightforward to extend them to fulfill the first property by tagging the out-metadata with the address of the node in the protocol tree.

[1] We show in Lemma 1 that uniformity of bi-protocols in CoSP corresponds to uniformity of bi-processes in the applied π-calculus.

6.2.2. Bridging the Gap from Trace Properties to Uniformity

The key observation for the connection to trace properties is that, given a bi-protocol Π, some computationally sound symbolic models enable the construction of a self-monitor $\mathsf{Mon}(\Pi)$ – a protocol, not a bi-protocol – that has essentially the same interface to the attacker as the bi-protocol Π and checks at run-time whether Π would behave uniformly. In other words, non-uniformity of bi-protocols can be formulated as a trace property `bad`, which can be detected by the protocol $\mathsf{Mon}(\Pi)$.

The self-monitor $\mathsf{Mon}(\Pi)$ of a bi-protocol Π behaves like one of the two variants of the bi-protocol Π, while additionally simulating the opposite variant such that $\mathsf{Mon}(\Pi)$ itself is able to detect whether Π would be distinguishable. (For instance, one approach to detect whether Π is distinguishable could consist of reconstructing the symbolic view of the attacker in the variant of Π that is not executed by $\mathsf{Mon}(\Pi)$.) At the beginning of the execution of the self-monitor, the attacker chooses if $\mathsf{Mon}(\Pi)$ should basically behave like $\mathit{left}(\Pi)$ or like $\mathit{right}(\Pi)$. We denote the chosen variant as $b \in \{\mathit{left}, \mathit{right}\}$ and the opposite variant as $\bar{b}$. After this decision, $\mathsf{Mon}(\Pi)$ executes the the b-variant $b(\Pi)$ of the bi-protocol Π, however, enriched with the computation nodes and the corresponding output nodes of the opposite variant $\bar{b}(\Pi)$.[2]

The goal of the self-monitor $\mathsf{Mon}(\Pi)$ is to detect whether the execution of $b(\Pi)$ would be distinguishable from $\bar{b}(\Pi)$ at the current state. If this is the case, $\mathsf{Mon}(\Pi)$ raises the event `bad`, which is the disjunction of two events `bad-branch` and `bad-knowledge`.

The event `bad-branch` corresponds to the case that the left and the right protocol of the bi-protocol Π take different branches. Since uniformity-enforcing protocols have a control node immediately after every computation node (see Definition 67), the attacker can always check whether $b(\Pi)$ and $\bar{b}(\Pi)$ take the same branch. We require (in Definition 69) the existence of a so-called *distinguishing subprotocol* $f_{\texttt{bad-branch},\Pi}$ that checks whether each destructor application in $b(\Pi)$ succeeds if and only if it succeeds in $\bar{b}(\Pi)$; if not, the distinguishing subprotocol $f_{\texttt{bad-branch},\Pi}$ raises `bad-branch`.

The event `bad-knowledge` captures that the messages sent by $b(\Pi)$ and $\bar{b}(\Pi)$ (via output nodes, i.e., not the out-metadata) are distinguishable. This distinguishability is only detectable by a protocol if the constructors and destructors, which are available to both the protocol and the symbolic attacker, capture all possible tests. We require (in Definition 69) the existence of a distinguishing subprotocol $f_{\texttt{bad-knowledge},\Pi}$ that raises `bad-knowledge` in $\mathsf{Mon}(\Pi)$ whenever a message, sent in Π, would allow the attacker to distinguish $b(\Pi)$ and $\bar{b}(\Pi)$.

Parameterized CoSP Protocols. For a bi-protocol Π, we formalize the distinguishing subprotocols $f_{\texttt{bad-knowledge},\Pi}$ and $f_{\texttt{bad-branch},\Pi}$ with the help of *parameterized CoSP protocols*, which have the following properties: Nodes in such protocols are not required to have successors and instead of other nodes, also formal parameters can be referenced. A parameterized CoSP protocol is intended to be plugged into another protocol; in that case

[2] This leads to the fact that whenever there is an output node in Π, there are two corresponding output nodes in $\mathsf{Mon}(\Pi)$, which contradicts the goal that the interface of Π and $\mathsf{Mon}(\Pi)$ should be the same towards the attacker. However, this technicality can be dealt with easily when applying our method. For example, in the computational proof for our case study, we use the self-monitor in an interaction with a filter machine that hides the results of the output nodes of $\bar{b}(\Pi)$ to create a good simulation towards the computational attacker, whose goal is to distinguish Π. The filter machine is then used as a computational attacker against $\mathsf{Mon}(\Pi)$.

the parameters references must be changed to references to nodes.

Definition 68 (Self-monitor)**.** *Let* Π *be a CoSP bi-protocol. Let* $f_{\texttt{bad-knowledge},\Pi}$ *and* $f_{\texttt{bad-branch},\Pi}$ *be functions that map execution traces to parameterized CoSP protocols*[3] *whose leaves are either* ok*, in which case they have open edges, or halt in distinguished states* `bad-knowledge`*, or* `bad-branch` *respectively, i.e., they end in an infinite sequence of control nodes with out-metadata* `bad-knowledge`*, or* `bad-branch` *respectively.*

Recall that nodes ν *of* Π *have bi-references (as defined in Definition 25) consisting of a left reference (to be used in the left protocol) and a right reference. We write* $left(\nu)$ *for the node with only the left reference and* $right(\nu)$ *analogously. For each node* ν *in* Π*, let* tr_ν *be the execution trace of* Π *that leads to* ν*, i.e., the list of node and edge identifiers on the path from the root of* Π *to* ν*, including* ν*. The* self-monitor $\mathsf{Mon}(\Pi)$ *protocol is defined as follows:*

Insert before the root node a control node with two copies of Π*, called the left branch (with* $b := left$*) and the right branch (with* $b := right$*). Apply the following modifications recursively for each node* ν*, starting at the root of* Π*:*

1. *If* ν *is a* computation node of Π*, replace* ν *with* $f_{\texttt{bad-branch},\Pi}(b, tr_\nu)$*. Append two copies* $left(\nu)$ *and* $right(\nu)$ *of the the computation node* ν *to each open edge of an* ok*-leaf. All left references that pointed to* ν *point in* $\mathsf{Mon}(\Pi)$ *to* $left(\nu)$*, and all right references that pointed to* ν *point in* $\mathsf{Mon}(\Pi)$ *to* $right(\nu)$*. The successor of* $right(\nu)$ *is the subtree rooted at the successor of* ν*.*
2. *If* ν *is an* output node *of* Π*, replace* ν *with* $f_{\texttt{bad-knowledge},\Pi}(b, tr_\nu)$*. Append the sequence of the two output nodes* $left(\nu)$ *(labeled with* `left`*) and* $right(\nu)$ *(labeled with* `right`*) to each open edge of an* ok*-leaf. All left references that pointed to* ν *point in* $\mathsf{Mon}(\Pi)$ *to* $left(\nu)$*, and all right references that pointed to* ν *point in* $\mathsf{Mon}(\Pi)$ *to* $right(\nu)$*. The successor of* $right(\nu)$ *is the subtree rooted at the successor of* ν*.*

The sub-sequence of a trace of a self-monitor that contains only those nodes that are labeled with `left` *is called the* left trace $left(tr)$ of $\mathsf{Mon}(\Pi)$*. Analogously, the sub-sequence of a trace of a self-monitor that contains only those nodes that are labeled with* `right` *is called the* right trace $right(tr)$ of $\mathsf{Mon}(\Pi)$*.*

Recall that our goal is to identify sufficient conditions under which computational soundness (CS) w.r.t. trace properties implies CS w.r.t. equivalence properties (see Theorem 7). We require three properties. Beside the requirement that the class of CoSP protocols has to be uniformity enforcing for which CS w.r.t. trace properties holds, we require two properties from the distinguishing subprotocols: *symbolic self-monitoring* and *computational self-monitoring* (see Figure 6.1). Symbolic self-monitoring states that whenever a bi-protocol Π is indistinguishable, the corresponding distinguishing subprotocol in $\mathsf{Mon}(\Pi)$ does not raise the event `bad`. Computational self-monitoring, in turn, states that whenever the distinguishing subprotocol in $\mathsf{Mon}(\Pi)$ does not raises the event `bad`, then Π is indistinguishable.

Shortened Protocols Π_i**.** We prove these three requirement sufficient by induction over the nodes in a bi-protocol. As a technical vehicle, we introduce a notion of *shortened protocols* in the definition of distinguishing subprotocols. For a (bi-)protocol Π, the

[3] These functions are candidates for distinguishing subprotocols for `bad-knowledge` and `bad-branch`, respectively, for the bi-protocol Π, as defined in Definition 69.

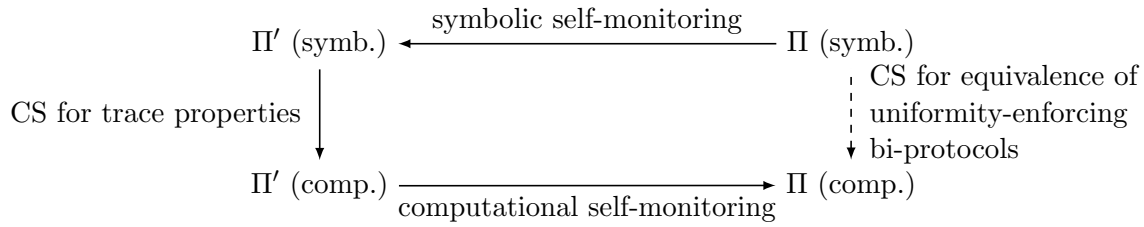

Figure 6.1.: Symbolic and computational self-monitoring.

shortened (bi-)protocol Π_i is for the first i nodes exactly like Π but that stops after the ith node that is either a control node or an output node.[4]

Definition 69 (Allowing for self-monitoring)**.** *Let* M *be a symbolic model and* Impl *a computational implementation of* $\mathbf{M}$*. Let* Π *be a bi-protocol and* $\mathsf{Mon}(\Pi)$ *its self-monitor. Let* $e \in \{\texttt{bad-knowledge}, \texttt{bad-branch}\}$ *and* $n_{\texttt{bad-knowledge}}$ *denote the node type output node and* $n_{\texttt{bad-branch}}$ *denote the node type control node. Then the function* $f_{e,\Pi}(b, tr)$*, which takes as input* $b \in \{left, right\}$ *and the path to the root node, including all node and edge identifiers, is a* distinguishing subprotocol for e *for* Π *and* M *if the following conditions hold for every* $i \in \mathbb{N}$*:*

1. symbolic self-monitoring: *If* Π_i *is symbolically indistinguishable,* bad *does symbolically not occur in* $\mathsf{Mon}(\Pi_{i-1})$*, and the* i*th node in* Π_i *is of type* n_e*, then the event* e *does not occur symbolically in* $\mathsf{Mon}(\Pi_i)$*.*
2. computational self-monitoring: *If the event* e *in* $\mathsf{Mon}(\Pi_i)$ *occurs computationally with at most negligible probability,* Π_{i-1} *is computationally indistinguishable, and the* i*th node in* Π_i *is of type* n_e*, then* Π_i *is computationally indistinguishable.*
3. *The difference in the running time of* Π_i *and* $\mathsf{Mon}(\Pi_i) - \mathrm{F}_b$ *polynomially bounded in* i*.*

We say that a model M*, an implementation* Impl*, and a protocol class* P allow for self-monitoring *if for every bi-protocol* Π *in the protocol class* P*, there are distinguishing subprotocols for* bad-branch *and* bad-knowledge *such that the resulting self-monitor* $\mathsf{Mon}(\Pi)$ *is also in* P*.*

Finally, we are ready to state our main theorem.

Theorem 7. *Let* M *be a symbolic model,* P *be an efficient uniformity-enforcing class of bi-protocols, and* Impl *be an implementation for* M*. If* $\mathsf{M}, \mathrm{Impl}, \mathrm{P}$ *allow for self-monitoring (in the sense of Definition 69), then the following holds: If* Impl *is a computationally sound implementation of a symbolic model* M *with respect to trace properties then* Impl *is also a computationally sound implementation with respect to equivalence properties.*

Proof. Let $\mathbf{A}$ be a computationally sound implementation of $\mathbf{M}$ for trace properties. Assume that Π is symbolically indistinguishable. The goal is show that Π is also computationally indistinguishable.

First, we show that bad does not happen symbolically in $\mathsf{Mon}(\Pi)$. To this end, we show by induction on i that bad does not happen symbolically in $\mathsf{Mon}(\Pi_i)$ for all $i \in \mathbb{N}$. For the

[4]Formally, the protocol only has an infinite chain of control nodes with single successors after this node.

induction base $i = 0$, i.e., the empty execution, this is clear. For $i > 0$, we know by the induction hypothesis that `bad` does not happen symbolically in $\mathsf{Mon}(\Pi_{i-1})$. Furthermore, Π_i is symbolically indistinguishable, because Π is symbolically indistinguishable. We distinguish cases on the type of the ith control or output node ν_i that is reached in Π. Assume it is an control node. Then we know by Item 2 of the uniformity-enforcing property that Π_i contains in contrast to Π_{i-1} only additional input and computation nodes (and ν_i itself). Thus by construction of $\mathsf{Mon}(\Pi)$, we know that $\mathsf{Mon}(\Pi_i)$ does not contain an additional distinguishing subprotocol for `bad-knowledge` in contrast to $\mathsf{Mon}(\Pi_{i-1})$, i.e., `bad-knowledge` does not happen in $\mathsf{Mon}(\Pi_i)$. Since ν_i is a control node, Item 1 of Definition 69 implies that `bad-branch` does not happen in $\mathsf{Mon}(\Pi_i)$ either. The case that ν_i is an output node is analogous. Taken together, `bad` does not happen in $\mathsf{Mon}(\Pi_i)$. This concludes the induction proof.

Since M and P allow for self-monitoring, the self-monitor $\mathsf{Mon}(\Pi)$ lies in the protocol class P. As `bad` does not happen symbolically in $\mathsf{Mon}(\Pi)$, CS for trace properties thus entails that `bad` happens computationally in $\mathsf{Mon}(\Pi)$ only with negligible probability. This implies computational indistinguishability with essentially the same arguments as in the symbolic execution apply in a reverse manner, by using Item 2 of Definition 69. □

6.3. Case Study: Encryption and Signatures with Lengths

We exemplify our method by proving a CS result for equivalence properties, which captures protocols that use public-key encryption and signatures. We use the CS result in [BMU12] for trace properties, which we extend by a length function, realized as a destructor. Since encryptions of plaintexts of different length can typically be distinguished,we must reflect that fact in the symbolic model.

6.3.1. The Symbolic Model

Lengths in the Symbolic Model. In order to express lengths in the symbolic model, we introduce *length specifications*, which are the result of applying a special destructor $len/1$. We assume that the bitlength of every computational message m_c is of the form $|m_c| = rk$ for some natural number r, where k is the security parameter, i.e., the length of a nonce. This assumption will be made precise. With this simplification, length specifications only encode r; this can be done using Peano numbers, i.e., the constructors O (zero) and S (successor).

Even though this approach leads admittedly to rather inefficient realizations from a practical point of view,[5] the aforementioned assumption can be realized using a suitable padding. Essentially, this assumption is similar to the one introduced by Comon-Lundh and Cortier [CC08] for a symbolic model for symmetric encryption. The underlying problem is exactly the same: while the length of messages in the computational model, in particular the length of ciphertexts, may depend on the security parameter, there is no equivalent concept in the symbolic model. For instance, let n and m be nonces, and let ek be an encryption key. For certain security parameters in the computational model, the

[5] Consider, e.g., a payload string that should convey n bits. This message must be encoded using at least kn bits.

computational message $pair(n, m)$ may have the same length as the message $enc(ek, n)$; for other security parameters this may not be the case. Thus it is not clear if the corresponding symbolic messages should be of equal symbolic length. Comon-Lundh, Hagiya, Kawamoto, and Sakurada [CHKS12] propose a different approach towards this problem, by labeling messages symbolically with an expected length and checking the correctness of these length computationally. However, it is not clear whether such a symbolic model can be handled by current automated verification tools.

Automated Verification: Combinding ProVerif and APTE. ProVerif is not able to handle recursive destructors such as *len*, e.g., $len(pair(t_1, t_2)) = len(t_1) + len(t_2)$. Recent work by Cheval, Cortier, and Plet [CCP13] extends the protocol verifier APTE, which is capable of proving trace equivalence of two processes in the applied π-calculus, to support such length functions. Since however trace equivalence is a weaker notion than uniformity, i.e., there are bi-processes that are trace equivalent but not uniform, our CS result does not carry over to APTE. Due to the lack of a tool that is able to check uniformity as well as to handle length functions properly, we elaborate and prove in Section 2.6 how APTE can be combined with ProVerif to make protocols on the symbolic model of our case study amenable to automated verification.

We consider the following symbolical model $\mathsf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D})$.

Constructors and Nonces. We define $\mathbf{C} := \{enc/3, ek/1, dk/1, sig/3, vk/1, sk/1, string_0/1,$
$string_1/1, empty/0, pair/2, O/0, S/1, garbageEnc/3, garbageSig/3, garb/2,$
$garbageInvalidLength/1\}$ and $\mathbf{N} := \mathbf{N_E} \uplus \mathbf{N_E}$ for countably infinite sets of protocol nonces $\mathbf{N_P}$ and attacker nonces $\mathbf{N_E}$. Encryption, decryption, verification, and signing keys are represented as $ek(r), dk(r), vk(r), sk(r)$ with a nonce r (the randomness used when generating the keys). The term $enc(ek(r'), m, r)$ encrypts m using the encryption key $ek(r')$ and randomness r. $sig(sk(r'), m, r)$ is a signature of m using the signing key $sk(r')$ and randomness r. The constructors $string_0$, $string_1$, and *empty* are used to model arbitrary strings used as payload in a protocol, e.g., a bitstring 010 would be encoded as $string_0(string_1(string_0($ $empty()))))$. Length specifications can be constructed using O representing zero and S representing the successor of a number. *garb*, *garbageInvalidLength*, *garbageEnc*, and *garbageSig* are not used by the protocol; they express invalid terms the attacker may send.

Message Type. We define $\mathbf{T}$ as the set of terms M according to this grammar:

$$\begin{aligned} M ::= {} & enc(ek(N), M, N) \mid ek(N) \mid dk(N) \mid \\ & sig(sk(N), M, N) \mid vk(N) \mid sk(N) \mid pair(M, M) \mid P \mid N \mid L \mid \\ & garb(N, L) \mid garbageInvalidLength(N) \\ & garbageEnc(M, N, L) \mid garbageSig(M, N, L) \\ P ::= {} & emp() \mid string_0(P) \mid string_1(P) \qquad L ::= O() \mid S(L) \end{aligned}$$

The nonterminals P, N, and L represent payloads, nonces, and length specifications, respectively. Note that the garbage terms carry an explicit length specification to enable the attacker to send invalid terms of a certain length.

Destructors. We define the set $\mathbf{D}$ of destructors of the symbolic model $\mathbf{M}$ by $\mathbf{D} :=$ $\{dec/2, isenc/1, isek/1, isdk/1, ekof/1, ekofdk/1, equals/2, verify/2, isvk/1, issk/1, issig/1,$ $vkofsk/1, vkof/1, fst/1, snd/1, unstring_0/1, unstring_1/1, len/1, unS/1\}$. The destructors *isek*,

$$
\begin{array}{rcll}
dec(dk(t_1), enc(ek(t_1), m, t_2)) & = & m & \\
isenc(enc(ek(t_1), t_2, t_3)) & = & enc(ek(t_1), t_2, t_3) & \\
isenc(garbageEnc(t_1, t_2, l)) & = & garbageEnc(t_1, t_2, l) & \\
isek(ek(t)) & = & ek(t) & \\
ekof(enc(ek(t_1), m, t_2)) & = & ek(t_1) & \\
ekof(garbageEnc(t_1, t_2, l)) & = & t_1 & \\
\\
equals(t_1, t_1) & = & t_1 & \\
verify(vk(t_1), sig(sk(t_1), t_2, t_3)) & = & t_2 & \\
isvk(vk(t_1)) & = & vk(t_1) & \\
issk(sk(t_1)) & = & sk(t_1) & \\
issig(sig(sk(t_1), t_2, t_3)) & = & sig(sk(t_1), t_2, t_3) & \\
issig(garbageSig(t_1, t_2, l)) & = & garbageSig(t_1, t_2, l) & \\
vkof(sig(sk(t_1), t_2, t_3)) & = & vk(t_1) & \\
vkof(garbageSig(t_1, t_2, l)) & = & t_1 & \\
vkofsk(sk(t_1)) & = & vk(t_1) & \\
fst(pair(x, y)) & = & x & \\
snd(pair(x, y)) & = & y & \\
unstring_b(string_b(s)) & = & s & \forall b \in \{0, 1\} \\
unS(S(t)) & = & t &
\end{array}
$$

Figure 6.2.: The destructor definitions for the symbolic models

isdk, *isvk*, *issk*, *isenc*, and *issig* realize predicates to test whether a term is an encryption key, decryption key, verification key, signing key, ciphertext, or signature, respectively. *ekof* extracts the encryption key from a ciphertext, *vkof* extracts the verification key from a signature. $dec(dk(r), c)$ decrypts the ciphertext c. $verify(vk(r), s)$ verifies the signature s with respect to the verification key $vk(r)$ and returns the signed message if successful. *ekofdk* and *vkofsk* compute the encryption/verification key corresponding to a decryption/signing key. The destructors *fst* and *snd* are used to destruct pairs, and the destructors $unstring_0$ and $unstring_1$ allow to parse payload-strings. The destructor *len* returns a the length of message, where the unit is the length of a nonce. The purpose of *unS* is destruct length specifications. (Destructors *ispair* and *isstring* are not necessary, they can be emulated using *fst*, $unstring_i$, and $equals(\cdot, empty)$.) The precise cancellation rules for destructors (except for *len*) are depicted in Figure 6.2; application matching none of these rules evaluates to $\bot$:

Length Destructor. Our result is parametrized over the destructor *len* that must adhere to the following restrictions:

1. Each message except for *garbageInvalidLength* is assigned a length:
   $len(t) \neq \bot$ for all terms $t \in \mathbf{T} \setminus \{garbageInvalidLength(t') \mid t' \in \mathbf{T}\}$.
2. The length of garbage terms (constructed by the attacker) is consistent:

$$
\begin{array}{rcll}
len(garb(t, l)) & = & l, & len(garbageEnc(t_1, t_2, l)) = l, \\
len(garbageSig(t_1, t_2, l)) & = & l, & len(garbageInvalidLength(t_1)) = \bot
\end{array}
$$

3. Let $[\cdot]$ be the canonical interpretation of Peano numbers, given by $[O()] = 0$ and $[S(l)] = [l] + 1$. We require the length destructor to be linear: For each constructor

$C/n \in \mathbf{C} \setminus \{garb,\ garbageInvalidLength, garbageEnc, garbageSig\}$ there are $a_i \in \mathbb{N}$ (where $i = 0, \ldots, n$) such that $len(t_i) = l_i$ for $i = 1, \ldots, n$ and $len(C(\underline{t})) = l$ together imply $[l] = \sum_{i=1}^{n} a_i \cdot [l_i] + a_0$.

Length specifications are ordinary messages that the protocol can send, receive and process. Thus we require length specifications to have a length itself.

6.3.2. Implementation Conditions

A computationally sound implementation of the symbolic model M has to adhere to the conditions given below, which are essentially the same as in [BMU12]. Since the message type $\mathbf{T}$ used here includes length specifications, i.e., an additional type of messages that represents natural numbers (see Section 6.3.1), we need basic implementation conditions such as $A_{unS}(A_S(m)) = m$ for messages m of type length specification. Furthermore, the algorithm that implements the length destructor must compute the bitlength of the argument correctly. These additional requirements are highlighted in blue. We stress that the strong requirements on the encryption scheme, which require the random oracle model, namely PROG-KDM security [Unr12], are used only to handle protocols that send and receive decryption keys. We refer to [BMU12] for more details. In principle, our proofs do not rely on this particular security definition. We conjecture that is possible to obtain a computational soundness result for uniformity using weaker implementation conditions (IND-CCA secure public-key encryption) but a restricted protocol class, by applying our proof technique to the computational soundness result for trace properties in [BHU09]; we leave a formal treatment for future work however.

For lengths in the computational model, we require that the destructor *len* as well as its computational implementation Impl_{len} compute indeed the bitlength of its argument. To connect the symbolic result of the destructor *len* to bitlengths in the computational world, the destructor must be consistent with its implementation.

The following list of conditions is copied verbatim from [BMU12] except for the parts that are marked in blue.

1. Impl is an implementation for M in the sense of Definition 6.
2. There are disjoint and efficiently recognizable sets of bitstrings representing the types nonces, ciphertexts, encryption keys, decryption keys, signatures, verification keys, signing keys, pairs, payload-strings, length specifications, and invalid-length. The set of all bitstrings of type nonce we denote Nonces_k.[6] (Here and in the following, k denotes the security parameter.)
3. The functions A_{enc}, A_{ek}, A_{dk}, A_{sig}, A_{vk}, A_{sk}, and A_{pair} are length-regular. We call an n-ary function f length regular if $|m_i| = |m_i'|$ for $i = 1, \ldots, n$ implies $|f(\underline{m})| = |f(\underline{m'})|$. All $m \in \mathrm{Nonces}_k$ have the same length.
4. A_N for $N \in \mathbf{N}$ returns a uniformly random $r \in \mathrm{Nonces}_k$.
5. Every image of A_{enc} is of type ciphertext, every image of A_{ek} and A_{ekof} is of type encryption key, every image of A_{dk} is of type decryption key, every image of A_{sig} is of type signature, every image of A_{vk} and A_{vkof} is of type verification key, every image of A_{empty}, A_{string_0}, and A_{string_1} is of type payload-string, every image of A_S

[6] This would typically be the set of all bitstrings with length $k - t$, with a tag of length t denoting nonces.

and A_O is of type length specification. Every $m \in \{0,1\}^*$ such that no $r \in \mathbb{N}$ with $|m| = rk$ exists, is of type invalid-length.

6. For all $m_1, m_2 \in \{0,1\}^*$ we have $A_{fst}(A_{pair}(m_1, m_2)) = m_1$ and $A_{snd}(A_{pair}(m_1, m_2)) = m_2$. Every m of type pair is in the range of A_{pair}. If m is not of type pair, $A_{fst}(m) = A_{snd}(m) = \bot$.
7. For all m of type payload-string we have that $A_{unstring_i}(A_{string_i}(m)) = m$ and $A_{unstring_i}(A_{string_j}(m)) = \bot$ for $i, j \in \{0,1\}$, $i \neq j$. For $m = A_{empty}()$ or m not of type payload-string, $A_{unstring_0}(m) = A_{unstring_1}(m) = \bot$. Every m of type payload-string is of the form $m = A_{string_0}(m')$ or $m = A_{string_1}(m')$ or $m = empty$ for some m' of type payload-string. For all m of type payload-string, we have $|A_{string_0}(m)|, |A_{string_1}(m)| > |m|$.
8. For all m of type length specification we have that $A_{unS}(A_S(m)) = m$. For $m = A_O()$ or m not of type number, $A_{unS}(m) = \bot$.
9. $A_{ekof}(A_{enc}(p, x, y)) = p$ for all p of type encryption key, $x \in \{0,1\}^*$, $y \in \text{Nonces}_k$. $A_{ekof}(e) \neq \bot$ for any e of type ciphertext and $A_{ekof}(e) = \bot$ for any e that is not of type ciphertext.
10. $A_{vkof}(A_{sig}(A_{sk}(x), y, z)) = A_{vk}(x)$ for all $y \in \{0,1\}^*$, $x, z \in \text{Nonces}_k$. $A_{vkof}(e) \neq \bot$ for any e of type signature and $A_{vkof}(e) = \bot$ for any e that is not of type signature.
11. $A_{enc}(p, m, y) = \bot$ if p is not of type encryption key.
12. $A_{dec}(A_{dk}(r), m) = \bot$ if $r \in \text{Nonces}_k$ and $A_{ekof}(m) \neq A_{ek}(r)$. (This implies that the encryption key is uniquely determined by the decryption key.)
13. $A_{dec}(d, c) = \bot$ if $A_{ekof}(c) \neq A_{ekofdk}(d)$ or $A_{ekofdk}(d) = \bot$.
14. $A_{dec}(d, A_{enc}(A_{ekofdk}(e), m, r)) = m$ if $r \in \text{Nonces}_k$ and $d := A_{ekofdk}(e) \neq \bot$.
15. $A_{ekofdk}(d) = \bot$ if d is not of type decryption key.
16. $A_{ekofdk}(A_{dk}(r)) = A_{ek}(r)$ for all $r \in \text{Nonces}_k$.
17. $A_{vkofsk}(s) = \bot$ if s is not of type signing key.
18. $A_{vkofsk}(A_{sk}(r)) = A_{vk}(r)$ for all $r \in \text{Nonces}_k$.
19. $A_{dec}(A_{dk}(r), A_{enc}(A_{ek}(r), m, r')) = m$ for all $r, r' \in \text{Nonces}_k$.
20. $A_{verify}(A_{vk}(r), A_{sig}(A_{sk}(r), m, r')) = m$ for all $r, r' \in \text{Nonces}_k$.
21. For all $p, s \in \{0,1\}^*$ we have that $A_{verify}(p, s) \neq \bot$ implies $A_{vkof}(s) = p$.
22. $A_{isek}(x) = x$ for any x of type encryption key. $A_{isek}(x) = \bot$ for any x not of type encryption key.
23. $A_{isvk}(x) = x$ for any x of type verification key. $A_{isvk}(x) = \bot$ for any x not of type verification key.
24. $A_{isenc}(x) = x$ for any x of type ciphertext. $A_{isenc}(x) = \bot$ for any x not of type ciphertext.
25. $A_{issig}(x) = x$ for any x of type signature. $A_{issig}(x) = \bot$ for any x not of type signature.
26. We define an encryption scheme $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ as follows: KeyGen picks a random $r \leftarrow \text{Nonces}_k$ and returns $(A_{ek}(r), A_{dk}(r))$. $\mathsf{Enc}(p, m)$ picks a random $r \leftarrow \text{Nonces}_k$ and returns $A_{enc}(p, m, r)$. $\mathsf{Dec}(k, c)$ returns $A_{dec}(k, c)$. We require that then $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is PROG-KDM secure.

27. Additionally, we require that $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is malicious-key extractable.
28. We define a signature scheme $(\mathsf{SKeyGen}, \mathsf{Sig}, \mathsf{Verify})$ as follows: $\mathsf{SKeyGen}$ picks a random $r \leftarrow \text{Nonces}_k$ and returns $(A_{vk}(r), A_{sk}(r))$. $\mathsf{Sig}(p, m)$ picks a random $r \leftarrow \text{Nonces}_k$ and returns $A_{sig}(p, m, r)$. $\mathsf{Verify}(p, s, m)$ returns 1 iff $A_{verify}(p, s) = m$. We require that then $(\mathsf{SKeyGen}, \mathsf{Sig}, \mathsf{Verify})$ is strongly existentially unforgeable.
29. For all e of type encryption key and all $m, m' \in \{0,1\}^*$, the probability that $A_{enc}(e, m, r) = A_{enc}(e, m', r')$ for uniformly chosen $r, r' \in \text{Nonces}_k$ is negligible.
30. For all $r_s \in \text{Nonces}_k$ and all $m \in \{0,1\}^*$, the probability that $A_{sig}(A_{sk}(r_s), m, r) = A_{sig}(A_{sk}(r_s), m, r')$ for uniformly chosen $r, r' \in \text{Nonces}_k$ is negligible.
31. A_{ekofdk} is injective. (That is, the encryption key uniquely determines the decryption key.)
32. A_{vkofsk} is injective. (That is, the verification key uniquely determines the signing key.)
33. For all $m \in \{0,1\}^*$ that are not of type invalid-length, $A_{len}(m) = A_S^{|m|/k}(A_O())$, where k is the security parameter and A_S^n is the n-fold application of A_S. For all m of type invalid-length, $A_{len}(m) = \bot$.
34. Given a security parameter k, a computational variant of a message $m \in \mathbf{T}$ is obtained by implementing each constructor C and nonce N in m by the corresponding algorithm A_C or A_N, respectively. (For example, for all random choices of $A_N()$, $A_{pair}(A_{string_0}(A_{empty}(), A_{ek}(A_N())))$ is a computational variant of the message $pair(string_0(empty(), ek(N)))$, where $N \in \mathbf{N}$.) We require that for each message $m \in \mathbf{T}$ and all of its computational variants m_k under security parameter k, we have that $len(m) \neq \bot$ implies $|m_k| = [len(m)] \cdot k$. Note that $|m_k|$ is well-defined, because length-regularity (implementation condition 3 in Section 6.3.2) ensures that $|m_k|$ does not depend on randomness.

6.3.3. Randomness-safe Bi-protocols

The CoSP bi-protocols we consider are almost exactly those bi-protocol for which both the left and the right variant are randomness-safe as defined in [BMU12]. The reason is that the self-monitor Π' for the case study, which uses the distinguishing subprotocols from Section 6.3.4 and Section 6.3.5, is in the protocol class of randomness-safe protocol considered in [BMU12] if the left and the right variant of the corresponding bi-protocol Π are randomness-safe. The exact definition is as follows, changes in comparison to [BMU12] are highlighted in blue. The new condition 6 ensures that no constructors are applied to messages with invalid lengths, i.e., terms of the form $garbageInvalidLength(n)$ for a nonce in the symbolic model. Such messages are not generated by the protocol (5). If they have been received at an input node, the protocol must check if at least one destructor does not fail before applying a constructor. This suffices as all destructors fail if one of their arguments has an invalid length.

The following definition is copied verbatim from [BMU12] except for the parts that are marked in blue.

Definition 70 (Randomness-safe Bi-Protocol)**.** *A CoSP* ***bi****-protocol* Π *is* randomness-safe *if both its variants, i.e.,* $left(\Pi)$ *and* $right(\Pi)$*, fulfill the following conditions:*

1. *The argument of every ek-, dk-, vk-, and sk-computation node and the third argument of every enc- and sig-computation node is an N-computation node with $N \in \mathbf{N_P}$, i.e., protocol-generated randomness. (Here and in the following, we call the nodes referenced by a protocol node its arguments.) We call these N-computation nodes* randomness nodes*. Any two randomness nodes on the same path are annotated with different nonces.*
2. *Every computation node that is the argument of an ek-computation node or of a dk-computation node on some path p occurs only as argument to ek- and dk-computation nodes on that path p.*
3. *Every computation node that is the argument of a vk-computation node or of an sk-computation node on some path p occurs only as argument to vk- and sk-computation nodes on that path p.*
4. *Every computation node that is the third argument of an E-computation node or of a sig-computation node on some path p occurs exactly once as an argument in that path p.*
5. *There are no computation nodes with the constructors garb, garbageEnc, garbageSig, garbageInvalidLength, or $N \in \mathbf{N_E}$.*
6. *Every computation node annotated ν with a constructor refers only to argument nodes ν' that fulfill one of the these conditions:*
   a) *Either ν' does not depend on an input node, i.e., no input node is reachable by following (transitively) the references to argument nodes, or*
   b) *ν is in the yes-branch of a computation node with a destructor that has ν' as one of its arguments.*

Accordingly, a CoSP protocol Π *is* randomness-safe *it is the left or right variant of a bi-protocol that is randomness-safe. Throughout the rest of the paper, we use* P *to denote the class of CoSP bi-protocols that are randomness-safe. In abuse of notation, if the context prevents confusion, we also use* P *for the class of CoSP protocols that are randomness-safe.*

Theorem 8. *Let* M *be the symbolic model from Section 6.3.1,* P *be the class of uniformity-enforcing of randomness-safe bi-protocols, and* Impl *an implementation that satisfies the conditions from Section 6.3.2. Then,* M *allows for self-monitoring for* P*. In particular, for each bi-protocol* Π*,* $f_{\texttt{bad-knowledge},\Pi}$ *and* $f_{\texttt{bad-branch},\Pi}$ *as described above are distinguishing subprotocols (see Definition 69) for* M *and* P*.*

We prove Theorem 8 in Section 6.3.6.

6.3.4. The branching monitor

In this section, we define the distinguishing subprotocol $f_{\texttt{bad-branch},\Pi}(b, tr)$, which we call the branching monitor $f_{\texttt{bad-branch},\Pi}(b, tr)$. The branching monitor $f_{\texttt{bad-branch},\Pi}(b, tr)$ reconstructs an attacker strategy by reconstructing a possible symbolic operation for every input message. In more detail, in the symbolic execution, $f_{\texttt{bad-branch},\Pi}(b, tr)$ parses the input message with all symbolic operations in the model M that the attacker could have performed as well, i.e., with all tests from the shared knowledge. This enables $f_{\texttt{bad-branch},\Pi}(b, tr)$ to simulate the symbolic execution of $\bar{b}(\Pi)$ on the constructed attacker strategy. In the computational execution of the self-monitor, the distinguishing subprotocol constructs the

symbolic operations (i.e., the symbolic inputs) by parsing the input messages with the implementations of all tests in the shared knowledge (i.e., lookups on output messages and implementations of the destructors). With this reconstructed symbolic inputs (i.e., symbolic operations, from messages that were intended for $b(\Pi)$, $f_{\texttt{bad-branch},\Pi}(b, tr)$ is able to simulate the symbolic execution of $\bar{b}(\Pi)$, called the extended symbolic execution, even in the computational execution. The distinguishing subprotocol $f_{\texttt{bad-branch},\Pi}(b, tr)$ then checks whether this extended symbolic execution of $\bar{b}(\Pi)$ takes the same branch as $b(\Pi)$ would take, for the computation node ν in question. If this is not the case, the event `bad-branch` is raised.

Symbolic self-monitoring follows by construction because the distinguishing subprotocol reconstructs a correct attacker strategy and correctly simulates a symbolic execution. Hence, $f_{\texttt{bad-branch},\Pi}(b, tr)$ found a distinguishing attacker strategy for $b(\Pi)$ and $\bar{b}(\Pi)$. We show computational self-monitoring by reducing a distinguishing event due to different branchings to computational soundness for trace properties. We then conclude that the symbolic simulation of $\bar{b}(\Pi)$ suffices to check whether $b(\Pi)$ computationally branches differently from $\bar{b}(\Pi)$.

Notation: interfaces for interactions between machines. Throughout this chapter, we use the notion of *interfaces* to connect network of machines. An interface is like a channel through which two machines communicate. In this chapter, all machines have a network interface, denoted as net. As a notational convention, we assume that for three machines A, B, C where B has a left and a right network interface (often call the execution network interface and the network interface, respectively) $\langle A \mid \langle B \mid C \rangle\rangle$ denotes that the network interface of A is connected to the left network interface (i.e., the execution network interface) of B and the right network interface (i.e., the network interface) of B is connected to the network interface of C. Moreover, we assume that the final output of a machine is sent over an output interface. Typically, $\langle A \mid \langle B \mid C \rangle\rangle$ denotes that B has a sub-output interface that is connected to the output interface of C, and for the interaction $\langle A \mid \langle B \mid C \rangle\rangle$ the output is the message that is sent over the output interface (of B). Similarly, in the case where we only have two machines A, B the output of $\langle A \mid B \rangle$ is typically the message sent over the output interface of B.

6.3.4.1. The construction of $f_{\texttt{bad-branch},\Pi}(b, tr)$

Recall that the distinguishing subprotocol for `bad-branch` tests for a bi-protocol Π whether the computation nodes in $\mathit{left}(\Pi)$ and $\mathit{right}(\Pi)$ take the same branch. The self-monitor only executes one of the two protocols of Π. For testing that both protocols always take the same branch, we perform a so-called *extended symbolic execution* of the other protocol. For this extended symbolic execution, we construct shapes of the messages that the attacker sends and use these shapes of the input messages for the extended symbolic execution.

Recall that for testing whether the two protocols of a bi-protocol Π always take the same branch at each computation node ν, we execute one of the two protocols, say $b(\Pi)$, in the self-monitor $\mathsf{Mon}(\Pi)$, reconstruct for every computation node the attacker strategy and perform with this attacker strategy a symbolic execution of the other protocol $\bar{b}(\Pi)$.

We reconstruct the attacker strategy by constructing shapes, so-called *extended symbolic operations*, of all attacker-messages of the real execution. These extended symbolic operations contain addresses of the trace of an execution of $\mathit{left}(\Pi)$ or $\mathit{right}(\Pi)$. In some

cases, it is not possible to completely reconstruct all operations that the attacker did to construct the term. As an example consider an ciphertext sent by the attacker. If the decryption key is not known to the protocol, the distinguishing subprotocol cannot reconstruct the plaintext message. Instead it creates a placeholder with the right length $\mathit{plaintextof}(t_1, \mathit{enc}(t_2, t_3, t_4))$, where t_1 corresponds to the length of the plaintext t_3. In the same manner, we create placeholders $\mathit{skofvk}(vk)$ for signature keys of signatures for which we only know the verification key, and we create placeholders $\mathit{nonceof}(m)$ for randomness that has been used, in order to create the same extended symbolic operation for equal attacker-messages m, e.g., ciphertext or signatures with the same key, the same message and the same randomness.

The core idea is to reconstruct a symbolic attacker strategy from the transcript in a manner that works, both, in the symbolic and in the computational model. Then, we use this reconstructed attacker strategy to internally run a symbolic execution of $\bar{b}(\Pi)$ and to check whether the internal symbolic execution of $\bar{b}(\Pi)$ branches differently than the real execution of $b(\Pi)$. As described above, this internal symbolic execution is the extended symbolic execution from Definition 72.

The shape of a message. For characterizing the symbolic knowledge that the adversary has about a message, we characterize the bitstring as detailed as possible in a term-like representation, which we call a *shape*. Technically, a shape is a tree, labelled with constructor and destructor names, but while constructing a shape for a given message m the algorithm Construct-shape, applies all possible tests that the adversary could apply as well to the message and inversely construct a shape, e.g., if for a message m a decryption operation dec (or dec') succeeds with the decryption key k, and to the plaintext $\mathit{unstring}_0$ succeeds, then the shape would be $\mathit{enc}(\mathit{ekofdk}(k), \mathit{string}_0(\mathit{empty}), n)$. In order to make shapes unique and to assign the same shape to the same bitstring, we refine the nonce n as a virtual constructor $\mathit{nonceof}/1$ that gets as an argument the so-called *dual symbolic operation* that lead to the term for which the randomness is retrieved: in our example $n = \mathit{nonceof}(\mathit{dec}(k), m)$. Moreover, in order to be able to obtain a attacker strategy, we have to get rid of all bistrings in a shape. For our example this means that m and k have to be replaced in the shape. For k, we recursively apply this construction of a shape to k. As an example, the decryption key could be a message that the adversary sent to the protocol, which was received at the jth node in the protocol states trace. For the message m, we know that it is the message is about to be sent at output node i Then, Construct-shape succeeds with an *isdk*-operation, and the corresponding shape would additionally point to the jth node in the protocol states trace: the shape is

$$\mathit{enc}(\mathit{ekofdk}(\mathit{dk}(\mathit{nonceof}(x_j))), \mathit{string}_0(\mathit{empty}), \mathit{nonceof}(\mathit{dec}(k), x_i))$$

Constructing the shared knowledge. The shared knowledge is a set of symbolic operations that is constructed by adding to the shared knowledge all messages that have been sent to the adversary and that have been received from the adversary. The algorithm Construct-knowledge (see Figure 6.4) slowly increases the knowledge set by iterating through the nodes of the protocol state trace, and by constructing the shapes for all previous input and output nodes and by adding these shapes to the knowledge. In this way, it happens that for the same input or output node several time a shape constructed. These different shapes for the same node are, however, consistent with each other. Since Construct-shape recursively parses a message and the knowledge

```
f_bad-branch,Π(b, tr) for computation node d
  V := CONSTRUCT-ATTACKER-STRATEGY(b, tr)
  run the extended symbolic execution of b̄(Π) with
  the attacker strategy V_In
  if d has not been reached in this execution then
    go to a distinct abort-state
  if the successor of d that is not at the x-edge has
  been reached then
    go to the state bad-branch
  return ok

CONSTRUCT-KNOWLEDGE(b, tr, K, j)
  for i = 1 to min(j, |tr|) do
    let ν_i be the ith node in b(tr)
    let tr' be the prefix-trace from ν_i to the root
    if ν_i is an input node then
      let m_i be message at the input node ν_i
      O_i := CONSTRUCT-SHAPE(m_i, x_i, K, b, tr', dec')
      /*The shared knowledge also increases if an
      attacker-message is received.*/
      K := K ∪ {O_i}
      /*saturate the knowledge after adding O_i*/
      K := FP-DESTRUCT(K, tr')
    else if ν_i is an output node then
      let m_i be message at the output node ν_i
      O_i := CONSTRUCT-SHAPE(m_i, x_i, K, b, tr', dec')
      K := K ∪ {O_i}
      /*saturate the knowledge after adding O_i*/
      K := FP-DESTRUCT(K, tr')
  return K

CONSTRUCT-ATTACKER-STRATEGY(b, tr)
  K := ∅
  for i = 1 to |tr| do
    K := CONSTRUCT-KNOWLEDGE(b, tr, K, i)
  V := ε /*initialization with the empty list*/
  for i = 1 to |tr| do
    let ν_i be the ith node in b(tr)
    let tr' be the prefix-trace from ν_i to the root
    if ν_i is an input node then
      let m_i be message at the input node ν_i
      O_i := CONSTRUCT-SHAPE(m_i, x_i, K, b, tr', dec')
      V := V :: (in, (∗, O_i))
    else if ν_i is an output node then
      let m_i be message at the output node ν_i
      O_i := CONSTRUCT-SHAPE(m_i, x_i, K, b, tr', dec')
      V := V :: (out, (∗, O_i))
    else if ν_i is a control node then
      let l' be the bitstring in the annotation of the
      edge between ν_i and the successor of ν_i that
      has been reached in tr (i.e., the in-metadata
      the attacker has sent)
      V := V :: (control, (∗, l'))
  return V
```

Figure 6.3.: The main loop of the branching monitor.

monotonically increases, each new shape for the same node only differs in that it is a refinement of the old shape, specifically *plaintextof* sub-shapes are potentially replaced by shapes that contain more information, e.g., $pair(string_1(empty), string_0(empty))$.

Extended shared knowledge. The algorithm EXTENDED-SHARED-KNOWLEDGE (see Figure 6.5) is defined that extends the shared knowledge with the plaintexts that the attacker knows because it can decrypt protocol-ciphertexts that use an attacker key. From a ciphertext $enc(ek, m, r')$ with attacker-generated ek, the attacker learns m and FP-DESTRUCT(K, tr') for the prefix of the trace tr' from the input node's to the root.

Constructing an attacker strategy. Given a shared knowledge K, constructing an attacker strategy is canonical. Using K, we again iterate over all nodes in the trace and apply CONSTRUCT-SHAPE for each input or output node and add the result together with a `in`- or `out`-tag to the attacker strategy. Upon encountering a control node the in-metadata is stored in the attacker strategy, together with a `control`-tag.

The monitor as a parametric CoSP protocol. We stress that the branching monitor can be constructed as a parametric CoSP protocol. The expressions $eval_O(tr)$ can be evaluated with sequences of the computation nodes that refer to the respective nodes. The references to previous nodes can also be checked using an *equals*-destructor node. Since CoSP protocols are infinite, loops only need an if-then-else command, and an if-then-else command can encoded into the deterministic polynomial-time algorithm be implemented in the that computes the next identifier (see Definition 26).

Leveraging the sub-algorithms for constructing the branching monitor. Given

FP-DESTRUCT(K, tr)
  repeat
    $K' := K$
    for all $O \in K$ **do**
      $K := K \cup$ DESTRUCT$(\mathbf{D}', O, tr)$
    for all pairs $(O, O') \in K^2$ **do**
      $K := K \cup$ DESTRUCT-BINARY$(\mathbf{D}', O, O', tr)$
    $K :=$ EXTENDED-SHARED-KNOWLEDGE(K, b, tr')
  until K reached a fixpoint, i.e.,
      $\forall O \in K. \exists O' \in K'. eval_O(tr) = eval'_O(tr)$
  return K

DESTRUCT(D, O, tr)
  $K'' := \emptyset$
  for all unary destructors d in D **do**
    $K'' := K'' \cup \{d(O)\}$
  return K''

DESTRUCT-BINARY(D, O, O', tr)
  $K'' := \emptyset$
  for all binary destructors d in D **do**
    $K'' := K'' \cup \{d(O, O')\}$
  return K''

Figure 6.4.: The algorithms FP-DESTRUCT, DESTRUCT, DESTRUCT-BINARY, $\mathbf{D}'$ is the extended set of destructors (see Section 6.3.4.2).

EXTENDED-SHARED-KNOWLEDGE(K, b, tr')
  for all $O \in K$ such that $O(b(tr'))$ is a ciphertext **do**
    if $\exists$ no $O' \in K$ such that $O'(b(tr'))$ and $ekof(O(b(tr'))) = ekofdk(dk)$ **then**
      if $ekof(O(b(tr')))$ is not the result of any ek-computation node in tr' **then**
        if $\exists$ an enc-computation node in tr' with the second argument from the jth node **then**
          $K := K \cup \{x_j\}$
  return K

Figure 6.5.: Extended shared knowledge.

the algorithms for constructing an attacker strategy, we run a modified symbolic execution, called the *extended symbolic execution*, with these shapes for $\bar{b}(\Pi)$ and check whether the real execution and the extended symbolic execution always take the same branch. The extended symbolic execution is defined Section 6.3.4.2 and Section 6.3.4.3.

6.3.4.2. Extended Symbolic Model

In order to be able to define the *extended symbolic execution*, we first define the extended symbolic model that contains extended terms with the extended constructors $skofvk/1, plaintextof/2, nonceof/1$ and its natural extension to terms and destructors.

The extended symbolic model M' of the symbolic model $\mathsf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D})$ is the natural extension of M with the constructors $skofvk/1, plaintextof/2, nonceof/1$. The set of *extended constructors* is defined as $\mathbf{C}' := \mathbf{C} \cup \{skofvk/1, plaintextof/2, nonceof/1\}$. The set of extended nonces are defined as $\mathbf{N}' := \mathbf{N} \cup \{nonceof(t) \mid t \in \mathbf{T}\}$.

Extended Message Type $\mathbf{T}'$. The set of *extended terms* $\mathbf{T}'$ is defined as the set of

```
CONSTRUCT-SHAPE(m, O_q, K, b, tr, dec')
  O_r := nonceof(O_q)
  switch m with
    case "∃O ∈ K. m = eval_O(b(tr))"
      return O
    case "isenc(m) ≠ ⊥"
      O_ek := CONSTRUCT-SHAPE(
                "ekof(m), ekof"(O_q), K, b, tr)
      if ∃dk.dk = ekof(m) ∧ ∃O_dk ∈ K. dk =
      eval_{O_dk}(b(tr)) then
        /*Decrypt with the known dk.*/
        if dec(dk, m) ≠ ⊥ then
          O_m' := CONSTRUCT-SHAPE(dec(dk, m),
                    "dec"(O, O_q), K, b, tr)
        else
          let l = len(m)
          return garbageEnc(O_ek, O_r, l)
      else if dec'(m, b(tr)) ≠ ⊥ then
        /*The ciphertext is protocol generated with an
        non-protocol key.*/
        O_m' := CONSTRUCT-SHAPE(dec'(m, b(tr)),
                  "dec'"(O_q), K, b, tr)
      else
        /*The plaintext is hidden.*/
        l := lenofdec(O_q)
        O_m' := plaintextof(l, O_q)
      return enc(O_ek, O_m', O_r)
    case "issig(m) ≠ ⊥"
      O_vk = CONSTRUCT-SHAPE(
               vkof(m), "vkof(O_q)", K, b, tr)
      if verify(eval_{O_vk}(b(tr)), m) = m' ≠ ⊥ then
        O_m' := CONSTRUCT-SHAPE(
                  m', "verify"(O_vk, O_q), K, b, tr)
        if ∃O.vkof(eval_O(b(tr))) = eval_{O_vk}(b(tr))
        then
          /*We know the signing key.*/
          O_sk = O
        else
          /*We know the matching vk.*/
          O_sk = skofvk(O')
        return sig(O_sk, O_m', O_r)
      else
        let l = len(m)
        return garbageSig(O_vk, O_r, l)
    case "issk(m) ≠ ⊥"
      if ∃O ∈ K. v = eval_O(b(tr)) ∧ v = vkof(m)
      then
        /*We know the matching vk.*/
        O_sk := skofvk(O)
      else
        /*The key is fresh in the shared knowledge.*/
        O_sk := sk(O_r)
      return O_sk
    case "isvk(m) ≠ ⊥"
      if ∃O ∈ K. eval_O(b(tr)) = s ∧ vkofsk(s) = m
      then
        /*We know the matching sk.*/
        O_vk := vkofsk(O)
      else
        /*The key is fresh in the shared knowledge.*/
        O_vk := vk(O_r)
      return O_vk
    case "isdk(m) ≠ ⊥"
      if ∃O ∈ K. ek = eval_O(b(tr)) ∧ ek =
      ekofdk(m) then
        /*We know the matching ek.*/
        O_dk := dkofek(O)
      else
        /*The key is fresh in the shared knowledge.*/
        O_dk := dk(O_r)
      return O_dk
    case "isek(m) ≠ ⊥"
      if ∃O ∈ K. k = eval_O(b(tr)) ∧ ekofdk(k) = m
      then
        /*We know the matching dk.*/
        O_vk := ekofdk(O)
      else
        /*The key is fresh in the shared knowledge.*/
        O_ek := ek(O_r)
      return O_ek
    case "unstring_q(m) ≠ ⊥ for q ∈ {0,1}"
      O_m' := CONSTRUCT-SHAPE(unstring_q(m),
                "unstring_q"(O_q), K, b, tr)
      return string_q(O_m')
    case "fst(m) ≠ ⊥"
      O_lt := CONSTRUCT-SHAPE(
                fst(m), "fst"(O_q), K, b, tr)
      O_rg := CONSTRUCT-SHAPE(
                snd(m), "snd"(O_q), K, b, tr)
      return pair(O_lt, O_rg)
    case "unS(m) ≠ ⊥"
      O_m' := CONSTRUCT-SHAPE(
                unS(m), "unS"(O_q), K, b, tr)
      return S(O_m')
    case "l = len(m) ≠ ⊥"
      return garb(O_r, l)
    case "default"
      return garbageInvalidLength(O_r)
lenofdec(O_q) := (len(O_q) − a_1 · len(ekof(O_q)) − a_0)/a_2
for len(enc(k, m, r)) = a_1 · len(k) + a_2 · len(m) + a_0
```

Figure 6.6.: The algorithm CONSTRUCT-SHAPE.

terms M according to the following grammar:

$$\begin{aligned} M ::= {} & enc(ek(N), M, N) \mid ek(N) \mid dk(N) \mid \\ & sig(sk(N), M, N) \mid vk(N) \mid sk(N) \mid pair(M, M) \mid P \mid N \mid L \mid \\ & garb(N, L) \mid garbageInvalidLength(N) \\ & garbageEnc(M, N, L) \mid garbageSig(M, N, L) \\ & skofvk(M) \mid plaintextof(M, M) \mid nonceof(M) \\ P ::= {} & emp() \mid string_0(P) \mid string_1(P) \qquad L ::= O() \mid S(L) \end{aligned}$$

$dec'(c, tr)$
Let tr be the trace produced by the protocol so far.
if $\exists$ an enc-node ν_i in tr with the result c', with $equals(c, c') \neq \bot$, with the messages k, m, r as arguments, and $\exists$ no ek-node in tr with result k' such that $equals(k, k') \neq \bot$ **then**
/*the ciphertext is protocol generated with an adversarially generated key*/
return m
else if c is not the result of any enc-node in tr $\wedge$ $\exists$ a node ν_i with the result $dk = x_i(tr)$ **then**
/*the attacker generated the ciphertext with a protocol encryption key*/
$m := dec(dk, c)$
return m
else
return $\bot$

Figure 6.7.: The shared knowledge algorithm dec' for decryption

The nonterminals P, N, and L represent payloads, nonces, and length specifications, respectively. Note that the garbage terms carry an explicit length specification to enable the attacker to send invalid terms of a certain length.

Additional Destructor Rules. The set $\mathbf{D}'$ of *extended destructors* is the set of partial functions that is defined by the destructor rules for $\mathbf{D}$, with the unary virtual destructor dec' (see Figure 6.7 and below) with the following additional rules for the extended constructors:

$$
\begin{aligned}
verify(vk(t_1), sig(skofvk(t_1), t_2, t_3)) &= t_2 \\
issk(skofvk(t_1)) &= skofvk(t_1) \\
issig(sig(skofvk(t_1), t_2, t_3)) &= sig(skofvk(t_1), t_2, t_3) \\
vkof(sig(skofvk(t_1), t_2, t_3)) &= t_1 \\
vkofsk(skofvk(t_1)) &= t_1 \\
len(enc(t_1, plaintextof(t_2, t_3), t_4)) &= len(t_3)
\end{aligned}
$$

In particular, decryption is undefined on extended terms that use $plaintextof(t, t')$ as a plaintext:

$$dec(dk(t_1), enc(ek(t_1), plaintextof(t_2, t_3), t_4)) = \bot$$

We define a unary virtual destructor dec' that looks up and outputs the plaintext as depicted in Figure 6.7.[7]

6.3.4.3. Extended Symbolic Execution

The only difference between the *extended symbolic execution* and the symbolic execution is that the extended symbolic execution operates on M' and, moreover, expects so-called *extended symbolic operations*, which operate on traces instead of views. Moreover, the extended symbolic model does not work on terms but on symbolic operations. In particular, input messages not immediately parsed as terms but rather only assigned as much structure as has already been tested by the protocol via successful destructor application tests, which are executed on the real bitstrings. Another important difference is that the extended symbolic execution is run inside the CoSP protocol $\mathsf{Mon}(\Pi)$. As a consequence,s the

[7] Formally, dec' is not a destructor, because its result depends on the trace that the protocol has been produced up to the invocation of dec'. However, we treat it like a destructor in the following to simplify presentation.

extended symbolic operation, if $\mathsf{Mon}(\Pi)$ is run in the computational execution, uses the computational implementations of the constructors and destructors, and, when $\mathsf{Mon}(\Pi)$ is considered in the symbolic model, it uses the symbolic constructors and destructors. Since the alarms in $\mathsf{Mon}(\Pi)$ are trace properties, the assumed computational soundness result implies that $\mathsf{Mon}(\Pi)$, and thereby the extended symbolic execution, in the symbolic model coincides with $\mathsf{Mon}(\Pi)$, and thereby the extended symbolic execution, in the computational execution.

The extended symbolic operations are defined as follows.

Definition 71 (Extended Symbolic Operation)**.** *An* extended symbolic operation O/n (of arity n) on M for a protocol Π *is a symbolic operation on* M' *(as defined in Section 6.3.4.2) except that the projections are evaluated on traces instead of views via the function* $eval_O$: *Traces*$(\Pi) \to \mathbf{T}'$ *(where Traces denotes the set of all finite traces in* Π*):*

$$eval_{x_i}(tr) = \begin{cases} m & \text{, if there is an } m \text{ associated to the } i\text{th node in } tr \\ \bot & \text{, otherwise} \end{cases}$$

$$eval_{f(O_1,\ldots,O_n)}(tr) = f(eval_{O_1}(tr), \ldots, eval_{O_n}(tr)) \text{ for } f \in \mathbf{D}' \text{ with arity } n$$

A symbolic execution of a protocol is basically a valid path through the protocol tree. It induces a *view*, which contains the communication with the attacker.

Definition 72 (Extended Symbolic Execution)**.** *Let* $\mathsf{M}' = (\mathbf{C}', \mathbf{N}', \mathbf{T}', \mathbf{D}')$ *be the extended symbolic model from Section 6.3.4.2. Let a CoSP protocol* I *be given. An* extended symbolic view *of the protocol* I *is a (finite) list* L *of triples* (V_i, ν_i, f_i) *with the following conditions: For the first triple, we have* $V_1 = \varepsilon$*,* ν_1 *is the root of* I*, and* f_1 *is an empty partial function, mapping node identifiers to terms. For every two consecutive tuples* (V, ν, f) *and* (V', ν', f') *in the list, let* $\underline{\tilde{\nu}}$ *be the nodes referenced by* ν *and define* $\underline{\tilde{t}}$ *through* $\tilde{t}_j := f(\tilde{\nu}_j)$*. We conduct a case distinction on* ν*.*

- ν ***is a computation node with constructor, destructor or nonce*** F***.*** *Let* $V' = V$*. If* $m := eval_F(\underline{\tilde{t}}) \neq \bot$*,* ν' *is the* `yes`*-successor of* ν *in* I*, and* $f' = f(\nu := m)$*. If* $m = \bot$*, then* ν' *is the* `no`*-successor of* ν*, and* $f' = f$*.*
- ν ***is an input node.*** *If there exists a term* $t \in \mathbf{T}$ *and an extended symbolic operation* O *on* M *with* $eval_O(L') = t$*, let* ν' *be the successor of* ν *in* I*,* $V' = V :: (\mathtt{in}, (t, O))$*, and* $f' = f(\nu := t)$*, where* L' *is the prefix of the extended symbolic view* L *up to the tuple* (V, ν, f)*.*
- ν ***is an output node.*** *Let* $V' = V :: (\mathtt{out}, \tilde{t}_1)$*,* ν' *is the successor of* ν *in* I*, and* $f' = f$*.*
- ν ***is a control node with out-metadata*** l***.*** *Let* ν' *be the successor of* ν *with the in-metadata* l' *(or the lexicographically smallest edge if there is no edge with label* l'*),* $f' = f$*, and* $V' = V :: (\mathtt{control}, (l, l'))$*.*

Here, V_{Out} *denotes the list of extended terms in* V *that have been sent at output nodes, i.e., the extended terms* t *contained in entries of the form* $(\mathtt{out}, t)$ *in* V*. Analogously,* $V_{Out\text{-}Meta}$ *denotes the list of out-metadata in* V *that has been sent at control nodes.*

The set of all symbolic views of I *is denoted by* $SViews(I)$*. Furthermore,* V_{In} *denotes the partial list of* V *that contains only entries of the form* $(\mathtt{in}, (*, O))$ *or* $(\mathtt{control}, (*, l'))$ *for some symbolic operation* O *and some in-metadata* l'*, where the input term and the*

out-metadata have been masked with the symbol $*$. The list V_{In} is called attacker strategy*. We write $[V_{In}]_{SViews(I)}$ to denote the class of all views $U \in SViews(I)$ with $U_{In} = V_{In}$.*

6.3.4.4. $f_{\texttt{bad-branch},\Pi}$ is a distinguishing subprotocol

Finally, we are able to prove that the branching monitor satisfies symbolic and computational self-monitoring.

Symbolic self-monitoring. Symbolic self-monitoring, at its core, follows from the insight that all operations that the branching monitor performs are in the shared knowledge, i.e., are in the symbolic knowledge (of the symbolic adversary). As a consequence, we conclude that a run of the extended symbolic execution induces a full symbolic trace.

Before, we can prove symbolic self-monitoring, we define, as a technical vehicle, a derived view. For an output node ν of a CoSP protocol, we call the output message m *associated with* ν, i.e., m is the message to which the output node references. For an input node or a computation node ν, we call the message m produced at ν *associated with* ν.

Definition 73 (Derived view)**.** *Let Π be a bi-protocol and Π' be the corresponding self-monitor. Then, the* derived *left* view *$left(tr)$ of Π' for a trace tr is iteratively constructed as follows:*

$left(tr) := \varepsilon$ (i.e., the empty list)
for** $i = 1$ to length of tr **do
  if** node i is an input node that is associated to a message m **then
    $left(tr) := left(tr) :: (\texttt{in}, m)$
  if** node i is a output node that is labeled as left and is associated to a message m **then
    $left(tr) := left(tr) :: (\texttt{out}, m)$
  if** node i is an control node with input metadata label l and the edge to node $i+1$ in tr is labeled with the metadata label l' **then
    $left(tr) := left(tr) :: (\texttt{control}, (l, l'))$

The derived *right* view *$right(tr)$ is defined analogously with right instead of left for output nodes. For symbolic traces the messages in the view are terms and for computational traces the messages in the view are bitstrings.*

Lemma 29 ($f_{\texttt{bad-branch},\Pi}$ Only Uses Symbolic Operations)**.** *For every extended symbolic operation O that $f_{\texttt{bad-branch},\Pi}$ uses in the extended symbolic execution, there is a symbolic operation O' such that for all traces tr we have $eval_O(tr) = eval_{O'}(V(tr))$, where V is the derived view of tr (similar to Definition 73).*

Proof. By construction of CONSTRUCT-SHAPE, we know that there is only one case in which projections are generated that do not point to output nodes: for the plaintexts m of ciphertexts c for which the attacker knows the decryption key dk. For these cases, there is a symbolic operation $dec(dk, c)$ that outputs the plaintext m.

As a next step, we show that the extended constructors *plaintextof*$/2$, *skofvk*$/1$, *nonceof*$/1$ are only used whenever the attacker could replace them by derivable messages. For *plaintextof*$/2$, we know by construction of CONSTRUCT-SHAPE that *plaintextof*$/2$ is only used on ciphertexts for which either the attacker or the protocol cannot know the plaintext. Hence, the attacker could already generate the ciphertext, and thus there is a symbolic operation that contains an actual message instead of *plaintextof*$/2$. An analogous argumentation

shows that whenever we use *skofvk*/1 or *nonceof*/1, the message was attacker-generated and the attacker new the secret key or the randomness, respectively. □

Lemma 30 ($f_{\texttt{bad-branch},\Pi}$ satisfies symbolic self-monitoring)**.** *Let* Π *be a bi-protocol from the protocol class* P*, and* M *be the symbolic model from Section 6.3.1. The parametric CoSP protocol* $f_{\texttt{bad-branch},\Pi}$ *(see Section 6.3.4.1) satisfies symbolic self-monitoring (see Definition 69).*

Proof. By Lemma 29, we know that for every extended symbolic operation used by $f_{\texttt{bad-branch},\Pi}$ there is a symbolic operation. As a consequence, we know that there is a full symbolic trace for each such run of the extended symbolic execution. Moreover, this symbolic trace trace equals the resulting trace from the run of the extended symbolic execution.

Hence, whenever an alarm is raised in $\mathsf{Mon}(\Pi)$, there is an attacker strategy such that (for $b \in \{\mathit{left}, \mathit{right}\}$) the symbolic execution of $b(\Pi)$ (which corresponds to the real execution in $\mathsf{Mon}(\Pi)$) and the symbolic execution of $\bar{b}(\Pi)$ branch at some point differently. Thus, there is a symbolic view for which symbolic indistinguishability is violated. □

Internalizing the filter into the protocol. As a technical vehicle for the proof, we define a protocol $\mathsf{Mon}(\Pi) - \mathrm{F}_b$ that combines the monitor with the filter F_b, which chooses the b-variant. Then, we show that this protocol $\mathsf{Mon}(\Pi) - \mathrm{F}_b$ is indistinguishable from $b(\Pi)$.

Definition 74 (The Filter F_b)**.** *The filter machine* F_b *is constructed by initially choosing the* b *branch in* $\mathsf{Mon}(\Pi)$ *and by only forwarding the messages from the output nodes that are labelled with* b*, thereby ignoring the messages from the output nodes labelled with* $\bar{b}$*.*

Lemma 31 (Indistinguishability of the Self-monitor)**.** *Let* Π *be an efficient bi-protocol and* F_b *be the filter machine from Definition 74. Then, for every* $b \in \{\mathit{left}, \mathit{right}\}$ *and every machine* $\mathcal{A}$*, we have*

$$\begin{aligned}&\left\langle \mathit{Exec}_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi)} \mid \mathrm{F}_{\mathit{left}} \right\rangle \approx_{\mathit{tic}} \left\langle \mathit{Exec}_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi)} \mid \mathrm{F}_{\mathit{right}} \right\rangle \\ &\iff \mathit{Exec}_{\mathsf{M},\mathrm{Impl},\mathit{left}(\Pi)} \approx_{\mathit{tic}} \mathit{Exec}_{\mathsf{M},\mathrm{Impl},\mathit{right}(\Pi)}\end{aligned}$$

Proof. Observe that in both relations, the messages sent to the adversary are the same for $b = \mathit{left}$ and for $b = \mathit{right}$. Hence, we only have to consider the running time to prove the two implications. (Indeed, if one of the tic-indistinguishabilities holds, because one of the machines needs a super-polynomial running time, then this tic-indistinguishability states nothing about the messages sent to the adversary after a super-polynomial amount of time. Thus, we could not use this tic-indistinguishability to prove the other one, because the machines machines in the desired tic-indistinguishability could potentially need only polynomial running time, which means that we have to prove a statement about all the messages sent to the adversary.) It suffices to show that the running time of the machines in the one tic-indistinguishability is related to the running time of the machines in the other tic-indistinguishability. Precisely, we show: For each adversary $\mathcal{A}$, for each output a, for all auxiliary information z, and for all polynomials p there is a polynomial q and a

negligible function μ such that

$$\begin{aligned}&\Pr[\langle\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi)}(k) \mid \mathrm{F}_b\rangle \mid \mathcal{A}(k,z)\rangle \Downarrow_{q(k)} a] \\ &\geq \Pr[\langle Exec_{\mathsf{M},\mathrm{Impl},b(\Pi)}(k) \mid \mathcal{A}(k,z)\rangle \Downarrow_{p(k)} a] - \mu(k)\end{aligned} \tag{6.1}$$

and vice versa, i.e., for all q there is a p such that the formula holds. In other words, the probability that the amount of joint computation steps is polynomially bounded in $\langle Exec_{\mathsf{M},\mathrm{Impl},b(\Pi)} \mid \mathcal{A}\rangle$ equals the probability that the joint computation steps are polynomially bounded in $\langle\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi)} \mid \mathrm{F}_b\rangle \mid \mathcal{A}\rangle$.

The direction "$\Longrightarrow$" follows from the fact that $Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi)-\mathrm{F}_b}$ computes internally, due to the extended symbolic execution, both variants and hence has always more computation steps than $Exec_{\mathsf{M},\mathrm{Impl},b(\Pi)}$. As a consequence, for all polynomials q, there is a $p \leq q$ such that equation 6.1 holds.

For the direction "$\Longleftarrow$", we have to show that the computation steps of the interaction $\langle\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi)}(k) \mid \mathrm{F}_b\rangle \mid \mathcal{A}(k,z)\rangle$ are polynomially bounded as well. This statement follows from the following three other statements: (*i*) in $f_{\texttt{bad-knowledge}}$ the running time of the fixpoint computation (FP-Destruct) is bounded polynomially in the sum of the length of the prefix trace tr_j and the security parameter; (*ii*) $\mathsf{Mon}(\Pi)$ is efficient in the sense of Definition 4; (*iii*) the extended symbolic execution of $\bar{b}(\Pi)$ takes super-polynomially many computation steps, St against $\bar{b}(\Pi)$ only needs poly many steps.

(*i*) follows from the construction of FP-Destruct, since we only destruct messages. (*ii*) also follows from the construction $\mathsf{Mon}(\Pi)$ and from $f_{\texttt{bad-knowledge}}$. It, hence, remains to show (*iii*)

Since $\mathsf{Mon}(\Pi)$ internally also computes $\bar{b}(\Pi)$, we have to exclude that the extended symbolic execution of $\bar{b}(\Pi)$ takes super-polynomially many computation steps. Observe that uniformity-enforcing, symbolic indistinguishability and computational soundness for trace properties imply that the distribution of traces induced by $\langle Exec_{\mathsf{M},\mathrm{Impl},\mathit{left}(\Pi_i)} \mid \mathcal{A}\rangle$ is computationally indistinguishable from the distribution of node traces induced by $\langle Exec_{\mathsf{M},\mathrm{Impl},\mathit{right}(\Pi_i)} \mid \mathcal{A}\rangle$; otherwise, we can construct a single protocol that executes both $\mathit{left}(\Pi_i)$ and $\mathit{right}(\Pi_i)$, checks in the second run whether all inputs satisfy the same destructor tests as the inputs of the first run, and raises an alarm if a branching was different. This protocol breaks the computational soundness for trace properties (which holds by Theorem 9).

We know that the length of the trace is polynomially bounded. Next, we have to show that also the call of the algorithms does not cause a super-polynomial computation. By Definition 6 all implementations are polynomial-time computable, and by the definition of a length destructor and Implementation Conditions 5 and 33 the length of all bitstrings is linear in the length of the input. The length of the input, in turn, is polynomially bounded in the sum of the length of tr_j and the security parameter. Hence, every call to an implementation algorithm only causes a polynomially bounded number of computation steps, in the sum of the security parameter and the length of the trace so far tr_j (where tr_j is the path from the computation node ν_j to the root). □

Definition 75 ($\mathsf{Mon}(\Pi) - \mathrm{F}_b$)**.** *For a protocol* Π*, we define the protocol* $\mathsf{Mon}(\Pi) - \mathrm{F}_b$ *as the unrolled variant of the protocol in which the initial node in* $\mathsf{Mon}(\Pi)$ *is removed and*

only the b-branch is taken and each the $\bar{b}$-labelled output node that points to a node ν is replaced by a so-called virtual output *node: a computation node of the pair constructor that points twice to ν. In $\mathsf{Mon}(\Pi) - \mathrm{F}_b$ the monitor $\mathsf{Mon}(\Pi)$ treats virtual output nodes as it treated output nodes.*

Corollary 3 ($\mathsf{Mon}(\Pi) - \mathrm{F}_b$ equivalence)**.** *For all CoSP protocols Π and all ppt adversaries $\mathcal{A}$, we have*

$$\begin{aligned} &Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi)-\mathrm{F}_{left}} \approx_{tic} Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi)-\mathrm{F}_{right}} \\ \iff &Exec_{\mathsf{M},\mathrm{Impl},left(\Pi)} \approx_{tic} Exec_{\mathsf{M},\mathrm{Impl},right(\Pi)} \end{aligned}$$

Proof. This immediately follows from the definition of $\mathsf{Mon}(\Pi) - \mathrm{F}_b$ and from Lemma 31. □

Computational self-monitoring. For computational self-monitoring, we reduce the distinguishability of different branchings to trace properties and then apply the computational soundness result for trace properties.

Lemma 32 ($f_{\texttt{bad-branch},\Pi}$ satisfies computational self-monitoring)**.** *The parametric CoSP protocol $f_{\texttt{bad-branch},\Pi}$ (see Section 6.3.4.1) satisfies computational self-monitoring (see Definition 69).*

Proof. We construct an *extended branching monitor* $\mathsf{Mon}'(\Pi)$. Where the branching monitor runs the extended symbolic execution for $\bar{b}(\Pi)$, the extended branching monitor Mon' completely runs $\bar{b}(\Pi)$ once again and additionally implements input guards in the second run (see below) and checks whether the last computation node, i.e., the node before ν, branches the same in both runs if the input guards succeed executes Π_i twice and compares which branch was taken at the node ν.

1: in the first run, apply CONSTRUCT-SHAPE at each input node and store each resulting shape
2: in the second run, if a branching is different from the first run, set a flag fail to *true*
3: in the second run, we additionally apply CONSTRUCT-SHAPE at each input node and check whether the resulting shape coincides with the shape from same input node in the first run. If the check fails, set the flag fail to *true*, as well.
4: **if** the flag fail equals *true* **then**
5: $\quad\Pi^\nu$ halts in a state `stop`

We consider the following trace property p_ν: if `stop` is not reached (in Π^ν), then ν is reached in both runs and use the same branch in both runs.

We define a filter F_b by a machine (which typically interacts with $\mathsf{Mon}(\Pi)$) that upon the initial query of $\mathsf{Mon}(\Pi)$ chooses the branch b and then forwards all messages to the adversary or the simulator, i.e., over the network interface. The extended filter EF_b extends the filter F_b by completely hiding the second run from $\mathcal{A}$ by replaying the same messages once again, also from the distinguisher, i.e., does not output the results of the second run.

Claim 1. *Let Π be an arbitrary bi-protocol in P. If and only if Π is indistinguishable, the extended branching monitor is indistinguishable, i.e.,*

$$\begin{aligned} &Exec_{\mathsf{M},\mathrm{Impl},b(\Pi)} \approx_{tic} Exec_{\mathsf{M},\mathrm{Impl},\bar{b}(\Pi)} \\ \iff &\left\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}'(\Pi)} \mid \mathrm{EF}_b \right\rangle \approx_{tic} \left\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}'(\Pi)} \mid \mathrm{EF}_{\bar{b}} \right\rangle \end{aligned}$$

Proof of Claim 1. This claim directly follows from Lemma 31 since no information is sent to the adversary $\mathcal{A}$ after the first run. ◇

As a next step, we consider a modification EF'_b of the extended filter EF_b that sends `firstRunDone` after the first run (with b) over the network interface net. Then, it waits for the string `beginSecondRun` and performs real a second run with $\bar{b}$, in contrast to EF_b not the attacker messages from the first run. Moreover, let the rewinding filter $\mathrm{RF}(\mathcal{A})$ be a machine that internally runs the adversary machine $\mathcal{A}$ and resets $\mathcal{A}$ after receiving the string `firstRunDone`. Then, it sends `beginSecondRun` over the left network interface net. We plug these machines together as follows: $\left\langle \left\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}'(\Pi)} \mid \mathrm{EF}'_b \right\rangle \mid \mathrm{RF}(\mathcal{A}) \right\rangle$, i.e., the left network interface of the rewinding filter is connected to the right network interface of EF'_b. Moreover, the output interface of $\mathcal{A}$ is connected to the sub-output interface of RF, and the output interface of RF is connected to the output interface of EF'_b. EF'_b outputs the guess of the adversary from the first run.

Claim 2. *The extended branching monitor* $\mathsf{Mon}'(\Pi)$ *is indistinguishable against the extended filter if and only if* $\mathsf{Mon}'(\Pi)$ *is indistinguishable against the rewinding filter, i.e., for all probabilistic polynomial-time machines* $\mathcal{A}$*, we have*

$$\begin{aligned}&\left\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}'(\Pi)} \mid \mathrm{EF}_b \right\rangle \approx^{\mathcal{A}}_{tic} \left\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}'(\Pi)} \mid \mathrm{EF}_{\bar{b}} \right\rangle \\ \iff &\left\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}'(\Pi)} \mid \mathrm{EF}'_b \right\rangle \approx^{\mathrm{RF}(\mathcal{A})}_{tic} \left\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}'(\Pi)} \mid \mathrm{EF}'_{\bar{b}} \right\rangle\end{aligned}$$

Proof of Claim 2. The statement directly follows from the construction of EF'_b against $\mathrm{RF}(\mathcal{A})$: the guess of EF'_b has exactly the same distribution as the guess of EF_b against $\mathcal{A}$. ◇

Claim 3. *If* Π_i *is distinguishable,* Π_{i-1} *is indistinguishable, the last node* ν *in* Π_i *is a control node, then both extended branching monitors computationally raise an alarm, i.e.,*

$$\begin{aligned}&Exec_{\mathsf{M},\mathrm{Impl},b(\Pi_i)} \not\approx_{tic} Exec_{\mathsf{M},\mathrm{Impl},\bar{b}(\Pi_i)} \text{ and} \\ &Exec_{\mathsf{M},\mathrm{Impl},b(\Pi_{i-1})} \approx_{tic} Exec_{\mathsf{M},\mathrm{Impl},\bar{b}(\Pi_{i-1})} \text{ and } \nu \text{ is a control node} \\ &\implies \Pr\left[\left\langle \left\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}'(\Pi_i)} \mid \mathrm{EF}'_b \right\rangle \mid \mathrm{RF}(\mathcal{A}) \right\rangle : p_\nu \text{ occurs}\right] \text{ is non-negligible}\end{aligned}$$

Moreover, we have

$$\begin{aligned}&\Pr\left[\left\langle \left\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}'(\Pi_i)} \mid \mathrm{EF}'_b \right\rangle \mid \mathrm{RF}(\mathcal{A}) \right\rangle : p_\nu \text{ occurs}\right] \\ &= \Pr\left[\left\langle \left\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}'(\Pi_i)} \mid \mathrm{EF}'_{\bar{b}} \right\rangle \mid \mathrm{RF}(\mathcal{A}) \right\rangle : p_\nu \text{ occurs}\right]\end{aligned}$$

Proof of Claim 3. Since Π_{i-1} is indistinguishable and by Claim 2, we know that the following holds

$$\left\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}'(\Pi_{i-1})} \mid \mathrm{EF}'_b \right\rangle \approx^{\mathrm{RF}(\mathcal{A})}_{tic} \left\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}'(\Pi_{i-1})} \mid \mathrm{EF}'_{\bar{b}} \right\rangle$$

By Claim 1 and Claim 2, we know that the extended branching monitor for Π_i is distinguishable against the rewinding filter, i.e.,

$$\left\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}'(\Pi_i)} \mid \mathrm{EF}'_b \right\rangle \not\approx_{tic}^{\mathrm{RF}(\mathcal{A})} \left\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}'(\Pi_i)} \mid \mathrm{EF}'_{\bar{b}} \right\rangle$$

By the definition of computational challenger (see Definition 20) and since by assumption the last node ν in Π_i is a control node we know that the only information that the distinguisher receives against Π_i that he does not receive against Π_{i-1} is the in-metadata. As a consequence, we know that the computation node ν' directly before ν branches differently in the execution of $b(\Pi_i)$ and $\bar{b}(\Pi_i)$ against $\mathcal{A}$, thus also in $\mathsf{Mon}'(\Pi_i)$ against EF'_b and $\mathrm{RF}(\mathcal{A})$. Thus, for the two runs in $\mathsf{Mon}'(\Pi_i)$ against the rewinding filter $\mathrm{RF}_b(\mathcal{A})$, the attacker produces with non-negligible probability, traces that pass all entry guards and cause the computation node ν' (just before ν) to branch differently. Since the rewinding filter resets the attacker, also the rewinding filter $\mathrm{RF}_b(\mathcal{A})$ finds such a trace with non-negligible probability, thus the statement follows:

$$\begin{aligned}
&\Pr\left[\left\langle\left\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}'(\Pi_i)} \mid \mathrm{EF}'_b\right\rangle \mid \mathrm{RF}(\mathcal{A})\right\rangle : p_\nu \text{ occurs}\right] \\
&= \Pr\left[\left\langle\left\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}'(\Pi_i)} \mid \mathrm{EF}'_{\bar{b}}\right\rangle \mid \mathrm{RF}(\mathcal{A})\right\rangle : p_\nu \text{ occurs}\right] \text{ and} \\
&\Pr\left[\left\langle\left\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}'(\Pi_i)} \mid \mathrm{EF}'_{\bar{b}}\right\rangle \mid \mathrm{RF}(\mathcal{A})\right\rangle : p_\nu \text{ occurs}\right] \text{ is non-negligible}
\end{aligned}$$

◇

Claim 4. *Let* Π *be an arbitrary bi-protocol. If the branching monitor* $\mathsf{Mon}(\Pi)$ *symbolically does not raise an alarm, then the extended branching monitor* $\mathsf{Mon}'(\Pi)$ *symbolically does not raise an alarm.*

Proof of Claim 4. We first show that in the extended symbolic execution the branching monitor $\mathsf{Mon}(\Pi)$ never produces an extended term *plaintextof*$(\_,\_)$, i.e., a term that the monitor does not know for the extended symbolic execution. By the construction of Construct-shape (see Section 6.3.4.1) we know that such an extended term is only inside a ciphertext for which the protocol does not know the key and the ciphertext is attacker-generated. However, the protocol will never be able to decrypt such an encryption, i.e., such an extended term *plaintextof*$(\_,\_)$ is never the result of an evaluation of a computation node in the extended symbolic execution.

If extended terms *plaintextof*$(\_,\_)$ are never produced by any evaluation in the extended symbolic execution, the shapes always produce the same branching in the extended symbolic execution as any term for which the input guards succeed, which concludes the statement. ◇

The contraposition of Claim 3 is the following statement:

$$\begin{aligned}
&Exec_{\mathsf{M},\mathrm{Impl},b(\Pi_{i-1})} \approx_{tic} Exec_{\mathsf{M},\mathrm{Impl},\bar{b}(\Pi_{i-1})} \text{ and } \nu \text{ is a control node} \\
&\wedge \Pr\left[\left\langle\left\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}'(\Pi_i)} \mid \mathrm{EF}'_b\right\rangle \mid \mathrm{RF}(\mathcal{A})\right\rangle : p_\nu \text{ occurs}\right] \text{ is negligible} \\
&\implies Exec_{\mathsf{M},\mathrm{Impl},b(\Pi_i)} \approx_{tic} Exec_{\mathsf{M},\mathrm{Impl},\bar{b}(\Pi_i)}
\end{aligned}$$

Furthermore, the following statement holds by assumption:

$$Exec_{\mathsf{M},\mathrm{Impl},b(\Pi_{i-1})} \approx_{tic} Exec_{\mathsf{M},\mathrm{Impl},\bar{b}(\Pi_{i-1})} \text{ and } \nu \text{ is a control node}$$

By Claim 4 and by the assumption that no branching alarm is raised (with more than negligible probability), we know that

$$\Pr\left[\left\langle\left\langle Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}'(\Pi_i)} \mid \mathrm{EF}'_b\right\rangle \mid \mathrm{RF}(\mathcal{A})\right\rangle : p_\nu \text{ occurs}\right] \text{ is negligible}$$

Hence, we conclude

$$Exec_{\mathsf{M},\mathrm{Impl},b(\Pi_i)} \approx_{tic} Exec_{\mathsf{M},\mathrm{Impl},\bar{b}(\Pi_i)}$$

□

6.3.5. The knowledge monitor

The distinguishing subprotocol $f_{\texttt{bad-knowledge},\Pi}(b, tr)$ for an output node ν starts like $f_{\texttt{bad-branch},\Pi}(b, tr)$ by reconstructing a (symbolic) attacker strategy and simulating a symbolic execution of $\bar{b}(\Pi)$. However, instead of testing the branching behavior of $\bar{b}(\Pi)$, the distinguishing subprotocol $f_{\texttt{bad-knowledge},\Pi}(b, tr)$ characterizes the message m that is output in $b(\Pi)$ at the output node ν in question, and then $f_{\texttt{bad-knowledge},\Pi}(b, tr)$ compares m to the message that would be output in $\bar{b}(\Pi)$. This characterization must honor that ciphertexts generated by the protocol are indistinguishable if the corresponding decryption key has not been revealed to the attacker so far. If a difference in the output of $b(\Pi)$ and $\bar{b}(\Pi)$ is detected, the event `bad-knowledge` is raised.

Symbolic self-monitoring for the distinguishing subprotocol $f_{\texttt{bad-knowledge},\Pi}(b, tr)$ follows by the same arguments as for $f_{\texttt{bad-branch},\Pi}(b, tr)$. We show computational self-monitoring by first applying the PROG-KDM property to prove that the computational execution of $b(\Pi)$ is indistinguishable from a *faking* setting: in the faking setting, all ciphertexts generated by the protocol do not carry any information about their plaintexts (as long as the corresponding decryption key has not been leaked). The same holds analogously for $\bar{b}(\Pi)$. We then consider all remaining real messages, i.e., all messages except ciphertexts generated by the protocol with unleaked decryption keys. We then show that in the faking setting, $f_{\texttt{bad-knowledge},\Pi}(b, tr)$ is able to characterize all information that is information theoretically contained in a message. We conclude by showing that with this characterization, the knowledge monitor raises the event `bad-knowledge` whenever the bi-protocol Π is distinguishable.

Section 6.3.5.1 presents the construction of the knowledge monitor. The proof of symbolic self-monitoring goes along the lines of the proof of self-monitoring for the branching monitor (see Section 6.3.5.2). For the proof of computational self-monitoring, we use several technical vehicles. We prove that the construction of a so-called *faking simulator* in the computational soundness proof w.r.t. trace properties from previous work [BMU12] can be reused. This faking simulator simulates a real computational execution but does not use sensitive information at all, i.e., for all ciphertext of which the decryption key has not been leaked yet the encryption operation does not use the actual plaintext. To this end, we first recall this faking simulator in Section 6.3.5.3. Then, in Section 6.3.5.5, we prove that the indistinguishability of the faking simulator and the real computational execution does carries over to the case with equivalence properties. Thereafter, in Section 6.3.5.6, we prove that for each symbolic operation there is exactly one bitstring in one run. As a final vehicle, we resolve, in Section 6.3.5.7, the natural double induction, over the length of the protocol and the structural size of the message to be sent by introducing so-called *unrolled variants*, which before sending a message first send all visible sub-messages this

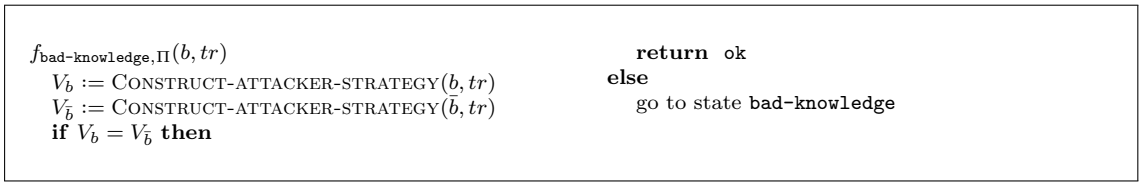

Figure 6.8.: Construction of the knowledge monitor

message to the adversary. Finally, in Section 6.3.5.8, we plug all these tools together and prove computational self-monitoring for the knowledge monitor.

6.3.5.1. Construction of the knowledge monitor

The knowledge monitor $f_{\texttt{bad-knowledge},\Pi}$ (see Figure 6.8), like the branching monitor $f_{\texttt{bad-branch},\Pi}$ (see Section 6.3.4.1), first constructs an attacker strategy, using CONSTRUCT-ATTACKER-STRATEGY (see Figure 6.4). With the this attacker strategy, the knowledge monitor $f_{\texttt{bad-knowledge},\Pi}$ obtains for the output node for $b(\Pi)$ and for $\bar{b}(\Pi)$ a shape such that all information that is visible from the shared knowledge is contained in these shape, i.e., these shapes have maximal information. The knowledge monitor then simply checks whether these shapes are equal or not. If the shapes are not equal, an alarm is raised.

6.3.5.2. Symbolic self-monitoring of the knowledge monitor

In this section, we show the symbolic self-monitoring of the knowledge monitor.

Definition 76 (Shared knowledge)**.** *A* shared test *is a symbolic operation for a trace tr (for its derived view V see Definition 73) that either*

- *does not use any nodes labelled with nonces , or*
- *with attacker nonces $\mathbf{N}_E$ that were used in protocol-constructed terms in tr.*

The shared knowledge function *is the knowledge function (see Definition 17) $K_{shV} : SO \to \{\top, \bot\}$ that is defined on all shared tests and undefined on all other symbolic operations.*

We say that a symbolic operation is in the shared knowledge *if it is a shared test.*

Lemma 33 (Symbolic self-monitoring for the knowledge monitor)**.** *Let Π be a bi-protocol. Let $i \in \mathbb{N}$. Let Π'_i be the self-monitor for Π_i. For all $i \in \mathbb{N}$ the following holds. If there is an attacker strategy such that in Π'_i the event* bad-knowledge *occurs but in Π'_{i-1} the event* bad *does not occur and Π_{i-1} is symbolically indistinguishable, then Π_i is symbolically distinguishable because of knowledge.*

Proof. We show this statement by induction over i. For $i = 1$, either the last node of Π_1 is a control node or an output node. If the last node is a control node, the statement follows. If the last node is a output node, then observe that $f_{\texttt{bad-knowledge},\Pi}$ only performs tests that are in the shared knowledge (see Lemma 29). Hence, there is an attacker strategy that distinguishes the pair of views obtained by the symbolic execution of Π_1 and Π_1 is symbolically distinguishable. Since the last node was an output node and Π_0 is

indistinguishable because it does not output anything to the attacker, Π_1 is symbolically distinguishable because of knowledge.

For $i > 1$ we know that in Π'_i for all but the last control node no alarm is raised, i.e., `bad` does not occur for any attacker strategy. Hence, in the tests of before the last control node `bad-knowledge` occurred. It remains to show that then Π_i is distinguishable because of knowledge. Observe that $f_{\texttt{bad-knowledge},\Pi}$ only performs tests that are in the shared knowledge (see Lemma 29). Hence, there is an attacker strategy that distinguishes the pair of views obtained by the symbolic execution of Π_i and Π_i is symbolically distinguishable. Since the last node was an output node and Π_{i-1} is indistinguishable by assumption, Π_1 is symbolically distinguishable because of knowledge. □

6.3.5.3. The Faking Simulator Sim_f

We show that we can reuse the simulator of the computational soundness result of Backes, Malik, and Unruh [BMU12]. In particular, we use the hybrid execution in which all messages are faked. Technically, however, the previous computational soundness result for trace properties needs indistinguishability trace properties and computational soundness for equivalence properties needs the communication is indistinguishable for the attacker.

Hybrid Execution for Trace Properties. A successful way of proving computational soundness (for trace properties) in the literature is the construction of a simulator that translates bitstrings to terms and vice versa. The simulator interacts with a modified symbolic execution, called the hybrid execution, on one side and the computational attacker on the other side. Whenever the simulator receives a term from the hybrid execution, the simulator constructs a corresponding bitstring and sends it to the attacker. Whenever the simulator receives a bitstring from the attacker, the simulator parses the bitstring and assigns a term to it, which it sends to the hybrid execution. This simulator thereby assigns a symbolic attacker strategy for a run agains the computational attacker.

This simulator runs against a modified symbolic execution, called a hybrid execution challenger, that lets the simulator decide the attacker strategy. In the work of Backes, Malik, and Unruh [BMU12], the hybrid execution challenger is a further modification in that it enables a lazy evaluation that works as follows: first, the hybrid execution challenger accepts incomplete terms with variables inside as attacker-terms for input nodes; second, whenever a term is evaluated the hybrid execution asks the simulator, so-called eval-questions, to evaluate the term, potentially using an assignment for the variables. At the end of the execution, the simulator has to send an assignment of all variables to terms that is consistent with the responses of the simulator to the evaluation queries.

Definition 77 (Hybrid Challenger *TrH-Exec* (Trace Properties))**.** *Let* Π *be a CoSP protocol. We define an interactive machine* $TrH\text{-}Exec_{\mathsf{M},\Pi}(k)$ *run on input* k*. It is called the hybrid protocol machine associated with* Π*. It internally maintains and finally outputs a (finite) lists of tuples* (S_i, ν_i, f_i)*, called the* full hybrid trace*, and runs a symbolic simulation of* Π *as follows:*

Initially $S_1 := S := \varepsilon$*,* $\nu_1 := \nu$ *is the root of* Π*, and* $f_1 := f$ *is a totally undefined partial function mapping node identifiers to* $\mathbf{T}$*. For* $i = 2, 3, \ldots$ *do the following (recall that* net *is the network interface):*

1. *Let* $\underline{\tilde{\nu}}$ *be the node identifiers in the label of* ν*. Define* $\underline{\tilde{t}}$ *through* $\tilde{t}_j := f(\tilde{\nu}_j)$*.*

2. *Proceed depending on the type of* ν*:*
   - ***If*** ν ***is a computation node with constructor, destructor, or nonce*** F*, then send the question* $F(\underline{\tilde{t}})$ *over* net *and wait for a response* m*. If* $m = \texttt{yes}$*, let* ν' *be the yes-successor of* ν *and let* $f' := f(\nu := F(\underline{\tilde{t}}))$*. If* $m = \texttt{no}$*, let* ν' *be the no-successor of* ν *and let* $f' := f$*. Set* $f := f'$ *and* $\nu := \nu'$*.*
   - ***If*** ν ***is an output node****, send* $\tilde{t}_1$ *over* net*. Let* ν' *be the unique successor of* ν *and let* $S' := S \cup \{\tilde{t}_1\}$*. Set* $\nu := \nu'$ *and* $S := S'$*.*
   - ***If*** ν ***is an input node****, wait on* net *to receive* $m \in \mathbf{T}$ *from Sim. Let* $f' := f(\nu := m)$*, and let* ν' *be the unique successor of* ν*. Set* $f := f'$ *and* $\nu := \nu'$*.*
   - ***If*** ν ***is a control node labeled with out-metadata*** l*, send* l *over* net*, and wait to receive a bitstring* l' *from* net*. Let* ν' *be the successor of* ν *along the edge labeled* l' *(or the edge with the lexicographically smallest label if there is no edge with label* l'*). Set* $\nu := \nu'$*.*
3. *Send* $(info, \nu, t)$ *over* net*. When receiving an answer* (*proceed*) *from* net*, continue.*
4. *If over* net *has output a final assignment from variables to symbolic operations is sent, check whether the final assignment is consistent with all the responses to the eval-questions. If the check fails, abort with* `inconsistency`*. Otherwise, hand the control back over* net*.*
5. *Otherwise, if over* net *a final assignment is not sent, let* $(S_i, \nu_i, f_i) := (S, \nu, f)$*.*

Indistinguishability and Dolev-Yaoness of the Faking Simulator. On one hand, in order to ensure that this translation from a transcript of bitstrings to a symbolic attacker strategy is accurate, the simulator needs to produce a symbolic attacker strategy that matches the interaction of the computational attacker with the computational execution challenger. In particular, the trace of protocol states that the hybrid execution produces has to be computationally indistinguishable from the one that the computational execution produces. In CoSP, this property is called *indistinguishability* of a simulator.

On the other hand, in spite of accurately modeling the computational execution, the produced symbolic attacker strategy has to obey the symbolic rules, i.e., w.r.t. the symbolic model it has to be a valid symbolic attacker strategy. In CoSP, this property is called *Dolev-Yaoness* of a simulator.

The Dolev-Yaoness is proven by showing that the computational execution challenger against the attacker $\mathcal{A}$ produces indistinguishable traces from the hybrid execution challenger against a faking simulator Sim_f[8] that fakes all ciphertexts and forwards all messages back and forth from the internally simulated attacker $\mathcal{A}$.

For the proof of the computational self-monitoring property, we use this faking simulator Sim_f. We review the construction of Sim_f, but for the sake of brevity we only describe in detail how Sim_f handles ciphertexts. The full description can be found in the work of Backes, Malik, and Unruh [BMU12].

For translating bitstrings to terms and vice versa, the simulator Sim_f has two efficiently computable stateful functions τ and β: τ assigns to each bitstring a term and β assigns to each term a bitstring. The simulator is constructed such that τ and β are partially inverse for each other, i.e., for all terms t that are sent by the hybrid execution (except for the

[8] In the paper by Backes, Malik, and Unruh [BMU12], this simulator is called Sim_7.

randomness) the equation $\tau(\beta(t)) = t$ and for all bitstrings m the equation $\beta(\tau(m)) = m$ holds.

The Honest Simulator *Sim*. For the sake of illustration, we present the definition of the encryption-related cases of the honest simulator *Sim*, and subsequently briefly describe how the faking simulator Sim_f differs from *Sim*.

Let $\mathcal{R}$ be the set of randomness terms that has been sent by the hybrid execution before the respective invokation of β or τ. In the following definition the first case that applies is taken.

The definition of τ for the encryption-related bitstrings is as follows:

1. $\tau(r) := N$ if $r = r_N$ for some $N \in \mathcal{N} \setminus \mathcal{R}$.
2. $\tau(r) := N^r$ if r is of type nonce.
3. $\tau(c) := enc(ek(M), \mathbf{T}, N)$ if c has earlier been output by $\beta(enc(ek(M), t, N))$ for some $M \in \mathbf{N}$, $N \in \mathcal{R}$.
4. $\tau(c) := enc(ek(N), \tau(m), N^c)$ if c is of type ciphertext and $\tau(A_{ekof}(c)) = ek(N)$ for some $N \in \mathcal{R}$ and $m := A_{dec}(A_{dk}(r_N), c) \neq \bot$.
5. $\tau(c) := garbageEnc(ek(N), N^c)$ if c is of type ciphertext and $\tau(A_{ekof}(c)) = ek(N)$ for some $N \in \mathcal{R}$ but $A_{dec}(A_{dk}(r_N), c) = \bot$.
6. $\tau(c) := x^c$ if c is of type ciphertext but $\tau(A_{ekof}(c)) \neq ek(N)$ for all $N \in \mathcal{R}$.
7. $\tau(e) := ek(N)$ if e has earlier been output by $\beta(ek(N))$ for some $N \in \mathcal{R}$.
8. $\tau(e) := ek(N^e)$ if e is of type encryption key.
9. $\tau(k) := dk(N)$ if k has earlier been output by $\beta(dk(N))$ for some $N \in \mathcal{R}$.
10. $\tau(k) := dk(N^e)$ if k is of type decryption key and $e := A_{ekofdk}(k) \neq \bot$.

The definition of β for the encryption-related terms is as follows:

1. $\beta(N) := r_N$ if $N \in \mathcal{N}$.
2. $\beta(N^m) := m$.
3. $\beta(enc(ek(t_1), \mathbf{T}_2, M)) := \mathrm{Impl}_{enc}(\beta(ek(t_1)), \beta(\mathbf{T}_2), r_M)$ if $M \in \mathcal{R}$.
4. $\beta(enc(ek(t_1), t, N^m)) := m$.
5. $\beta(x^c) := c$.
6. $\beta(ek(N)) := \mathrm{Impl}_{ek}(r_N)$ if $N \in \mathcal{R}$.
7. $\beta(ek(N^m)) := m$.
8. $\beta(dk(N)) := A_{dk}(r_N)$ if $N \in \mathcal{R}$.[9]
9. $\beta(dk(N^m)) := A^{-1}_{ekofdk}(m)$. (Note that due to Implementation Condition 31, there is at most one value $A^{-1}_{ekofdk}(m)$. And see below for a discussion of the polynomial-time computability of $A^{-1}_{ekofdk}(m)$.)

By keeping a record of all decryption keys d, the simulator can efficiently compute $A^{-1}_{ekofdk}(e)$ in Case 9.

Due to the limited compositionality of tic-indistinguishability (see Definition 23), we need to ensure that even machines that are connected to the network and execution interface cannot distinguish the honest simulator *Sim* from the faking simulator Sim_f. We say that a term t *is used in the randomness of the term* t' if there are terms t_1, t_2 such that $t' \in \{ek(t), dk(t), vk(t), sk(t)\} \cup \{sig(t_1, t_2, t) \mid t_1, t_2 \text{ are terms}\} \cup \{enc(t_1, t_2, t) \mid t_1, t_2 \text{ are terms}\}$.

[9] Technically, before returning the value $\beta(dk(N))$ invokes $\beta(ek(N))$ and discards it return value. (This is to guarantee that $A_{ek}(N)$ can only be guessed when $\beta(ek(N))$ was invoked.) We refer to [BMU12] for a detailed explanation.

Sim checks whether each term t that is used in randomness position of a term t' is only used inside t'. If t is used somewhere else or sent in plain, *Sim* aborts. If *Sim* interacts with a hybrid execution, these checks will always succeed; in these cases *Sim* behaves as the simulator in the work of Backes, Malik, and Unruh [BMU12].

Modifications for the Faking Simulator. In the faking simulator, the ciphertext simulator CS is used (see Implementation Conditions 26) instead of the encryption algorithm. Technically, Sim_f maintains for each $N \in \mathbf{N}_P$ an instance CS_N of the ciphertext simulator. Whenever the encryption scheme algorithm is honestly applied in *Sim*, Sim_f uses CS_N as follows: for $\beta(ek(N))$, Sim_f sends a $\mathtt{getek}_{\mathtt{ch}}$-query to CS_N; for $\beta(enc(ek(N), t_2, M))$ with $N, M \in \mathcal{R}$, Sim_f either sends a $\mathtt{fakeenc}_{\mathtt{ch}}$-query (if there was no $\mathtt{getdk}_{\mathtt{ch}}$-query to CS_N yet) or a *enc*-query to CS_N (if there already was a $\mathtt{getdk}_{\mathtt{ch}}$-query to CS_N; for $\beta(dk(N))$, Sim_f sends a $\mathtt{getdk}_{\mathtt{ch}}$-query to CS_N.

In addition, the faking simulator internally runs an instance of the random oracle $\mathcal{O}$ and gives all ciphertext simulators CS_N access to $\mathcal{O}$ and to get the list of queries and programming capabilities for $\mathcal{O}$. To the attacker $\mathcal{A}$ the simulator Sim_f grants access to $\mathcal{O}$.

Beside these modifications, Sim_f performs book keeping for mapping honestly generated ciphertexts to their respective plaintexts, and Sim_f queries a signing oracle instead of using the signing algorithms. A detailed description of the modifications can be found in [BMU12].

The Hybrid Execution for Equivalence Properties.

Definition 78 (Hybrid Challenger *H-Exec* (Equivalence))**.** *Let* Π *be a CoSP protocol, and let Sim be a simulator. We define an interactive machine* $H\text{-}Exec_{\mathsf{M},\Pi}(k)$ *run on input* k*. It is called the hybrid protocol machine associated with* Π*. It internally maintains on (finite) lists of tuples* (S_i, ν_i, f_i)*, called the* full hybrid trace*, and runs a symbolic simulation of* Π *as follows:*

Initially $S_1 := S := \varepsilon$*,* $\nu_1 := \nu$ *is the root of* Π*, and* $f_1 := f$ *is a totally undefined partial function mapping node identifiers to* $\mathbf{T}$*. For* $i = 2, 3, \ldots$ *do the following (recall that* net *is the network interface):*

1. *Let* $\underline{\tilde{\nu}}$ *be the node identifiers in the label of* ν*. Define* $\underline{\tilde{t}}$ *through* $\tilde{t}_j := f(\tilde{\nu}_j)$*.*
2. *Proceed depending on the type of* ν*:*
   - ν ***is a computation node with constructor, destructor or nonce*** F***.*** *Let* $V' = V$*. Send the question* $F(\underline{\tilde{t}})$ *over* net *and wait for a response* m*. If* $m = \mathtt{yes}$*,* ν' *is the* yes*-successor of* ν *in* I*, and* $f' = f(\nu := F(\underline{\tilde{t}}))$*. If* $m = \mathtt{no}$*, then* ν' *is the* no*-successor of* ν*, and* $f' = f$*.*
   - ν ***is an input node.*** *Wait over* net *for a symbolic operation* O*. Let* $t := xeval_O(V_{Out})$*, where* $xeval_O(V_{Out})$ *is defined just like* $eval_O(V_{Out})$ *except that nodes of type variables are evaluated to the variable with the name that is annotated in the node. Let* $f' := f(\nu := t)$*, and let* ν' *be the unique successor of* ν*. Set* $f := f'$*,* $V' = V :: (\mathtt{in}, (t, O))$*, and* $\nu := \nu'$*.*
   - ν ***is an output node.*** *Send* $\tilde{t}_1$ *over* net*. Let* ν' *be the unique successor of* ν *and let* $S' := S \cup \{\tilde{t}_1\}$*. Set* $\nu := \nu'$*,* $V' = V :: (\mathtt{out}, \tilde{t}_1)$*, and* $S := S'$*.*
   - ν ***is a control node with out-metadata*** l***.*** *Let* ν' *be the successor of* ν *with the in-metadata* l' *(or the edge with the lexicographically smallest label if there is no edge with label* l'*),* $f' = f$*, and* $V' = V :: (\mathtt{control}, (l, l'))$*.*

3. *Send* $(info, \nu, t)$ *over* net. *When receiving an answer* (*proceed*) *over* net, *continue.*
4. *If over* net *has output a final assignment from variables to symbolic operations is sent, check whether the final assignment is consistent with all the responses to the eval-questions. If the check fails, abort with* `inconsistency`. *Otherwise, hand the control back over* net.
5. *Otherwise, if over* net *a final assignment is not sent, let* $(S_i, \nu_i, f_i) := (S, \nu, f)$.

Here, V_{Out} *denotes the list of terms in* V *that have been sent at output nodes, i.e., the terms* t *contained in entries of the form* $(\texttt{out}, t)$ *in* V. *Analogously,* $V_{Out\text{-}Meta}$ *denotes the list of out-metadata in* V *that has been sent at control nodes.*

Furthermore, V_{In} *denotes the partial list of* V *that contains only entries of the form* $(\texttt{in}, (*, O))$ *or* $(\texttt{control}, (*, l'))$ *for some symbolic operation* O *and some in-metadata* l', *where the input term and the out-metadata have been masked with the symbol* $*$. *The list* V_{In} *is called* attacker strategy. *We write* $[V_{In}]_{SViews(I)}$ *to denote the class of all views* $U \in SViews(I)$ *with* $U_{In} = V_{In}$.

The Faking Simulator Sim'_f for Equivalence Properties. We define a faking simulator Sim'_f for equivalence properties as the faking simulator Sim_f for trace properties except that the final output, i.e., the guess, of the attacker $\mathcal{A}$ sent over the output interface is also output by Sim'_f and all terms that are sent to the hybrid challenger from Sim_f are parsed in Sim'_f to symbolic operations, using CONSTRUCT-SHAPE.

6.3.5.4. CS for Trace Properties with Length Functions

The following theorem follows from the CS result of Backes, Unruh, and Malik [BMU12]. We discuss the differences to the CS proof in [BMU12].

Theorem 9. *Let* $\mathbf{A}$ *be a computational implementation fulfilling the implementation conditions (see Appendix 6.3.2), i.e., in particular* $\mathbf{A}$ *is length-consistent. Then,* $\mathbf{A}$ *is a computationally sound implementation of the symbolic model* M *for the class of randomness-safe protocols (see Definition 70).*

Proof. The implementation conditions that we require are a superset of the conditions in the work of Backes, Malik, and Unruh (our additional conditions are marked blue). Hence, every implementation that satisfies our implementation condition also satisfies the implementation condition of their work. Moreover, we add one protocol condition that excludes *garbageInvalidLength*-terms as arguments for constructors. This protocol condition only further restricts the class; hence, without length functions, their CS result would still hold.

We extend the simulator *Sim* in the CS proof of the work of Backes, Malik, and Unruh to also parse (τ) and to produce (β) length functions. The rules are straight-forward and follow the same pattern as for $string_b$ and $unstring_b$.

Length functions are constant functions that the attacker can produce on its own, just like payload strings ($string_b$). Consequently, the Dolev-Yaoness of the simulator also holds in the presence of length functions

For the translation functions $\beta : \mathbf{T} \to \{0,1\}^*$ and $\tau : \{0,1\}^* \to \mathbf{T}$ of the simulator, $\beta(len(m)) = A_{len}(\beta(m))$, $\tau(\beta(t)) = t$, $\beta(t) \neq \bot$, and $\beta(\tau(b)) = b$ follow from the imple-

DFC(k) : **upon out-metadata from a decision node**
  respond with the in-metadata `continue`

DFC(k) : **upon another message** m
  send m via the network interface

  if a message m' is received over the network interface **then**
    forward the message m' to execution network interface
  else if a guess b is received over the output interface **then**
    output the guess b

DFNT(k) : **upon out-metadata from a decision node**
  if a guess b is received vai the output interface **then**
    respond with the in-metadata `decision-`b
  else
    respond with the in-metadata `continue`

DFNT(k) : **upon another message** m
  if no guess was received over the output interface yet **then**
    send m over the network interface
    if a message m' is received over the network interface **then**
      send the message m' to the execution network interface
    else if a guess b is received over the output interface **then**
      store the guess b
  else
    respond with dummy messages, i.e., for an input node send the constant 0 bitstring, for a control node that is not a decision node send one of the possible in-metadata, and for an output node give back control to the execution interface party

Figure 6.9.: Code for the Decision Node Filter for Communication and for Node Traces

mentation condition. The indistinguishability of the simulator follows from these equation (see [BMU12]).

As shown in the initial work on CoSP (see Theorem 1), a simulator that satisfies Dolev-Yaoness and Indistinguishability implies computational soundness. □

6.3.5.5. Decision variant of a protocol

In this section, we show how to reduce indistinguishability of the transcripts to indistinguishability of nodes traces. To this end, we define the decision variant of a protocol.

Definition 79 (Decision Node Filter for Communication and for Node Traces)**.** *The* decision node filter for communication *is the interactive machine* DFC *that is defined in Figure 6.9. The* decision node filter for node traces *is the interactive machine* DFNT *that is defined in Figure 6.9.*

Both machines expect two communication partners. In our notation, there will be a left communication partner and a right communication partner. Each of these partners offer a network interface: the network interface of the left partner (typically the execution) is called the execution network interface, and the network interface of the right partner (typically the adversary or simulator) is called the network interface. The right communication partner additionally offers an output interface (over which it typically sends its distinguisher-guess). We call this interface the output interface.

Definition 80 (Decision-Variant)**.** *Given a bi-protocol* Π*, the* decision-variant $\tilde{\Pi}$ *is defined like* Π *but with the modification that each node* ν *that appears in* Π *is preceded by an additional control node, called* decision node*, as follows: The decision node has three successors with the edges to them labeled by* `decision-0`*,* `decision-1`*, and* `continue`*. The* `continue`*-successor is* ν *here (i.e., the protocol continues). Both the* `decision-0`*-successor*

and the `decision-1`*-successor are roots of infinite chains of dummy control nodes with only one successor (i.e., the protocol stops).*[10]

Lemma 34. *Let* M *be a symbolic model,* Impl *be an implementation,* $\mathcal{A}$ *be a ppt machine, and* P *be the class of all bi-protocols that are the left or the right variant of an efficient randomness-safe bi-protocol Definition 70. If there is a ppt machine* Sim_f *such that for all protocols* $\Pi \in \mathrm{P}$*, the node traces* $\langle TrExec_{\mathsf{M},\mathrm{Impl},\Pi} \mid \mathcal{A}\rangle$ *and* $\langle TrH\text{-}Exec_{\mathsf{M},\mathrm{Impl},\Pi} \mid \langle Sim_f \mid \mathcal{A}\rangle\rangle$ *are computationally indistinguishable, then for all protocols* $\Pi \in \mathrm{P}$*, the executions* $\langle Exec_{\mathsf{M},\mathrm{Impl},\Pi} \mid \mathcal{A}\rangle$ *and* $\left\langle H\text{-}Exec_{\mathsf{M},\mathrm{Impl},\Pi} \mid \left\langle Sim'_f \mid \mathcal{A}\right\rangle\right\rangle$ *are computationally indistinguishable.*

Proof. For convenience, we consider a computational challenger *TrExec* and hybrid challenger *TrH-Exec* for trace properties with the following modifications: instead of calling the distinguisher with the node trace after the interaction between the (respective) challenger and the adversary (or the simulator), the challenger assumes that the attacker consists of two machines, the normal adversary and the distinguisher, that do not share their state and it sends the node trace to the second machine. The distinguisher then ends the interaction by outputting a guess $b \in \{0,1\}$.

Accordingly, we define for an adversary $\mathcal{A}$ another machine $\mathcal{A}'$ that consists of two parts: first, a modification of $\mathcal{A}$ that instead of outputting its final guess b chooses via in-metadata the `decision-`b node and then stops; second, a distinguisher part that checks which `decision-`b was taken and outputs a guess b.

Since the simulator need to be compatible with these potentially two parts of $\mathcal{A}$ forwards, we define a variant Sim_f^* of the faking simulator Sim_f for trace properties: Sim_f^* forwards the node trace to the distinguisher part and outputs the final guess of the distinguisher part.

The following interactions are perfectly indistinguishable for the adversary $\mathcal{A}$.

$$
\begin{aligned}
& Exec_{\mathsf{M},\mathrm{Impl},\Pi} \\
\overset{(1)}{\approx}_{time}\ & \left\langle Exec_{\mathsf{M},\mathrm{Impl},\tilde{\Pi}} \mid \langle \mathrm{DFC} \mid \mathcal{A}\rangle\right\rangle \\
\overset{(2)}{\approx}_{time}\ & \left\langle Exec_{\mathsf{M},\mathrm{Impl},\tilde{\Pi}} \mid \langle \mathrm{DFNT} \mid \mathcal{A}\rangle\right\rangle \\
\overset{(3)}{\approx}_{time}\ & \left\langle TrExec_{\mathsf{M},\mathrm{Impl},\tilde{\Pi}} \mid \left\langle \mathrm{DFNT} \mid \mathcal{A}'\right\rangle\right\rangle
\end{aligned}
$$

[10] Formally, $\tilde{\Pi}$ is the limes of the operation that recursively replaces each node ν by a decision node followed by ν.

$$
\begin{aligned}
& \left\langle \mathit{TrH\text{-}Exec}_{\mathsf{M},\mathrm{Impl},\tilde{\Pi}} \mid \left\langle \left\langle \mathit{Sim}_f^* \mid \mathrm{DFNT} \right\rangle \mid \mathcal{A}' \right\rangle \right\rangle \\
\overset{(4)}{\approx}_{time} & \left\langle \mathit{TrH\text{-}Exec}_{\mathsf{M},\mathrm{Impl},\tilde{\Pi}} \mid \left\langle \left\langle \mathrm{DFNT} \mid \mathit{Sim}_f^* \right\rangle \mid \mathcal{A}' \right\rangle \right\rangle \\
\overset{(3)}{\approx}_{time} & \left\langle \mathit{H\text{-}Exec}_{\mathsf{M},\mathrm{Impl},\tilde{\Pi}} \mid \left\langle \left\langle \mathrm{DFNT} \mid \mathit{Sim}'_f \right\rangle \mid \mathcal{A} \right\rangle \right\rangle \\
\overset{(2)}{\approx}_{time} & \left\langle \mathit{H\text{-}Exec}_{\mathsf{M},\mathrm{Impl},\tilde{\Pi}} \mid \left\langle \left\langle \mathrm{DFC} \mid \mathit{Sim}'_f \right\rangle \mid \mathcal{A} \right\rangle \right\rangle \\
\overset{(1)}{\approx}_{time} & \left\langle \mathit{H\text{-}Exec}_{\mathsf{M},\mathrm{Impl},\Pi} \mid \left\langle \mathit{Sim}'_f \mid \mathcal{A} \right\rangle \right\rangle
\end{aligned}
$$

(1): The decision-variant $\tilde{\Pi}$ of a protocol Π sends the same messages as the original protocol Π to the adversary except for adding a special kind of control nodes: decision nodes (see Definition 80). The messages with the out-metadata from the decision nodes is filtered by the machine DFC, and except for filtering these additional messages DFC does nothing more than forwarding all messages to $\mathcal{A}$. Hence, the indistinguishability holds.

(2): Until $\mathcal{A}$ stops with outputting a bitstring b, the machine DFNT behaves just like DFC. Hence, the indistinguishability holds.

(3): The two executions *Exec* and *TrExec* behave exactly the same towards the communication partner (i.e., $\langle \mathrm{DFNT} \mid \mathcal{A} \rangle$ or $\langle \mathrm{DFNT} \mid \langle \mathit{Sim}_f \mid \mathcal{A} \rangle \rangle$, respectively, in our case). Hence, the interactions are indistinguishable for $\mathcal{A}$. The same holds for *H-Exec* and *TrH-Exec* except that in this case we additionally replace Sim_f with Sim'_f, which, by the construction of Sim'_f (see Section 6.3.5.3), is not observable to $\mathcal{A}$.

(4): By the construction of Sim_f^*, we can see that the sub-machine $\mathcal{A}$ does not see a difference if first the simulator Sim_f^* is invoked, as in $\langle \mathit{Sim}_f^* \mid \langle \mathrm{DFNT} \mid \mathcal{A} \rangle \rangle$ or first the decision nodes are filtered, as in $\langle \mathrm{DFNT} \mid \langle \mathit{Sim}_f^* \mid \mathcal{A} \rangle \rangle$. Moreover, Sim'_f outputs the output of its sub-machine, which is in one case $\mathcal{A}$ and in the other case $\langle \mathrm{DFNT} \mid \mathcal{A} \rangle$. Hence, the behavior towards *TrH-Exec* is indistinguishable, as well.

Moreover, whenever the attacker $\mathcal{A}$ (or the simulator $\langle \mathit{Sim}'_f \mid \mathcal{A} \rangle$) outputs a guess b, DFNT eventually sends to a decision node the in-metadata `decision-`b. Hence, the indistinguishability relations (1)–(4) imply that the node traces output by

$$\left\langle \mathit{TrExec}_{\mathsf{M},\mathrm{Impl},\tilde{\Pi}} \mid \langle \mathrm{DFNT} \mid \mathcal{A} \rangle \right\rangle \text{ and } \left\langle \mathit{TrH\text{-}Exec}_{\mathsf{M},\mathrm{Impl},\tilde{\Pi}} \mid \langle \mathit{Sim}_f^* \mid \langle \mathrm{DFNT} \mid \mathcal{A} \rangle \rangle \right\rangle$$

are distinguishable if the adversary can distinguish

$$\langle \mathit{Exec}_{\mathsf{M},\mathrm{Impl},\Pi} \mid \mathcal{A} \rangle \text{ from } \left\langle \mathit{H\text{-}Exec}_{\mathsf{M},\mathrm{Impl},\Pi} \mid \left\langle \mathit{Sim}'_f \mid \mathit{attacker} \right\rangle \right\rangle$$

This statement is the contraposition of the claim of the lemma.

□

6.3.5.6. Uniqueness of a symbolic operation

For a CoSP bi-protocol Π and an adversary $\mathcal{A}$, we say that a symbolic operation O *occurs in a run* of $\left\langle \mathit{Exec}_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi)-\mathrm{F}_b} \mid \mathcal{A} \right\rangle$ *for a message* m for one invokation of

CONSTRUCT-ATTACKER-STRATEGY(b, tr) if it is a sub-symbolic operation of a symbolic operation that was generated by one monitor call. More precisely, a symbolic operation O occurs in a run of $\langle Exec_{\mathsf{M,Impl,Mon}(\Pi)-\mathrm{F}_b} \mid \mathcal{A} \rangle$ if in that run there is an invokation of $V :=$ CONSTRUCT-ATTACKER-STRATEGY(b, tr) such that there is a symbolic operation context C with $V_j = (\_, C[O])$ and $eval_{\bar{O}}(tr) = m$. Analogously, we talk about several symbolic operations $O_1, \ldots, O_l$ occurring in a run of $\langle Exec_{\mathsf{M,Impl,Mon}(\Pi)-\mathrm{F}_b} \mid \mathcal{A} \rangle$ for messages $m_1, \ldots, m_l$ if each O_j (for $j \in \{1, \ldots, l\}$) occurs in the same run of $\langle Exec_{\mathsf{M,Impl,Mon}(\Pi)-\mathrm{F}_b} \mid \mathcal{A} \rangle$ for m_j. If it is clear from the context which computational execution is meant and which invokation is meant, will only say that a symbolic operation O *occurs in a run for a message* m.

Lemma 35. *Let O_{left} and O_{right} be two output shapes that are generated by the knowledge monitor. If $O_{left} \neq O_{right}$, then the knowledge monitor raises an alarm* `bad-knowledge`.

In contraposition: whenever two output shapes O_{left} and O_{right}, generated by an invokation of the knowledge monitor, do not cause an alarm `bad-knowledge`*, we have $O_{left} = O_{right}$. We call this shape the* joint symbolic operation O_b *of that invokation.*

Proof. In the construction of the monitor, in Figure 6.8, an alarm is only raised if the symbolic operations are not equal. □

Modifications to CONSTRUCT-SHAPE. As a technical vehicle, we run a modification of the knowledge monitor. We modify CONSTRUCT-SHAPE such that it additionally outputs all sub-shapes and all symbolic operations $((O_m, \text{“}x_i''\text{”}), (O_1, \text{“}O_1''\text{”}), \ldots, (O_n, \text{“}O_n\text{”}))$ that characterize how the messages to those sub-shapes have been computed (in Figure 6.6 denoted in quotes: “O”). We stress that, due to the recursive implementation of CONSTRUCT-SHAPE, the sequence of sub-shapes and symbolic operations is a list in which, if reversed, the sub-shapes always occur before their parent shapes. All other parts of the knowledge monitor remain the same except that they ignore these sub-shapes and the symbolic operations and only use the (top-level) shape, as before. These sub-shapes and the symbolic operations are need to compute the visible submessages of an output message (see Figure 6.10).

Corollary 4. *Let Π be a CoSP bi-protocol and $\mathcal{A}$ be an adversary. If O is not a sub-symbolic operation, i.e., $C = \cdot$, then we have the following property: For each invokation of* CONSTRUCT-ATTACKER-STRATEGY(b, tr) *in any run of $\langle Exec_{\mathsf{M,Impl,Mon}(\Pi)-\mathrm{F}_b} \mid \mathcal{A} \rangle$, we have that for every symbolic operations O for m, if m' is the actual message for which O was generated, i.e., the invokation of $O =$* CONSTRUCT-SHAPE$(m', x_j, K, b, tr,\ dec')$*, then*

$$m = m'$$

Proof. The dual symbolic operation output by CONSTRUCT-SHAPE$'$ simply points with the projection x_j to the very message m' in the computational view. □

Lemma 36. *Let Π be a CoSP bi-protocol and $\mathcal{A}$ be an adversary. For each invokation of* CONSTRUCT-ATTACKER-STRATEGY(b, tr) *in any run of $\langle Exec_{\mathsf{M,Impl,Mon}(\Pi)-\mathrm{F}_b} \mid \mathcal{A} \rangle$, we have that for every pair of sub-symbolic operations O and O' that occur in a run of $\langle Exec_{\mathsf{M,Impl,Mon}(\Pi)-\mathrm{F}_b} \mid \mathcal{A} \rangle$ for m and m', respectively, the symbolic operations are unique, i.e.,*

$$O \neq O' \Leftrightarrow m \neq m'$$

```
SUBMESSAGE(m, tr)
  let shapes = ((O_m, "x''_i), (O_1, "O''_1), ..., (O_n, "O_n"))
  be the output of the last call (for m)
  CONSTRUCT-SHAPE(m, x_i, K, left, tr, dec')
reverse shapes
for each (O_j, "O''_j) in shapes do
  let m_j = eval_"O''_j (left(tr))
  output m_j
```

Figure 6.10.: The algorithm SUBMESSAGE(m, tr) that is called in the unrolled variant $\tilde{\Pi}$

Proof. Assume towards contradiction that there is a pair O, O' such that $O \neq O'$ but $m = m'$. Let m, m' be the first pair of messages in the run for which this violation occurs. W. l.o.g. assume that first m and then m' was constructed. While constructing O', the algorithm CONSTRUCT-SHAPE first checks whether there is already a symbolic operation O'' in the knowledge K such that $eval_{\bar{O}''} = m'$. The evaluation *eval* of the dual operation $\bar{O}''$ uses exactly the same deterministic algorithms for retrieving the bitstring that were used to construct O''. Hence, m' coincides with $eval_{\bar{O}''}$, i.e., the test $eval_{\bar{O}''} = m'$ is well-defined. Thus, if the same bitstring was already parsed, the corresponding symbolic operation is found and used and no new symbolic operation is used.

Moreover, a non-constant minimal symbolic operations O' always uses as randomness $nonceof(\bar{O}'')$ with the dual operation $\bar{O}''$ of O'', which prevents two symbolic operations to be the same for two different bitstrings.

Observe that the knowledge K is already fully saturated; hence, it cannot happen that while constructing O' in CONSTRUCT-SHAPE a new symbolic operation is learned that leads to a larger K', i.e., $K \neq K'$ and $K \subset K'$. As a consequence, if O' occurs for m', $m = m'$, and O occurs for m, O is always found; hence $O' = O'$.

For showing the contradiction, we also have to consider the case where $O = O'$ but $m \neq m'$. By construction this cannot happen because $m = eval_{\bar{O}}(tr) = eval_{\bar{O}'}(tr) = m'$. □

6.3.5.7. Unrolled variants

In order to simplify the main proof, we define, for a given protocol Π, its unrolled variant $\tilde{\Pi}$, which before each output node of Π sends all visible sub-messages of the message from that output node.

Definition 81 (Unrolled Variant)**.** *Let* SUBMESSAGE(m, tr) *be the algorithm that is depicted in Figure 6.10. Given a protocol* Π*, we define its* unrolled variant $\tilde{\Pi}$ *as the protocol in which each output node with a message* m *is replaced by , where* tr *is the trace from the output node to the root.*[11]

Corollary 5 (Unrolled variants preserve uniformity-enforcing)**.** *For all uniformity-enforcing protocols* Π*, the unrolled variant* $\tilde{\Pi}$ *of* Π *is uniformity-enforcing.*

Proof. The inserted sub-protocol is uniformity-enforcing since before each computation node a single-successor control node is placed that has a unique out-metadata. □

[11] Formally, $\tilde{\Pi}$ is the limes of the recursive replacement operation that replaces each output node with SUBMESSAGE(m, tr).

Lemma 37 (Symbolic indistinguishability is preserved in unrolled variants)**.** *For all efficient pairs of protocols* Π_1 *and* Π_2*, if* Π_1 *and* Π_2 *are symbolically indistinguishable, then the corresponding unrolled variants* $\tilde{\Pi}_1$ *and* $\tilde{\Pi}_2$ *are symbolically indistinguishable.*

Proof. First, we show the following statememt.

Claim 1. *For each CoSP protocol* Π*, let* $\tilde{\Pi}$ *be the unrolled variant of* Π*. For all attacker-strategies* V_{In} *and for all views* $V \in [V_{In}]_{SViews(\Pi)}$ *and* $V' \in [V_{In}]_{SViews(\tilde{\Pi})}$ *the symbolic knowledge* K_V *of* Π *and* $K_{V'}$ *of* $\tilde{\Pi}$ *is the same, i.e.,* $K_V = K_{V'}$*.*

Proof of Claim 1. The statement follows from the fact that the only difference of an unrolled variant $\tilde{\Pi}$ and the original protocol Π is that $\tilde{\Pi}$ additionally sends the output messages computed by $\textsc{Submessage}(m, tr)$. Since the additional messages sent by $\textsc{Submessage}(m, tr)$ are in the symbolic knowledge after m is sent, the statement follows. ◇

Assume that the unrolled variants $\tilde{\Pi}_1$ and $\tilde{\Pi}_2$ are not symbolically indistinguishable. W.l.o.g., there is an attacker strategy V_{In} and a view $\tilde{V} \in [V_{In}]_{SViews(\tilde{\Pi}_1)}$ of $\tilde{\Pi}_1$ under V_{In} such that for all views $\tilde{V}' \in [V_{In}]_{SViews(\tilde{\Pi}_2)}$ of $\tilde{\Pi}_2$ under V_{In} it holds that $\tilde{V} \not\sim V'$. Let $V \in [V_{In}]_{SViews(\Pi_1)}$ and $V' \in [V_{In}]_{SViews(\Pi_2)}$ be the corresponding views for Π_1 and Π_2.

For all $\tilde{V}' \in [V_{In}]_{SViews(\tilde{\Pi}_2)}$, one of the following three statements does not hold:

1. (Same structure) $\tilde{V}_i$ is of the form $(s, \cdot)$ if and only if V_i' is of the form $(s, \cdot)$ for some $s \in \{\texttt{out}, \texttt{in}, \texttt{control}\}$.
2. (Same out-metadata) $\tilde{V}_{\textit{Out-Meta}} = \tilde{V}'_{\textit{Out-Meta}}$.
3. (Same symbolic knowledge) $K_{\tilde{V}} = K_{\tilde{V}'}$.

If the structure is not the same (Case 1), then we first consider the following cases, which are related to invokations of Submessage: either a different different branch was taken in an invokation of Submessage or a different amount of output messages were sent by an invokation of Submessage. Since Submessage internally only branches when computing Construct-shape, all tests that Submessage performs are in the shared knowledge and hence in the symbolic knowledge of Π_i. Thus, in these cases, the symbolic knowledge is already different for Π_1 and Π_2. In all other cases, the unrolled variant $\tilde{\Pi}_i$ coincides with the original protocol Π_i; hence the same difference occurs in Π_i.

For the case that the out-metadata is different (Case 2), observe that the unrolled variant does not sends the same out-metadata as the original protocol. Hence, the statement follows.

For the case that the symbolic knowledge (Case 3), the statement follows from the claim above. □

Lemma 38 (Unrolled variants preserve computational indistinguishability)**.** *For all efficient protocols* Π_1*,* Π_2*, if the corresponding unrolled variants* $\tilde{\Pi}_1$ *and* $\tilde{\Pi}_2$ *are indistinguishable, then* Π_1 *and* Π_2 *are indistinguishable. Moreover, if the corresponding unrolled variants* $\tilde{\Pi}_1$ *and* $\tilde{\Pi}_2$ *are perfectly indistinguishable, then* Π_1 *and* Π_2 *are perfectly indistinguishable.*

Proof. We construct a reduction R from a distinguisher $\mathcal{A}$ of two original protocols Π_1 and Π_2 to a distinguisher $R_{\mathcal{A}}$ of the two unrolled variants $\tilde{\Pi}_1$ and $\tilde{\Pi}_2$. The reduction

$R_{\mathcal{A}}$ internally runs $\mathcal{A}$ but drops all sub-messages and only sends the final message to the attacker. Finally, $R_{\mathcal{A}}$ outputs the same guess as the distinguisher $\mathcal{A}$. The view of $\mathcal{A}$ is perfectly indistinguishable to an interaction with Π_1 and Π_2, and the success probability of the reduction $R_{\mathcal{A}}$ to distinguish $\tilde{\Pi}_1$ from $\tilde{\Pi}_2$ equals the success probability of $\mathcal{A}$ for distinguishing Π_1 from Π_2. □

6.3.5.8. Computational self-monitoring for the knowledge monitor

Finally, we are in a position present the proof of the computational self-monitoring.

Symbolic distinguishability. In the proof of computational self-monitoring, we need the property that the monitor is symbolically distinguishing, i.e., that it symbolically always raises an alarm if a pair of protocols is symbolically distinguishable.

Definition 82 (Symbolically distinguishing Monitor)**.** *A monitor is* symbolically distinguishing *if for all bi-protocols* Π *the following holds: if* Π *is not equivalent, then the monitor raises an alarm.*

Lemma 39 (Knowledge monitor is symbolically distinguishing)**.** *The distinguishing sub-protocol* $f_{\texttt{bad-knowledge},\Pi}$ *is symbolically distinguishing in the sense of Definition 82.*

Proof. We have to show that if the monitor $f_{\texttt{bad-knowledge},\Pi}$ does not raise an alarm, then a bi-protocol is equivalent (see Definition 82). By Lemma 35, we know that if the monitor does not raise an alarm, then the symbolic operation that is computed for the output message for $\mathit{left}(\Pi)$ and $\mathit{right}(\Pi)$ is the same. We denote this symbolic operations as O_b.

We stress that for each run the symbolic operations output by Construct-shape are unique w.r.t. the messages that are sent or received for the following reason: for all symbolic operations O that use randomness, Construct-shape places $\mathit{nonceof}(\bar{O})$ as the symbolic operation for the randomness, where $\bar{O}$ is the respective dual symbolic operations of O. As a consequence, it cannot happen that Construct-shape assigns for two different messages in one run the same symbolic operation.

As a next step, we perform an induction over the length i of the protocol and show that for all cases O_b is equivalent. Assume that the monitor did not raise an alarm and Π_{i-1} is equivalent but Π_i is not equivalent.

1. O_b **is a message that is in the attacker-knowledge against** Π_{i-1}**.** Messages that are in the attacker-knowledge against Π_{i-1} are messages that the attacker could have sent after an interaction Π, i.e., with $\Pi_{i-1,\mathit{left}}$ or $\Pi_{i-1,\mathit{right}}$. For these messages the equivalence follows by induction hypothesis.

   This case, in particular, covers garbage terms, garbage encryptions, garbage signatures, and attacker-generated keys. For attacker-generated ciphertexts and signatures, since we consider unrolled variants, all visible sub-messages do not lead to distinguishing symbolic operations by induction hypothesis. Recall that, due to the recursive implementation of Construct-shape, the sequence of sub-shapes and symbolic operations is a list in which, since reversed, the sub-shapes always occur before their parent shapes. Thus, the same holds for Submessage and thus for unrolled variants. Hence, attacker-generated ciphertexts and signatures are also covered by this case. Consequently, we exclude such attacker-generated messages in the following cases.

2. **Empty string:** $O_b = \mathit{empty}()$**.** If $O_{\mathit{left}} = O_{\mathit{right}} = \mathit{empty}()$, then $t_{\mathit{left}} = t_{\mathit{right}} = \mathit{empty}()$. Hence, the equivalence follows.
3. **Protocol nonce:** $O_b = n_P$**.** This case cannot occur. In Construct-shape, a protocol nonce is always parsed as a projection.
4. **Protocol-generated encryption or verification key:** $O_b \in \{\mathit{ek}(\mathit{nonceof}(O_1)), \mathit{vk}(\mathit{nonceof}(O_1))\}$**.** For protocol-generated keys, equality of the symbolic operation means that the corresponding message is equal. Either the encryption key is fresh in the shared knowledge, i.e., never used before, then learning it does not help distinguishing Π_{i-1}, or the encryption key was used earlier, then it already was in the shared knowledge for Π_{i-1} and by induction hypothesis the messages are equivalent. The same argumentation holds for the case if O_b characterizes a verification key.
5. **Protocol-generated decryption key:** $O_b \in \{\mathit{dk}(\mathit{nonceof}(O_1)), \mathit{dkofek}(O_1)\}$**.** For protocol-generated keys, equality of the symbolic operation means that the corresponding message is equal. The decryption key can only be used to decryption ciphertexts with the corresponding encryption key. The Construct-attacker-strategy reconstructs all symbolic operations after learning the output message of node i, which O_b characterizes, and the knowledge monitor checks by whether all resulting symbolic operations are equal. Hence, whenever a decryption key is learned, all symbolic operations that characterize ciphertexts that are constructed with the matching encryption key contain the respective plaintext. As a consequence, every distinguishing test leads to different symbolic operations (see Lemma 36). By contraposition, since `bad-knowledge` was not raised and hence all symbolic operations are equal, there is no distinguishing test.
6. **Protocol-generated signing key:** $O_b \in \{\mathit{sk}(\mathit{nonceof}(O_1)), \mathit{skofvk}(O_1)\}$**.** By induction hypothesis, Π_{i-1} is equivalent; hence, there is no distinguishing symbolic operation for $\Pi_{i-1,\mathit{left}}$ or $\Pi_{i-1,\mathit{right}}$. Since $O_{\mathit{left}} = O_{\mathit{right}} = O_b$, learning the message that corresponds to O_b does not offer a distinguishing equality test. Since only certain plaintexts of ciphertexts (i.e., encryption terms) and randomness terms are unaccessible. Plaintexts can only be accessed via decryption keys, and in the symbolic model there is no destructor that grants access to the randomness, *sk* does not introduce novel distinguishing symbolic operations for Π_i.
7. **A protocol-generated, derived public-key:** $O_b \in \{\mathit{vkofsk}(O_1), \mathit{ekofdk}(O_1)\}$**.** Either these protocol-generated keys are already in the adversary's knowledge, or they have never been used anywhere yet. Consequently, equivalence follows from the equivalence of Π_{i-1}.
8. **The virtual term:** $O_b = \mathit{plaintextof}(O_1, O_2)$**.** This case cannot occur since $\mathit{plaintextof}(O_1, O_2)$ only occurs inside an encryption symbolic operation and it is never a visible sub-message.
9. **Protocol-generated encryption:** $O_b = \mathit{enc}(O_1, \mathit{plaintextof}(O_2, O_3), O_4)$**.** Observe that by the construction of Construct-shape this case only happens for honestly generated ciphertexts for which the decryption key has not been leaked. More precisely, if the decryption key is not in the symbolic knowledge of the adversary. Hence, it suffices to show that the length of O_2 is the same in the term that corresponds to O_b in $\mathit{left}(\Pi_i)$ and $\mathit{right}(\Pi_i)$. If no alarm was raised, the length of both terms are the same because the knowledge monitor implicitly performs these

length tests by comparing the symbolic operations, which in turn are unique for each length of a message.

10. **Protocol-generated encryption:** $O_b = enc(O_1, O_2, O_3)$ **where** $O_2 \neq plaintextof(O_4, O_5)$**.** By the construction of CONSTRUCT-SHAPE, we know that this case can occur in two cases: first, if the corresponding the decryption key is in the shared knowledge and second if the encryption was produced with an attacker-key. Since we consider unrolled variants, we know that with previous output nodes the bitstrings that correspond to O_1 and O_2 have been sent to the adversary. Recall that in the unrolled variants sub-messages are always sent before their parent messages. By assumption we know that for Π_{i-1} the statement holds. By the construction of CONSTRUCT-SHAPE, we know that the term that corresponds to $enc(O_1, O_2, O_3)$ is an honestly generated ciphertext. As a consequence, Π_i did not leak more information that Π_{i-1} since by the protocol conditions the protocol randomness is a freshly chosen nonce. Hence, Π_i is equivalence as well.
11. **Protocol-generated signature:** $O_b \in \{sig(sk(O_1), O_2, O_3), sig(skofvk(O_1), O_2, O_3)\}$**.** Since we consider unrolled protocols, with the last two output nodes the protocol already sent $vkof(O_b)$ and O_2 to the adversary. Recall that in the unrolled variants sub-messages are always sent before their parent messages. In other words, there is no distinguishing symbolic operation against Π_{i-1}, i.e., no symbolic operation that has a different outcome in the left and the right protocol. But then there can also not be any distinguishing symbolic operation against $sig(sk(O_1), O_2, O_3)$ or $sig(skofvk(O_1), O_2, O_3)$, since the symbolic model only has for signatures the verification operation ($verify$), the length test (len), and the retrieval of the verification key ($vkof$).
12. **Pairs:** $pair(O_1, O_2)$**.** Since we solely consider unrolled variants of protocols, we know that both O_1 and O_2 have already been sent in Π_{i-1}. Recall that in the unrolled variants sub-messages are always sent before their parent messages. Hence, the equivalence immediately following from induction hypothesis about Π_{i-1}.
13. **Payload strings:** $O_b \in \{string_0(O_1), string_1(O_1)\}$**.** This case follows by induction hypothesis, since Lemma 38 shows that it suffices to consider unrolled variants of protocols and O_b can be computed out of O_1.
14. **Length term:** $O_b = S(O_1)$**.** This case follows by induction hypothesis, since Lemma 38 shows that it suffices to consider unrolled variants of protocols and O_b can be computed out of O_1.

□

Lemma 40. *The class of randomness-safe (Definition 70) bi-protocols is closed under taking decision-variants and unrolled variants of bi-protocols:*

1. *A bi-protocol is randomness-safe if and only if its decision-variant is randomness-safe.*
2. *A bi-protocol is randomness-safe if and only if its unrolled variant is randomness-safe.*

Proof.

1. By inspection of Definition 70, randomness-safety is invariant under adding and removing control nodes (decision nodes and dummy nodes).
2. By inspection of Definition 70, randomness-safety is invariant under removing output

nodes. However, adding output nodes could potentially violate protocol conditions 2 to 4 in Definition 70. Since Definition 81 adds only output nodes that do not refer to a randomness node, these conditions are not violated.

□

The main lemma. All the preparations in this section culminate in the following technical lemma.

Lemma 41 (Same conditional distribution for faking simulator.)**.** *Let Tr_{break} be the event that there is a trace property that holds symbolically but does not hold in the current run.*

Let $\Omega(\Pi, \mathcal{A})_b$ be defined as the following probability space:

$$guess \leftarrow \left\langle H\text{-}Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi)-\mathrm{F}_b} \mid \left\langle Sim'_f \mid \mathcal{A} \right\rangle \right\rangle$$

where $\mathsf{Mon}(\Pi) - \mathrm{F}_b$ is defined as in Definition 75, $b \in \{left, right\}$, and the output guess of this interaction denotes the output of $Sim_{f,\mathcal{A}}$ (and thereby of $\mathcal{A}$).

For all ppt adversaries $\mathcal{A}$, for all $i \in \mathbb{N}$, and for all uniformity-enforcing, randomness-safe efficient bi-protocols Π: If

$$\begin{aligned}&\Pr[\Omega(\Pi_{i-1}, \mathcal{A})_{left} : guess = 1 \mid \neg Tr_{break} \wedge \neg \mathtt{bad}]\\ &= \Pr[\Omega(\Pi_{i-1}, \mathcal{A})_{right} : guess = 1 \mid \neg Tr_{break} \wedge \neg \mathtt{bad}]\end{aligned}$$

holds, then the following holds:

$$\begin{aligned}&\Pr[\Omega(\Pi_{i}, \mathcal{A})_{left} : guess = 1 \mid \neg Tr_{break} \wedge \neg \mathtt{bad}]\\ &= \Pr[\Omega(\Pi_{i}, \mathcal{A})_{right} : guess = 1 \mid \neg Tr_{break} \wedge \neg \mathtt{bad}]\end{aligned}$$

Proof. Let Π_i be the shortened protocol that halts after the ith output node. By Lemma 38 and Item 2 in Lemma 40 we can assume that Π is unrolled without loss of generality. For $i = 0$, the equality holds, because $\mathsf{Mon}(\Pi_0)$ does not send a message. For $i > 0$, we know that the equality of the probabilities for $\Omega(\Pi_{i-1}, \mathcal{A})_b$ holds by induction hypothesis. Let m_{left} and m_{right} be the terms the are sent at output node i. For m_{left} and m_{right}, let O_{left} and O_{right} be their respective output shapes. Lemma 35 shows that O_{left} and O_{right} are equal; hence, we denote them as O_b.

We stress that for each run the symbolic operations output by Construct-shape are unique w.r.t. the bitstrings that are sent or received for the following reason: for all symbolic operations O that use randomness, Construct-shape places $nonceof(\bar{O})$ as the symbolic operation for the randomness, where $\bar{O}$ is the respective dual symbolic operations of O. As a consequence, it cannot happen that Construct-shape assigns for two different bitstrings in one run the same symbolic operation (see Lemma 36).

We conduct a case distinction over O_b.

1. **Projections: $O_b = x_j$.** The projection x_j corresponds to the jth message that has been sent by the protocol; hence, this message refers to a former message. For all attacker strategies $Strat_{i,O_b}$ that lead to this case for O_b for the ith output node, we construct an reduction R against the induction hypothesis, i.e., that the probabilities for $\Omega(\Pi_{i-1}, \mathcal{A})_{left}$ and $\Omega(\Pi_{i-1}, \mathcal{A})_{right}$ are the same.

R forwards all messages from $H\text{-}Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_{i-1})-\mathrm{F}_b}$ to and from $\langle Sim'_f \mid \mathcal{A} \rangle$ until the first response after the $i-1$st output node. Then, R potentially responds with the out-metadata that the protocol would send for control nodes between the $i-1$st output node and the ith output node.[12] Then, R responds with jth protocol message to the attacker. Finally, upon a *guess* by the attacker, R also outputs *guess* as a distinguishing guess for $H\text{-}Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_{i-1})-\mathrm{F}_b}$.

Observe that since no alarm was raised, the branching monitor, in particular, did not raise an alarm. Thus, since by Corollary 5 Π is uniformity-enforcing, the control flow of both programs is the same. Moreover, if this case (with $O_b = x_j$) is reached with non-zero probability, there is a non-zero probability that x_j is the response that the protocol would have given as well. As a consequence, if the attacker succeeds with non-zero probability to distinguish $H\text{-}Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_b}$ in this case, i.e., for $O_b = x_j$, then R also succeeds with non-zero probability against $H\text{-}Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_{i-1})-\mathrm{F}_b}$.

2. O_b **is a message that is in the attacker-knowledge against** Π_{i-1}**.** Messages that are in the attacker-knowledge are messages such that there is an extended symbolic operation (see Definition 71) that is a shared test (see Definition 76) O with the following property: the symbolic operation that characterizes the message $eval_O(tr)$ is O_b (see the discussion about the uniqueness in Section 6.3.5.6 and Lemma 36). We distinguish two cases: first, the message m characterized by O_b is not only known through a protocol-generated ciphertext that uses an attacker-generated key; second, m is only known through such a ciphertext.

   Formally, in the first case O does not contain dec', i.e., there is a shared test O such that the symbolic operation that characterizes $eval_O(tr)$ is O_b we have that for all C such that $O = C[O']$ we have that $O' \neq dec'(O'')$. In the second case for all O such that the symbolic operation that characterizes $eval_O(tr)$ is O_b we have that there is a C such that $O = C[O']$ we have that $O' = dec'(O'')$.

   In the first case, the reduction to the induction hypothesis can evaluate O, i.e., compute $eval_O(tr)$, resulting in a projection to an input node after the jth output node, for $j \leq i-1$. Then, we construct a reduction against P_{i-1}, along the lines of the reduction from Case 1. Thus, the statement follows by the induction hypothesis.

   In the second case, the reduction cannot directly evaluate O because it would need to evaluate dec', which directly accesses a computation node. Recall that there is a C and a shared test O_c such that $O = C[dec'(O_c)]$. In particular, we know that in each run the bitstring c characterized by O_c is known by the attacker and is protocol-generated. In particular, by the construction of unrolled variants, we know that c is a visible sub-message of a message that has been sent to the attacker. Recall that in the unrolled variants sub-messages are always sent before their parent messages. By the construction of Construct-shape (in Line 16), we know that the symbolic operation that characterizes c is of the form $enc(O_1, O_2, O_3)$ with $O_2 \neq plaintextof(O'_2)$. Since m is a visible sub-message of c by the construction of unrolled variants (see Defintion 81), we know that then its plaintext m characterized by O_2 and by O_b has been sent in an earlier output node as well. In other words, there is a projection that points to the output node that output m. As a consequence,

[12] This is efficiently computable since R is specific for the protocol Π and the out-metadata for all control nodes is efficiently computable.

we can construct a reduction as in Case 1, and the statement follows by the induction hypothesis.

We stress that these cases in particular cover garbage message, garbage ciphertexts, garbage signatures, and attacker-generated keys. For attacker-generated ciphertexts and signatures, since we consider unrolled variants, all visible sub-messages do not lead to distinguishing symbolic operations by induction hypothesis. Hence, attacker-generated ciphertexts and signatures are covered by this case, as well. Consequently, we exclude such attacker-generated messages in the subsequent cases.

3. **The empty payload string:** $O_b = \mathit{empty}()$**.** The bitstring the corresponds to O_b is the same for $\mathrm{F}_{\mathit{left}}$ and $\mathrm{F}_{\mathit{right}}$. Along the lines of Case 1, the statement follows by the induction hypothesis.
4. **Protocol nonce:** $O_b = n_P$**.** This case cannot occur. A protocol nonce is always parsed as a projection.
5. **Protocol-generated encryption or verification key:** $O_b \in \{\mathit{ek}(\mathit{nonceof}(O_1)), \mathit{vk}(\mathit{nonceof}(O_1))\}$**.** For protocol-generated keys, equality of the symbolic operations O_{left} and O_{right} means that the corresponding bitstrings has the same distribution. Either the encryption key is fresh in the shared knowledge, i.e., never used before, then learning it (information theoretically) does not help distinguishing Π_{i-1}, or the encryption key was used earlier, then it already was in the shared knowledge for Π_{i-1} and along the lines of Case 1 we can show by induction hypothesis that the equality holds. The same argumentation holds for the case if O_b characterizes a verification key.
6. **Protocol-generated decryption key:** $O_b \in \{\mathit{dk}(\mathit{nonceof}(O_1)), \mathit{dkofek}(O_1)\}$**.** For protocol-generated keys, equality of the symbolic operations O_{left} and O_{right} means that the corresponding bitstrings have the same distribution. The decryption key can only be used to decryption ciphertexts with the corresponding encryption key. In an unrolled all plaintext messages that are in the symbolic knowledge after learning the decryption key (the message corresponding to) O_b are sent directly before (the message corresponding to) O_b is sent. Recall that in the unrolled variants sub-messages are always sent before their parent messages. Since we can consider an unbounded simulator and by Implementation Condition 31 there is exactly one decryption key for each encryption key, we can reduce this case to the induction hypothesis (along the lines of Case 1).
7. **Protocol-generated signing key:** $O_b \in \{\mathit{sk}(\mathit{nonceof}(O_1)), \mathit{skofvk}(O_1)\}$**.** For protocol-generated keys, equality of the symbolic operations O_{left} and O_{right} means that the corresponding bitstrings have the same distribution. If $O_b = \mathit{skofvk}(O_1)$, the equality follows from the induction hypothesis, i.e., because verification key is equally distributed, and because for every verification key there is exactly one signing key (Implementation Condition 32). If $O_b = \mathit{sk}(\mathit{nonceof}(O_1))$, then by the construction of Construct-shape, the signing key (characterized by O_b) was never used before. Hence, the equality follows from the fact that the corresponding bitstring have the same distribution.
8. **A protocol-generated, derived public-key:** $O_b \in \{\mathit{vkofsk}(O_1), \mathit{ekofdk}(O_1)\}$**.** For protocol-generated keys, the distribution of the bitstring that corresponds to O_b is equally distributed in $\Omega(\Pi_i, \mathcal{A})_{\mathit{left}}$ and $\Omega(\Pi_i, \mathcal{A})_{\mathit{right}}$. By induction hypothesis, the

statement follows. These are public keys and $O_b = O_{left} = O_{right}$. Hence, either these keys are already in the adversary's knowledge, or they have never been used anywhere yet. Consequently, with an argument along the lines of Case 1 the indistinguishability follows from the indistinguishability of Π_{i-1}.

9. **A virtual term:** $O_b = \mathit{plaintextof}(O_1, O_2)$**.** This case cannot occur since $\mathit{plaintextof}(O_1, O_2)$ only occurs inside an encryption symbolic operation.
10. **Encryption:** $O_b = \mathit{enc}(O_1, \mathit{plaintextof}(O_2, O_3), O_4)$**.** Observe that by the construction of CONSTRUCT-SHAPE this case only happens for honestly generated ciphertexts for which the decryption key has not been leaked. More precisely, if the decryption key is not in the approximation of the symbolic knowledge of the adversary that the monitor internally computes. By Lemma 39, we know that if the monitor does not raise an alarm then there is symbolically no distinguishing symbolic operation. Hence, if the key is not in the approximated symbolic knowledge, then the key is also not in the full symbolic knowledge (otherwise a counterexample to Lemma 39 can be constructed). If the decryption key is not in the symbolic knowledge, then by Lemma 1 from [BMU12] we know that the simulator (i.e., $\beta^\dagger$) never called $\texttt{getdk}_{\texttt{ch}}$. If in turn $\texttt{getdk}_{\texttt{ch}}$ was never called, then for computing a ciphertext with the respective encryption key only $\texttt{fakeenc}_{\texttt{ch}}(R, l)$ is used, where l equals the bitstring that corresponds to $\mathit{plaintextof}(O_2, O_3)$ and R is some internal register of the PROG-KDM challenger. In other words, the faking PROG-KDM challenger faked the encryption, i.e., it did not use the plaintext for constructing the ciphertext. Hence, by construction of the PROG-KDM challenger, the bitstrings corresponding to O_{left} and O_{right} are equally distributed. By induction hypothesis (for Π_{i-1}), we can conclude that the success probability for $\Omega(\Pi_i, \mathcal{A})_{left}$ is thus the same as for $\Omega(\Pi_i, \mathcal{A})_{right}$.
11. **Encryption:** $O_b = \mathit{enc}(O_1, O_2, O_3)$ **where** $O_2 \neq \mathit{plaintextof}(O_4, O_5)$**.** Since we consider unrolled variants (see Lemma 38), we know that with previous output nodes the bitstrings that correspond to O_1 and O_2 have been sent to the adversary. Recall that in the unrolled variants sub-messages are always sent before their parent messages. By induction hypothesis we know that for Π_{i-1} the statement holds. By the construction of CONSTRUCT-SHAPE, we know that the bitstring that corresponds to $\mathit{enc}(O_1, O_2, O_3)$ is an honestly generated ciphertext. As a consequence, Π_i did not leak more information that Π_{i-1} since the randomness of the encryption algorithm is chosen uniformly at random (Item 1 in Definition 70 and Item 4 in Section 6.3.2) Hence, the probabilities for $H\text{-}\mathit{Exec}_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_{left}}$ and $H\text{-}\mathit{Exec}_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_{right}}$ are equal.
12. **Signature:** $O_b \in \{\mathit{sig}(\mathit{sk}(O_1), O_2, O_3), \mathit{sig}(\mathit{skofvk}(O_1), O_2, O_3)\}$**.** Since we consider unrolled protocols, with in the last two output nodes the protocol already sent $\mathit{vkof}(O_b), O_2$ to the adversary. Recall that in the unrolled variants sub-messages are always sent before their parent messages. By induction hypothesis for Π_{i-1}, we know that with Π_{i-1} the distribution is the same as long as `bad` has not been raised. Assume that there is an adversary $\mathcal{A}$ that can distinguish $H\text{-}\mathit{Exec}_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_{left}}$ from $H\text{-}\mathit{Exec}_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_{right}}$ but not the same scenario for $i-1$. Information theoretically, if the two distributions that already send $\mathit{vkof}(O_b), O_2$ are the same, then also the two distributions that additionally send $\mathit{vkof}(O_b), O_2, O_b$ are the same.
13. **Garbage encryption or signature:** $O_b \in \{\mathit{garbageEnc}(O_1, O_2, l), \mathit{garbageSig}(O_1,$

$O_2, l)\}$**.** By the construction of CONSTRUCT-SHAPE and the protocol conditions, we know that the protocol cannot have produced a bitstring with a garbage encryption or garbage signature as an output shape. Hence, O_b characterizes a previously observed attacker-generated ciphertext or signature. Hence, the equality immediately follows from induction hypothesis, with a reduction along the lines of Case 1.

14. **Pairs:** $pair(O_1, O_2)$**.** Since we solely consider unrolled variants of protocols (see Lemma 38), we know that both O_1 and O_2 have already been sent in Π_{i-1}. Recall that in the unrolled variants sub-messages are always sent before their parent messages. Hence, the equality immediately follows from induction hypothesis, with a reduction along the lines of Case 1.
15. **Payload strings:** $O_b \in \{string_0(O_1), string_1(O_1)\}$**.** This case follows by induction hypothesis along the lines of Case 1, since Lemma 38 shows that it suffices to consider unrolled variants of protocols and O_b can be computed out of O_1.
16. **Length term:** $O_b = S(O_1)$**.** This case follows by induction hypothesis along the lines of Case 1, since Lemma 38 shows that it suffices to consider unrolled variants of protocols and O_b can be computed out of O_1.

□

Lemma 42. *For the honest simulator Sim and the faking simulator Sim_f (see Section 6.3.5.3), we have*

$$Sim \approx_{comp} Sim_f$$

Proof. As a corollary of Lemma 34, we know that if the execution interface is connected to a hybrid execution for some efficient randomness-safe CoSP protocol Π, the statement holds for all machines that are connected to the network interface of *Sim* or Sim_f. For distinguishers that are directly connected to the execution interface and behave differently, the only inputs that cause a distinguishing behavior are inputs of protocols that do not obey the protocol conditions, in particular if terms that are used in a randomness position, i.e., randomness messages, are additionally sent in clear. If such randomness messages would be sent in clear the distinguisher could, e.g., reconstruct an encryption and check whether this randomness message has been used (as in *Sim*), or an external oracle has been used to construct a known ciphertext (as in Sim_f). However, we constructed the honest simulator *Sim* and the faking simulator Sim_f such that it checks whether these protocol conditions are met (see the construction of the honest simulator in Section 6.3.5.3). If this check fails, the execution is aborted, both in *Sim* and in Sim_f. □

Lemma 43 ($f_{\texttt{bad-knowledge},\Pi}$ satisfies computational self-monitoring)**.** *The parametric CoSP protocol $f_{\texttt{bad-knowledge},\Pi}$ satisfies computational self-monitoring (see Definition 69).*

Proof. Recall that for computational self-monitoring, we have to show that if `bad-knowledge` in $\mathsf{Mon}(\Pi_i)$ occurs computationally with at most negligible probability, Π_{i-1} is computationally indistinguishable, and the ith node in Π_i is an output node, then Π_i is time-insensitively computationally indistinguishable:

$$Exec_{\mathsf{M},\mathrm{Impl},left(\Pi_i)} \approx_{tic} Exec_{\mathsf{M},\mathrm{Impl},right(\Pi_i)}$$

By assumption, we know that `bad-knowledge` occurs with at most negligible probability. Hence, the following indistinguishability relations hold (for $b \in \{left, right\}$):

$$\begin{array}{ll} & Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_b} \\ \overset{\text{Lemma 34 \& [BMU12]}}{\approx_{time}} & \langle H\text{-}Exec'_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_b} \mid Sim' \rangle \end{array}$$

By the computational soundness for trace properties (Theorem 9) of the symbolic model, Tr_{break} only happens with negligible probability. Furthermore, since `bad` only happens with negligible probability and $Exec_{\mathsf{M},\mathrm{Impl},left(\Pi_{i-1})}$ is indistinguishable from $Exec_{\mathsf{M},\mathrm{Impl},right(\Pi_{i-1})}$, Lemma 41 implies that $\langle H\text{-}Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_{left}} \mid Sim'_f \rangle$ is indistinguishable from $\langle H\text{-}Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_{right}} \mid Sim'_f \rangle$.

$$\langle H\text{-}Exec'_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_{left}} \mid Sim'_f \rangle \quad \approx_{tic} \quad \langle H\text{-}Exec'_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_{right}} \mid Sim'_f \rangle$$

By Lemma 42, we know that $Sim'_f \approx_{comp} Sim'$. Unruh has proven that if $\langle C \mid A \rangle \approx_{tic} \langle D \mid A \rangle$ and $A \approx_{comp} B$ holds and A, B have polynomially bounded running-time then $\langle C \mid B \rangle \approx_{tic} \langle D \mid B \rangle$ holds [Unr11, Lemma 22]. As a consequence, we get

$$\begin{array}{lrcl} & \langle H\text{-}Exec'_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_{left}} \mid Sim'_f \rangle & \approx_{tic} & \langle H\text{-}Exec'_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_{right}} \mid Sim'_f \rangle \\ \wedge & Sim'_f & \approx_{comp} & Sim' \\ \Longrightarrow & \langle H\text{-}Exec'_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_{left}} \mid Sim' \rangle & \approx_{tic} & \langle H\text{-}Exec'_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_{right}} \mid Sim' \rangle \end{array}$$

Unruh has, furthermore, proven that if $C \approx_{time} A$ and $D \approx_{time} B$ and $A \approx_{tic} B$, then $C \approx_{tic} D$ [Unr11, Corollary 13]. Thus, we conclude the proof as follows:

$$\begin{array}{lrcl} & Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_{left}} & \approx_{time} & \langle H\text{-}Exec'_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_{left}} \mid Sim' \rangle \\ \wedge & Exec_{\mathsf{M},\mathrm{Impl},,\mathsf{Mon}(\Pi_i)-\mathrm{F}_{right}} & \approx_{time} & \langle H\text{-}Exec'_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_{right}} \mid Sim' \rangle \\ \wedge & \langle H\text{-}Exec'_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_{left}} \mid Sim' \rangle & \approx_{tic} & \langle H\text{-}Exec'_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_{right}} \mid Sim' \rangle \\ \Longrightarrow & Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_{left}} & \approx_{tic} & Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi_i)-\mathrm{F}_{right}} \end{array}$$

With the application of Corollary 3, we conclude the proof:

$$\begin{array}{lrcl} & Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi)-\mathrm{F}_{left}} & \approx_{tic} & Exec_{\mathsf{M},\mathrm{Impl},\mathsf{Mon}(\Pi)-\mathrm{F}_{right}} \\ \Longrightarrow & Exec_{\mathsf{M},\mathrm{Impl},left(\Pi)} & \approx_{tic} & Exec_{\mathsf{M},\mathrm{Impl},right(\Pi)} \end{array}$$

□

6.3.6. CS for Uniform Bi-processes in the Applied π-calculus

Finally, we can prove that the symbolic model allows for self-monitoring.

Theorem 8. *Let* M *be the symbolic model from Section 6.3.1,* P *be the class of uniformity-enforcing of randomness-safe bi-protocols, and* Impl *an implementation that satisfies the conditions from Section 6.3.2. Then,* $\mathsf{M}, \mathrm{Impl}, \mathrm{P}$ *allow for self-monitoring. In particular, for*

each bi-protocol Π*,* $f_{\texttt{bad-knowledge},\Pi}$ *and* $f_{\texttt{bad-branch},\Pi}$ *as described above are distinguishing subprotocols (see Definition 69) for* M *and* P*.*

Proof. By construction of the self-monitors (see Definition 68) and the construction of the family of self-monitors $f_{\texttt{bad-knowledge},\Pi}$ and $f_{\texttt{bad-branch},\Pi}$ (see Section 6.3.4.1 and 6.3.5.1), and by inspection of the protocol conditions for the class P of uniformity-enforcing (see Definition 67) randomness-safe CoSP bi-protocols (see Definition 70), we can see that for all $\Pi \in \mathrm{P}$, $\mathsf{Mon}(\Pi) \in \mathrm{P}'$, where P' is the class of randomness-safe CoSP protocols (see Definition 70). By Lemma 30 and Lemma 32, we know that for any $\Pi \in \mathrm{P}$, each member of the family of branching monitors $f_{\texttt{bad-branch},\Pi}$ satisfies symbolic and computational self-monitoring. By Lemma 33 and Lemma 43, we know that for any $\Pi \in \mathrm{P}$, each member of the family of branching monitors $f_{\texttt{bad-knowledge},\Pi}$ satisfies symbolic and computational self-monitoring. □

Theorem 10. *Let* M *be the symbolic model from Section 6.3.1,* P *be the class of uniformity-enforcing of randomness-safe bi-protocols, and* Impl *an implementation that satisfies the conditions from Section 6.3.2. Then,* Impl *is computationally sound w.r.t. equivalence properties for* M *and* P*.*

Proof. By Theorem 9, we know that Impl is computationally sound for M the class of randomness-safe CoSP protocols. By Theorem 8, we know that $\mathsf{M}, \mathrm{Impl}, \mathrm{P}$ allow for self-monitoring. By Theorem 7, we can finally conclude that Impl is computationally sound w.r.t. equivalence properties for P. □

Combining our results, we conclude CS for protocols in the applied π-calculus that use signatures, public-key encryption, and corresponding length functions.

Theorem 11 (CS for Enc. and Signatures in the Applied π-calculus)**.** *Let* M *be as defined in Section 6.3. Let* Q *be a randomness-safe bi-process in the applied* π*-calculus, and let* $\mathbf{A}$ *of* $\mathbf{M}$ *be an implementation that satisfies the conditions from above. Let* e *be the embedding from bi-processes in the applied* π*-calculus to CoSP bi-protocols. If* Q *is uniform, then* $\mathit{left}(e(Q)) \approx_c \mathit{right}(e(Q))$*.*

Proof. The class of the embedding of the applied π-calculus is uniformity-enforcing by Lemma 1; thus, Theorem 10 entails the claim. □

6.4. Conclusion

In this work, we provided the first result that allows to leverage existing CS results for trace properties to CS results for uniformity of bi-processes in the applied π-calculus. Our result, which is formulated in an extension of the CoSP framework to equivalence properties, holds for Dolev-Yao models that fulfill the property that all distinguishing computational tests are expressible as a process on the model. We exemplified the usefulness of our method by applying it to a Dolev-Yao model that captures signatures and public-key encryption.

We moreover discussed how computationally sound, automated analyses can still be achieved in those frequent situations in which ProVerif does not manage to terminate

whenever the Dolev-Yao model supports a length function. We propose to combine ProVerif with the recently introduced tool APTE [CCP13].

Our results are formulated in the CoSP framework, which decouples the CS of Dolev-Yao models from the calculi, such as the applied π-calculus. We extended this framework with a notion of CS for equivalence properties, which might be of independent interest. Moreover, we proved the existence of an embedding from the applied π-calculus to CoSP that preserves uniformity of bi-processes, using a slight variation of the embedding of the original CoSP paper.

We leave as a future work to prove for more comprehensive Dolev-Yao models (e.g., for zero-knowledge proofs) the sufficient conditions for deducing from CS results for trace properties the CS of uniformity. Another interesting direction for future work is the extension of our result to observational equivalence properties that go beyond uniformity.

Part II.

Anonymous Communication Protocols

Provable Security & Efficient Key-Exchange

Chapter 7.

Provably Secure and Practical Onion Routing

[This chapter is based on a paper [BGKM12] with Michael Backes, Ian Goldberg, and Aniket Kate. I am the main contributor of all parts that occur in this chapter.]

7.1. Motivation

Over the last few years the onion routing (OR) network Tor [Tor03] has emerged as a successful technology for anonymous web browsing. It currently employs more than two thousand dedicated relays, and serves hundreds of thousands of users across the world. Its impact is also evident from the media coverage it has received over the last few years [Gre11]. Despite its success, the existing Tor network still lacks a rigorous security analysis, as its circuit construction as well as network transmission delays are found to be large [RG09; vS07], the current infrastructure is not scalable enough for the future users [MB09; MTHK09; PRR09], and from the cryptographic point of view its security properties have neither been formalized cryptographically nor proven. (See [CL05; FJS07b; MVV04] for previous attempts and their shortcomings.) In this paper, we define security for the third-generation OR protocol Tor, and construct a provably secure and practical OR protocol.

An OR network consists of a set of routers or OR nodes that relay traffic, a large set of users, and directory servers that provide routing information for the OR nodes to the users. A user (say Alice) constructs a *circuit* by choosing a small sequence of (usually three) OR nodes, where the chosen nodes route Alice's traffic over the path formed. The crucial property of an OR protocol is that a node in a circuit can determine no circuit nodes other than its predecessor and its successor. Alice sends data over the constructed circuit by sending the first OR node a message wrapped in multiple layers of symmetric encryption (one layer per node), called an *onion*, using symmetric keys agreed upon during an initial *circuit construction* phase. Consequently, given a public-key infrastructure (PKI), cryptographic challenges in onion routing are to securely agree upon such symmetric keys, and then to use the symmetric keys to achieve confidentiality and integrity.

In the first generation onion routing [RSG98], circuits are constructed in a single pass. However, the scalability issues while pursuing forward secrecy [DOW92] in the single-pass construction prompted Dingledine, Mathewson and Syverson [DMS04] to use a telescoping approach for the next-generation OR protocol Tor. In this telescoping approach, they employed a forward secret, *multi-pass* key agreement protocol called the Tor authentication protocol (TAP) to negotiate a symmetric session key between user Alice and a node. Here,

the node's public key is only used to initiate the construction, and the compromise of this public key does not invalidate the secrecy of the session keys once the randomness used in the protocol is erased.Goldberg [Gol06] presented a security proof for individual instances of TAP. The security of TAP, however, does not automatically imply the security of the Tor protocol. (For a possible concurrent execution attack, see [Zha09].) The Tor protocol constitutes a sequential execution of multiple TAP instances as well as onion construction and processing algorithms, and thus its security has to be analyzed in a composability setting.

In this direction, Camenisch and Lysyanskaya [CL05] defined an anonymous message transmission protocol in the universal composability (UC) framework, and presented a protocol construction that satisfies their definition. They motivated their choice of the UC framework for a security definition by its versatility as well as its appropriateness for capturing protocol compositions. However, Feigenbaum, Johnson and Syverson [FJS07b; FJS11] observe that the protocol definition presented by Camenisch and Lysyanskaya [CL05] does not correspond to the OR methodology, and a rigorous security analysis of an OR protocol still remains an unsolved problem.

Studies on OR anonymity such as [FJS07b; MVV04; Shm04] assume simplified OR black-box models to perform an analysis of the anonymity guarantees of these models. Due to the complexity of an OR network's interaction with the network and the adversary, such black-box models are not trivially realized by deployed OR networks, such as Tor. As a result, there is a gap between deployed OR protocols and anonymity analysis research that has to be filled.

Our Contributions. Our contribution is threefold. First, we present a security definition for the OR methodology as an ideal functionality $\mathcal{F}_{\text{OR}}$ in the UC framework. This ideal functionality in particular gives appropriate considerations to the goals of various system entities. After that, we identify and characterize which cryptographic primitives constitute central building blocks of onion routing, and we give corresponding security definitions: a one-way authenticated key exchange (1W-AKE) primitive, and onion construction and processing algorithms. We then describe an OR protocol Π_{OR} that follows the current Tor specification and that relies on these building blocks as black boxes. We finally show that Π_{OR} is secure in the UC framework with respect to $\mathcal{F}_{\text{OR}}$, provided that these building blocks are instantiated with secure realizations (according to their respective security definitions).

Second, we present a practical OR protocol by instantiating Π_{OR} with the following OR modules: a 1W-AKE protocol *ntor* [GSU12], employed onion construction and processing algorithms in Tor with a slightly enhanced integrity mechanism. We show that these instantiations fulfill the security definitions of the individual building blocks that we identified before. This yields the first practical and provably secure OR protocol that follows the Tor specification. As part of these proofs, we identify a novel security definition of symmetric encryption notion we show to be sufficient for showing Π_{OR} secure. This notion strictly lies between CPA-security and CCA-security and characterizes stateful deterministic countermode encryptions. We call this notion *predictably malleable encryptions*, which might be of an independent interest.

Third, we illustrate the applicability of the abstraction $\mathcal{F}_{\text{OR}}$ by introducing the first cryptographic definition of forward circuit secrecy for onion routing, which might be of independent interest. We utilize the abstraction $\mathcal{F}_{\text{OR}}$ and the UC composability theorem for proving that Π_{OR} satisfies forward circuit secrecy by means of a simple proof. As a

second application, we close the gap between the OR black-box model, prevalently used in anonymity analyses [FJS07b; FJS11; MVV04; Shm04], and a cryptographic model (Π_{OR}) of onion routing. Again, we utilize our abstraction $\mathcal{F}_{\text{OR}}$ and the UC composability theorem for proving that against local, static attackers the recent analysis of the OR black-box model [FJS11] also applies to our OR protocol Π_{OR} instantiated with secure core building blocks.

Compared to previous work [CL05], we construct an OR circuit interactively in multiple passes, whereas previous work did not consider circuit construction at all, and hence does not model the widely used Tor protocol. The previous approach, and even single-pass circuit construction in general, restricts the protocol to eventual forward secrecy, while a multi-pass circuit construction ensures forward secrecy immediately after the circuit is closed. Second, we show that their hop-to-hop integrity verification is not mandatory, and that an end-to-end integrity verification suffices for onion routing. Finally, they do not consider backward messages (from web-servers to Alice), and their onion wrapping and unwrapping algorithms also do not work in the backward direction.

There has also been work on universally composable Mix-Nets by Wikström [Wik04]. That work has some similarities to our work, but it only considers Mix-Nets, e.g., it does not need to cope with circuits and sessions.

Another important approach for analyzing onion routing has been conducted by Feigenbaum, Johnson, and Syverson [FJS07a]. In contrast to our work, the authors analyze an I/O automaton that use idealized encryption, pre-shared keys, and assume that every party only constructs one circuit to one destination. Moreover, the result in that work only holds in the stand-alone model against a local attackers whereas our result holds in the UC model against global and partially global attackers. In particular, by the UC composability theorem our result even holds with arbitrary protocols surrounding and against an attacker that controls parts of the network.

Outline of the Paper. Section 7.2 provides background information relevant to onion routing, 1W-AKE, and the UC framework. Section 7.3, presents our security definition for onion routing. Section 7.4, presents cryptographic definitions for predictably malleable encryptions and secure onion construction and processing algorithms. Section 7.5, states that given a set of secure OR modules we can construct a secure OR protocol. Section 7.7 utilizes our security definition to analyze some security and anonymity properties of onion routing. Finally, we discuss some further interesting directions in Section 7.8. In this work, many proofs have been omitted due to space constraints, which can be found in the full version [BGKM12].

7.2. Background

In this paper, we often omit the security parameter κ when calling an algorithm A; i.e., we abbreviate $A(1^\kappa, x)$ by $A(x)$. We write $y \leftarrow A(x)$ for the assignment of the result of $A(x)$ to a variable y, and we write $y \leftarrow S$ for the assignment of a uniformly chosen element from S to y. For a given security parameter κ, we assume a message space $M(\kappa)$ that is disjoint from the set of onions. We assume a distinguished error message $\perp$; in particular, $\perp$ is not in the message space. For some algorithms, we write $Alg(a, b, c, [d])$ and mean that the argument d is optional. Finally, for stateful algorithms, we write $y \leftarrow A(x)$ but we actually

mean $(y, s') \leftarrow A(x, s)$, where s' is used in the next invocation of A as a state, and s is the stored state from the previous invocation.We assume that for all algorithms $s \in \{0, 1^\kappa\}$. We abbreviate probabilistic polynomial-time as PPT.

7.2.1. Onion Routing Circuit Construction

In the original Onion Routing project [GRS96; GRS99; RSG98; STRL00], circuits were constructed in a single pass. However, such a single-pass circuit construction does not provide *forward secrecy*: if an adversary corrupts a node and obtains the private key, the adversary can decrypt all of the node's past communication. Although changing the public/private key pairs for all OR nodes after a predefined interval is a possible solution (*eventual* forward secrecy), this solution does not scale to realistic OR networks such as Tor, since at the start of each interval every user has to download a new set of public keys for all the nodes.

A user (Alice) chooses a path of OR nodes to a receiver, and creates a *forward onion* with several layers. Each onion layer is targeted at one node in the path and is encrypted with that node's public key. A layer contains that node's symmetric session key for the circuit, the next node in the path, and the next layer. Each node decrypts a layer using its secret key, stores the symmetric key, and forwards the next layer of the onion along to the next node. Once the last node in the path, i.e., the receiver, gets its symmetric session key, it responds with a confirmation message encrypted with its session key. Every node in the path wraps (encrypts) the *backward onion* using its session key in the reverse order, and the message finally reaches Alice. A circuit that is constructed in this way, i.e., the sequence of established session keys, is thereafter used for constructing and sending onions via this circuit.

There are attempts to solve this scalability issue. Kate, Zaverucha and Goldberg [KZG07] suggested the use of an identity-based cryptography (IBC) setting and defined a pairing-based onion routing (PB-OR) protocol. Catalano, Fiore and Gennaro [CFG09] suggested a certificateless cryptography (CLC) setting [AP03] instead, and defined two certificateless onion routing protocols (CL-OR and 2-CL-OR). However, both approaches do not yield satisfactory solutions: CL-OR and 2-CL-OR suffer from the same scalability issues as the original OR protocol [KG10b]; PB-OR requires a distributed private-key generator [KG10a] that may lead to inefficiency in practice.

Another problem with the single-pass approach is its intrinsic restriction to *eventual* forward secrecy [KZG10]; i.e., if the current private key is leaked, then past sessions remain secret only if their public and private keys have expired. A desirable property is that all past sessions that are closed remain secret even if the private key is leaked; such a property is called *immediate* forward secrecy.

In the current Tor protocol, circuits are constructed using a multi-pass approach that is based on TAP. The idea is to use the private key only for establishing a temporary session key in a key exchange protocol. Together with the private key, additional temporary (random) values are used for establishing the key such that knowing the private key does not suffice for reconstructing the session key. These temporary values are erased immediately after the session key has been computed. This technique achieves immediate forward secrecy in multi-pass constructions, which however was never formally defined or proven before.

The multi-pass approach incurs additional communication overhead. However, in practice, almost all Tor circuits are constructed for a circuit length of $\ell = 3$, which merely causes an overhead of six additional messages.[1] With this small overhead, the multi-pass circuit construction is the preferred choice in practice, due to its improved forward secrecy guarantees. Consequently, for our OR security definition we consider a multi-pass circuit construction as in Tor.

7.2.2. One-Way Authenticated Key Exchange – 1W-AKE

In a multi-pass circuit construction, a session key is established via a Diffie–Hellman key exchange. However, the precise properties required of this protocol were not formalized until recently. Goldberg, Stebila and Ustaoglu [GSU12] formalized the concept of 1W-AKE, presented an efficient instantiation, and described its utility towards onion routing. We review their work here and we refer the readers to [GSU12] for a detailed description.

An authenticated key exchange (AKE) protocol establishes an authenticated and confidential communication channel between two parties. Although AKE protocols in general aim for key secrecy and mutual authentication, there are many practical scenarios such as onion routing where mutual authentication is undesirable. In such scenarios, two parties establish a private shared session key, but only one party authenticates to the other. In fact, as in Tor, the unauthenticated party may even want to preserve its anonymity. Their 1W-AKE protocol constitutes this precise primitive.

The 1W-AKE protocol consists of three procedures: *Init*, *Resp*, and *CompKey*. With procedure *Init*, Alice (or her onion proxy) generates and sends an authentication challenge to the server (an OR node). The OR node responds to the challenge by running the *Resp* procedure, and returning the authentication response. The onion proxy (OP) then runs the *CompKey* procedure over the received response to authenticate the OR node and compute the session key.

The security of a 1W-AKE is defined by means of a *challenger* that represents all honest parties. The attacker is then allowed to query this challenger. If the attacker is not able to distinguish a fresh session key from a randomly chosen session key, we say that the 1W-AKE is *secure*. This challenger is constructed in a way that security of the 1W-AKE implies one-way authentication of the responding party.

For the definition of *one-way anonymity* we introduce a proxy, called the anonymity challenger, that relays all messages from and to the usual 1W-AKE challenger except for a challenge party C. The attacker can choose two challenge parties, out of which the anonymity challenger randomly picks one, say i^*. Then, the anonymity challenger relays all messages that are sent to C to P_{i^*} (via the 1W-AKE challenger).

In the one-way anonymity experiment, the adversary can issue the following queries to the challenger C. All other queries are simply relayed to the 1W-AKE challenger. The session Ψ^* denotes the challenge session. The two queries are for activation and communication during the test session. We say that a 1W-AKE is *one-way anonymous* if the attacker cannot guess which party has been guessed with more than $1/2 + \mu(\kappa)$ probability, where μ is a negligible function.

[1] The overhead reduces to four additional messages if we consider the "CREATE_FAST" option available in Tor.

In terms of instantiation, Goldberg et al. showed that an AKE protocol suggested for Tor—the fourth protocol in [vS07]—can be attacked, leading to an adversary determining all of the user's session keys. They then fixed the protocol (see Figure 7.17) and proved that the fixed protocol (*ntor*) satisfies the formal properties of 1W-AKE. In our OR analysis, we use their formal definition and their fixed protocol.

7.2.3. Generalized UC Framework

As presented in Section 4.4.1.1, the UC framework is designed to enable a modular analysis of security protocols. In this framework, the security of a protocol is defined by comparing it with a setting in which all parties have a direct and private connection to a trusted machine that computes the desired functionality. In this chapter we omit the port-based notation.

In contrast to typical UC proofs, our attacker model considers a more fine-grained network topology. Typically, a global attacker is assumed in UC; however, as we also want to be able to argue about local attackers, we prove our result for partially global attackers, i.e., in particular also for completely global attackers. A network over which the attacker does not have full control is modelled by a network functionality $\mathcal{F}_{\text{NET}^q}$ in which the attacker can adaptively compromise up to q links between honest onion routers. This network functionality is a global setup assumption; consequently, we have to consider the generalized UC framework (GUC) by Canetti, Dodis, Pass, and Walfish [CDPW07].[2] Throughout this chapter, if we say that a protocol ρ UC realizes a protocol π we actually mean that ρ *GUC* realizes π. (For a thorough definition of GUC, we refer to [CDPW07].)

7.2.4. The OR Protocol

We describe an OR protocol Π_{OR} that follows the Tor specification [DM08]. We do not present the cryptographic algorithms, e.g., wrapping and unwrapping onions, in this section but only present the skeleton of the protocol. A thorough characterization of these cryptographic algorithms follows in Section 7.4.

We describe our protocols using pseudocode and assume that a node maintains a state for every execution and responds (changes the state and/or sends a message) upon receiving a message as per its current state. In Figure 7.1, we give an overview of the setup that we consider.

As an attacker model we consider a partially global attacker in contrast to the global attacker that is typically used in UC analyses. For modelling a partially global attacker, we introduce an ideal functionality $\mathcal{F}_{\text{NET}^q}$ that allows the attacker to compromise at most q links.

There are two types of messages that the protocol generates and processes: the first type contains *input actions*, which carry inputs to the protocol from the user (Alice), and *output actions*, which carry outputs of the protocol to Alice. The second message type is a point-to-point *network message* (a cell in the OR literature), which is to be delivered by one protocol node to another. To enter a wait state, a thread may execute a command of the form **wait for** a network message.

[2]The authors show that composability also holds true in the presence of global functionalities as long as

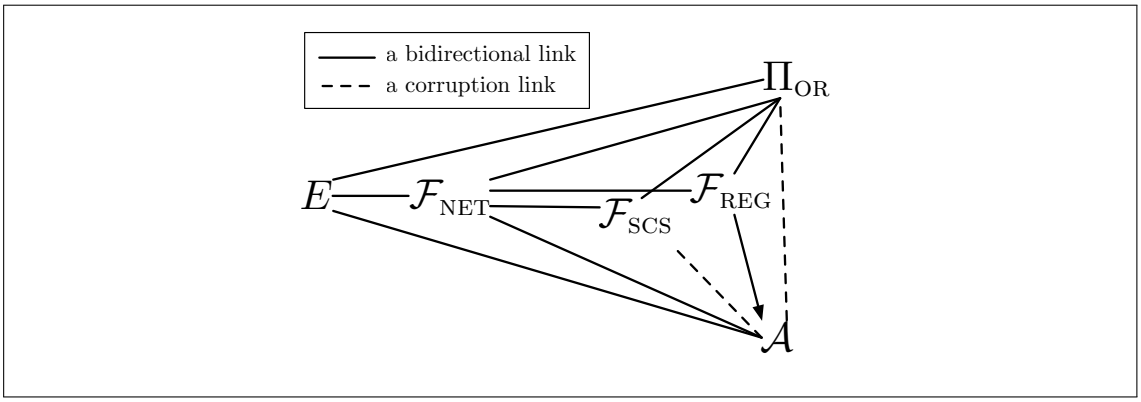

Figure 7.1.: Overview of the set-up

upon an input (setup):
Generate an asymmetric key pair $(sk_P, pk_P) \leftarrow G(1^\eta)$.
send a cell (register, P, pk) to the $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ functionality
wait for a cell (registered, $\langle P_j, pk_j \rangle_{j=1}^n$) from $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$
output (ready, $\mathcal{N} = \langle P_j \rangle_{j=1}^n$)

upon an input (createcircuit, $\mathcal{P} = \langle P, \langle P_j \rangle_{j=1}^\ell \rangle$):
store $\mathcal{P}$ and $\mathcal{C} \leftarrow \langle P \rangle$; call *ExtendCircuit*$(\mathcal{P}, \mathcal{C})$

upon an input (send, $\mathcal{C} = \langle P \overset{cid_1}{\Longleftrightarrow} P_1 \Longleftrightarrow \cdots P_\ell \rangle, m$):
if $(m \notin M(\kappa))$ **then** abort
if $Used(cid_1) < ttl_C$ **then**
look up the keys $(\langle k_j \rangle_{j=1}^\ell)$ for cid_1
$O \leftarrow WrOn(m, (k_j)_{j=1}^\ell)$; $Used(cid_1)$++
send a cell (cid_1, relay, O) to P_1 over $\mathcal{F}_{\text{SCS}}$
else
call *DestroyCircuit*$(\mathcal{C}, cid_1)$; output (destroyed, $\mathcal{C}, m$)

Figure 7.2.: Π_{OR}: The OR Protocol for Party P – Input/Output Commands

With this methodology, we are able to effortlessly extract an OR protocol (Π_{OR}) from the Tor specification by categorizing actions based on the OR cell types (see Figure 7.2). For ease of exposition, we only consider Tor cells that are cryptographically important and relevant from the security definitional perspective. In particular, we consider create, created and destroy cells among control cells, and data, extend and extended cells among relay cells. We also include two input messages createcircuit and send, where Alice uses createcircuit to create OR circuits and uses send to send messages m over already-created circuits. We do not consider streams and the SOCKS interface in Tor as they are extraneous to the basic OR methodology. We unify instructions for an OP (onion proxy) node and an OR node for the simplicity of discussion. Moreover, for the sake of brevity, we restrict ourselves to messages $m \in M(\kappa)$ that fit exactly in one cell. It is straight-forward to extend our result to a protocol that accepts larger messages. The only difference is that the onion proxy and the exit node divide messages into smaller pieces and recombine them in an appropriate way.

Function calls *Init*, *Resp* and *CompKey* correspond to 1W-AKE function calls described in Section 7.2.2. Function calls *WrOn* and *UnwrOn* correspond to the principal onion algorithms. *WrOn* creates a layered encryption of a payload (plaintext or onion) for given an ordered list of ℓ session keys for $\ell \geq 1$. *UnwrOn* removes ℓ layers of encryptions from an onion to output a plaintext or an onion given an input onion and a ordered list of ℓ

the environment has access to these functionalities, i.e., they are not simulated by the simulator.

upon receiving a cell $(cid, \mathsf{create}, X)$ from P_i over $\mathcal{F}_{\text{SCS}}$:
  $\langle Y, k_{\mathsf{new}} \rangle \leftarrow Resp(pk_P, sk_P, X)$
  store $\mathcal{C} \leftarrow \langle P_i \overset{cid,k_{\mathsf{new}}}{\Longleftrightarrow} P \rangle$
  send a cell $(cid, \mathsf{created}, Y, t)$ to P_i over $\mathcal{F}_{\text{SCS}}$
upon receiving a cell $(cid, \mathsf{created}, Y, t)$ from P_i over $\mathcal{F}_{\text{SCS}}$:
  if $prev(cid) = (P', cid', k')$ **then**
    $O \leftarrow WrOn(\langle \mathsf{extended}, Y, t \rangle, k')$
    send a cell $(cid', \mathsf{relay}, O)$ to P' over $\mathcal{F}_{\text{SCS}}$
  else if $prev(cid) = \perp$ **then**
    $k_{\mathsf{new}} \leftarrow CompKey(pk_i, Y, t)$
    update $\mathcal{C}$ with k_{new}; call $ExtendCircuit(\mathcal{P}, \mathcal{C})$
upon receiving a cell (cid, relay, O) from P_i over $\mathcal{F}_{\text{SCS}}$:
  if $prev(cid) = \perp$ **then**
    if $getkey(cid) = (k_j)_{j=1}^{\ell'}$ **then**
      (type, m) **or** $O \leftarrow UnwrOn(O, (k_j)_{j=1}^{\ell'})$
      (P', cid') **or** $\perp \leftarrow next(cid)$
  else if $prev(cid) = (P', cid', k')$ **then**
    $O \leftarrow WrOn(O, k')$ /* a backward onion */
  switch (type)
  case extend:
    get $\langle P_{next}, X \rangle$ from m; $cid_{next} \leftarrow \{0,1\}^\kappa$
    update $\mathcal{C} \leftarrow \langle P_i \overset{cid,k}{\Longleftrightarrow} P \overset{cid_{next}}{\Longleftrightarrow} P_{next} \rangle$
    send a cell $(cid_{next}, \mathsf{create}, X)$ to P_{next} over $\mathcal{F}_{\text{SCS}}$
  case extended:
    get $\langle Y, t \rangle$ from m; get P_{ex} from $(\mathcal{C}, \mathcal{P})$
    $k_{\mathsf{ex}} \leftarrow CompKey(pk_{\mathsf{ex}}, Y, t)$
    update $\mathcal{C}$ with (k_{ex}); call $ExtendCircuit(\mathcal{P}, \mathcal{C})$
  case data:
    if (P = OP) **then** output $(\mathsf{received}, \mathcal{C}, m)$
    else if $m = (S, m')$
      generate or lookup the unique sid for cid
      send (P, S, sid, m') to $\mathcal{F}_{\text{NET}^q}$
  case *corrupted*: /*corrupted onion*/
    call $DestroyCircuit(\mathcal{C}, cid)$
  case default: /*encrypted forward/backward onion*/
    send a cell $(cid', \mathsf{relay}, O)$ to P' over $\mathcal{F}_{\text{SCS}}$
upon receiving a msg (sid, m) from $\mathcal{F}_{\text{NET}^q}$:
  get $\mathcal{C} \leftarrow \langle P' \overset{cid,k}{\Longleftrightarrow} P \rangle$ for sid; $O \leftarrow WrOn(m, k)$
  send a cell (cid, relay, O) to P' over $\mathcal{F}_{\text{SCS}}$
upon receiving a cell $(cid, \mathsf{destroy})$ from P_i over $\mathcal{F}_{\text{SCS}}$:
  call $DestroyCircuit(\mathcal{C}, cid)$

Figure 7.3.: Π_{OR}: The OR Protocol for Party P – Network Commands

session keys for $\ell \geq 1$. Moreover, the onion algorithm *UnwrOn* also ensures end-to-end integrity. Along with the plaintext of an onion *UnwrOn* outputs a flag type that indicates whether an onion has been corrupted (type = *corrupted*) or has integrity (type = default). The cryptographic requirements for these onion algorithms are presented in Section 7.4.2.

Tor uses a centralized approach to determine valid OR nodes and distribute their public keys. Every OR node has to be registered in so-called directory servers, where each registration is checked by an administrator. These directory servers then distribute the list of valid OR nodes and the respective public keys. We abstract the key registration procedure by assuming that the directory servers expect a fixed set of parties upon setup. Formally, we model these directory servers as an ideal functionality $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$, which is basically defined as by Canetti [Can01] except that $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ rejects all parties that are not in $\mathcal{N}$ and only sends the public keys around once all parties in $\mathcal{N}$ registered.[3] Tor does not guarantee any anonymity once these directory servers are compromised. Therefore, we concentrate on the case in which these directory servers cannot be compromised.[4] As in Tor, we assume that the list of valid OR nodes is given to the directory servers from outside, in our case from the environment. However, for the sake of simplicity we assume that the OR list is only synchronized initially. In detail, we slightly extend the functionality as follows. $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ initially receives a list of OR nodes from the environment, waits for each of these parties for a public key, and distributes the list of OR nodes and their public keys as $(\mathsf{registered}, \langle P_j, pk_j \rangle_{j=1}^n)$. Each OR node P, on the other hand, initially computes its long-term keys (sk_P, pk_P) and registers the public part at $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$. Then, the node waits to

[3] Technically, we also extend $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ such that upon each $(\mathsf{register}, sid, v)$-message, $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ notifies the attacker. And only after the attacker confirmed this message, $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ registers v with P.

[4] Formally, this ideal functionality $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ does not accept `compromise`-requests from the attacker.

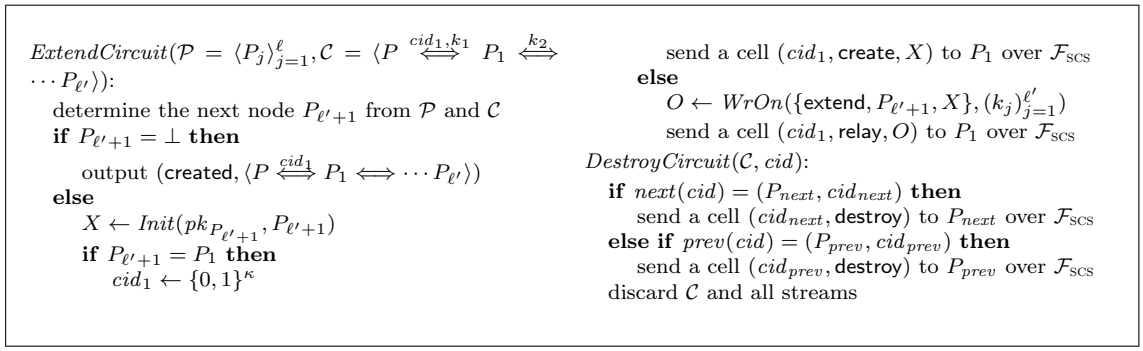

Figure 7.4.: Subroutines of Π_{OR} for Party P

receive the message (registered, $\langle P_j, pk_j \rangle_{j=1}^{n}$) from $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ before declaring that it is ready for use.[5]

OPs develop circuits incrementally, one hop at a time, using the *ExtendCircuit* function defined in Figure 7.4. To create a new circuit, an OP sends a create cell to the first node, after calling the *Init* function of 1W-AKE; the first node responds with a created cell after running the *Resp* function. The OP then runs the *CompKey* function. To extend a circuit past the first node, the OP sends an extend relay cell after calling the *Init* function, which instructs the last node in the circuit to send a create cell to extend the circuit.

Circuits are identified by circuit IDs ($cid \in \{0,1\}^{\kappa}$) that associate two consecutive circuit nodes. We denote circuit at a node P_i using the terminology $\mathcal{C} = P_{i-1} \overset{cid_i,k_i}{\Longleftrightarrow} P_i \overset{cid_{i+1}}{\Longleftrightarrow} P_{i+1}$, which says that P_{i-1} and P_{i+1} are respectively the predecessor and successor of P_i in a circuit $\mathcal{C}$. k_i is a session key between P_i and the OP, while the absence of k_{i+1} indicates that a session key between P_{i+1} and the OP is not known to P_i; analogously the absence of a circuit id cid in that notation means that only the first circuit id is known, as for OP, for example. Functions *prev* and *next* on cid correspondingly return information about the predecessor or successor of the current node with respect to cid; e.g., $next(cid_i)$ returns (P_{i+1}, cid_{i+1}) and $next(cid_{i+1})$ returns $\perp$. The OP passes on to Alice $\langle P \overset{cid_1}{\Longleftrightarrow} P_1 \Longleftrightarrow \cdots P_{\ell} \rangle$.

Within a circuit, the OP and the exit node use relay cells created using *WrOn* to tunnel end-to-end commands and connections. The exit nodes use some additional mechanisms (abstracting the streams used in Tor) to synchronize communication between the network and a circuit $\mathcal{C}$. We represent that using sid. With this auxiliary synchronization, end-to-end communication between OP and the exit node happens with a *WrOn* call with multiple session keys and a series of *UnwrOn* calls with individual session keys in the forward direction, and a series of *WrOn* calls with individual session keys, and finally a *UnwrOn* call with multiple session keys in the backward direction. Communication in the forward direction is initiated by a send message by Alice to the OP, while communication in the backward direction is initiated by a network message to the exit node. Cells are exchanged between OR nodes over a secure and authenticated channels, e.g., a TLS connection. We abstract such a channel in the UC framework by a functionality $\mathcal{F}_{\text{SCS}}$ as proposed by Canetti [Can01] with the only difference that $\mathcal{F}_{\text{SCS}}$ does not output the leakage

[5] The functionality $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ additionally answers upon a request `retrieve` with the full list of participants $\langle P_j, pk_j \rangle_{j=1}^{n}$.

upon receiving a msg $(\texttt{compromise}, \mathsf{N_A})$ from $\mathbf{A}$:
set $compromised(P) \leftarrow true$ **for** every $P \in \mathsf{N_A}$
set $b \leftarrow \frac{|\mathsf{N_A}|}{|\mathsf{N_{OR}}|}$
upon an input (send, S) from the environment for party U:
with probability b^2, send (sent, U, S) to $\mathbf{A}$
with probability $(1-b)b$, send $(\mathsf{sent}, -, S)$ to $\mathbf{A}$
with probability $b(1-b)$, send $(\mathsf{sent}, U, -)$ to $\mathbf{A}$
with probability $(1-b)^2$, send $(\mathsf{sent}, -, -)$ to $\mathbf{A}$

Figure 7.5.: Black-box OR Functionality $\mathcal{B}_{\text{OR}}$ [FJS11]

to the attacker but to $\mathcal{F}_{\text{NET}^q}$, i.e., the network functionality.[6] We write $storeX \leftarrow v$ for either introducing a new variable X with value v, or assign a value v to a variable X in case X was not previously defined.

To tear down a circuit completely, an OR or OP sends a destroy cell to the adjacent nodes on that circuit with appropriate *cid* using the *DestroyCircuit* function defined in Figure 7.4. Upon receiving an outgoing destroy cell, a node frees resources associated with the corresponding circuit. If it is not the end of the circuit, it sends a destroy cell to the next node in the circuit. Once a destroy cell has been processed, the node ignores all cells for the corresponding circuit. Note that if an integrity check fails during *UnwrOn*, the destroy cells are sent in the forward and backward directions in a similar way.

In the Tor the OP has a time limit (of ten minutes) for each established circuit; thereafter, the OP constructs a new circuit. However, the UC framework does not provide a notion of time. We model such a time limit in the UC framework by only allowing a circuit to transport at most a constant number (say ttl_C) of messages measured using the *used* function call. Afterwards, the OP discards the circuit and establishes a fresh circuit.

7.2.5. An OR Black Box Model

Anonymity in a low-latency OR network does not only depend upon the security of the onions but also upon the magnitudes and distributions of users and their destination servers. In the OR literature, considerable efforts have been put towards measuring the anonymity of onion routing [FJS07a; FJS07b; FJS11; MVV04; Shm04].

Feigenbaum, Johnson, and Syverson used for an analysis of the anonymity properties of onion routing an ideal functionality $\mathcal{B}_{\text{OR}}$ [FJS11]. This functionality emulates an I/O-automata model for onion routing from [FJS07a; FJS07b]. Figure 7.5 presents this functionality $\mathcal{B}_{\text{OR}}$.

Let $\mathsf{N_{OR}}$ be the set of onion routers, and let $\mathsf{N_A}$ of those be eavesdropped, where $b = |\mathsf{N_A}|/|\mathsf{N_{OR}}|$ defines the fraction of compromised nodes. It takes as input from each user U the identity of a destination S. For every such connection between a user and a destination, the functionality may reveal to the adversary the identity of the user $(\mathsf{sent}, U, -)$ (i.e., the first OR router is compromised), the identity of the destination $(\mathsf{sent}, -, S, [m])$ (i.e., the exit node is compromised), both $(\mathsf{sent}, U, S, [m])$ (i.e., the first OR router and the exit node are compromised) or only a notification that something has been sent $(\mathsf{sent}, -, -)$ (i.e., neither the first OR router nor the exit node is compromised).

We stress that this functionality only abstracts an OR network against local attackers.

[6] As leakage function l for $\mathcal{F}_{\text{SCS}}$, we choose $l(m) := |m|$.

As the distribution of the four cases only depends on the first and the last router being compromised but not on the probability that the attacker controls sensitive links between honest parties, $\mathcal{B}_{\text{OR}}$ only models OR against local adversaries. As an example consider, the case in which the attacker only wiretaps the connection between the exit node and the server. In this case, the attacker is able to determine which message has been sent to whom, i.e., the abstraction needs to leak $(\mathsf{sent}, -, S, [m])$; however, the probability of this event is c, where c is the fraction of observed links between honest onion routers and users and servers. Therefore, $\mathcal{B}_{\text{OR}}$ cannot be used as an abstraction for onion routing against partially global attackers. In Section 7.7.1.2, we present an extension of $\mathcal{B}_{\text{OR}}$ that models onion routing against partially global attackers and prove that it constitutes a sound abstraction.

We actually present $\mathcal{B}_{\text{OR}}$ in two variants. In the first variant $\mathcal{B}_{\text{OR}}$ does not send an actual message but only a notification. This variant has been analyzed by Feigenbaum, Johnson, and Syverson. We additionally consider the variant in which $\mathcal{B}_{\text{OR}}$ sends a proper message m. We denote these two variants by marking the message m as optional, i.e., as $[m]$.

In order to justify these OR anonymity analyses that consider an OR network as a black box, it is important to ascertain that these black boxes indeed model onion routing. In particular, it is important under which adversary and network assumptions these black boxes model and securely abstract real-world OR networks. In this work, we show that the black box $\mathcal{B}_{\text{OR}}$ can be UC-realized by a simplified version of the Tor network.

7.3. Security Definition of OR

In this section, we first describe our system and adversary model for all protocols that we analyze (Section 7.3.1). Thereafter, we present a composable security definition of OR by introducing an ideal functionality (abstraction) $\mathcal{F}_{\text{OR}}$ in the UC framework (Section 7.3.2).

Tor was designed to guarantee anonymity even against partially global attackers, i.e., attackers that do not only control compromised OR nodes but also a portion of the network. Previous work, however, only analyzed local, static attackers [FJS07a; FJS07b; FJS11], such as the abstraction $\mathcal{B}_{\text{OR}}$ presented in Figure 7.5. In contrast, we analyze onion routing against partially global, active attackers. As our resulting abstraction $\mathcal{F}_{\text{OR}}$ has to faithfully reflect that an active attacker can hold back all onions that it observes, $\mathcal{F}_{\text{OR}}$ is naturally more complex than $\mathcal{B}_{\text{OR}}$.

7.3.1. System and Adversary Model

We consider a fully connected network of n parties $\mathsf{N} = \{P_1, \ldots, P_n\}$. For simplicity of presentation, we consider all parties to be OR nodes that also can function as OPs to create circuits and send messages. It is also possible to use our formulation to model separate user OPs that only send and receive messages but do not relay onions.

Tor has not been designed to resist against global attackers. Such an attacker is too strong for many practical purposes as it can simply break the anonymity of an OR protocol by holding back all but one onion and tracing that one onion though the network. However, in contrast to previous work, we do not only consider local attackers, which do not control more than the compromised OR routers, but also partially global attackers that control a certain portion of the network. Analogous to the network functionality $\mathcal{F}_{\text{SYN}}$ proposed

upon an input (setup):
  draw a fresh handle h; set registered_flag $\leftarrow$ *true*
  store $lookup(h) \leftarrow (\mathsf{dir}, \mathsf{registered}, \mathcal{N})$
  send $(h, \mathsf{register}, P)$ to **A**
  wait for the message $(h, \mathsf{register}, P)$ from **A**
  output $(\mathsf{ready}, \mathcal{N})$

upon an input $(\mathsf{createcircuit}, \langle P, Q_1, \ldots, Q_\ell \rangle)$:
  call $ExtendCircuit(\langle P, Q_1, \ldots, Q_\ell \rangle, (P))$

upon an input $(\mathsf{send}, \langle P \overset{cid_1}{\Longleftrightarrow} Q_1 \Longleftrightarrow \cdots Q_\ell \rangle, m)$:
  if $(m \notin M(\kappa))$ **then** abort
  if $Used(cid_1) < ttl_C$ **then**
    $Used(cid_1)$++
    $SendMessage(Q_1, cid_1, \mathsf{relay}, \langle data, m \rangle)$
  else
    $DestroyCircuit(\langle P \overset{cid_1}{\Longleftrightarrow} Q_1 \Longleftrightarrow \cdots Q_\ell \rangle, cid_1)$
    output $(\mathsf{destroyed}, \langle P \overset{cid_1}{\Longleftrightarrow} Q_1 \Longleftrightarrow \cdots Q_\ell \rangle, m)$

Figure 7.6.: The ideal functionality $\mathcal{F}_{\text{OR}}^{\mathcal{N}}$ (short $\mathcal{F}_{\text{OR}}$) for Party P – Input/Output Messages

by Canetti [Can01], we model the network as an ideal functionality $\mathcal{F}_{\text{NET}^q}$, which bounds the number of attacker-controlled links to $q \in [0, \binom{n}{2}]$. For attacker-controlled links the messages are forwarded to the attacker; otherwise, they are directly delivered. In Section 7.7 we show that previous black-box analyses of onion routing against local attackers applies to our setting as well by choosing $q := 0$. Let S represent all possible destination servers $\{S_1, \ldots, S_\Delta\}$ which reside in the network abstracted by a network functionality $\mathcal{F}_{\text{NET}^q}$.

We stress that the UC framework does not provide a notion of time; hence, the analysis of timing attacks, such as traffic analysis, is not in the scope of this work.

Adaptive Corruptions. Forward secrecy [DOW92] is an important property for onion routing. In order to analyze this property, we allow adaptive corruptions of nodes by the attacker **A**. Such an adaptive corruption is formalized by a message `compromise`, which is sent to the respective party. Upon such a `compromise` message the internal state of that party is deleted and a long-term secret key *sk* for the node is revealed to the attacker. **A** can then impersonate the node in the future; however, **A** cannot obtain the information about its ongoing sessions. We note that this restriction arises due to the currently available security proof techniques and the well-known selective opening problem with symmetric encryptions [Hof11], and the restriction is not specific to our constructions [BMP00; GL01]. We could also restrict ourselves to a static adversary as in previous work [CL05]; however, that would make an analysis of forward secrecy impossible.

7.3.2. Ideal Functionality

The presentation of the ideal functionality $\mathcal{F}_{\text{OR}}$ is along the lines of the description OR protocol Π_{OR} from Section 7.2.4. We continue to use the message-based state transitions from Π_{OR}, and consider sub-machines for all n nodes in the ideal functionality. To communicate with each other through messages and data structures, these sub-machines share a memory space in the functionality. The sub-machine pseudocode for the ideal functionality appears in Figure 7.6 and three subroutines are defined in Figure 7.8. As the similarity between pseudocodes for the OR protocol and the ideal functionality is obvious, rather than explaining the OR message flows again, we concentrate on the differences.

The only major difference between Π_{OR} and $\mathcal{F}_{\text{OR}}$ is that cryptographic primitives such as message wrapping, unwrapping, and key exchange are absent in the ideal world; we do not have any keys in $\mathcal{F}_{\text{OR}}$, and the OR messages *WrOn* and *UnwrOn* as well as the 1W-AKE messages *Init*, *Resp*, and *CompKey* are absent.

upon receiving $\langle Q_0, \ldots, Q_u, \_, h\rangle$ from $\mathcal{F}_{\text{NET}^q}$:
send $(msg) \leftarrow lookup(h)$ to a receiving submachine Q_u inside $\mathcal{F}_{\text{OR}}^{\mathcal{N}}$
upon receiving $(Q, cid, \mathsf{create})$ through a handle:
$SendMessage(Q, cid, \mathsf{created})$
upon receiving $(Q, cid, \mathsf{created})$ through a handle:
if $prev(cid) = (Q', cid')$ **then**
$SendMessage(Q', cid', \mathsf{relay}, \mathsf{extended})$
else if $prev(cid) = \bot$ **then**
$ExtendCircuit(\mathcal{P}, circuit(cid))$
upon receiving $(Q, cid, \mathsf{relay}, O)$ through a handle:
switch $(O, prev(cid), next(cid))$:
case $((\mathsf{extend}, Q_{next}), (Q, \_), \bot)$:
$cid_{next} \leftarrow \{0,1\}^{\kappa}$;
$next(cid) \leftarrow (Q_{next}, cid_{next})$
$prev(cid_{next}) \leftarrow (P, cid)$
$SendMessage(Q_{next}, cid_{next}, \mathsf{create})$
case $((\mathsf{extended}, \_), \_, (Q, \bot))$:
$ExtendCircuit(\mathcal{P}, circuit(cid))$
case $((data, m), (P, \bot), (Q, \_))$: /*message for the onion proxy*/
output $(\mathsf{received}, circuit(cid), m)$
case $((data, (S, m')), (Q, \_), (P, \bot))$: /*message for the server*/
if $(SID(cid) = \bot)$ draw a fresh sid; $SID(cid) \leftarrow sid$
else $sid \leftarrow SID(cid)$
set $m'' \leftarrow (sid, m')$; send (P, S, m'') to $\mathcal{F}_{\text{NET}^q}$
case $(\_, (Q, \_), (P, cid_{next}))$: /*encrypted forward onion*/
$(Q_{next}, \_) \leftarrow next(cid_{next})$ /*retrieve the next node in the circuit*/
$SendMessage(Q_{next}, cid_{next}, \mathsf{relay}, O)$
case $(\_, (P, cid_{prev}), (Q, \_))$: /*encrypted backward onion*/
$(Q_{prev}, \_) \leftarrow prev(cid_{prev})$ /*retrieve the previous node in the circuit*/
$SendMessage(Q_{prev}, cid_{prev}, \mathsf{relay}, O)$
case $((corrupted, \_), \_, \_)$: /*corrupted onion*/
$DestroyCircuit(\mathcal{C}, cid)$
upon receiving a msg (sid, m) from $\mathcal{F}_{\text{NET}^q}$:
obtain $\mathcal{C} = \langle P' \overset{cid}{\Longleftrightarrow} P\rangle$ for sid
$SendMessage(P', cid, \mathsf{relay}, \langle \mathsf{data}, m\rangle)$
upon receiving a msg $(P_i, cid, \mathsf{destroy})$ through a handle:
$DestroyCircuit(\mathcal{C}, cid)$
upon receiving a msg $(P_i, P, h, [\mathsf{corrupt}, T(\cdot)])$ from $\mathbf{A}$:
$(message) \leftarrow lookup(h)$
if $\mathsf{corrupt} = true$ **then**
$message \leftarrow T(msg)$;
set $corrupted(message) \leftarrow true$
process $message$ as if the receiving submachine was P
upon receiving $(\mathtt{compromise}, P)$ from $\mathbf{A}$:
set $compromised(P) \leftarrow true$
delete all local information at P

Figure 7.7.: The ideal functionality $\mathcal{F}_{\text{OR}}^{\mathcal{N}}$ (short $\mathcal{F}_{\text{OR}}$) for Party P – Network Messages, where the function $circuit(cid)$ is defined in Figure 7.8

The ideal functionality also abstracts the directory server and expects on the input/output interface of $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ (from the setting with Π_{OR}) an initial message with the list $\langle P_i\rangle_{i=1}^{n}$ of valid nodes. This initial message corresponds to the list of onion routers that have been approved by an administrator. We call the part of $\mathcal{F}_{\text{OR}}$ that abstracts the directory servers dir. For the sake of brevity, we do not present the pseudocode of dir. Upon an initial message with a list $\langle P_i\rangle_{i=1}^{n}$ of valid nodes, dir waits for all nodes P_i ($i \in \{1, \ldots, n\}$) for a message $(\mathsf{register}, P_i)$. Once all nodes registered, dir sends a message $(\mathsf{registered}, \langle P_i\rangle_{i=1}^{n})$ with a list of valid and registered nodes to every party that registered, and to every party that sends a `retrieve` message to dir.

Using $\mathcal{F}_{\text{NET}^q}$ with the ideal functionality. To unify the presentation, our ideal functionality also uses the network functionality $\mathcal{F}_{\text{NET}^q}$. While it is non-standard to use an ideal functionality (here, $\mathcal{F}_{\text{NET}^q}$) with an ideal functionality (here, $\mathcal{F}_{\text{OR}}$), this setting can be reduced to a scenario where a modified ideal functionality $\mathcal{F}_{\text{OR}}{}'$ internally runs $\mathcal{F}_{\text{NET}^q}$. This reduction shows, in particular, that the composability theorem of the UC-framework also applied to our setting.

Messages from A and $\mathcal{F}_{\text{NET}^q}$. In Figure 7.6 and Figure 7.9, we present the pseudocode for the attacker messages and the network functionality, respectively. For our basic analysis, we model an adversary that can control all communication links and servers in $\mathcal{F}_{\text{NET}^q}$, but cannot view or modify messages between parties due to the presence of the secure and

ExtendCircuit$((Q_j)_{j=0}^{\ell}, (Q_j)_{j=0}^{\ell'})$:
  if $\ell' = \ell$ **then**
    output $(\mathsf{created}, (Q_j)_{j=0}^{\ell'})$
  else
    if $\ell' = 0$ **then**
      $cid_1 \leftarrow \{0,1\}^{\kappa}$; *SendMessage*$(Q_1, cid_1, \mathsf{create})$
    else
      SendMessage$(Q_1, cid_1, \mathsf{relay}, (\mathsf{extend}, Q_{\ell'+1}))$

DestroyCircuit$(\mathcal{C}, cid)$:
  if $\bot \neq next(cid) = (Q_{next}, cid_{next})$ **then**
    SendMessage$(Q_{next}, cid_{next}, \mathsf{destroy})$
  else if $\bot \neq prev(cid) = (Q_{prev}, cid_{prev})$ **then**
    SendMessage$(Q_{prev}, cid_{prev}, \mathsf{destroy})$
  discard $\mathcal{C}$

SendMessage$(Q, cid, command[, \mathtt{msg}])$:
  if ($\mathtt{msg}$ is defined) draw a fresh handle h and set $lookup(h) \leftarrow \mathtt{msg}$
  if $compromised(Q) = true$ **then**
    let Q_u be the last node in the contiguous *compromised* path starting in $Q_1 \leftarrow Q$
    /* if Q_u is the onion proxy or Q_u is the exit node */
    if $prev(cid) = (Q_u, \bot)$ or $next(cid) = (Q_u, \bot)$ **then**
      send $\langle P, Q_1, \ldots, Q_u, cid, command, \mathtt{msg}\rangle$ to $\mathbf{A}$
    else
      send $\langle P, Q_1, \ldots, Q_u, cid, command, h\rangle$ to $\mathbf{A}$
  else
    send $\langle P, Q_1, h\rangle$ to $\mathcal{F}_{\text{NET}^q}$

circuit(cid):
  reconstruct the nodes of the circuit that has already been established by recursively applying $next(cid)$ and $prev(cid)$, obtaining a tuple $(Q_j)_{j=0}^{\ell'}$
  let cid_0 be the circuit id such that $prev(cid) = (Q_0, \bot)$
  return $\langle Q_0 \overset{cid_0}{\Longleftrightarrow} Q_1 \Longleftrightarrow \cdots Q_{\ell'}\rangle := (Q_j, cid_j)_{j=0}^{\ell'}$, where $cid_j = \bot$ for $j > 0$

Figure 7.8.: Subroutines of $\mathcal{F}_{\text{OR}}$ for Party P

upon receiving a msg $(\mathsf{observe}, P, Q)$ for $Q \in \mathcal{N}$ from $\mathbf{A}$:
  set $observedLink(P, Q) \leftarrow true$
upon receiving a msg $(\mathtt{compromise}, Q)$ for $Q \in \mathsf{S}$ from $\mathbf{A}$:
  set $compromised(Q) \leftarrow true$; send $SIDS(Q)$ to $\mathbf{A}$
upon receiving a msg (P, Q, m) from $\mathcal{F}_{\text{OR}}$:
  if Q is a $\mathcal{F}_{\text{OR}}$ node **then**
    if $observedLink(P, Q) = true$ **then**
      forward the msg (P, Q, m) to $\mathbf{A}$
    else
      reflect the msg (P, Q, m) to $\mathcal{F}_{\text{OR}}$
  else if Q is a $\mathcal{F}_{\text{NET}^q}$ server **then**
    if $compromised(Q) = true$ **then**
      forward the msg (P, Q, m) to $\mathbf{A}$
    else
      if m is of the form (sid, m') **then**
        $SIDS(Q) \leftarrow SIDS(Q) \cup \{sid\}$
        output $(P, Q, (sid, m'))$
upon receiving a msg (P, Q, m) from $\mathbf{A}$:
  forward the msg (P, Q, m) to $\mathcal{F}_{\text{OR}}$

Figure 7.9.: The Network Functionality $\mathcal{F}_{\text{NET}^q}$

authenticated channel. Therefore, sub-machines in the functionality store their messages in the shared memory, and create and send handles $\langle P, P_{next}, h\rangle$ for these messages in $\mathcal{F}_{\text{NET}^q}$. The message length does not need to be leaked as we assume a fixed message size (for all $M(\kappa)$). Here, P is the sender, P_{next} is the receiver and h is a handle or a pointer to the message in the shared memory of the ideal functionality. In our analysis, all $\mathcal{F}_{\text{NET}^q}$ messages flow to $\mathbf{A}$, which may choose to return these handles back to $\mathcal{F}_{\text{OR}}$ through $\mathcal{F}_{\text{NET}^q}$ at its own discretion. However, $\mathcal{F}_{\text{NET}^q}$ also maintains a mechanism through *observedLink* flags for the non-global adversary $\mathbf{A}$. The adversary may also corrupt or replay the corresponding messages; however, these active attacks are always detected by the receiver due to the presence of a secure and authenticated channel between any two communicating parties and we need not model these corruptions.

The adversary can compromise a party P or server S by sending a `compromise` message to respectively $\mathcal{F}_{\text{OR}}$ and $\mathcal{F}_{\text{NET}^q}$. For party P or server S, the respective functionality then sets the *compromised* tag to *true*. Furthermore, all input or network messages that are

supposed to be visible to the compromised entity are forwarded to the adversary. In principle, the adversary runs that entity for the rest of the protocol and can send messages from that entity. In that case, it can also propagate corrupted messages which in Π_{OR} can only be detected during *UnwrOn* calls at OP or the exit node. We model these corruptions using $\mathit{corrupted}(msg) = \{\mathit{true}, \mathit{false}\}$ status flags, where $\mathit{corrupted}(msg)$ status of messages is maintained across nodes until they reach end nodes. Furthermore, for every corrupted message, the adversary also provides a modification function $T(\cdot)$ as the end nodes run by the adversary may continue execution even after observing a *corrupted* flag. In that case, $T(\cdot)$ captures the exact modificaiton made by the adversary.

We stress that $\mathcal{F}_{\text{OR}}$ does not need to reflect reroutings and circuit establishments initiated by the attacker, because the attacker learns, loosely speaking, no new information by rerouting onions.[7] Similar to the previous work [CL05], a message is directly given to the adversary if all remaining nodes in a communication path are under adversary control.

7.4. Secure OR modules

We identify the core cryptographic primitives for a secure OR protocol. In this section, we present a cryptographic characterization of these core cryptographic primitives, which we call *secure OR modules*. We believe that proving the security of OR modules is significantly less effort than proving the UC security of an entire protocol. Secure OR modules consist of two parts: first, secure onion algorithm, and second, a one-way authenticated key exchange primitive (1W-AKE), a notion recently introduced by Goldberg, Stebila, and Ustaoglu [GSU12].

Onion algorithms typically use several layers of encryptions and possibly integrity mechanisms, such as message authentication codes. Previous attempts [CL05] for proving the security OR protocols use mechanisms to ensure hop-to-hop integrity, such as non-malleable encryption schemes. The widely-used Tor network, however, does not use hop-to-hop integrity but only end-to-end integrity. In the analysis of OR protocols with only end-to-end integrity guarantees, we also have to consider the cases in which the end node is compromised, thus no integrity check is performed at all. In order to cope with these cases, we identify a new notion of predictably malleable encryption schemes. Predictable malleability allows the attacker to change the ciphertexts but requires the resulting changes to the plaintext to be efficiently predictable given only the changes of the ciphertext. In Section 7.4.1 we rigorously define the notion of *predictably malleable* encryption schemes.

Inspired by Section 7.4.1, we introduce in Section 7.4.2 the notion of *secure onion algorithms*.

In the following definitions, we assume the PPT machines to actually be oracle machines. We write A^B to denote that A has oracle access to B.

[7] More formally, the simulator can compute all responses for rerouting or such circuit establishments without requesting information from $\mathcal{F}_{\text{OR}}$ because the simulator knows all long-term and session keys. The only information that the simulator does not have is the routing information, which the simulator gets in case of rerouting or circuit establishment.

```
upon (initialize)
  k ← G(1^η)
  s_d ← ε; s_e ← ε

upon (encrypt, m)
  if b = 0 then
     (c, s) ← E(0^|m|, s_e, k)
     if q(s_e) ≠ ⊥ then
        (d, u) ← q(s_e)
        c ← d
  else if b = 1 then
     (c, s) ← E(m, s_e, k)
  q(s_e) ← (c, m)
  s_e ← s; respond c

upon (decrypt, c)
  (d, u) ← q(s_d)
  T ← M(c, d)
  if b = 0 then
     if q(s_d) = ⊥ then
        (m, s)
           ← D(c, s_d, k)
        q(s_d) ← (c, m)
     else
        if q(s_d) ≠ ⊥ then
           m ← T(u)
  else if b = 1 then
     (m, s) ← D(c, s_d, k)
  s_d ← s; respond (m, T)
```

Figure 7.10.: The IND-PM Challenger $PM\text{-}Ch_b^{\mathcal{E}}$

7.4.1. Predictably Malleable Encryption

Simulation-based proofs often face their limits when dealing with malleable encryption schemes. The underlying problem is that malleability induces an essentially arbitrarily large number of possibilities to modify ciphertexts, and the simulator has no possibility to predict the resulting changes to the corresponding plaintext.

We characterize the property of predicting the changes to the plaintext merely given the modifications on the ciphertext. Along the lines of the IND-CCA definition for stateful encryption schemes, we define the notion of *predictably malleable* (IND-PM) encryption schemes.[8] The attacker has access to an encryption and a decryption oracle, and either all encryption and decryption queries are honestly answered (the honest game) or all are faked (the faking game), i.e., $0^{|m|}$ is encrypted instead of a message m. In the faking game, the real messages are stored in some shared datastructure q, and upon a decryption query only look-ups in q are performed. The IND-PM challenger maintains a separate state, e.g., a counter, for encryption and decryption. These respective states are updated with each encryption decryption query. Predicable malleability is similar to HCCA [PR08] except that it is defined for stateful encryption schemes.

In contrast to the IND-CCA challenger, the IND-PM challenger (see Figure 7.10) additionally stores the produced ciphertext together with the corresponding plaintext for each encryption query. Moreover, for each decryption call the challenger looks up the stored ciphertexts and messages. The honest decryption ignores the stored values and performs an honest decryption, but the faking decryption compares the stored ciphertext with the ciphertext from the query and tries to predict the modifications to the plaintext. Therefore, we require the existence of an efficiently computable algorithm M that outputs the description of an efficient transformation procedure T for the plaintext given the original ciphertext as well as the modified ciphertext.

Definition 83 (Predictable malleability)**.** *An encryption scheme $\mathcal{E} := (G, E, D)$ is IND-PM if there is a negligible function μ such that there is a deterministic polynomial-time*

[8] The name predictable malleability is justified as it can be shown that every IND-CCA secure scheme is also IND-PM, and every IND-PM scheme in turn is IND-CPA secure. In Section 7.6.1, we present detCTR and state that it is IND-PM secure.

algorithm M such that for all PPT attackers $\mathcal{A}$

$$\Pr[b' \leftarrow \{0,1\}, b \leftarrow \mathcal{A}(1^\kappa)^{PM\text{-}Ch_b^{\mathcal{E}}} : b = b'] \leq 1/2 + \mu(\kappa)$$

Moreover, we require that for all $m, c, s, k, k' \in \{0,1\}^*$

$$\Pr[(c', s') \leftarrow E(m, k, s), \\ (m', s'') \leftarrow D(c, k', s) : s' = s''] = 1$$

$PM\text{-}Ch_0^{\mathcal{E}}$ and $PM\text{-}Ch_1^{\mathcal{E}}$ are defined in Figure 7.10.

We stress that the definition implies a super-polynomial length for state-cycles; otherwise there is in the faking game at least one repeated state s for which the two `encrypt` queries output the same ciphertext for any two plaintexts.

In Section 7.6.1, we show that deterministic counter-mode is IND-PM.

7.4.2. Secure Onion Algorithms

We identify the onion wrapping (*WrOn*) and unwrapping (*UnwrOn*) algorithms as central building blocks in onion routing. We identify four core properties of onion algorithms. The first property is *correctness*, i.e., if all parties behave honestly, the result is correct. The second property is the security of statefulness, coined *synchronicity*. It roughly states that whenever a wrapping and an unwrapping algorithms are applied to a message with unsynchronous states, the output is completely random. The third property is *end-to-end integrity*. The fourth property states that for all modifications to an onion the resulting changes in the ciphertext are predictable. We this property *predictable malleability*.

Onion Correctness. The first property of secure onion algorithms is *onion correctness*. It states that honest wrapping and unwrapping results in the same message. Moreover, the correctness states that whenever the unwrapping algorithm has a `fake` flag, it does not care about integrity, because for $m \in M(\kappa)$ the integrity measure is always added, as required by the end-to-end integrity. But for $m \notin M(\kappa)$ but of the right length, the wrapping is performed without an integrity measure. The `fake` flag then causes the unwrapping to ignore the missing integrity measure. Then, we also require that the state transition is independent from the message or the key.

Definition 84 (Onion correctness)**.** *Recall that $M(\kappa)$ is the message space for the security parameter κ. Let $\langle k_i \rangle_{i=1}^{\ell}$ be a sequence of randomly chosen bitstrings of length κ.*

Forward: $\Omega_f(m)$
- $O_1 \leftarrow WrOn(m, \langle k_i \rangle_{i=1}^{\ell})$
- ***for*** $i = 1$ *to* ℓ ***do***
  - $O_{i+1} \leftarrow UnwrOn(O_i, k_i)$
- $x \leftarrow O_{\ell+1}$

Backward: $\Omega_b(m)$
- $O_\ell \leftarrow WrOn(m, k_\ell)$
- ***for*** $i = \ell - 1$ *to* 1 ***do***
  - $O_i \leftarrow WrOn(O_{i+1}, k_i)$
- $x \leftarrow UnwrOn(O_1, \langle k_i \rangle_{i=1}^{\ell})$

Let Ω'_f be the defined as Ω_f except that UnwrOn additionally uses the `fake` *flag. Analogously, Ω'_b is defined. We say that a pair of onion algorithms $(WrOn, UnwrOn)$ is* correct *if the following three conditions hold:*

(i) $\Pr[x \leftarrow \Omega_d(m) : x = m] = 1$ *for $d \in \{f, b\}$ and $m \in M(\kappa)$.*

(*ii*) $\Pr[x \leftarrow \Omega'_d(m) : x = m] = 1$ *for* $d \in \{f, b\}$ *and all* $m \in M'(\kappa) := \{m' | \exists m'' \in M(\kappa).|m'| = |m''|\}$.

(*iii*) *For all* $m \in M'(\kappa)$, $k, k' \in \{0,1\}^\kappa$ *and* $c, s \in \{0,1\}^*$ *such that* c *is a valid onion and* s *is a valid state*

$$\Pr[(c', s') \leftarrow \mathit{WrOn}(m, k, s), \\ (m', s'') \leftarrow \mathit{UnwrOn}(c, k', s) : s' = s''] = 1$$

(*iv*) *WrOn and UnwrOn are polynomial-time computable and randomized algorithms.*

Synchronicity. The second property is synchronicity. In order to achieve replay resistance, we have to require that once the wrapping and unwrapping do not have synchronized states anymore, the output of the wrapping and unwrapping algorithms is indistinguishable from randomness.

Definition 85 (Synchronicity)**.** *For a machine* $\mathbf{A}$*, let* $\Omega_{l,\mathbf{A}}$ *and* $\Omega_{r,\mathbf{A}}$ *be defined as follows:*

Left: $\Omega_{l,\mathbf{A}}(\kappa)$
$(m_1, m_2, st) \leftarrow \mathbf{A}(1^\kappa)$
$k, s, s' \leftarrow \{0,1\}^\kappa$
$O \leftarrow \mathit{WrOn}(m_1, k, s)$
$O' \leftarrow \mathit{UnwrOn}(O, k, s')$
$b \leftarrow \mathbf{A}(O', st)$

Right: $\Omega_{r,\mathbf{A}}(\kappa)$
$(m_1, m_2, st) \leftarrow \mathbf{A}(1^\kappa)$
$k, s, s' \leftarrow \{0,1\}^\kappa$
$O \leftarrow \mathit{WrOn}(m_2, k, s)$
$O' \leftarrow \mathit{UnwrOn}(O, k, s')$
$b \leftarrow \mathbf{A}(O', st)$

For all PPT machines $\mathbf{A}$ *the following is negligible in* κ*:*

$$|\Pr[b \leftarrow \Omega_{l,\mathbf{A}}(\kappa) : b = 1] - \Pr[b \leftarrow \Omega_{r,\mathbf{A}}(\kappa) : b = 1]|$$

End-to-end integrity. The third property that we require is *end-to-end integrity*; i.e., the attacker is not able to produce an onion that successfully unwraps unless it compromises the exit node. For the following definition, we modify $\mathsf{OS\text{-}Ch}^0$ such that, along with the output of the attacker, also the state of the challenger is output. In turn, the resulting challenger $\mathsf{OS\text{-}Ch}^{0'}$ can optionally get a state s as input. In particular, $(a, s) \leftarrow A^B$ denotes in the following definition the pair of the outputs of A and B.

For the following definition we use the modified challenger $\mathsf{OS\text{-}Ch}^{0'}$, which results from modifying $\mathsf{OS\text{-}Ch}^0$ such that along with the output of the attacker also the state of the challenger is output. The resulting challenger $\mathsf{OS\text{-}Ch}^{0'}$ can, moreover, optionally get a state s as input.

Definition 86 (End-to-end integrity)**.** *Let* $S(O, cid)$ *be the machine that sends a* (destruct, O) *query to the challenger and outputs the response. Let* $Q'(s)$ *be the set of answers to* construct *queries from the challenger to the attacker. Let the last onion* $O_{\ell'}$ *of an onion* O_1 *be defined as follows:*

$\mathsf{Last}(O_1)$***:***
for $i = 1$ *to* $\ell' - 1$ ***do***
$O_{i+1} \leftarrow \mathit{UnwrOn}(O_i)$

Let $Q(s) := \{\mathsf{Last}(O_1) \mid O_1 \in Q'(s)\}$ *be the set of last onions answers to the challenger. We say a set of onion algorithms has* end-to-end integrity *if for all PPT attackers* $\mathbf{A}$ *the*

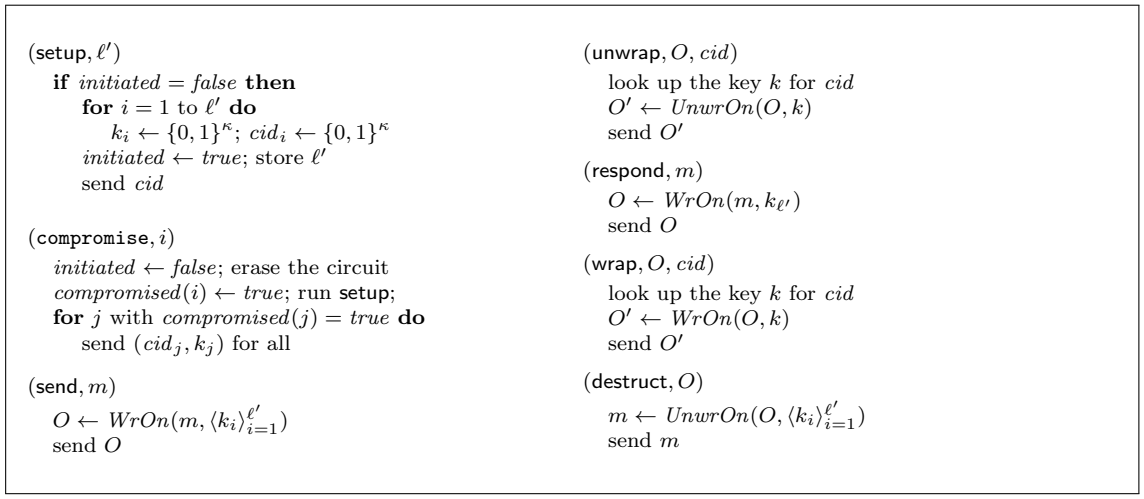

Figure 7.11.: The Honest Onion Secrecy Challenger $\mathsf{OS\text{-}Ch}^0$: $\mathsf{OS\text{-}Ch}^0$ only answers for honest parties

following is negligible in κ

$$\Pr[(O,s) \leftarrow \mathbf{A}(1^\kappa)^{\mathsf{OS\text{-}Ch}^{0\prime}}, (m,s') \leftarrow S(O,cid)^{\mathsf{OS\text{-}Ch}^{0\prime}(s)} : m \in M(\kappa) \wedge P_{\ell'} \textit{ is honest} \wedge O \notin Q(s')].$$

Predictably Malleable Onion Secrecy. The fourth property that we require is *predictably malleable onion secrecy*, i.e., for every modification to a ciphertext the challenger is able to compute the resulting changes for the plaintext. This even has to hold for faked plaintexts.

In detail, we define a challenger $\mathsf{OS\text{-}Ch}^0$ that provides, a wrapping, a unwrapping and a send and a destruct oracle. In other words, the challenger provides the same oracles as in the onion routing protocol except that the challenger only provides one single session. We additionally define a faking challenger $\mathsf{OS\text{-}Ch}^1$ that provides the same oracles but fakes all onions for which the attacker does not control the final node.

For $\mathsf{OS\text{-}Ch}^1$, we define the maximal paths that the attacker knows from the circuit. A visible subpath of a circuit $(P_i, k_i, cid_i)_{i=1}^{\ell}$ from an honest onion proxy is a minimal subsequence of corrupted parties $(P_i)_{i=u}^{s}$ of $(P_i)_{i=1}^{\ell}$ such that P_{i-1} is honest and either $s = \ell$ or P_{s+1} is honest as well. The parties P_{i-1} and, if existent, P_{s+1} are called the guards of the visible subpath $(P_i)_{i=u}^{s}$. We store visible subpaths by the first $cid = cid_u$.

Figure 7.11 and 7.12 presents $\mathsf{OS\text{-}Ch}^0$, and $\mathsf{OS\text{-}Ch}^1$, respectively.[9]

Definition 87 (Predictably malleable onion secrecy)**.** *Let* onionAlg *be a pair of algorithms WrOn and UnwrOn. We say that the algorithms* onionAlg *satisfy* predictably malleable onion secrecy *if there is a negligible function* μ *such that there is a efficiently computable function* M *such that for all PPT machines* $\mathcal{A}$ *and sufficiently large* κ

$$\Pr[b \leftarrow \{0,1\}, b' \leftarrow \mathbf{A}(1^\kappa)^{\mathsf{OS\text{-}Ch}^b} : b = b'] \leq 1/2 + \mu(\kappa)$$

[9]We stress that in Figure 7.12 the onion O_u denotes the onion from party P_j to party P_{j+1}.

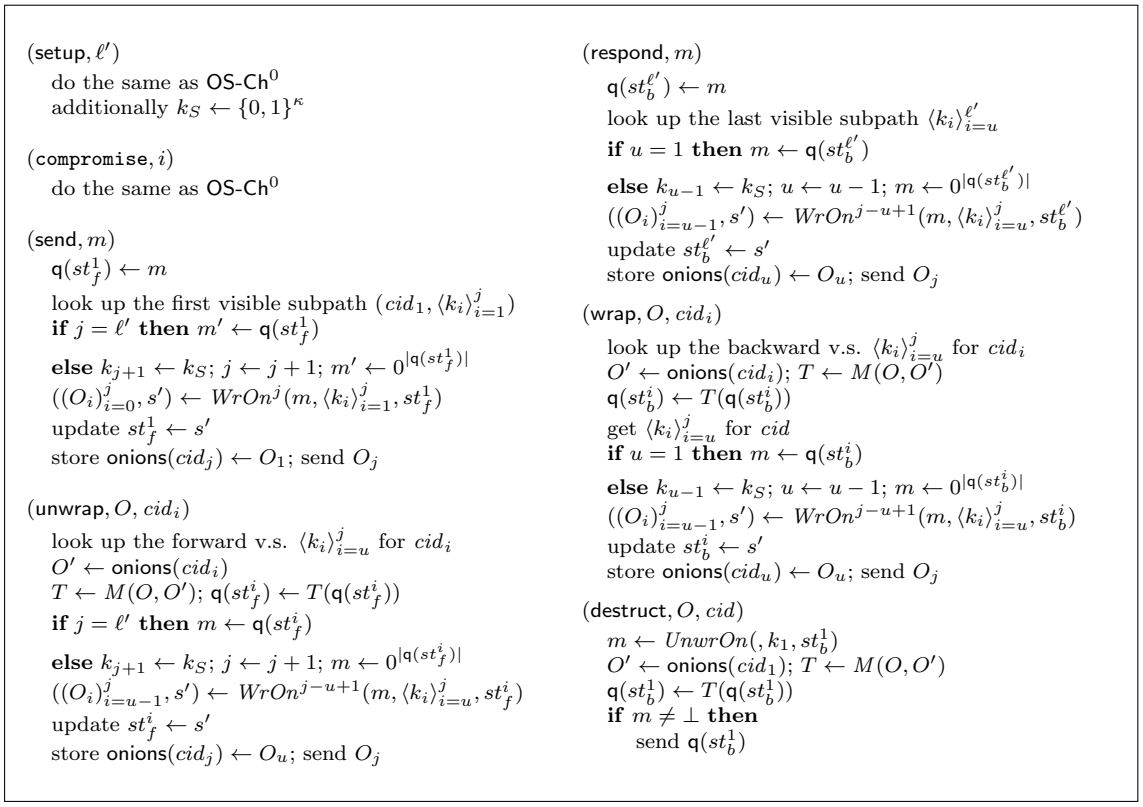

Figure 7.12.: The Faking Onion Secrecy Challenger OS-Ch^1: OS-Ch^1 only answers for honest parties. st_f^i, st_b^i is the current forward, respectively backward, state of party i. $((O_i)_{i=u-1}^j, s') \leftarrow WrOn^{j-u+1}(m, \langle k_i\rangle_{i=u}^j, st)$ is defined as $O_{u-1} \leftarrow m$; **for** i = u **to** j **do** $(O_i, s') \leftarrow WrOn(O_{i-1}, k_{j+u-i}, st)$

Definition 88 (Secure onion algorithms)**.** *A pair of onion algorithms* (*WrOn*, *UnwrOn*) *is* secure *if it satisfies onion correctness, synchronicity, predictably malleable onion secrecy, and end-to-end integrity.*

In Section 7.6.2, we show that the Tor algorithms are secure onion algorithms.

7.4.3. One-Way Authenticated Key-Exchange

We introduce the 1W-AKE primitive in Section 7.2.2, and later use the 1W-AKE algorithms *Init*, *Resp*, and *CompKey* in the OR protocol Π_{OR} in Section 7.2.4. In this section, we give an informal description of the security requirements for 1W-AKE. The rigorous definitions can be found in [GSU12].

The 1W-AKE establishes a symmetric key between two parties (an initiator and a responder) such that the identity of the initiator cannot be derived from the protocol messages. Moreover, given a public-key infrastructure, the 1W-AKE guarantees that the responder cannot be impersonated. *Init* takes as input the public key of the responder and generates a challenge. *Resp* takes as input the responder's secret key and the received challenge, and outputs a session key and a response. The algorithm *CompKey* runs on the

response and the responder's public key, and outputs the session key or an error message of authentication failure.

We assume a public-key infrastructure; i.e., every party knows a secret key whose corresponding public key has been distributed in a verifiable manner. Let pk_P be the public key of party P and sk_P be its secret key.

The first property that a 1W-AKE has to satisfy is correctness: if all parties behave honestly, then the protocol establishes a shared key.

Definition 89 (Correctness of 1W-AKE)**.** *Let a public-key infrastructure be given; i.e., for every party P every party knows a (certified) public key pk_P and P itself also knows the corresponding secret key sk_P. Let $AKE := (Init, Resp, CompKey)$ be a tuple of polynomial-time bounded randomized algorithms. We say that AKE is a* correct one-way authenticated key-agreement *if the following holds for all parties A, B:*

$$\begin{aligned}\Pr[&(ake, B, m_1, \Psi_A) \leftarrow Init(pk_B, B, m),\\ &((ake, B, m_2), (k_2, \star, \vec{v})) \leftarrow Resp(pk_B, sk_B, m_1),\\ &(k_1, B, \vec{v}') \leftarrow CompKey(pk_B, m_2,)\\ &: k_1 = k_2 \text{ and } \vec{v} = \vec{v}'] = 1.\end{aligned}$$

The 1W-AKE Challenger for Security. Goldberg, Stebila, and Ustaoglu [GSU12] formalize the security of a 1W-AKE by defining a *challenger* that represents all honest parties. The attacker is then allowed to query this challenger. If the attacker is not able to distinguish a fresh session key from a randomly chosen session key, we say that the 1W-AKE is *secure*. This challenger is constructed in a way that security of the 1W-AKE implies one-way authentication of the responding party.

The challenger answers the following queries of the attacker. Internally, the challenger runs the algorithms of AKE. All queries are directed to some party P; we denote this party in a superscript. If the party is clear from the context, we omit the superscript, e.g., we then write $\texttt{send}(m)$ instead of $\texttt{send}^P(m)$.

- $\texttt{send}^P(\mathsf{params}, P')$*:* Compute $(m, \mathsf{st}) \leftarrow Init\ (pk_P, P', \mathsf{params})$. Send m to the attacker.
- $\texttt{send}^P(\Psi, msg, P')$*:* If $P' = P$ and $\mathsf{akest}(\Psi) = \perp$, compute $(m, \mathsf{result}) \leftarrow Resp(sk_P, P, msg, \Psi)$. Otherwise, if $msg = (msg', Q)$ compute $(m, \mathsf{result}) \leftarrow CompKey\ (pk_Q, msg', \mathsf{akest}(\Psi), \Psi)$. Then, send m to the attacker.
- $\texttt{compromise}^P$*:* The challenger returns the long-term key of P to the attacker.

If any verification fails, i.e. one of the algorithms outputs $\perp$, then the challenger erases all session-specific information for that party and aborts the session.

Additionally, the attacker has access to the following oracle in the 1W-AKE security experiment:

$\texttt{test}(P, \Psi)$: Abort if party P has no key stored for session Ψ or the partner for session Ψ is anonymous(i.e., P is not the initiator of session Ψ). Otherwise, choose $b \leftarrow \{0, 1\}$. If $b = 1$, then return the session key k; otherwise, if $b = 0$, return a randomly chosen element from the key space. Only one call to `test` is allowed.

We say that a session Ψ at a party i is *fresh* if no party involved in that session is compromised.

Definition 90 (One-way-AKE-security)**.** *Let κ be a security parameter and let $n \geq 1$. A protocol π is said to be* one-way-AKE-secure *if, for all PPT adversaries M, the advantage that M distinguishes a session key of a one-way-AKE-fresh session from a randomly chosen session key is negligible (in κ).*

The 1W-AKE Challenger for One-Way Anonymity. For the definition of *one-way anonymity* we introduce a proxy, called the anonymity challenger, that relays all messages from and to the 1W-AKE challenger except for a challenge party C. The attacker can choose two challenge parties, out of which the anonymity challenger randomly picks one, say i^*. Then, the anonymity challenger relays all messages that are sent to C to P_{i^*} (via the 1W-AKE challenger).

In the one-way anonymity experiment, the adversary can issue the following queries to the challenger C. All other queries are simply relayed to the 1W-AKE challenger. The session Ψ^* denotes the challenge session. The two queries are for activation and communication during the test session.

$\texttt{start}^C(i, j, \mathsf{params}, P)$: Abort if $i = j$. Otherwise, set $i \leftarrow \{i, j\}$ and $(\Psi^*, msg) \leftarrow \texttt{send}^{P_{i^*}}(\mathsf{params}, P)$; return msg'. Only one message $\texttt{start}^C$ is processed.

$\texttt{send}^C(msg)$: Relay $\texttt{send}^{P_{i^*}}(msg)$ to the 1W-AKE challenger. Upon receiving an answer msg', forward msg' to the attacker.

Definition 91 (One-way anonymity)**.** *Let κ be a security parameter and let $n \geq 1$. A protocol π is said to be* one-way anonymous *if, for all PPT adversaries M, the advantage that M wins the following experiment $\mathsf{Expt}^{1w-anon}_{\pi,\kappa,n}(M)$ is negligible (in κ).*

1. *Initialize parties $P_1, \ldots, P_n$.*
2. *The attacker M interacts with the anonymity challenger, finishing with a message $(\mathsf{guess}, \hat{i})$.*
3. *Suppose that M made a $\mathsf{Start}^C(i, j, \mathsf{params}, P)$ query which chose i^*. If $\hat{i} = i^*$, and M's query satisfy the following constraints, then M wins; otherwise M loses.*
   - *No $\texttt{compromise}^P$ query for P_i and P_j.*
   - *No $\mathsf{Send}(\Psi^*, \cdot)$ query to P_i or P_j.*

A set of algorithms *AKE* is said to be a *one-way authenticated key-exchange primitive* (short 1W-AKE) if it satisfies Definitions 89, 90, and 91.

In Section 7.6.3 we show that ntor is a 1W-AKE.

7.5. Π_{OR} UC-Realizes $\mathcal{F}_{\text{OR}}$

In this section, we show that Π_{OR} can be securely abstracted as the ideal functionality $\mathcal{F}_{\text{OR}}$.

Recall that π securely realizes $\mathcal{F}$ in the $\mathcal{F}'$-hybrid model if each party in the protocol π has a direct connection to $\mathcal{F}'$. $\mathcal{F}^{\mathcal{N}}_{\text{REG}}$ is the key registration, $\mathcal{F}_{\text{SCS}}$ is the secure channel functionality, and $\mathcal{F}_{\text{NET}^q}$ is the network functionality, where q is the upper bound on the corruptable parties. We prove our result in the $\mathcal{F}^{\mathcal{N}}_{\text{REG}}$, $\mathcal{F}_{\text{SCS}}$-hybrid model; i.e., our result holds for any key registration and secure channel protocol securely realizing $\mathcal{F}^{\mathcal{N}}_{\text{REG}}$, and

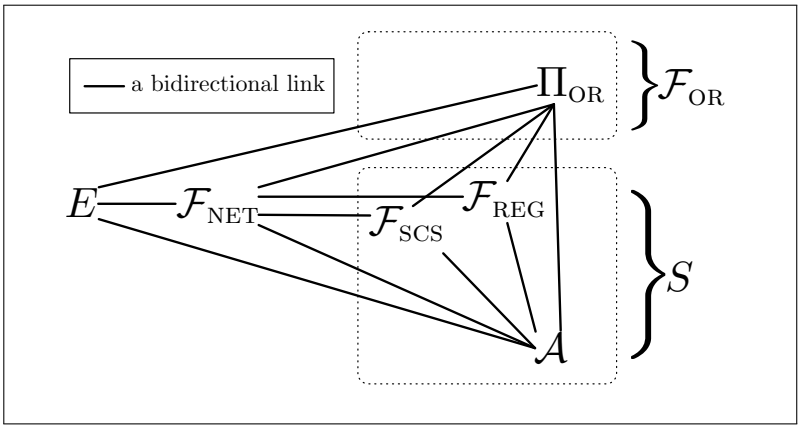

Figure 7.13.: Overview of the set-up

$\mathcal{F}_{\text{SCS}}$, respectively. The network functionality $\mathcal{F}_{\text{NET}^q}$ abstract partially global attacker and is a global functionality.[10]

Theorem 12. *If Π_{OR} uses secure OR modules $\mathcal{M}$, then with the global functionality $\mathcal{F}_{\text{NET}^q}$ the resulting protocol Π_{OR} in the $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$,$\mathcal{F}_{\text{SCS}}$-hybrid model securely realizes the ideal functionality $\mathcal{F}_{\text{OR}}$ for any q.*

As a next step, we give a proof outline in order to highlight at which places we apply the required security properties or the secure OR modules. The full proof can be found in [BGKM12].

Proof. We have to show that for every PPT attacker $\mathbf{A}$ there is a PPT simulator S such that no PPT environment E can distinguish the interaction with $\mathbf{A}$ and Π_{OR} from the interaction with S and $\mathcal{F}_{\text{OR}}$. Given a PPT attacker $\mathbf{A}$, we construct a simulator S that internally runs $\mathbf{A}$ and simulates the public key infrastructure; i.e, the functionality $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$. The crucial part in this proof is that the ideal functionality $\mathcal{F}_{\text{OR}}$ provides the simulator with all necessary information for the simulation. We prove this indistinguishability by examining a sequence of six games and proving their pairwise indistinguishability for the environment E.

Game 1: Game_1 is the original setting in which the environment E interacts with the protocol $\Pi_1 = \Pi_{\text{OR}}$ and the attacker $\mathbf{A}$. Moreover, Π_{OR} and $\mathbf{A}$ have access to a certification authority $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ and a secure channel functionality $\mathcal{F}_{\text{SCS}}$, and the network messages of all honest parties are sent via $\mathcal{F}_{\text{NET}^q}$.

Game 2: In Game_2 the simulator S_2 internally runs the attacker $\mathbf{A}$ and the functionalities $\mathcal{F}_{\text{SCS}}$ and $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$. All messages from these entities are forwarded on the corresponding channels and all messages to these entities are forwarded to the corresponding channels. Since S_2 honestly computes the attacker $\mathbf{A}$, $\mathcal{F}_{\text{SCS}}$, and $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$, Game_1 and Game_2 are perfectly indistinguishable for the environment E.

[10] We stress that Π_{OR} (with any modules) is $\mathcal{F}_{\text{NET}^q}$-subroutine respecting; hence the GUC composition theorem holds.

Game 3: In the protocol Π_3, we modify the session keys that have been established between two uncompromised parties. All parties are one machine and share some state. Instead of using the established key, Π_2 stores a randomly chosen value in the shared state for each established key k. This random value is used as a session key instead of k.

Assume that there is a PPT machine that can distinguish an execution with a randomly chosen session key from an execution with an honestly generated session key non-negligible probability (in the security parameter κ), given the key-exchange's transcript of messages. Then, using a hybrid argument, it can be shown that there is an attacker that breaks the security of the 1W-AKE, which in turn contradicts the assumption that the OR modules are secure. Hence, Game_2 and Game_3 are computationally indistinguishable.

Game 4: In Game_4, the onions do not contain the real messages anymore but only the constant zero bitstring. Π_4 maintains a shared datastructure q in which the real messages are stored.

Recall that a visible subpath of a circuit $(P_i, k_i, cid_i)_{i=1}^{\ell}$ from an honest onion proxy is a minimal subsequence of corrupted parties $(P_i)_{i=u}^{s}$ of $(P_i)_{i=1}^{\ell}$ such that P_{i-1} is honest and either $s = \ell$ or P_{s+1} is honest as well. The parties P_{i-1} and, if existent, P_{s+1} are called the guards of the visible subpath $(P_i)_{i=u}^{s}$. In particular, the onion proxy is also a guard. Every circuit can be split into a sequence of visible subpaths and guards. Π_4 stores for every circuit $(P_i, k_i, cid_i)_{i=1}^{\ell}$ such a splitting into visible subpaths and guards. These splittings are updated upon each `compromise` command.

Upon receiving a send input or a response from a network, Π_4 stores an input message m in a shared datastructure q as follows. For a guards P, let cid_P be the circuit id for which P knows the key. Let s the state of the wrapping algorithms of the sender before computing the onion. Then, we store $\mathsf{q}(cid_P, s) \leftarrow m$ for each P.

The attacker might be able to corrupt onions such that the contained plaintext is changed. Π_4, however, does not rely on the content of the onions anymore but rather looks up the message in the shared memory. Therefore, Π_4 needs a way to derive the changes to the plaintext due to possible modifications of the ciphertexts. At this point our predictable malleability applies, and we use the algorithm D from the onion secrecy definition for computing the changes in the plaintext. However, for computing the changes in the plaintext, we need to store the onions that the receiving guard has to expect. Hence, Π_4 maintains a shared datastructure onions indexed by the cid of the receiving guard that stores the expected onions.

Π_4 initially draws some distinguished random key k_S, which is later used for a distinguished last wrapping-layer of the constant zero bitstring. Whenever in Π_3 a guard P that is neither the exit node nor the onion proxy would unwrap an onion O with key k and circuit id cid, P looks up $O' = \mathit{pending}(cid)$. Then, it runs $T \leftarrow S(O, O')$ and replaces the real message $m \leftarrow \mathsf{q}(cid, st)$ in the shared memory with $T(m)$, where st is the state of the onion algorithms in the forward direction. Then, P unwraps O with the `fake` flag, i.e., $(O'', st') \leftarrow \mathit{UnwrOn}(O, k_S, \texttt{fake}, st)$ instead of $\mathit{UnwrOn}(O, k, st)$. We set the `fake` flag, because the unwrapping has to skip the integrity check; otherwise a corrupted onion would already in the middle of the circuit be stopped in Π_4. However, instead of forwarding O'', P constructs a new onion either for the attacker or for the next guard as follows. P looks up the adjacent visible subpath $(P_i)_{i=u}^{s}$ in forward direction. If $s = \ell$, then P constructs the onion for the attacker. P reads the real message $m \leftarrow \mathsf{q}(cid, st)$ from the shared memory

and sends a forward onion O_j for the subcircuit $(P_i, k_i, cid_i)_{i=u}^{\ell}$ that contains the message m and is constructed as follows:

$O_{u-1} \leftarrow m$
for $i = u$ **to** ℓ **do** $(O_i, st') \leftarrow WrOn(O_{i-1}, k_{j+u-i}, st)$

Only then, P updates the forward state $st \leftarrow st'$. Thereafter, P stores $\mathsf{q}(cid_{P_{j+1}}, st') \leftarrow O_u$, where $cid_{P_{j+1}}$ is the circuit id of the guard P_{j+1}.

If $s < \ell$, P sends a forward onion for the subcircuit $(P_i, k_i, cid_i)_{i=u}^{s+1}$ that contains $0^{|m|}$ instead of m, where we replace for the last layer k_{s+1} by the distinguished key k_S. Again only then, P updates the forward state $st \leftarrow st'$. Analogously, guards that are onion proxies, i.e., construct an onion in forward direction, also only construct an onion for the attacker or the next guard.

Similar to the forward direction, guards that receive an onion O in backward direction do not wrap it further as in Π_3 but first unwrap O with the `fake` flag and the distinguished key k_S, i.e., $O' \leftarrow UnwrOn(O, k_S, \texttt{fake})$. Instead of wrapping O as in Π_3, the guard constructs an onion for the adjacent subpath in backward direction as follows. Since P is a guard for the circuit, also the onion proxy is honest, thus $u > 1$. P looks up the adjacent visible subpath $(P_i)_{i=u}^{s}$ in backward direction. Let $m \leftarrow \mathsf{q}(cid, st)$ be the real message stored in the shared memory, cid be the circuit id for which P knows the key and s be the state of the onion algorithms in the backward direction. Then, P sends an onion $(O, st') \leftarrow UnwrOn(0^{|m|}, \langle k_i \rangle_{i=u-1}^{s}, st)$, where $k_{u-1} := k_S$. Thereafter, update the backward state $st \leftarrow st'$.

It might happen that the attacker compromised a node in the middle of the circuit and the exit node. Then the attacker sends a random message to an honest node P. In this case, P would honestly unwrap the message. Since the attacker controls the exit node the broken integrity is not realized. But from that point on the guard P is out of sync, i.e., P has a different unwrapping state than the predecessor guards. Consequently, by the synchronicity of the onion algorithms all future messages that are sent from the onion proxy will be garbage. For guards that are out of sync, we only send randomly chosen messages of appropriate length.

Then, by a hybrid argument it follows that any attacker distinguishing Game_3 from Game_4 can be used for breaking onion secrecy or synchronicity, where the hybrids are indexed by the circuits of honest onion proxies in the order in which the circuits are initiated. Hence, Game_3 and Game_4 are indistinguishable.

Game 5: In this setting the simulator remains unchanged, i.e., $S_5 = S_4$, but the protocol Π_5 in addition internally runs the ideal functionality $\mathcal{F}_{\text{OR}}$. We construct Π_5 such that it does not touch information in the send message, i.e., the message to be sent and the circuit, more than forwarding the send message to $\mathcal{F}_{\text{OR}}$. Instead, Π_5 only uses the messages that it receives from $\mathcal{F}_{\text{OR}}$.

$\mathcal{F}_{\text{OR}}$ outputs for every message from a guard to a visible subpath the entire visible subpath, both in forward an backward direction. If the visible subpath contains the exit node, $\mathcal{F}_{\text{OR}}$ even sends the message. Hence, Π_5 can just compute all onions to the next guard or the attacker in the same way as Π_4.

We also have to cope with the case in which Π_4 modifies the real message in the shared state with the transformation T that M computed from the differences in the expected

and the received onion. In this case, Π_5 sends a message $(\mathsf{corrupt}, T, h)$ to $\mathcal{F}_{\text{OR}}$.

Moreover, upon the message $(\mathsf{register}, P)$ the simulator computes a pk for party P and sends a message $(\mathsf{register}, P, pk)$ to the internally emulated functionality $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$. Upon a response $(\mathsf{registered}, \langle P_j, pk_j \rangle_{j=1}^{v})$ from $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$, we send $(\mathsf{registered}, \langle P_j \rangle_{j=1}^{v})$

Π_5 behaves like Π_4 except for the key agreement messages, which is computed by the simulator instead of the real party. But by the anonymity of the 1W-AKE primitive, the attacker cannot identify the sender with more than negligible probability. Consequently, Game_4 and Game_5 are indistinguishable.

Game 6: In this setting, we replace the protocol with the ideal functionality; i.e., $\Pi_6 = \mathcal{F}_{\text{OR}}$. The simulator $S := S_6$ in Game_6 additionally computes all network messages exactly as Π_5. As Π_5 did not touch the messages from the environment to the ideal functionality, S can compute Π_5 as well.

The ideal functionality behaves towards the environment exactly as Π_{OR}; consequently, it suffices to show that the network messages are indistinguishable. However, as the simulator S just internally runs Π_5, Game_5 and Game_6 are indistinguishable.

□

As our primitives are proven secure in the random oracle model (ROM), the main theorem uses the ROM.

Theorem 13. *If pseudorandom permutations exist, there are secure OR modules* (*ntor*, onionAlgs) *such that the protocol* Π_{OR} *in the* $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$, $\mathcal{F}_{\text{SCS}}$, $\mathcal{F}_{\text{NET}^q}$*-hybrid model using* (*ntor*, onionAlgs) *securely realizes in the ROM the ideal functionality* $\mathcal{F}_{\text{OR}}$ *in the* $\mathcal{F}_{\text{NET}^q}$*-hybrid model for any* q*.*

Proof. If pseudorandom permutations exist Lemma 45 implies that secure onion algorithms exist. Lemma 46 shows that in the ROM 1W-AKE exist. Then, Theorem 12 implies the statement. □

Note that we could not prove 1W-AKE security for the TAP protocol currently used in Tor as it uses a CCA-insecure version of the RSA encryption scheme.

7.6. Instantiating Secure OR Modules

We present a concrete instantiation of OR modules and show that this instantiation constitutes a set of secure OR modules. As onion algorithms we use the algorithms that are used in Tor with a strengthened integrity mechanism, and as 1W-AKE we use the recently proposed *ntor* protocol [GSU12].

We prove that the onion algorithms of Tor constitute secure onion algorithms, as defined in Definition 88. The crucial part in that proof is to show that these onion algorithms are predictably malleable, i.e., for every modification of the ciphertext the changes in the resulting plaintext are predictable by merely comparing the modified ciphertext with the original ciphertext. We first show that the underlying encryption scheme, the deterministic counter-mode, is predictably malleable (Section 7.6.1). Thereafter, we show the security of Tor's onion algorithms (Section 7.6.2).

$G_c(1^\eta)$
  output $k \leftarrow G(1^\eta)$

$E_c((x_1,\ldots,x_t),(k,\mathsf{ctr})) = D_c((x_1,\ldots,x_t),(k,\mathsf{ctr}))$
  if $\mathsf{ctr} = \varepsilon$ **then** $\mathsf{ctr} = 0$
  output $(PRP(s,k) \oplus x_1, \ldots, PRP(s+t-1,k) \oplus x_t, (k, \mathsf{ctr}+t))$

Figure 7.14.: The stateful deterministic counter-mode (detCTR) $\mathcal{E}_c = (G_c, E_c, D_c)$

In Section 7.6.3, we briefly present the *ntor* protocol and cite the result from Goldberg, Stebila, and Ustaoglu that *ntor* constitutes a 1W-AKE. The proofs of the lemmas in this section are postponed to the full version [BGKM12].

7.6.1. Deterministic Counter Mode and Predictable Malleability

We show that the deterministic counter-mode (detCTR) scheme is predictably malleable, as defined in Definition 83.

Lemma 44. *If pseudorandom permutations exist, the deterministic counter mode (detCTR) with $\mathcal{E}_c = (G_c, E_c, D_c)$ as defined in Figure 7.14 predictably malleable.*

Proof. We show the result with $t = 1$. This can, however, be extended to larger t in a straight-forward way.

Game_1 is the original game of $\mathcal{A}$ against $PM\text{-}Ch_1$.

Game_2 is the game in which $PM\text{-}Ch_1$ is replaced by the machine B_1 and the PRP Challenger $PRP\text{-}Ch_1$ such that B_1 communicates with $\mathcal{A}$ and $PRP\text{-}Ch_1$, which generates a key applies the PRP candidate algorithms. $\mathcal{A}$ cannot distinguish Game_1 from Game_2, as $\mathcal{A}$'s view is the same in both scenarios.

Game_3 is the game in which $PRP\text{-}Ch_1$ is replaced by $PRP\text{-}Ch_0$, which uses a randomly chosen permutation instead of the PRP candidate. As PRP is a pseudorandom permutation, the attacker cannot distinguish Game_2 from Game_3.

Game_4 is the game in which B_1 is replaced by B_0. Upon a query $(\texttt{decrypt}, c)$, the B_1 outputs $x \oplus c$ whereas B_0 outputs $c \oplus d \oplus u = c \oplus 0^{|u|} \oplus x \oplus u = c \oplus x \oplus u$. c can be represented as $c = d \oplus c'$ for some bitstring c'. Then, B_1 outputs $x \oplus x \oplus u \oplus c' = u \oplus c'$, and B_0 outputs $x \oplus c' \oplus x \oplus u = u \oplus c'$. Hence, the responses of $(\texttt{decrypt}, c)$ queries are the same for B_1 and B_0.

Upon a query $(\texttt{encrypt}, m)$, the B_1 responds $r \oplus m$ whereas B_0 outputs $r \oplus 0^{|m|} = r$. Since, r is randomly chosen and $\oplus$ is a group operation the attacker cannot distinguish $r \oplus m$ from r.[11] Game_3 and Game_4 only differ in B_b; hence, these two games are indistinguishable

Game_5 is the game in which $PRP\text{-}Ch_0$ is replaced by $PRP\text{-}Ch_1$, which uses the PRP candidate instead of a randomly chosen permutation. As PRP is a pseudorandom permutation, the attacker cannot distinguish Game_4 from Game_5.

[11] Since we use a random permutation, $\mathcal{A}$ can try the following: before starting the challenge phase, he sends as many encryption queries as he is allowed to and computes the corresponding $\mathsf{Enc}(\mathsf{ctr}_e)$. For the challenge response c_b he computes $c_b \oplus m$ and checks whether the result equals one of the $\mathsf{Enc}(\mathsf{ctr}_e)$ he has observed before. If so, either the encryption function is not a permutation or $b = 0$. This, however, only happens with a negligible probability.

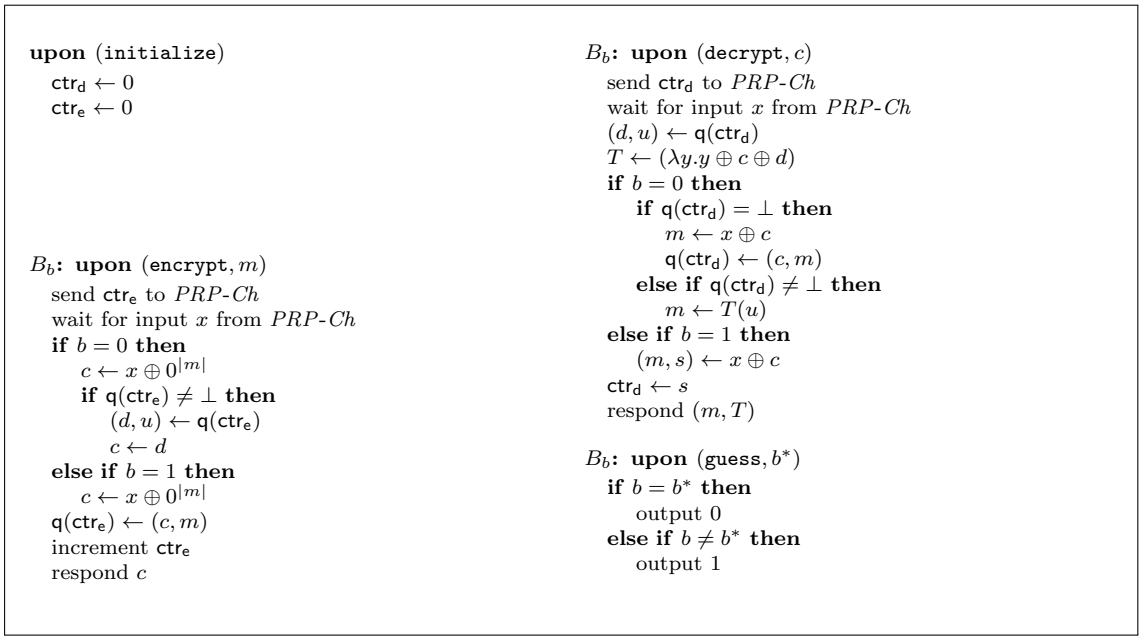

Figure 7.15.: The machine B_b

Game_6 is again the original game of $\mathcal{A}$ against $PM\text{-}Ch_0$. The attacker cannot distinguish Game_5 and Game_6, because the view of $\mathcal{A}$ is the same.

We conclude that Game_1 and Game_6, and therefore $PM\text{-}Ch_1$ and $PM\text{-}Ch_0$, are indistinguishable. □

7.6.2. Security of Tor's Onion Algorithms

Let $E := (Gen_e, Enc, Dec)$ be a stateful deterministic encryption scheme, and let $M := (Gen_m, Mac, V)$ be a deterministic MAC. Let PRG be a pseudo random generator such that for all $x \in \{0,1\}^*$ $|PRG(x)| = 2 \cdot |x|$. We write $PRG(x)_1$ for the first half of $PRG(x)$ and $PRG(x)_2$ the second half. Moreover, for a randomized algorithm A, we write $A(x; r)$ for a call of $A(x)$ with the randomness r.

As a PRP candidate we use AES, as in Tor, and as a MAC use H-MAC with SHA-256. We use that in detCTR encrypting two blocks separately results in the same ciphertext as encrypting the pair of the blocks at once. Moreover, we assume that the output of H-MAC is exactly one block.

The correctness follows by construction. The synchronicity follows, because a PRP is used for the state. The end-to-end integrity directly follows from the SUF of the Mac. And the predictable malleability follows from the predictable malleability of the deterministic counter-mode.

Lemma 45. *Let* onionAlg $= \{UnwrOn_I, WrOn_I\}$. *If pseudorandom permutations exist,* onionAlg *are secure onion algorithms.*

Proof. The correctness follows directly from the construction. The synchronicity can be reduced to the pseudorandomness of PRP of the detCTR scheme. In detail, we can

$WrOn_I(O,k)$, **for** $O \notin M(\kappa)$
  $O' \leftarrow \mathsf{Enc}_{ctr}(O,k)$; return O'

$WrOn_I(m,k)$, **for** $m \in M(\kappa)$
  $(r,r') \leftarrow PRG(k)$; $k_m \leftarrow Gen_m(r)$
  $k_e \leftarrow Gen_e(r')$
  $O \leftarrow \mathsf{Enc}_{ctr}(m,k_e)$
  $O' \leftarrow Mac(O,k_m)$; return O'

$WrOn_I(m,\langle k_i\rangle_{i=1}^{\ell})$, **for** $m \in M(\kappa)$
  $O_2 \leftarrow WrOn_I(m,k_1)$
  for $i = 2$ to ℓ **do**
    $O_{i+1} \leftarrow WrOn_I(O_i,k_i)$
  return O_ℓ

$UnwrOn_I(O,k)$
  $(r,r') \leftarrow PRG(k)$; $k_m \leftarrow Gen_m(r)$
  $k_e \leftarrow Gen_e(r')$
  $O' \leftarrow \mathsf{Dec}_{ctr}(O,k_e)$
  if $O' = m||t$ and $m \in M(\kappa)$ **then**
    if $V(m,t,k_m) = 1$ **then**
      return $(\mathsf{default}, O'')$
    else
      return $(corrupted, O'')$
  else
    $O' \leftarrow \mathsf{Dec}_{ctr}(O,k)$; return $(\mathsf{default}, O')$

$UnwrOn_I(O,k,\mathtt{fake})$
  $O' \leftarrow \mathsf{Dec}_{ctr}(O,k)$; return $(\mathsf{default}, O')$

$UnwrOn_I(O,\langle k_i\rangle_{i=1}^{\ell})$
  for $i = 1$ to ℓ **do**
    $(\mathsf{type}_{i+1}, O_{i+1}) \leftarrow UnwrOn_I(O_i,k_i)$
  return $(\mathsf{type}_\ell, O_\ell)$

Figure 7.16.: The Onion Algorithms onionAlg

construct a machine that breaks the pseudorandomness of PRP if there is an attacker that breaks the synchronicity.

For showing the end-to-end integrity, assume that there is a ppt attacker that is able to produce an onion O such that $\perp \neq m \leftarrow UnwrOn(O,k)$ and O is not an answer of a query. Then, we can construct a machine that breaks the SUF of the MAC by internally running the attacker against the end-to-end integrity and computing all detCTR call on our own and forwarding all *Mac* calls to the SUF challenger. Finally, after unwrapping the onion, we send the tag t to the SUF challenger as a guess. If the attacker against the end-to-end integrity wins with non-negligible probability, then we also win with non-negligible probability.

For showing the predictable malleability, we present a sequence of games and show that they are indistinguishable for any ppt attacker. In Game_0 the challenger is exactly defined as $\mathsf{OS\text{-}Ch}^0$. In Game_1 additionally the message m is stored in a shared memory $\mathsf{q}(st) \leftarrow m$ (st being the corresponding state), and the challenger maintains a separation into visible subpaths. Obviously, Game_1 is indistinguishable from Game_0 for any ppt attacker.

In Game_2, the challenger initially draws a distinguished key k_S. Then, the challenger looks up for every query $(\mathsf{unwrap}, O, cid_i$ the adjunct visible subpath $\langle P_i,k_i\rangle_{i=u}^{j}$ in forward direction. Then, the challenger completely unwraps the onion and checks whether the stored message $\mathsf{q}(st)$ equals the unwrapped message. If this check fails, the challenger proceeds with the onion as in Game_1. If the this check succeeds, however, and P_j is not the exit node the challenger computes $((O_i)_{i=u}^{j+1}, s') \leftarrow WrOn_I^{j-u+2}(m,\langle k_i\rangle_{i=u}^{j+1}, st_f^i)$, where $WrOn_I^{j-u+2}$ is defined as in Definition 87, $k_{j+1} := k_S$, and st_f^i denotes the forward state of party i. Thereafter, the challenger updates the forward state $st_f^i \leftarrow s'$ of party i. If P_j is the exit node, then, we only use $\langle k_i\rangle_{i=u}^{j}$, and perform one $WrOn$ operation less. (send, m) queries are processed in the same way except that additionally the message m is stored as $\mathsf{q}(st_f^1) \leftarrow m$.

For the backward direction, i.e., queries $(\mathsf{respond}, m)$ and $(\mathsf{wrap}, O, cid_i)$, the challenger proceeds analogously except that it is not checked whether $P_{\ell'}$ is compromised but whether P_1 is compromised. Accordingly, $k_{u-1} := k_S$ is used in the backward direction. Game_2 is indistinguishable from Game_1 because malicious onions are not touched and the length of a circuit is not leaked by an onion.

In Game_3 the challenger, loosely spoken, fakes all onions for which the message is not visible to the attacker. For all queries, the challenger additionally also stores the onion that the next guard has to expect. For example, consider an onion in forward direction with an adjunct visible subpath $\langle P_i, k_i \rangle_{i=u}^{j}$ for which P_j is not the exit node. Then, the challenger always stores $\mathsf{onions}(cid_i) \leftarrow O_u$ the onion that the guard P_{j+1} expects. In the backward direction the challenger analogously stores the expected onion for the next guard. Upon a $(\mathsf{unwrap}, O, cid_i)$ query, the challenger runs the predictor $T \leftarrow M(O, \mathsf{onions}(cid_i))$ of detCTR. The resulting transformations T is applied to the stored message $\mathsf{q}(st_f^i)$. The challenger proceeds analogously for the query $(\mathsf{wrap}, O, cid_i)$ in backward direction. Moreover, the challenger fakes all queries in forward direction for which the last node P_j in the visible subpath $\langle P_i, k_i \rangle_{i=u}^{j}$ is not the exit node, i.e., instead of the actual message $\mathsf{q}(st_f^i)$ the constant zero bitstring $0^{|\mathsf{q}(st_f^i)|}$ is used.

We can construct a machine M that breaks the predictable malleability of detCTR given an attacker that distinguishes Game_3 from Game_2. M internally runs the attacker computes the challenger Game_3 except for detCTR encryption and decryption calls, which are forwarded to the IND-PM challenger $PM\text{-}Ch_i$. Then, M breaks the predictable malleability of detCTR if the attacker distinguishes Game_3 from Game_2.

The challenger in Game_3 is exactly defined as $\mathsf{OS\text{-}Ch}^1$. Since Game_0 and Game_3 are indistinguishable, also $\mathsf{OS\text{-}Ch}^0$ and $\mathsf{OS\text{-}Ch}^1$ are indistinguishable. Hence, $\mathsf{onionAlg}$ satisfy predictably malleable onion secrecy. □

7.6.3. *ntor*: A 1W-AKE

Øverlier and Syverson [vS07] proposed a 1W-AKE for use in the next generation of the Tor protocol with improved efficiency. Goldberg, Stebila, and Ustaoglu found an authentication flaw in this proposed protocol, fixed it, and proved the security of the fixed protocol [GSU12]. We use this fixed protocol, called *ntor*, as a 1W-AKE.

The protocol *ntor* [GSU12] is a 1W-AKE protocol between two parties P (client) and Q (server), where client P authenticates server Q. Let (pk_Q, sk_Q) be the static key pair for Q. We assume that P holds Q's certificate (Q, pk_Q). P initiates an *ntor* session by calling the *Init* function and sending the output message m_P to Q. Upon receiving a message m'_P, server Q calls the *Resp* function and sends the output message m_Q to P. Party P then calls the *CompKey* function with parameters from the received message m'_Q, and completes the *ntor* protocol. We assume a unique mapping between the session ids Ψ_P of the *cid* in Π_{OR}.

Lemma 46 (*ntor* is anonymous and secure [GSU12])**.** *The ntor protocol is a one-way anonymous and secure 1W-AKE protocol in the random oracle model (ROM).*

Init(pk_Q, Q):

1. Generate ephemeral keys $(x, X \leftarrow g^x)$.
2. Set session id $\Psi_P \leftarrow H_{\mathsf{st}}(X)$.
3. Update $\mathsf{st}(\Psi_P) \leftarrow (ntor, Q, x, X)$.
4. Set $m_P \leftarrow (ntor, Q, X)$.
5. Output m_P.

Resp(pk_Q, sk_Q, X):

1. Verify that $X \in G^*$.
2. Generate ephemeral keys $(y, Y \leftarrow g^y)$.
3. Set session id $\Psi_Q \leftarrow H_{\mathsf{st}}(Y)$.
4. $(k', k) \leftarrow H(X^y, X^{sk_Q}, Q, X, Y, ntor)$.
5. $t_Q \leftarrow H_{mac}(k', Q, Y, X, ntor, \texttt{server})$.
6. Set $m_Q \leftarrow (ntor, Y, t_Q)$.
7. Delete y and output m_Q.

CompKey(pk_Q, Ψ_P, t_Q, Y):

1. Retrieve Q, x, X from $\mathsf{st}(\Psi_P)$ if it exists.
2. Verify that $Y \in G^*$.
3. $(k', k) \leftarrow H(Y^x, pk_Q^x, Q, X, Y, ntor)$.
4. If $t_Q = H_{mac}(k', Q, Y, X, ntor, \texttt{server})$,
5. delete $\mathsf{st}(\Psi_P)$ and output k.

If any verification fails, the party erases all session-specific information and aborts the session.

Figure 7.17.: The *ntor* protocol

7.7. Forward Secrecy and Anonymity Analysis

In this section, we show that our abstraction $\mathcal{F}_{\text{OR}}$ allows for applying previous work on the anonymity analysis of onion routing to Π_{OR}. Moreover, we illustrate that $\mathcal{F}_{\text{OR}}$ enables a rigorous analysis of forward secrecy of Π_{OR}.

In Section 7.7.1, we show that the analysis of Feigenbaum, Johnson, and Syverson [FJS11] of Tor's anonymity properties in a black-box model can be applied to our protocol Π_{OR}. Feigenbaum, Johnson, and Syverson show their anonymity analysis an ideal functionality $\mathcal{B}_{\text{OR}}$ (see Figure 7.5). By proving that the analysis of $\mathcal{B}_{\text{OR}}$ applies to $\mathcal{F}_{\text{OR}}$, the UC composition theorem and Theorem 12 imply that the analysis applies to Π_{OR} as well.

In Section 7.7.2, we present the result that immediate forward secrecy for Π_{OR} holds, by merely by analyzing $\mathcal{F}_{\text{OR}}$.

7.7.1. OR Anonymity Analysis

In this section, we discuss the applicability of our result to OR anonymity analysis techniques. First, we show that an anonymity analysis of a black-box OR abstraction $\mathcal{B}_{\text{OR}}$ by Feigenbaum, Johnson and Syverson [FJS11] carries over to the OR protocol presented in this work (see Section 7.7.1.1). Second, we present a generalization of $\mathcal{B}_{\text{OR}}$ that also accounts for partially global attackers, i.e., attackers that can observe links between honest ORs (see Section 7.7.1.2). We show that also all results about the generalization of $\mathcal{B}_{\text{OR}}$ carry over to the OR protocol presented in this work.

7.7.1.1. Π_{OR} realizes $\mathcal{B}_{\text{OR}}$

Feigenbaum, Johnson and Syverson [FJS11] analyzed the anonymity properties of OR networks. In their analysis, the authors abstracted an OR network against attackers that are local, static as a black-box functionality $\mathcal{B}_{\text{OR}}$. We reviewed their abstraction $\mathcal{B}_{\text{OR}}$ in Section 7.2.5. In this section, we show that the analysis of $\mathcal{B}_{\text{OR}}$ is applicable to Π_{OR} against local, static attackers.

There is a slight mismatch in the user-interface of $\mathcal{B}_{\text{OR}}$ and Π_{OR}. The main difference is that Π_{OR} expects separate commands for creating a circuit and sending a message whereas

upon the first input m
send N_{OR} to $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ in Π; send setup to Π
wait for $(\mathsf{ready}, \langle P_i \rangle_{i=1}^{n})$; further process m

upon an input $(\mathsf{send}, S, [m])$
draw $P_1, \ldots, P_\ell$ at random from N_{OR}
store $(S, m_{\mathtt{dummy}})$ [or (S, m)] in the queue for $\langle P, P_1, \ldots, P_\ell \rangle$
send $(\mathsf{createcircuit}, \langle P, P_1, \ldots, P_\ell \rangle)$ to Π

upon $(\mathsf{created}, \langle P \overset{cid_1}{\Longleftrightarrow} P_1 \Longleftrightarrow \ldots P_\ell \rangle)$ **from** Π
look up (S, m) from the queue for $\langle P, P_1, \ldots, P_\ell \rangle$
send $(\mathsf{send}, \langle P \overset{cid_1}{\Longleftrightarrow} P_1 \Longleftrightarrow \ldots P_\ell \rangle, (S, m))$ to Π

upon $(\mathsf{received}, \mathcal{C}, m)$ **from** Π
do nothing /*$\mathcal{B}_{\text{OR}}$ does not allow responses to messages*/

upon a message m **from** $\mathcal{F}_{\text{NET}^0}$ **to the environment**
do nothing /*$\mathcal{B}_{\text{OR}}$ does not deliver messages*/

Figure 7.18.: User-interface $U(\Pi)$ for party P

$\mathcal{B}_{\text{OR}}$ only expects a command for sending a message. We construct for every party P a wrapper U for Π_{OR} that adjusts Π_{OR}'s user-interface. Recall that we consider two versions of $\mathcal{B}_{\text{OR}}$ and U simultaneously: one version in which no message is sent and one version in which a message is sent (denoted as $[m]$).

Instead of Π_{OR}, U only expects one command: $(\mathsf{send}, S, [m])$. We fix the length ℓ of the circuit.[12] Upon $(\mathsf{send}, S, [m])$, $U(\Pi)$ draws the path $P_1, \ldots, P_\ell \leftarrow N_{\text{OR}}$ at random, sends a $(\mathsf{createcircuit}, \langle P, P_1, \ldots, P_\ell \rangle)$ to Π, waits for the *cid* from Π, and sends a (send, cid, m) command, where m is a dummy message if no message is specified. Moreover, in contrast to $\mathcal{B}_{\text{OR}}$ the protocol Π_{OR} allows a response for a message m and therefore additionally sends a session id *sid* to a server.[13]

In addition to the differences in the user-interface, $\mathcal{B}_{\text{OR}}$ assumes the weaker threat model of a local, static attacker whereas Π_{OR} assumes a partially global attacker. We formalize a local attacker by considering Π_{OR} in the $\mathcal{F}_{\text{NET}^0}$-hybrid model, and connect the input/output interface of $\mathcal{F}_{\text{NET}^0}$ to the wrapper U as well. For considering a static attacker, we make the standard UC-assumption that every party only accepts `compromise` requests at the beginning of the protocol. Moreover, we also need to assume that $\mathcal{B}_{\text{OR}}$ is defined for a fixed set of onion routers $\mathcal{N}$ in the same way as $\mathcal{F}_{\text{OR}}^{\mathcal{N}}$.

Finally, our work culminates in the connection of previous work on black-box anonymity analyses of onion routing with our cryptographic model of onion routing.

Lemma 47 ($U(\Pi_{\text{OR}})$ UC realizes $\mathcal{B}_{\text{OR}}$)**.** *Let $U(\Pi_{\text{OR}})$ be defined as in Figure 7.18. If Π_{OR} uses secure OR modules, then $U(\Pi_{\text{OR}})$ in the $\mathcal{F}_{\text{NET}^0}$-hybrid model UC realizes $\mathcal{B}_{\text{OR}}$ against static attackers.*

Proof. Applying the UC composition theorem, it suffices to prove that $U(\mathcal{F}_{\text{OR}})$ in the $\mathcal{F}_{\text{NET}^0}$-hybrid model UC realizes $\mathcal{B}_{\text{OR}}$ against static attackers. We construct a simulator $S_{\mathbf{A}}$ as in Figure 7.19 that internally runs $\mathcal{F}_{\text{NET}^0}$ and $U'(\mathcal{F}_{\text{OR}})$ and an attacker $\mathbf{A}$. Then, we show that $\mathcal{B}_{\text{OR}}$ against $S_{\mathbf{A}}$ is indistinguishable from $U(\mathcal{F}_{\text{OR}})$ against $\mathbf{A}$ for any PPT environment E.

[12] We fix the length for the sake of brevity. This choice is rather arbitrary. The analysis can be adjusted to the case in which the length is chosen from some efficiently computable distribution or specified by the environment for every message.

[13] It is also possible to modify Π_{OR} such that Π_{OR} does not accept responses and does not draw a session id *sid*. However, for the sake of brevity we slightly modify $\mathcal{B}_{\text{OR}}$.

upon the first input m
  set $N_{\mathbf{A}} := \emptyset$; send N_{OR} to $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ in $\mathcal{F}_{\text{OR}}$
  send setup to $\mathcal{F}_{\text{OR}}$; wait for $(\mathsf{ready}, \langle P_i \rangle_{i=1}^{n})$
  further process m

upon $(\texttt{compromise}, P)$ **from A**
  if all previous messages only were compromise messages **then**
    set $N_{\mathbf{A}} := N_{\mathbf{A}} \cup \{P\}$
    forward $(\texttt{compromise}, P)$ to party P in $U(\mathcal{F}_{\text{OR}})$

upon the first message m **that is not** compromise **from A**
  send $(\texttt{compromise}, N_A)$ to $\mathcal{B}_{\text{OR}}$
  further process m

upon any other message m **from A to** $\mathcal{F}_{\text{NET}^0}$
  forward m to $\mathcal{F}_{\text{NET}^0}$

upon any other message m **from A to** $U'(\mathcal{F}_{\text{OR}})$
  forward m to $U'(\mathcal{F}_{\text{OR}})$

upon a message m **from** $U'(\mathcal{F}_{\text{OR}})$ **to the environment**
  do nothing /* $\mathcal{B}_{\text{OR}}$ already outputs the message */

upon $(\mathsf{sent}, U, S, [m])$ **from** $\mathcal{B}_{\text{OR}}$
  choose $P_1 \leftarrow \mathsf{N}_{\mathbf{A}}$ and $P_\ell \leftarrow \mathsf{N}_{\mathbf{A}}$
  choose $P_2, \ldots, P_{\ell-1} \leftarrow N_{\text{OR}}$
  send $(\mathsf{send}, \langle P_i \rangle_{i=1}^{\ell}, [m])$ to $U'(\mathcal{F}_{\text{OR}})$

upon $(\mathsf{sent}, -, S, [m])$ **from** $\mathcal{B}_{\text{OR}}$
  choose $P_1 \leftarrow \mathsf{N}_{\text{OR}} \setminus \mathsf{N}_{\mathbf{A}}$ and $P_\ell \leftarrow \mathsf{N}_{\mathbf{A}}$
  choose $P_2, \ldots, P_{\ell-1} \leftarrow N_{\text{OR}}$
  send $(\mathsf{send}, \langle P_i \rangle_{i=1}^{\ell}, [m])$ to $U'(\mathcal{F}_{\text{OR}})$

upon $(\mathsf{sent}, U, -)$ **from** $\mathcal{B}_{\text{OR}}$
  choose $P_1 \leftarrow \mathsf{N}_{\mathbf{A}}$ and $P_\ell \leftarrow \mathsf{N}_{\text{OR}} \setminus \mathsf{N}_{\mathbf{A}}$
  choose $P_2, \ldots, P_{\ell-1} \leftarrow N_{\text{OR}}$
  send $(\mathsf{send}, \langle P_i \rangle_{i=1}^{\ell}, [m_{\texttt{dummy}}])$ to $U'(\mathcal{F}_{\text{OR}})$

upon $(\mathsf{sent}, -, -)$ **from** $\mathcal{B}_{\text{OR}}$
  choose $P_1 \leftarrow \mathsf{N}_{\text{OR}} \setminus \mathsf{N}_{\mathbf{A}}$ and $P_\ell \leftarrow \mathsf{N}_{\text{OR}} \setminus \mathsf{N}_{\mathbf{A}}$
  choose $P_2, \ldots, P_{\ell-1} \leftarrow N_{\text{OR}}$
  send $(\mathsf{send}, \langle P_i \rangle_{i=1}^{\ell}, [m_{\texttt{dummy}}])$ to $U'(\mathcal{F}_{\text{OR}})$

Figure 7.19.: The simulator $S_{\mathbf{A}}$: U' gets the path as input instead of drawing it at random

We show that the following sequence of games is indistinguishable for the environment E. The first game Game_1 is the original setting with $U(\mathcal{F}_{\text{OR}})$ and $\mathbf{A}$ in the $\mathcal{F}_{\text{NET}^0}$-hybrid model. In the second game Game_2, the simulator S_2 honestly simulates $\mathcal{F}_{\text{NET}^0}$ and the attacker $\mathbf{A}$. As S_1 honestly simulates $\mathcal{F}_{\text{NET}^0}$ the two games Game_1 and Game_2 are indistinguishable.

In the third game Game_3, the simulator S_3 honestly runs $U(\mathcal{F}_{\text{OR}})$ as well. As S_3 honestly simulates $U(\mathcal{F}_{\text{OR}})$ the two games Game_2 and Game_3 are indistinguishable.

In the fourth game Game_4, the simulator S_4 maintains a set of compromised parties $N_{\mathbf{A}}$. S_4 runs U' instead of U, where U' gets the path as input instead of drawing the path at random. Then, the simulator S_4 upon an input $(\mathsf{send}, S, [m])$ to U draws the first onion router P_1 (not the onion proxy) and the exit node P_ℓ as follows with $b := |N_{\mathbf{A}}|/|N_{\text{OR}}|$.

- (i) with probability b^2, S_4 draws $P_1, P_\ell \leftarrow N_{\mathbf{A}}$
- (ii) with probability $b(1-b)$, S_4 draws $P_1 \leftarrow N_{\mathbf{A}}$ and $P_\ell \leftarrow N_{\text{OR}} \setminus N_{\mathbf{A}}$
- (iii) with probability $(1-b)b$, S_4 draws $P_1 \leftarrow N_{\text{OR}} \setminus N_{\mathbf{A}}$ and $P_\ell \leftarrow N_{\mathbf{A}}$
- (iv) with probability $(1-b)^2$ S_4 draws $P_1, P_\ell \leftarrow N_{\text{OR}} \setminus N_{\mathbf{A}}$

The nodes $P_2, \ldots, P_{\ell-1} \leftarrow N_{\text{OR}}$ are drawn uniformly at random. Then, S_4 sends $(\mathsf{send}, \langle P_i \rangle_{i=1}^{\ell}, [m])$ to U'.

Game_4 is indistinguishable from Game_3 as the distribution of compromised parties remains the same, and in Game_4 the modified wrapper U' together with the simulator S_4 have the same input/output behavior as U.

The game Game_5 is the scenario in which $S_{\mathbf{A}}$ communicates with $\mathcal{B}_{\text{OR}}$. The simulator does not directly communicate with the environment over the protocol interface anymore but $\mathcal{B}_{\text{OR}}$ communicates with the environment instead. The simulator S_5 behaves as $S_{\mathbf{A}}$ in Figure 7.19.

The difference between the input/output behavior of $\mathcal{B}_{\text{OR}}$ and the part of S_4 that communicates with U' is minimal. Only for the cases in which the last onion router is not compromised the message m is not sent to U'. In these cases $S_{\mathbf{A}}$ chooses $m_{\texttt{dummy}}$ as a message. But as the ideal functionality does not reveal any information about m if the last node is not compromised, Game_5 and Game_4 are indistinguishable. □

7.7.1.2. Generalizing $\mathcal{B}_{\text{OR}}$ to partially global attackers

The result from the previous section can be generalized to an onion routing network against partially global attackers. In order to cope with the partially compromised network, the black-box needs to maintain the amount of compromised links, in addition to the number of compromised parties. In this section, we prove that even for $q > 0$ the onion routing protocol $U(\Pi_{\text{OR}})$ realizes this modified black-box $\mathcal{B}_{\text{OR}'}$, which is defined in Figure 7.20.

The realization proof goes along the lines of the proof of Lemma 47. However, in order to bound the probability that a link between a user and the first onion router or an exit node and a server is compromised, we need to restrict the number of users and servers. Let m be the amount of users and o be the amount of servers.

Lemma 48 ($U(\Pi_{\text{OR}})$ UC realizes $\mathcal{B}_{\text{OR}'}$)**.** *Let $U(\Pi_{\text{OR}})$ be defined as in Figure 7.18. If Π_{OR} uses secure OR modules, then $U(\Pi_{\text{OR}})$ in the $\mathcal{F}_{\text{NET}^q}$-hybrid model UC realizes $\mathcal{B}_{\text{OR}}$ against static attackers for any $q \in \{0, \ldots, n\}$, where n is the number of onion routers.*

Proof. Applying the UC composition theorem, it suffices to prove that $U(\mathcal{F}_{\text{OR}})$ in the $\mathcal{F}_{\text{NET}^q}$-hybrid model UC realizes $\mathcal{B}_{\text{OR}'}$ against static attackers. We construct a simulator $S'_{\mathbf{A}}$ as in Figure 7.21 that internally runs $\mathcal{F}_{\text{NET}^0}$ and $U'(\mathcal{F}_{\text{OR}})$ and an attacker $\mathbf{A}$. Then, we show that $\mathcal{B}_{\text{OR}'}$ against $S'_{\mathbf{A}}$ is indistinguishable from $U(\mathcal{F}_{\text{OR}})$ against $\mathbf{A}$ for any ppt environment E.

We show that the following sequence of games is indistinguishable for the environment E. The first game Game_1 is the original setting with $U(\mathcal{F}_{\text{OR}})$ and $\mathbf{A}$ in the $\mathcal{F}_{\text{NET}^q}$-hybrid model. In the second game Game_2, the simulator S_2 honestly simulates $\mathcal{F}_{\text{NET}^q}$ and the attacker $\mathbf{A}$. As S_1 honestly simulates $\mathcal{F}_{\text{NET}^q}$ the two games Game_1 and Game_2 are indistinguishable.

In the third game Game_3, the simulator S_3 honestly runs $U(\mathcal{F}_{\text{OR}})$ as well. As S_3 honestly simulates $U(\mathcal{F}_{\text{OR}})$ the two games Game_2 and Game_3 are indistinguishable.

In the fourth game Game_4, the simulator S_4 maintains a set of compromised parties $N_{\mathbf{A}}$. S_4 runs U' instead of U, where U' gets the path as input instead of drawing the path at random. Then, the simulator S_4 upon an input $(\mathsf{send}, S, [m])$ to U draws the first onion router P_1 (not the onion proxy) and the exit node P_ℓ as follows with $n := |\mathsf{N}_{\text{OR}}|$, $b \leftarrow \frac{|\mathsf{N}_{\mathbf{A}}|}{n}$, $L'_{\mathbf{A}} := L_{\mathbf{A}} \cap (N_{\text{OR}} \setminus N_{\mathbf{A}})^2$, and $c \leftarrow \frac{|L'_{\mathbf{A}}|}{n(n-1)/2}$:

(i) with probability $(b+c)^2$, S_4 draws

$$(P_1, P_\ell) \leftarrow (\mathsf{N}_{\mathbf{A}} \cup \{P \mid \exists Q.(P,Q) \in L_{\mathbf{A}}\}) \times (\mathsf{N}_{\mathbf{A}} \cup \{P \mid \exists Q.(P,Q) \in L_{\mathbf{A}}\})$$

(ii) with probability $(b+c)(1-(b+c))$, S_4 draws

$$(P_1, P_\ell) \leftarrow ((\mathsf{N}_{\text{OR}} \setminus \mathsf{N}_{\mathbf{A}}) \cap \{P \mid (U,P) \notin L_{\mathbf{A}}\}) \times (\mathsf{N}_{\mathbf{A}} \cup \{P \mid (P,S) \in L_{\mathbf{A}}\})$$

upon receiving a msg $(\texttt{compromise}, \mathsf{N}_{\mathbf{A}}, L_{\mathbf{A}})$ from $\mathbf{A}$:
set $\mathit{compromised}(P) \leftarrow \mathit{true}$ **for** every $P \in \mathsf{N}_{\mathbf{A}}$
set $n \leftarrow |\mathsf{N}_{\text{OR}}|$; set $b \leftarrow \frac{|\mathsf{N}_{\mathbf{A}}|}{n}$
set $L'_{\mathbf{A}} \leftarrow L_{\mathbf{A}} \cap (\{(P, P')^n |$
$(P$ is a user $\wedge\, P' \in (N_{\text{OR}} \setminus N_{\mathbf{A}})) \vee$
$(P \in (N_{\text{OR}} \setminus N_{\mathbf{A}}) \wedge P'$ is a server$\}))$
set $c \leftarrow \frac{|L'_{\mathbf{A}}|}{nm+no}$
upon an input $(\mathsf{send}, S, [m])$ from the environment for party U:
with probability $(b+c)^2$,
choose $P_\ell \leftarrow \mathsf{N}_{\mathbf{A}}$
send $(\mathsf{sent}, U, S, [m])$ to $\mathbf{A}$
with probability $(1-(b+c))(b+c)$,
choose $P_\ell \leftarrow \mathsf{N}_{\mathbf{A}}$
send $(\mathsf{sent}, -, S, [m])$ to $\mathbf{A}$
with probability $(b+c)(1-(b+c))$,
choose $P_\ell \leftarrow \mathsf{N}_{\text{OR}} \setminus \mathsf{N}_{\mathbf{A}}$
send $(\mathsf{sent}, U, -)$ to $\mathbf{A}$
with probability $(1-(b+c))^2$,
choose $P_\ell \leftarrow \mathsf{N}_{\text{OR}} \setminus \mathsf{N}_{\mathbf{A}}$
send $(\mathsf{sent}, -, -)$ to $\mathbf{A}$

Figure 7.20.: Black-box OR Functionality $\mathcal{B}_{\text{OR}'}$ for partially global attackers: N_{OR} is the set of all parties

(*iii*) with probability $(1-(b+c))(b+c)$, S_4 draws

$$(P_1, P_\ell) \leftarrow (\mathsf{N}_{\mathbf{A}} \cup \{P \mid (U,P) \in L_{\mathbf{A}}\}) \times (\mathsf{N}_{\text{OR}} \setminus \mathsf{N}_{\mathbf{A}} \cap \{P \mid (P,S) \notin L_{\mathbf{A}}\})$$

(*iv*) with probability $(1-(b+c))^2$, S_4 draws

$$(P_1, P_\ell) \leftarrow (\mathsf{N}_{\text{OR}} \setminus \mathsf{N}_{\mathbf{A}} \cap \{P \mid (U,P) \notin L_{\mathbf{A}}\}) \times (\mathsf{N}_{\text{OR}} \setminus \mathsf{N}_{\mathbf{A}} \cap \{P \mid (P,S) \notin L_{\mathbf{A}}\})$$

The nodes $P_2, \ldots, P_{\ell-1} \leftarrow N_{\text{OR}}$ are drawn uniformly at random. Then, S_4 sends $(\mathsf{send}, \langle P_i \rangle_{i=1}^{\ell}, [m])$ to U'.

Game_4 is indistinguishable from Game_3 as the distribution of compromised parties remains the same, and in Game_4 the modified wrapper U' together with the simulator S_4 have the same input/output behavior as U.

The game Game_5 is the scenario in which $S'_{\mathbf{A}}$ communicates with $\mathcal{B}_{\text{OR}'}$. The simulator does not directly communicate with the environment over the protocol interface anymore but $\mathcal{B}_{\text{OR}'}$ communicates with the environment instead. The simulator S_5 behaves as $S'_{\mathbf{A}}$ in Figure 7.21.

The difference between the input/output behavior of $\mathcal{B}_{\text{OR}'}$ and the part of S_4 that communicates with U' is minimal. Only for the cases in which the last onion router is not compromised and the last link is not observed the message m is not sent to U'. In these cases $S'_{\mathbf{A}}$ chooses $m_{\texttt{dummy}}$ as a message. But as the ideal functionality does not reveal any information about m if the last node is not compromised, Game_5 and Game_4 are indistinguishable. □

Extending $\mathcal{B}_{\text{OR}'}$ to reusing circuits. Reusing a circuit, in particular accepting answers, raises the problem that the attacker might learn something by observing activities at the same places. This problem suggests that the resulting abstraction cannot be much simpler than abstraction $\mathcal{F}_{\text{OR}}$.

7.7.2. Forward Secrecy

Forward secrecy [DOW92] in cryptographic constructions ensures that a session key derived from a set of long-term public and private keys will not be compromised once the

upon the first input m
  set $N_{\mathbf{A}} := \emptyset$
  set $L_{\mathbf{A}} := \emptyset$
  send N_{OR} to $\mathcal{F}_{\text{REG}}^{\mathcal{N}}$ in $\mathcal{F}_{\text{OR}}$
  send setup to $\mathcal{F}_{\text{OR}}$
  wait for $(\mathsf{ready}, \langle P_i \rangle_{i=1}^{n})$
  further process m

upon $(\texttt{compromise}, P)$ **from A**
  if all previous messages were only compromise or observe messages **then**
    set $N_{\mathbf{A}} := N_{\mathbf{A}} \cup \{P\}$
    forward $(\texttt{compromise}, P)$ to party P in $U(\mathcal{F}_{\text{OR}})$

upon $(\mathsf{observe}, P_1, P_2)$ **from A to** $\mathcal{F}_{\text{NET}^q}$
  if all previous messages were only compromise or observe messages **then**
    set $L_{\mathbf{A}} := L_{\mathbf{A}} \cup \{(P_1, P_2)\}$
    forward $(\mathsf{observe}, P_1, P_2)$ to $\mathcal{F}_{\text{NET}^q}$

upon the first message m **that is not** compromise **from A**
  send $(\mathit{compromise}, N_{\mathbf{A}}, L_{\mathbf{A}})$ to $\mathcal{B}_{\text{OR}'}$
  further process m

upon any other message m **from A to** $\mathcal{F}_{\text{NET}^q}$
  forward m to $\mathcal{F}_{\text{NET}^q}$

upon any other message m **from A to** $U'(\mathcal{F}_{\text{OR}})$
  forward m to $U'(\mathcal{F}_{\text{OR}})$

upon a message m **from** $U'(\mathcal{F}_{\text{OR}})$ **to the environment**
  do nothing /* $\mathcal{B}_{\text{OR}'}$ already outputs the message */

upon $(\mathsf{sent}, U, S, [m])$ **from** $\mathcal{B}_{\text{OR}'}$
  choose $(P_1, P_\ell) \leftarrow$
    $(\mathsf{N}_{\mathbf{A}} \cup \{P \mid \exists Q.(P,Q) \in L_{\mathbf{A}}\})$
    $\times(\mathsf{N}_{\mathbf{A}} \cup \{P \mid \exists Q.(P,Q) \in L_{\mathbf{A}}\})$
  choose $P_2, \ldots, P_{\ell-1} \leftarrow N_{\text{OR}}$
  send $(\mathsf{send}, \langle P_i \rangle_{i=1}^{\ell}, [m])$ to $U'(\mathcal{F}_{\text{OR}})$

upon $(\mathsf{sent}, -, S, [m])$ **from** $\mathcal{B}_{\text{OR}'}$
  choose $(P_1, P_\ell) \leftarrow$
    $((\mathsf{N}_{\text{OR}} \setminus \mathsf{N}_{\mathbf{A}}) \cap \{P \mid (U,P) \notin L_{\mathbf{A}}\})$
    $\times(\mathsf{N}_{\mathbf{A}} \cup \{P \mid (P,S) \in L_{\mathbf{A}}\})$
  choose $P_2, \ldots, P_{\ell-1} \leftarrow N_{\text{OR}}$
  send $(\mathsf{send}, \langle P_i \rangle_{i=1}^{\ell}, [m])$ to $U'(\mathcal{F}_{\text{OR}})$

upon $(\mathsf{sent}, U, -)$ **from** $\mathcal{B}_{\text{OR}'}$
  choose $(P_1, P_\ell) \leftarrow$
    $(\mathsf{N}_{\mathbf{A}} \cup \{P \mid (U,P) \in L_{\mathbf{A}}\})$
    $\times(\mathsf{N}_{\text{OR}} \setminus \mathsf{N}_{\mathbf{A}} \cap \{P \mid (P,S) \notin L_{\mathbf{A}}\})$
  choose $P_2, \ldots, P_{\ell-1} \leftarrow N_{\text{OR}}$
  send $(\mathsf{send}, \langle P_i \rangle_{i=1}^{\ell}, [m_{\texttt{dummy}}])$ to $U'(\mathcal{F}_{\text{OR}})$

upon $(\mathsf{sent}, -, -)$ **from** $\mathcal{B}_{\text{OR}'}$
  choose $(P_1, P_\ell) \leftarrow$
    $(\mathsf{N}_{\text{OR}} \setminus \mathsf{N}_{\mathbf{A}} \cap \{P \mid (U,P) \notin L_{\mathbf{A}}\})$
    $\times(\mathsf{N}_{\text{OR}} \setminus \mathsf{N}_{\mathbf{A}} \cap \{P \mid (P,S) \notin L_{\mathbf{A}}\})$
  choose $P_2, \ldots, P_{\ell-1} \leftarrow N_{\text{OR}}$
  send $(\mathsf{send}, \langle P_i \rangle_{i=1}^{\ell}, [m_{\texttt{dummy}}])$ to $U'(\mathcal{F}_{\text{OR}})$

Figure 7.21.: The simulator $S'_{\mathbf{A}}$: U' gets the path as input instead of drawing it at random

session is over, even when one of the (long-term) private keys is compromised in the future. Forward secrecy in onion routing typically refers to the privacy of a user's circuit against an attacker that marches down the circuit compromising the nodes until he reaches the end and breaks the user's anonymity.

It is commonly believed that for achieving forward secrecy in OR protocols it is sufficient to securely erase the local circuit information once a circuit is closed, and to use a key exchange that provides forward secrecy. Π_{OR} uses such a mechanism for ensuring forward secrecy. Forward secrecy for OR, however, has never been proven, and not even rigorously defined.

In this section, we present a game-based definition for OR forward secrecy (Definition 95) and show that Π_{OR} satisfies our forward secrecy definition (Lemma 51). We require that a local attacker does not even learn anything about a closed circuit if he compromises all system nodes. The absence of knowledge about a circuit is formalized in the notion of *OR circuit secrecy* (Definition 95), a notion that might be of independent interest.

Recall that we formalize a local attacker by considering Π_{OR} in the $\mathcal{F}_{\text{NET}^0}$-hybrid model, i.e., the attacker cannot observe the link between any pair of nodes without compromising any of the two nodes.

Definition 92 (Local attackers)**.** *We say that we consider* a protocol Π against local

$\mathsf{CS\text{-}Ch}_b^\Pi$: (setup) **from A**
  if initial $= \perp$ **then**
    send (setup) to Π
    challenge $\leftarrow$ *false*
    initial $\leftarrow$ *true*

$\mathsf{CS\text{-}Ch}_b^\Pi$: (compromise, P) **from A**
  if challenge $=$ *false* **then**
    store that P is compromised
    forward (compromise, P) to Π

$\mathsf{CS\text{-}Ch}_b^\Pi$: (close_initial) **from A**
  challenge $\leftarrow$ *true*

$\mathsf{CS\text{-}Ch}_b^\Pi$: (createcircuit, $\mathcal{P}^0, \mathcal{P}^1, P$) **from A**
  if challenge $=$ *true* **then**
    if $\mathcal{P}^0$ and $\mathcal{P}^1$ visibly coincide **then**
      forward (createcircuit, $\mathcal{P}^b, P$) to Π

$\mathsf{CS\text{-}Ch}_b^\Pi$: **for every other message** m **from A**
  forward m to Π

$\mathsf{CS\text{-}Ch}_b^\Pi$: **for every message** m **from** Π
  if challenge $=$ *true* and
    $m = (\mathsf{created}, \langle P \overset{cid}{\Longleftrightarrow} P_1 \Longleftrightarrow \cdots P_{\ell'} \rangle, P)$
  then
    store P for *cid*
  forward m to **A**

Figure 7.22.: OR Circuit Secrecy Game

$\mathsf{FS\text{-}Ch}_b^\Pi$ behaves exactly like $\mathsf{CS\text{-}Ch}_b^\Pi$ except for the following message:

$\mathsf{FS\text{-}Ch}_b^\Pi$: **upon** (close_challenge) **from A**
  if challenge $=$ *true* **then**
    challenge $\leftarrow$ *false*
    for every circuit *cid* created in the challenge phase **do**
      look up onion proxy P for *cid*; send $(cid, \mathsf{destroy}, P)$ to Π

Figure 7.23.: OR Forward Secrecy Challenger: $\mathsf{FS\text{-}Ch}_b^\Pi$

attackers *if we consider* Π *in the* $\mathcal{F}_{\text{NET}^0}$*-hybrid model.*

The definition of circuit secrecy compares a pair of circuits and requires that the attacker cannot tell which one has been used. Of course, we can only compare two circuits that are not trivially distinguishable. The following notion of *visibly coinciding circuits* excludes trivially distinguishable pairs of circuits. Recall that a visible subpath of a circuit is a maximal contiguous subsequence of compromised nodes.

Definition 93 (Visibly coinciding circuits)**.** *A subsequence* $\langle P_j \rangle_{j=u}^{s}$ *of a circuit* $\langle P_i \rangle_{i=1}^{\ell}$ *is an* extended visible subpath *if* $\langle P_j \rangle_{j=u+1}^{s-1}$ *is a visible subpath or* $s = \ell$ *and* $\langle P_j \rangle_{j=u+1}^{s}$ *is a visible subpath.*

We say that two circuits $\mathcal{P}^0 = \langle P_i^0 \rangle_{i=0}^{\ell^0}$, $\mathcal{P}^1 = \langle P_i^1 \rangle_{i=0}^{\ell^1}$ *are trivially distinguishable if the following three conditions hold:*

(*i*) *the onion proxies* P_0^0, P_0^1 *are not compromised,*

(*ii*) *the sequences of extended visible subpaths of* $\mathcal{P}^0$ *and* $\mathcal{P}^1$ *are the same, and*

(*iii*) *the exit nodes of* $\mathcal{P}^0$ *and* $\mathcal{P}^1$ *are the same, i.e.,* $P_{\ell^0}^0 = P_{\ell^1}^1$.

For the definition of circuit secrecy of a protocol Π, we define a challenger that communicates with the protocol Π and the attacker. The challenger C_b is parametric in $b \in \{0, 1\}$. C_b forwards all requests from the attacker to the protocol except for the createcircuit commands. Upon a createcircuit command C_b expects a pair $\mathcal{P}^0$, $\mathcal{P}^1$ of node sequences, checks whether $\mathcal{P}^0$ and $\mathcal{P}^1$ are visibly coinciding circuits, chooses $\mathcal{P}^b$, and

forwards ($\mathsf{createcircuit}, \mathcal{P}^b$) to the protocol Π. We require that the attacker does not learn anything about visibly coinciding circuits.

A protocol can be represented without loss of generality as an interactive Turing machine that internally runs every single protocol party as a submachine, forwards each messages for a party P to that submachine, and sends every message from that submachine to the respective communication partner. We assume that upon a message (setup), a protocol responds with a list of self-generated party identifiers. The protocol expects for every message from the communication partner a party identifier and reroutes the message to the corresponding submachine. In the following definition, we use this notion of a *protocol*.

Definition 94. *Let* Π *be a protocol and* $\mathsf{CS\text{-}Ch}$ *be defined as in Figure 7.22. An OR protocol has circuit secrecy if there is a negligible function* μ *such that the following holds for all PPT attackers* $\mathbf{A}$ *and sufficiently large* κ

$$\Pr[b \leftarrow \{0,1\}, b' \leftarrow \mathbf{A}(\kappa)^{\mathsf{CS\text{-}Ch}_b^{\Pi}(\kappa)} : b = b'] \leq 1/2 + \mu(\kappa)$$

Forward secrecy requires that even if all nodes are compromised after closing all challenge circuits the attacker cannot learn anything about the challenge circuits.

Definition 95. *Let* Π *be a protocol and* $\mathsf{FS\text{-}Ch}$ *be defined as in Figure 7.23. An OR protocol has circuit secrecy if there is a negligible function* μ *such that the following holds for all PPT attackers* $\mathbf{A}$ *and sufficiently large* κ

$$\Pr[b \leftarrow \{0,1\}, b' \leftarrow \mathbf{A}(\kappa)^{\mathsf{FS\text{-}Ch}_b^{\Pi}(\kappa)} : b = b'] \leq 1/2 + \mu(\kappa)$$

Lemma 49. *$\mathcal{F}_{\mathrm{OR}}$ against local attackers satisfies OR circuit secrecy (see Definition 94).*

Proof. As we consider a local attacker the attacker can only observe the communication with compromised nodes, i.e., a guard sends a message to the first compromised node in a visible subpath. For such messages we distinguish two kinds of scenarios: either the visible subpath contains the exit node or not. If the visible subpath contains the exit node, $\mathcal{F}_{\mathrm{OR}}$ sends to the attacker the visible subpath together with the actual message to be transmitted. As any pair of challenge circuits visibly coincides, the visible subpaths are the same; hence, also the messages of $\mathcal{F}_{\mathrm{OR}}$ are the same for $b = 0$ or $b = 1$.

In the case that the visible subpath does not contain the exit node, the circuit contains an adjacent guard on both sides of the visible subpath. In these cases, $\mathcal{F}_{\mathrm{OR}}$ sends the visible subpath, the command relay, and the *cid* to the attacker. As any pair of challenge circuits visibly coincides, the visible subpaths are the same. As the *cid* is randomly chosen, the distributions of the *cid* is the same in the scenario with $b = 0$ and $b = 1$. Consequently, the distribution of network messages is the same in the scenario with $b = 0$ and $b = 1$. □

The protocol as introduce in Section 7.2.4 presents Π_{OR} as one (sub-)machine for every protocol party. Equivalently, Π_{OR} can be represented as one interactive Turing machine that runs all parties as submachines, upon a message (setup) from the communication partner, sends (setup) to every party, and sends an answer with a list of party identifiers to the communication partner. In the following definition, Π_{OR} is represented as one interactive Turing machine that internally runs all protocol parties.

Lemma 50. Π_{OR} *instantiated with secure OR modules against local attackers satisfies OR circuit secrecy (see Definition 94).*

Proof. By Theorem 12, we know that there is a simulator S such that the communication with $\mathsf{CS\text{-}Ch}_b^{\Pi_{\text{OR}}}$ and $\mathsf{CS\text{-}Ch}_b^{\mathcal{F}_{\text{OR}}+S}$ is indistinguishable for any PPT attacker.[14] An attacker $A^{\mathsf{CS\text{-}Ch}_b^{\mathcal{F}_{\text{OR}}+S}}$ communicating with $\mathsf{CS\text{-}Ch}_b^{\mathcal{F}_{\text{OR}}+S}$ can be represented as $S'(A)^{\mathsf{CS\text{-}Ch}_b^{\mathcal{F}_{\text{OR}}}}$ for a wrapping machine S' that upon every network message runs the simulator S and reroutes the network messages of S to the environment to A. By Lemma 49, $S'(A)$ cannot guess b with significantly more than a probability of $1/2$, hence also not $A^{\mathsf{CS\text{-}Ch}_b^{\mathcal{F}_{\text{OR}}+S}}$. As $\mathsf{CS\text{-}Ch}_b^{\Pi_{\text{OR}}}$ and $\mathsf{CS\text{-}Ch}_b^{\mathcal{F}_{\text{OR}}+S}$ are indistinguishable, we conclude that there is no attacker that can guess b with significantly more than a probability of $1/2$.

□

It is easy to see that in $\mathcal{F}_{\text{OR}}$, once a circuit is closed, all information related to the circuit at the uncompromised nodes is deleted. Therefore, forward secrecy for $\mathcal{F}_{\text{OR}}$ is obvious from the circuit secrecy in Lemma 50. Hence, the following lemma immediately follows.

Lemma 51. Π_{OR} *instantiated with secure OR modules against local attackers satisfies OR forward secrecy (see Definition 95).*

7.8. Conclusion

We have proven that the core cryptographic parts in a OR protocol are a one-way anonymous authenticated key exchange primitive (1W-AKE), and secure onion algorithms. We have presented an improved version of the existing Tor protocol using the efficient *ntor* protocol as a secure 1W-AKE [GSU12] and by proposing provably secure fixes for the Tor onion algorithms with a minimal overhead. We have shown that this improved protocol provides precise security guarantees in a composable setting (UC [Can01]).

We have further presented an elegant proof technique for the analysis of OR protocols, which leverages an OR abstraction $\mathcal{F}_{\text{OR}}$ that is induced by our UC security result. We show that the analysis of OR protocol boils down to the analysis of the abstraction $\mathcal{F}_{\text{OR}}$. As an example we have introduced a definition for forward secrecy of onion routing circuits and shown that $\mathcal{F}_{\text{OR}}$ satisfies this definition. Furthermore, we have proven that our abstraction $\mathcal{F}_{\text{OR}}$ satisfies the black-box criteria of Feigenbaum, Johnson and Syverson [FJS11], which in turn implies that their anonymity analysis also applies to the OR protocol presented in this paper.

It is well known that the UC framework lacks a notion of time; consequently any UC security analysis neglects timing attacks, in particular traffic analysis. A composable security analysis that also covers, e.g., traffic analysis, is an interesting task for future work. Although our work proposes a provably secure and practical next generation Tor network, users' anonymity may still be adversely affected if different users run different versions. Hence it is an important direction for future work to develop a anonymity-preserving methodology for updating OR clients. Michael Backes, Praveen Manoharan, and Esfandiar

[14] Actually, we do not only consider Π_{OR} but Π_{OR} together with the dummy attacker that only reroutes all messages from the environment to the protocol.

Mohammadi introduced a time-sensitive universal composability framework that is tailored towards the analysis of anonymous communication protocols [BMM14]. Their work uses the abstraction of onion routing presented in this work as a motivating example.

Chapter 8.

Ace: An Efficient Key-Exchange Protocol for Onion Routing

[This chapter is based on a paper with Michael Backes and Aniket Kate [BKM12]. I contributed the idea of this work, and I am the main contributor of all parts that occur in this chapter.]

8.1. Motivation

The onion routing (OR) network Tor [Tor03] has been immensely successful as a privacy enhancing technology. It currently employs nearly six thousand dedicated routers (or OR nodes) and serves million of users all over the world. With its recently observed utility as a censorship-resistant tool these numbers are bound to grow swiftly.

While utilizing the Tor network to access the Internet anonymously, a user constructs a *circuit* choosing a small ordered subset of (usually three) OR nodes, such that the chosen nodes route the user's traffic over the path formed. For the anonymity in onion routing, it is important that an OR node should not be able to determine the circuit nodes other than its predecessor and successor, while routing the user's messages. In the OR protocol, the user achieves this property by sending her every message in the form of an *onion*—a message wrapped in multiple layers of symmetric-key encryption (one layer per selected node). The symmetric keys are agreed upon during an initial *circuit construction* phase using a public-key infrastructure (PKI) implemented using a small set of *directory servers* that also provide routing information for the OR nodes to the users. The key cryptographic challenges in the OR protocol are to securely agree upon the symmetric keys, and then to use those to achieve confidentiality and integrity [BGKM12]. In this work, we concentrate on the first challenge.

Tor currently uses an interactive forward-secret key-exchange protocol called the Tor authentication protocol (TAP) in a *telescoping (or multi-pass)* fashion to agree upon the required symmetric keys [DMS04]. However, with its atypical use of an RSA encrypted group element (or pseudonym), TAP is considered to be inefficient. Øverlier and Syverson [vS07] suggested an efficient replacement for TAP (their fourth protocol) using a half-certified Diffie-Hellman (DH) key agreement [MOV97, Sec. 12.6]. Recently Goldberg, Stebila and Ustaoglu showed an attack on the fourth protocol in [vS07] that allows an adversary to impersonate an honest server (a router in Tor) to an honest client (a user in Tor) [GSU12]. They also defined the concept of one-way authenticated key exchange (1W-AKE), fixed the fourth protocol [vS07] to obtain a provably secure construction called the *ntor* protocol, and described its utility towards onion routing. However, while obtaining a provably secure

construction, they sacrificed computational efficiency to a certain extend. In particular, every *ntor* instance requires two *online* discrete logarithmic (*DLog*) exponentiations on the client side and 1.33 exponentiations on the server side in *ntor*, where only one online exponentiation each was required on the both sides in the original fourth protocol in [vS07]. In this paper, we work towards a computationally more efficient 1W-AKE protocol using a practical concession provided by the Tor protocol.

Contributions. We present a novel 1W-AKE protocol *Ace* (anonymous circuit establishment) that achieves an efficiency improvement of 46% at the client-side and of nearly 19% at the server-side, compared to the *ntor* protocol. The crux is to use as a pseudonym two randomly chosen group elements on the client side instead of one. In this way, we are able to use Shamir's multi-exponentiation trick on both the server and the client side, requiring only 1.17 online exponentiations for the key-exchange. These requirements can be further dropped to 1.08 exponentiations in the elliptic curve cryptographic (ECC) setting as it provides *DLog* group inverses for free.

Our requirement of sending two group elements from the client to the server may look an impeding factor in terms of communication. However, thanks to the fixed sized packets (or cells) of size 512 bytes in Tor, two group elements of size 32 bytes each in the ECC setting can easily be accommodated in a single cell. Given that the ECC setting is under consideration for the Tor protocol [Mat12b], our protocol does not affect the practical communication time of Tor circuit construction at all. We also prove *Ace* secure using the definition for 1W-AKE that has been introduced by Goldberg, Stebila, and Ustaoglu [GSU12].

Outline. Section 8.2 discusses the previous work on 1W-AKE. Section 8.3 introduces the *Ace* protocol. Section 8.4 compares the computational efficiency and the message sizes of the *Ace* protocol with the previous protocols. Section 8.5 reviews the security requirements for a 1W-AKE protocol, and shows that *Ace* indeed constitutes a 1W-AKE protocol. Section 8.6 concludes this chapter.

8.2. Background

This section discusses previous work on 1W-AKE protocols. Section 8.2.1 present the current Tor Authentication protocol (TAP). Section 8.2.2 discusses a one-way authentication protocol by Shoup that enriches the DH key exchange with a public-key signature. Section 8.2.3 illustrates the ØS protocol by Øverlier and Syverson, which is efficient but insecure. Section 8.2.4 presents the *ntor* protocol, which fixes the issues of the ØS protocol but is less efficient. In the end, Section 8.2.5 briefly discusses why we do not consider non-interactive key exchange methods.

A comparative overview of these four key exchange protocols and our protocol *Ace* is presented in Figure 8.6.

8.2.1. The current Tor Authentication Protocol

The current Tor authentication protocol (TAP) basically performs a DH key-exchange where the authentication of the server is ensured by encrypting the first DH message g^x under the public key of the server, for a generator g. Using public-key encryption, however, it is inefficient; therefore, a more efficient key exchange is desirable.

8.2.2. The A-DHKE Protocol

Shoup presented a 1W-AKE protocol A-DHKE that relies on public-key signatures [Sho99] and proved A-DHKE secure[1]. In A-DHKE basically, the DH key exchange is enriched in the second message with a signature of the server on the ephemeral key g^x of the client and the ephemeral g^y of the server. The key derivation function is computed as in a DH key exchange. This protocol only needs 1 online exponentiation as in the usual DH key exchange, but it additionally requires the protocol to compute 1 online signature. Therefore, the efficiency of A-DHKE depends upon the efficiency of the signature scheme.

8.2.3. The ØS Protocol

Øverlier and Syverson proposed a series of more efficient key-exchange protocols for future deployment in Tor, culminating in their fourth protocol [vS07]. This fourth protocol basically enhances the DH protocol with a long-term key g^b of the server. Neglecting the session id and the key-confirmation message, the protocol works as follows. The client sends a fresh ephemeral key g^x to the server. The server draws a fresh ephemeral key g^y, computes the session key $(g^x)^{b+y} = g^{x(b+y)}$, and sends g^y back to the client, which compute $(g^b g^y)^x = g^{(b+y)x}$.

An Attack on the ØS Protocol. Unfortunately, there is a man-in-the-middle attack against this protocol [GSU12]. The attacker intercepts the initial message g^x, draws a fresh g^y, and responds with $g^y/g^b = g^{y-b}$, where g^b is the public key of the server. Then, the client computes the session key $(g^b g^{y-b})^x = g^{yx}$ and the attacker computes $(g^x)^y$. Figure 8.6 in the appendix illustrates this attack.

8.2.4. The ntor Protocol

Goldberg, Stebila, and Ustaoglu [GSU12] present a fixed version of the ØS protocol, the *ntor* protocol. Moreover, the authors proved that *ntor* is 1W-AKE secure (see Section 8.5.1).

A closer look at the session key g^{xy+xb} in the ØS protocol reveals that for fixing the protocol it suffices to separate the term xy from the term xb. In *ntor*, this separation is achieved by applying a hash function H to these terms: $H(g^{xy}, g^{xb})$.[2] Neglecting the session keys and the key confirmation message, in *ntor* the client sends a fresh ephemeral key g^x to the server, The server draws a fresh ephemeral key g^y, computes the session key as $H((g^x)^y, (g^x)^b)$, and responds with g^y to the client.

The security of *ntor* is bought at a price of efficiency: the client has to compute 2 full exponentiations, and the server has to compute 1.33 exponentiations, using square-and-multiply optimizations since the base of $(g^x)^y$ and $(g^x)^b$ is the same.

8.2.5. A Note on Non-Interactive KE

In contrast to the presented interactive key-exchange a single-pass construction using a non-interactive key exchange is possible as well. However, achieving forward secrecy of the user's

[1] Technically, Shoup proved A-DHKE secure in another model, but it can be shown that A-DHKE also satisfies the 1W-AKE definition.

[2] g^{xy}, g^{xb} denotes the concatenation of g^{xy} and g^{xb}.

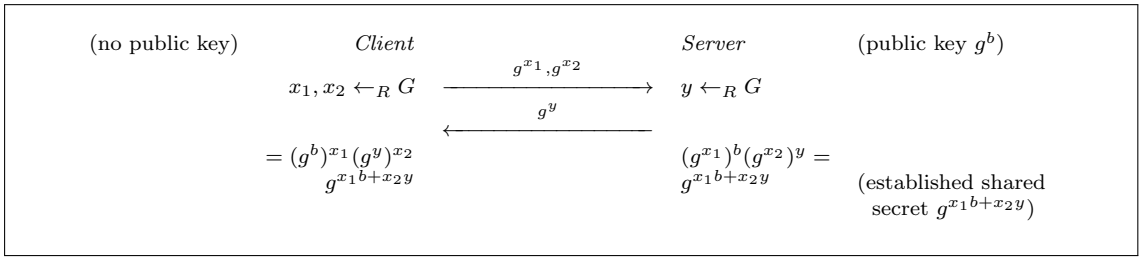

Figure 8.1.: An overview of *Ace*: G is the exponent group.

circuits without regularly rotating the PKI keys for all Tor nodes is not possible [KZG10], and the periodic public key rotation should be avoided for scalability reasons. There has also been attempts to solve this problem by introducing the identity-based setting [KZG07; KZG10] or the certificate-less cryptography setting [CFG09]. However, as discussed in Section 7.2.1, the key authorities required in these constructions can be difficult to implement in practice.

8.3. The Ace Procotol

Along the lines of the ØS protocol and the *ntor* protocol, the *Ace* protocol constitutes a one-way authenticated variant of the DH key exchange. The crux of *Ace* is that it trades computational efficiency with communication efficiency. However, we aim at improving the key-exchange in the Tor protocol, and it turns out that in the ECC setting a large fragment of the Tor packets are actually unused during the key exchange. As the ECC setting is currently under consideration for deployment in Tor, using *Ace* will not produce a communication overhead for the Tor protocol's key-exchange yet gaining efficiency in terms of computation.

Notation. We often denote the long-term secret key of a party A as a and correspondingly g^a as the public key, for a given generator g. In the security analysis and the detailed presented of *Ace*, we also talk about A as an identifier. At these place, we denote the long-term public key of a party A as pk_A and the corresponding secret key as sk_A.

For assigning the result of a (possibly randomized) computation A to a variable v, we write $v \leftarrow A$. Similarly, we write $v \leftarrow V$ to denote the assignation of a value V to the variable v. Moreover, we write $v \leftarrow_R S$ for denoting that a uniformly chosen element from a set S is assigned to the variable v. For the security parameter we use the letter η. In this work, we only consider probabilistic polynomial-time bounded machines, denoted as *ppt machines*. In abuse of notation, we also call a randomized algorithm that is polynomially bounded a *ppt algorithm*. We denote the hash of the concatenation of several values $v_1, \ldots, v_2$ as $H(v_1, \ldots, v_2)$.

8.3.1. The Construction

Recall that in the ØS protocol the client sends an ephemeral key g^x and the server responds with an ephemeral key g^y, resulting in a session key g^{xb+xy} (g^b being the server's certified long-term key). The main problem in ØS is that the two terms xy and xb are not separated, allowing the attacker to choose $y := y' - b$, by computing $g^y := g^{y'}/g^b$, and impersonating

any server. In the *ntor* protocol this separation is achieved by letting the session key be the hash of g^{xb} and g^{xy}. This remedy comes at the price of a loss in efficiency. In *Ace* we achieve this separation by letting the ephemeral key be a pair (g^{x_1}, g^{x_2}): the session key is then $g^{x_1 b + x_2 y}$.

In *Ace* the client sends an ephemeral pair (g^{x_1}, g^{x_2}) and the server responds an ephemeral key g^y. Then, the client computes the session key as $(g^b)^{x_1}(g^y)^{x_2} = g^{bx_1+yx_2}$ and the server as $(g^{x_1})^b(g^{x_2})^y = g^{x_1 b + x_2 y}$. Figure 8.1 gives an overview of the *Ace* protocol.

The Protocol in Detail. Figure 8.2 presents the *Ace* protocol in detail by showing the pseudo code for the initialization algorithm *Init*, the response algorithm *Resp*, and the final key computation algorithm *CompKey*. Let $(\mathsf{pk}_Q, \mathsf{sk}_Q)$ be the static key pair for Q. We assume that P holds Q's certificate (Q, pk_Q). Recall that cs is the queue of already chosen ephemeral key pairs (x, g^x) of which g^x is already leaked to the attacker.

The algorithm *Init* is called for initiating a new session. It expects as input the server's identity, two strings `new_session`, *Ace*, and the queue cs. Then, either a fresh ephemeral keys is chosen or an already chosen key is popped from the queue cs. Thereafter, the local session identifier is set by applying the ephemeral pair (x_1, x_2) to a collision resistant hash function H_{st}.[3] Then, the session information of the client is stored in the variable $\mathsf{st}(\Psi)$, and the client's ephemeral keys (g^{x_1}, g^{x_2}) together with the server identity Q and a string *Ace* is output as a network message m along with the session's state $\mathsf{st}(\Psi)$ and the session identifier Ψ.

The algorithm *Resp* is called by the server Q for responding to a session initialization. *Resp* expects as input the secret key sk_Q of Q, Q's identity, the network message (*Ace*, X_1, X_2) from the initialization, and the queue cs. First, again a ephemeral key (y, g^y) is chosen, either freshly or from cs. Then, it is verified that X_1, X_2 are in the public key group G_η^*. Thereafter, the local session identifier Ψ_Q is computed by applying the hash function H_{st} to the ephemeral key g^y. Thereafter, the two keys k_m, k_s are derived from $g^{x_1 \mathsf{sk}_Q + x_2 y}$ by applying the hash function H to $g^{x_1 \mathsf{sk}_Q + x_2 y}$, the session information $X_1, X_2, Y, \mathsf{pk}_Q$, and the protocol name *Ace*. Since we need k_m and k_s for the proof to be independent, we consider H is a random oracle in the proof. k_m is used for the key-confirmation message, and k_s is the resulting session key. Then, the key confirmation message is computed by basically applying a *Mac* to all public information using the key k_m. Thereafter, we output the session identifier, the session information, and as a network message the *Mac* tag, the ephemeral key g^y and a string *Ace*.

The algorithm *CompKey* is called by the client for completing the key-exchange. *CompKey* expects the public key of the server Q, the network message (Ace, Y, t_Q) from Q, and the temporary session state $(Q, (x_1, x_2), (g^{x_1}, g^{x_2}))$. First, we verify that Y is indeed a group element of G_η^*. Then, we compute the key confirmation key k_m and the session key k_s similar as in *Resp*, and verify the key confirmation message t_Q from the server with k_m. Thereafter, we output the session information $(k_s, Q, (X_1, X_2), (Y, \mathsf{pk}_Q))$.

[3] The purpose of this hash function is merely to reduce the space of the session key; therefore, we only need to require collision resistance.

Init$(Q, (\eta, \texttt{new\_session}, Ace), \mathsf{cs})$**:**
  if $\mathsf{cs} = \emptyset$ **then**
    $x_1, x_2 \leftarrow_R G_\eta$
    compute g^{x_1},g^{x_2}
  else
    $(x_1, g^{x_1}), (x_2, g^{x_2}) \leftarrow \mathsf{pop}(\mathsf{cs})$
  set session id $\Psi \leftarrow H_{\mathsf{st}}(g^{x_1}, g^{x_2})$
  set $\mathsf{st}(\Psi) \leftarrow (Q, (x_1, x_2), (g^{x_1}, g^{x_2}))$
  set $m \leftarrow (Ace, Q, (g^{x_1}, g^{x_2}))$
  output $(m, \mathsf{st}(\Psi), \Psi)$

Resp$(\mathsf{sk}_Q, Q, (\eta, Ace, X_1, X_2), \mathsf{cs})$**:**
  if $\mathsf{cs} = \emptyset$ **then** $y \leftarrow_R G_\eta$ **else** $y \leftarrow \mathsf{pop}(\mathsf{cs})$
  verify that $X_1, X_2 \in G_\eta^*$
  set session id $\Psi_Q \leftarrow H_{\mathsf{st}}(g^y)$
  compute $(k_m, k_s) \leftarrow H(X_1^{\mathsf{sk}_Q} \cdot X_2^y, g^{x_1}, g^{x_2}, g^y, g^b, Ace)$
  compute $t_Q \leftarrow Mac(k_m, (Q, g^y, X_1, X_2, Ace, \texttt{server}))$
  set $m_Q \leftarrow (Ace, g^y, t_Q)$
  $out \leftarrow (k_s, \star, (X_1, X_2), (g^y, g^{\mathsf{sk}_Q}))$
  output (m_Q, out, Ψ_Q)

CompKey$(\mathsf{pk}_Q, (\eta, Ace, Y, t_Q), \Psi, (Q, (x_1, x_2), (g^{x_1}, g^{x_2})))$**:**
  verify that $Y \in G_\eta^*$
  compute $(k_m, k_s) \leftarrow H(\mathsf{pk}_Q^{x_1} \cdot Y^{x_2}, g^{x_1}, g^{x_2}, g^y, g^b, Ace)$
  if $Mac(k_m, (Q, Y, g^{x_1}, g^{x_2}, Ace, \texttt{server})) = t_Q$ **then**
    $out \leftarrow (k_s, Q, (g^{x_1}, g^{x_2}), (Y, \mathsf{pk}_Q))$
    output out

If any verification fails, the party erases all session-specific information and aborts the session.

Figure 8.2.: The *Ace* protocol: G_η is the group of the secret keys, G_η^* is the group of the public keys. $Gen(1^\eta)$ outputs a pair (x, g^x) for a random element x of G_η.

8.4. Performance Comparison

In this section, we compare the performance of the *Ace* protocol with the relevant key agreement protocols.

We consider $\eta = 128$-bit security and use the elliptic curve cryptographic (ECC) setting with points (compressed form) of size $p = 256$ bits, such as provided by Dan Bernstein's Curve25519 [Ber06]. For the finite field setting ($\mathbb{F}$), we consider a DH modulus of size just $p = 2048$ bits to model 128-bit security. In these setting, we compare computational efficiency and message sizes of our protocol with the TAP protocol, the A-DKHE protocol [Sho99], the fourth protocol by Øverlier and Syverson, the multi-pass pairing-based onion routing (PB-OR) protocol [KZG10] and the *ntor* protocol.

8.4.1. Computational Efficiency

Table 8.1 compares computational efficiency and security of the above mentioned relevant key exchange schemes. We also include the unauthenticated and insecure Diffie-Hellman (DH) key exchange protocol to set the baseline for the required computation, where one (online) exponentiation is enough on both client and server sides. The TAP protocol also

Table 8.1.: Comparison between computational cost of relevant key exchange schemes for 128-bit security

Protocol	**Exponentiations (client)**		**Exp. (server)**		**Security**
	Off-line	**On-line**	**Off-line**	**On-line**	
DH	1	1	1	1	insecure
A-DHKE [Sho99]	1	$1 + 1^*$	1	$1 + 1^*$	secure
TAP [DMS04]	1	$1 + 1^\dagger$	1	$1 + 1^\dagger$	secure
ØS [vS07]	1	1	1	1	insecure
Multi-pass PB-OR [KZG10]	1	$1 + 1^\ddagger$	1	$1 + 1^\ddagger$	secure
ntor [GSU12]	1	2	1	1.33	tight
Ace **(this paper)**	2	1.08(1.17)	1	1.08(1.17)	tight

* A-DHKE requires a signature generation on the server side and a signature verification on the client side.
† TAP requires an RSA encryption on the client side and an RSA decryption on the server side.
‡ Multi-pass PB-OR requires a bilinear pairing on both client and server sides.

requires one exponentiation on both sides; however, it requires one RSA encryption on the client side and one RSA decryption on the server side, and the latter operation increases the server-side computational cost significantly.

The fourth protocol by Øverlier and Syverson is although as efficient as the unauthenticated DH key exchange, it is insecure. The *ntor* protocol requires two exponentiations on both client and server sides; however, the two exponentiations on the server side use the same base, and they can be parallelized [MN96] to reduce the computational cost to 1.33 exponentiations for $\eta = 128$. Although our *Ace* protocol naïvely also requires two exponentiations on both client and server sides, exponentiations on both sides are actually multi-exponentiations and using Shamir's trick [MOV97, Algo. 14.88] can be reduced to only 1.17 exponentiations on both sides for $\eta = 128$. In the ECC setting, where group inverses come for free, the number of exponentiations can be further reduced to 1.08 exponentiations using Avanzi's algorithm [Ava05] based on a sliding windows method for the joint sparse form [Sol01].

The A-DHKE protocol uses one signature generation on the server side and one signature verification on the client side alone with one exponentiation on each side for the session-key computation, and its efficiency depends upon the efficiency of the signature scheme used. In our ECC Curve25519 setting, the signature generation is expensive and A-DHKE is significantly inefficient than the *Ace* protocol. However, Bernstein et al. find that high-speed signatures are possible using table lookups and a twisted Edwards curve [BDLSY11]. Using this signature scheme, server-side computation for A-DHKE may become nearly equal to a single exponentiation. (See the discussion on the tor-dev mailing list [Mat12a].) Nevertheless, we observe that the multi-exponentiation techniques used in the Ace protocol can also benefit from table lookups; hence, the performance of Ace protocol will remain comparable to A-DHKE over the twisted Edwards curve.

We also include the multi-pass PB-OR protocol in our comparison for completeness. It asks for a bilinear pairing along with an exponentiation on both sides. However, the protocol belongs to the identity-based setting and the capability of the Tor network to implement the required setup assumptions is not clear.

upon $\texttt{send}^P(\mathsf{params}, Q)$**:**
  $(\texttt{msg}, \mathsf{st}, \Psi) \leftarrow \mathit{Init}(Q, \mathsf{params}, \mathsf{cs})$
  $\mathsf{akest}^P(\Psi) \leftarrow (Q, \mathsf{st})$; send $(\texttt{msg}, \Psi)$

upon $\texttt{send}^P(\Psi, \texttt{msg})$ **and** $\mathsf{akest}^P(\Psi) = \perp$**:**
  $(\texttt{msg}', (k, \star, \mathsf{st}), \Psi) \leftarrow \mathit{Resp}(\mathsf{sk}_P, P, \texttt{msg}, \mathsf{cs})$
  $\mathsf{resust}^P(\Psi) \leftarrow (k, \star, \mathsf{st})$
  send $\texttt{msg}'$

upon $\texttt{send}^P(\Psi, \texttt{msg})$ **and** $\mathsf{akest}^P(\Psi) \neq \perp$**:**
  $(Q, \mathsf{st}) \leftarrow \mathsf{akest}^P(\Psi)$; check for a valid pk_Q
  $(k, Q, \mathsf{st}) \leftarrow \mathit{CompKey}(\mathsf{pk}_Q, \texttt{msg}, \Psi, (Q, \mathsf{st}))$
  erase $\mathsf{akest}^P(\Psi)$; $\mathsf{resust}^P(\Psi) \leftarrow (k, Q, \mathsf{st})$

upon $\texttt{reveal\_next}^P$**:**
  $(x, X) \leftarrow \mathit{Gen}(1^\eta)$; append (x, X) to cs
  send X

upon $\texttt{partner}^P(X)$**:**
  if a key pair (x, X) is in the memory
    then send x

upon $\texttt{sk\_reveal}^P(\Psi)$**:**
  if $\mathsf{resust}^P(\Psi) = (k, Q, \mathsf{st})$ **then** send k

upon $\texttt{establish\_certificate}(Q, \mathsf{pk}_Q)$**:**
  register the public key pk_Q for the party Q

upon $\texttt{test}^P(\Psi)$**:** *(one time query)*
  $(k, Q, \mathsf{st}) \leftarrow \mathsf{resust}^P(\Psi)$
  if $k \neq \perp$ and $Q \neq \star$ and Ψ is 1W-AKE fresh **then**
    if $b = 1$
      then send k
      else send $k' \leftarrow_R \{0,1\}^{|k|}$

for every query (consistency check)
  if in a client session only one key X_i is partnered **then**
    send X_{3-i}

Figure 8.3.: 1W-AKE Security Challenger: $\mathit{Ch}^{\mathrm{KE}}{}_b(1^\eta)$. If any invocation outputs $\perp$, the challenger erases all session-specific information for that session and aborts that session.

8.4.2. Message Sizes

All of the above discussed relevant key exchange protocols except our *Ace* protocol require one group element to be communicated from the client to the server.[4] For $\eta = 128$, this asks for 256 bytes in the finite field setting, and 32 bytes in the ECC setting. In our *Ace* protocol, the client communicates two group elements to the server. In the finite field setting, this asks for 512 bytes in the finite field setting, and 64 bytes in the ECC setting.

However, the Tor uses cells of size 512 bytes, and in the ECC setting, sending 64 bytes instead of 32 bytes does not affect the Tor protocol. As the ECC setting is under consideration for the Tor protocol [Mat12b], we find our protocol to be more aptly suited to replace TAP instead of *ntor*.

8.5. Security Analysis

This sections presents a security analysis of *Ace*. First, Section 8.5.1 reviews the security requirements for 1W-AKE protocol. Second, Section 8.5.2 discusses the security of *Ace*.

8.5.1. Security Definition of Anonymous 1W-AKE

We already presented these definitions in Section 7.4.3 but recall them here for the sake of readability. Goldberg, Stebila, and Ustaoglu [GSU12] formalize the security of a 1W-AKE protocol between an anonymous client and an authenticated server by requiring the following three properties. First, the protocol should produce correct results if both parties are honest (*correctness*). Second, even a malicious attacker that can compromise

[4] Note that the client communicates an RSA-encrypted group element in TAP.

single sessions and introduce fake identities cannot learn anything about the session key of uncompromised sessions (*1W-AKE security*). In particular, 1W-AKE security implies that the attacker cannot impersonate a server. Third, a server should not be able to see any difference while communicating with two different clients (*1W-anonymity*). In this section, we review the three notions Correctness, 1W-AKE security, and 1W-anonymity.

A 1W-AKE protocol is a tuple of ppt algorithms $AKE = (Gen, Init, Resp, CompKey)$. *Gen* is called for generating temporary asymmetric keys, *Init* is called at the client for starting a 1W-AKE, *Resp* is called at the server for responding to a 1W-AKE initialization, and *CompKey* is again called at the client for verifying the key confirmation message and computing the key. We assume that every party $P \in \{P_1, \ldots, P_n\}$ can register public keys, and every party can obtain certificates for other parties' public keys and verify them.

Correctness of 1W-AKE. Correctness states that if all parties behave honestly, the protocol succeeds. Also, the correctness property requires the 1W-AKE algorithms to finally output a vector $\vec{v} = (v_1, v_2)$ that contains all ephemeral information and the long-term public key. For *Ace*, $\vec{v} = ((g^{x_1}, g^{x_2}), (g^y, g^{\mathsf{sk}_P}))$, where x_1, x_2 are the ephemeral secret keys of the client for that session, y is the ephemeral secret key of the server $P \in \{P_1, \ldots, P_n\}$ for that session, and sk_P is the secret long-term key of the server. Moreover, the 1W-AKE algorithms output the session key k_s, and the ID of the peer party, where the client outputs the actual ID of the server, and the server only outputs $\star$, since the client is anonymous.

Definition 89. (Correctness of 1W-AKE) *Let a PKI be given, i.e., for every party $P \in \{P_1, \ldots, P_n\}$ every party knows a (certified) public key pk_P and P itself also knows the corresponding secret key sk_P. Let $AKE := (Gen, Init, Resp, CompKey)$ be a tuple of polynomial-time bounded randomized algorithms. We say that AKE is a* correct one-way authenticated key exchange *protocol if the following holds for all parties A, B:*

$$\begin{aligned}\Pr\big[&(\mathit{msg}, \mathsf{st}, \Psi) \leftarrow \mathit{Init}(Q, m, \mathsf{cs}),\\ &(\mathit{msg}', (k, \star, \vec{v}), \Psi_Q) \leftarrow \mathit{Resp}(\mathsf{sk}_Q, Q, \mathit{msg}, \mathsf{cs}),\\ &(k', Q, \vec{v}') \leftarrow \mathit{CompKey}(\mathsf{pk}_Q, \mathit{msg}', \Psi, \mathsf{st})\\ &: k = k' \text{ and } \vec{v} = \vec{v}'\big] = 1.\end{aligned}$$

1W-AKE Security. We require that the attacker does not learn anything about the key and is not able to impersonate honest parties. This notion is formalized by requiring that even in the presence of an attacker that can send commands to each party, establish several concurrent sessions, compromise servers and issue fake identities servers, cannot learn a single bit of each party's session key once the key exchange is successfully completed.

More precisely, we construct a ppt machine Ch^{KE}, called the challenger, that represents honest parties ($P_1, \ldots, P_n$) and allows the attacker a fixed set of queries (see Figure 8.3). This challenger internally runs the 1W-AKE algorithms AKE. The definition basically states that an attacker breaks the 1W-AKE security if in the end it successfully distinguishes a randomly chosen session key from the actually established session key for an uncompromised session Ψ. For this challenge, the attacker sends a query $\mathtt{test}^P(\Psi)$ to Ch^{KE}. For triggering the initiation of a session, triggering the response to a key exchange, and for completing a key exchange, Ch^{KE} allows the attacker to send a query $\mathtt{send}^P(m)$. For

compromising parties, the attacker can query three different types of messages. First, the attacker can ask party P to reveal the next public key that will be chosen with the query $\texttt{reveal\_next}^P$. Second, the attacker can ask for a secret key for a corresponding public key X using the query $\texttt{partner}^P(X)$. Third, the attacker can ask for the session key of a session Ψ with the query $\texttt{sk\_reveal}^P(\Psi)$. Finally, the attacker can also register new long-term public keys pk_Q for unused identities Q with the query $\texttt{establish\_certificate}(Q, \mathsf{pk}_Q)$.

The challenger Ch^{KE} maintains several variables for every party P. A variable v for a party P is denoted as v^P. First, the challenger maintains the key exchange state $\mathsf{akest}^P(\Psi)$ for a party P and a session Ψ. This key exchange state stores the ephemeral secret keys that will be erased after the key exchange is completed. Then, Ch^{KE} gets as input the public parameters params, typically containing the security parameter η and the name of the protocol. The challenger furthermore maintains for every party P the result state $\mathsf{resust}^P(\Psi)$ of a completed session Ψ. This result state contains the established key, the peer party, which is $\star$ for the server P since the client is anonymous, and a state st that typically contains two vectors v_1, v_2 that contain the ephemeral public keys and the long-term keys used for establishing the session key of Ψ. In the case of *ntor*, v_1 contains the client's ephemeral key $X = v_1$ and v_2 contains the server's long-term key B and ephemeral key Y, i.e., $(Y, B) = v_2$. Recall that in the case of *Ace* v_1 contains the two ephemeral keys $(X_1, X_2) = v_1$ and v_2 is the same as in *ntor*, i.e., $(Y, B) = v_2$.

For characterizing those secret keys that are used in a key exchange and have not been leaked to the attacker yet, we introduce the notion of the attacker not being a *partner* to a ephemeral public key X. Formally, the attacker is a partner for a public value X if one of the following conditions hold true.

- X was not used yet.
- X is public key that the attacker registered using the query $\texttt{establish\_certificate}(Q, X)$.
- X was the response of a query $\texttt{send}^P$ or $\texttt{reveal\_next}^P$ and there is a successive query $\texttt{partner}^P(X)$.

We stress that unused values also include all values that are only chosen by the attacker; hence the attacker is with overwhelming probability a partner to all self-chosen values.

Moreover, we assume that if an attacker learns from a client one ephemeral key X_i of a session Ψ, then the attacker also learns the other ephemeral key X_{3-i} of that session. We ensure this by making a consistency check for all partnered values for every query. Even though this modification is particular to key exchange protocol in which the client sends two ephemeral keys, this modification looks natural to us.

Goldberg, Stebila, and Ustaoglu proposed a freshness notion for the challenge session, in order to prevent the attacker from trivially winning the game. We call their freshness condition *single value 1W-AKE freshness*. We say that a session Ψ at a party P is *single value 1W-AKE fresh* if the following two conditions hold:

1. Let $(k, Q, \mathsf{st}) \leftarrow \mathsf{resust}^P(\Psi)$ (see Figure 8.3). For every vector v_j in st there is at least one element X in v_j such that the attacker $\mathcal{A}$ is not a partner to X.
2. For the session Ψ such that $\mathsf{akest}^P(\Psi) = (v, Q)$, the adversary did not issue $\texttt{sk\_reveal}^Q(\Psi')$ for any Ψ' such that $\mathsf{akest}^Q(\Psi') = (v, \star)$.

The protocol *Ace* presented in this work, uses two ephemeral keys for the client, i.e.,

upon $\texttt{start}(i, j, params, Q)$**:** *(one time query)*
  if $i \neq j$ **then**
    if $b = 1$ **then** $i^* \leftarrow i$ **else** $i^* \leftarrow j$
    send $\texttt{send}^{P_{i^*}}(\mathsf{params}, Q)$ to $I\,Ch^{\mathrm{KE}}{}_1(1^\eta)$
    wait for the response $(\Psi^*, \texttt{msg}')$
    send $\texttt{msg}'$ to $\mathcal{A}$

upon $\texttt{send}(\texttt{msg})$**:**
  forward $\texttt{send}^{P_{i^*}}$ to $I\,Ch^{\mathrm{KE}}{}_1(1^\eta)$

upon `reveal_next`**:**
  forward $\texttt{reveal\_next}^{P_{i^*}}$ to $I\,Ch^{\mathrm{KE}}{}_1(1^\eta)$

upon `sk_reveal`**:**
  forward $\texttt{sk\_reveal}^{P_{i^*}}(\Psi^*)$ to $I\,Ch^{\mathrm{KE}}{}_1(1^\eta)$

upon $\texttt{partner}(X)$**:**
  forward $\texttt{partner}^{P_{i^*}}(X)$ to $I\,Ch^{\mathrm{KE}}{}_1(1^\eta)$

Figure 8.4.: The anonymizing machine $Ch^{\mathrm{AN}}{}_b(1^\eta)$: $I\,Ch^{\mathrm{KE}}{}_1(1^\eta)$ is an internally emulated copy of $Ch^{\mathrm{KE}}{}_1(1^\eta)$

$v_1 = (g^{x_1}, g^{x_2})$ and as *ntor* one ephemeral key g^y for the server and the long-term key g^b of the server, i.e., $v_2 = (g^y, g^b)$. The key is then deterministically derived from $g^{bx_1+yx_2}$. Since $g^{bx_1+yx_2}$ is computable for any attacker that is a partner to the pair (x_2, b) or the pair (x_1, y), we need to exclude these cases in order to prevent the attacker from trivially winning. We say that a session is *double value 1W-AKE fresh* if it is single value 1W-AKE fresh and the following condition holds:

3. Let $(k, Q, ((X_1, X_2), (Y, B))) \leftarrow \mathsf{resust}^P(\Psi)$. The attacker is not a partner of the pair (X_1, Y) or (X_2, B).

We call a double value 1W-AKE fresh session a *fresh 1W-AKE* session.

As depicted in Figure 8.3, we consider two challengers $Ch^{\mathrm{KE}}{}_0$ and $Ch^{\mathrm{KE}}{}_1$. $Ch^{\mathrm{KE}}{}_0$ sends a randomly chosen key as a response, and $Ch^{\mathrm{KE}}{}_1$ sends the actually established key as a response. Upon successful key exchange with a server Q, a key k, and the transcript v_1, v_2, a client outputs a tuple $(k, Q, (v_1, v_2))$. A server outputs $(k, \star, (v_1, v_2))$ for denoting that the peer party is anonymous.

Definition 90. (1W-AKE-security) *Let η be the security parameter. A protocol π is said to be* 1W-AKE-secure *if, for all ppt adversaries $\mathcal{A}$, the following difference is negligible in η:*

$$\begin{aligned}&|\Pr[b^* \leftarrow \langle A(1^\eta), Ch^{\mathrm{KE}}{}_0(1^\eta)\rangle : b^* = 1]\\ &-\Pr[b^* \leftarrow \langle A(1^\eta), Ch^{\mathrm{KE}}{}_1(1^\eta)\rangle : b^* = 1]|\end{aligned}$$

1W-Anonymity. A one-way authenticated key exchange is able to provide anonymity for the unauthorized client; this client-side anonymity is called *1W-anonymity*. Formally, 1W-anonymity means that the attacker cannot link a key exchange through an anonymized channel (e.g., Tor) with a key exchange through a direct connection. More formally, we consider the following scenario.

The attacker can communicate with all parties directly, which is modeled by the 1W-AKE challenger $Ch^{\mathrm{KE}}{}_1$ (see Figure 8.3). In addition, the attacker chooses two candidate parties for a key exchange challenge session Ψ^* over an anonymous channel. This anonymous channel is modeled by the ppt machine Ch^{AN} (presented in Figure 8.4). Ch^{AN} selects one of the two candidate parties. Finally, the attacker has to guess which of the two parties has been selected in the challenge session.

In order to prevent the attacker from trivially learning the identity of the correct candidate, we have to exclude the cases in which the attacker peaks into the state of the candidate parties. Formally, we require that Ch^{AN} internally runs a copy of the 1W-AKE challenger Ch^{KE}. We denote the internal copy of Ch^{KE} as $I\,Ch^{\mathrm{KE}}$ (see Figure 8.4).

Definition 91. (1W-anonymity) *Let η be the security parameter. Let M, N be ppt interactive turing machines. Let $v \leftarrow \langle A(1^\eta), M(1^\eta), N(1^\eta)\rangle$ denote the interaction between A and M and A and N and v be the output of A. A protocol AKE is said to be* 1W-anonymous *if, for all PPT adversaries $\mathcal{A}$, the following difference is negligible in η*

$$\begin{aligned}&|\Pr[b^* \leftarrow \langle A(1^\eta), {Ch^{\mathrm{AN}}}_0(1^\eta), {Ch^{\mathrm{KE}}}_1(1^\eta)\rangle : b^* = 1]\\ &-\Pr[b^* \leftarrow \langle A(1^\eta), {Ch^{\mathrm{AN}}}_1(1^\eta), {Ch^{\mathrm{KE}}}_1(1^\eta)\rangle : b^* = 1]|\end{aligned}$$

8.5.2. The Security of Ace

At this point, we are able to analyze the security of *Ace*. We first show that no information about the session key is leaked by proving 1W-AKE security for *Ace*. Then, we show that a *Ace* session cannot be linked to another *Ace* session by proving 1W-anonymity for *Ace*.

Lemma 52 (*Ace* is 1W-AKE secure)**.** *If H_{st} is a collision resistant hash function, Mac is universally unforgeable against chosen message attacks (CMA-UF), and H is a random oracle, the protocol Ace is 1W-AKE-secure in the sense of Definition 90 under the GDH assumption.*[5]

More precisely, for every machine $\mathcal{A}$ that breaks the 1W-AKE security of Ace with probability μ and runs in time t, there exists a bound $q \geq 1$ on the number of sessions and a machine S_q that breaks the GDH assumption with a probability of more than $\binom{2}{|P|}\mu/(q|P|)$ and runs in time $O(t)$, where $|P|$ is the number of honest servers.

Proof. Let $Game_1$ be the original setup with the ${Ch^{\mathrm{KE}}}_1$ challenger against the attacker $\mathcal{A}$.

$Game_2$ is the faking game with the ${Ch^{\mathrm{KE}}}_0$ challenger: upon a $\mathtt{test}^P(\Psi)$-query ${Ch^{\mathrm{KE}}}_0$ sends a randomly chosen key k instead of the real key k_s for $(k_m, k_s) \leftarrow H(g^{bx_1+yx_2}, g^{x_1}, g^{x_2}, g^y, g^b, Ace)$. Since H is a random oracle k_m is completely independent of k_s; hence, no information about the challenge key k_s is leaked by using k_m for the Mac. Moreover, we show below that by the gap DH assumption $g^{bx_1+yx_2}$ cannot be computed from g^{x_1}, g^{x_2}, g^y, and g^b.

We construct a ppt reduction S_q against the GDH challenger for an attacker $\mathcal{A}$ that distinguishes $Game_1$ from $Game_2$ but only allows q session-queries. Moreover, this reduction S_q also simulates the random oracle. Let $P = \{P_1, \ldots, P_n\}$ be the set of parties. We show that there is a q such that S_q that solves the GDH problem with probability $\binom{2}{|P|}\mu/(q|P|)$ if $\mathcal{A}$ breaks the 1W-AKE security with probability μ, where $q \leq p(\eta)$ and p is the runtime polynomial of the attacker $\mathcal{A}$. Let (g, g^u, g^v) be the GDH challenge. Moreover, the runtime of S_q is asymptotically the same as the runtime of $\mathcal{A}$.

[5] The GDH assumption states that the computational DH assumption holds even against an attacker that has access to a decisional DH oracle [OP01].

S_q: **upon initialization**
ask the GDH challenger for a DH tuple (g, g^u, g^v)
draw $b \leftarrow_R \{0,1\}$
if $b = 1$ **then**
draw $i \leftarrow_R \{1, \ldots, q\}$
else
draw $i \leftarrow_R \{1, \ldots, |P|\}$
replace $\mathsf{pk}_{P_i} = g^b$ of party P_i with g^v
draw $j \leftarrow_R \{1, \ldots q\}$

S_q: **upon** $\mathtt{send}^P(\mathsf{params}, Q)$:
/* if the client is called for the first protocol message */
if $b = 1$ and it is the ith session **then**
replace g^{x_2} with g^u
honestly choose g^{x_1}
if $b = 0 \wedge Q = P_i$ and it is the jth session **then**
replace g^{x_1} with g^u
honestly choose g^{x_2}
proceed as in $(\mathtt{msg}, \mathsf{st}, \Psi)$
$\leftarrow \mathit{Init}(Q, (\eta, \mathtt{new\_session}, \mathit{Ace}), \mathsf{cs})$
/* recall that cs is maintained by the challenger */
$\mathsf{akest}^P(\Psi) \leftarrow (Q, \mathsf{st})$; send $(\mathtt{msg}, \Psi)$

S_q: **upon** $\mathtt{send}^P(\Psi, (\eta, \mathit{Ace}, g^{x_1}, g^{x_2}))$ **and** $\mathsf{akest}^P(\Psi) = \bot$
/* if the server is called */
if $b = 1$ and Ψ is the ith session **then**
replace g^y with g^v
honestly choose g^b
draw $r = (k_m, k_s)$ at random from the range of RO
store $\mathsf{faked}(g^{x_1}, g^{x_2}, g^v, g^b, \mathit{Ace}) \leftarrow r$
else
replace $\mathsf{pk}_{P_i} = g^b$ of party P_i with g^v
honestly choose g^y
draw $r = (k_m, k_s)$ at random from the range of RO
store $\mathsf{faked}(g^{x_1}, g^{x_2}, g^y, g^v, \mathit{Ace}) \leftarrow r$
proceed as in $(\mathtt{msg}', (k, \star, \mathsf{st}), \Psi) \leftarrow \mathit{Resp}(\mathsf{sk}_P, P, \mathtt{msg}, \mathsf{cs})$
$\mathsf{resust}^P(\Psi) \leftarrow (k, \star, \mathsf{st})$; send $\mathtt{msg}'$

S_q **upon** $\mathtt{send}^P(\Psi, (\eta, \mathit{Ace}, Y, t_Q))$ **and** $\mathsf{akest}^P(\Psi) \neq \bot$
/* if the client is called with the response of the server */
lookup $(Q, (x_1, x_2), (g^{x_1}, g^{x_2})) \leftarrow \mathsf{akest}^P(\Psi)$
check for a valid pk_Q
if $\mathsf{faked}(g^{x_1}, g^{x_2}, g^y, g^b, \mathit{Ace})$ is defined **then**
lookup $(k_m, k_s) = r \leftarrow \mathsf{faked}(g^{x_1}, g^{x_2}, g^y, g^b, \mathit{Ace})$
if $g^v = g^b$ $(\mathsf{pk}_Q = g^b)$ **then**
query $(g, g^b, g^{x_1}, Z/g^{yx_2})$ to the DDH oracle
else if $g^v = Y = g^y$ **then**
query $(g, g^y, g^{x_2}, Z/g^{bx_1})$ to the DDH oracle
if the DDH oracle confirms **then**
program $\mathrm{RO}(Z, g^{x_1}, g^{x_2}, g^y, g^b, \mathit{Ace}) := r$
proceed as in (k, Q, st)
$\leftarrow \mathit{CompKey}(\mathsf{pk}_Q, \mathtt{msg}, \Psi, (Q, \mathsf{st}))$
erase $\mathsf{akest}^P(\Psi)$; $\mathsf{resust}^P(\Psi) \leftarrow (k, Q, \mathsf{st})$

S_q **simulating the RO: upon** $(Z, g^{x_1}, g^{x_2}, g^y, g^b, \mathit{Ace})$
if $(g^u, g^v) = (g^{x_1}, g^b)$ **then**
query $(g, g^u, g^v, Z/g^{yx_2})$ to the DDH oracle
if the DDH oracle confirms **then**
send Z/g^{yx_2} as a guess and stop
else if $(g^u, g^v) = (g^{x_2}, g^y)$ **then**
query $(g, g^u, g^v, Z/g^{bx_1})$ to the DDH oracle
if the DDH oracle confirms **then**
send Z/g^{bx_1} as a guess and stop
if $\mathrm{RO}(Z, g^{x_1}, g^{x_2}, g^y, g^b, \mathit{Ace}) = r$ is defined **then**
respond with r
else
draw $r = (k_m, k_s)$ at random from the range of RO
program $\mathrm{RO}(Z, g^{x_1}, g^{x_2}, g^y, g^b, \mathit{Ace}) := r$

Figure 8.5.: The simulator S_q

The reduction S_q answers all queries honestly, except for $\mathtt{partner}(g^u)$ or $\mathtt{partner}(g^v)$ queries. In these cases S_q aborts the simulation. If the attacker stops, S_q draws a random group element and sends it as a blind guess to the GDH challenger. The simulator S_q cannot compute $g^{bx_1+yx_2}$ if the GDH challenge exponent v equals b. Since, however, $g^{bx_1+yx_2}$ is never sent in plain but always hashed and S_q also simulates the random oracle, these hashes can be faked without knowing the input.

Besides, the simulator acts as specified in Figure 8.5.

Recall that we required the challenge session to be fresh. Let (g^{x_1}, g^{x_2}) be the ephemeral keys of the client, g^y be the ephemeral key of the server, and g^b be the long-term key of the server. By the freshness of the challenge session, we conclude that with overwhelming probability the attacker at most a partner to (x_2, y) or to (x_1, b). Hence, it suffices to consider these two cases in which the attacker is not a partner to (x_2, y) or not to (x_1, b).

If $\mathcal{A}$ is not a partner to (x_1, b), then S_q either knows x_2 or y and can hence compute g^{yx_2}. Moreover, with probability $1/(q|P|)$ we have $b = v$ and $x_1 = u$. Then, the simulator

guesses g^{uv} correctly if Z is the shared secret, i.e., if $Z = g^{bx_1+yx_2}$, since

$$Z/g^{yx_2} = g^{bx_1+yx_2-yx_2} = g^{uv+yx_2-yx_2} = g^{uv}.$$

If $\mathcal{A}$ is not a partner to (x_2, y), then S_q knows x_1 or b and can hence compute g^{bx_1}. We stress that the attacker cannot send a maliciously chosen y' (such as $y' = 0$) because a successful forgery of the MAC tag would lead to an attack against the CMA-UF property of *Mac*. Then, with probability $1/q$ we have $x_2 = u$ and $y = v$. Then, again the simulator guesses g^{uv} correctly if Z is the shared secret, i.e., if $Z = g^{bx_1+yx_2}$, since

$$Z/g^{bx_1} = g^{bx_1+yx_2-bx_1} = g^{bx_1+uv-bx_1} = g^{uv}.$$

Note that for $z = uv$, S_q is indistinguishable from $Game_1$ as long as $\texttt{partner}(g^u)$ and $\texttt{partner}(g^v)$ is not queried. Similarly for a randomly chosen z, S_q is indistinguishable from $Game_2$ as long as $\texttt{partner}(g^u)$ and $\texttt{partner}(g^v)$ is not queried. Below, we denote this event that $\texttt{partner}(g^u)$ and $\texttt{partner}(g^v)$ is not queried as T. The probability that T occurs is more than $\binom{2}{|P|} = \frac{2}{|P|(|P|-1)}$, where P is the set of parties.

We conclude that if the attacker can distinguish $Game_1$ from $Game_2$ with more than negligible probability, then the attacker queried the random oracle with

$$(g^{bx_1+yx_2}, g^{x_1}, g^{x_2}, g^y, g^b, Ace).$$

The overall winning probability of the simulator S_q can, hence, be computed as follows. Let E_1 be the event that the attacker is not a partner to (x_2, y) and E_2 the event that the attacker is not a partner to (x_1, b). Recall T is the event that $\texttt{partner}(g^u)$ and $\texttt{partner}(g^v)$ has not been queried. Moreover, let W be the event that the simulator S_q wins against the GDH challenger, and μ be the probability that the attacker distinguishes $Game_1$ from $Game_2$. Then, we get

$$\begin{aligned}
\Pr[W] &= \Pr[E_1] \cdot \Pr[W \mid E_1] + \Pr[E_2] \cdot \Pr[W \mid E_2] \\
&= \Pr[E_1] \cdot \frac{\mu}{q|P|} \cdot \Pr[T] + \Pr[E_2] \cdot \frac{\mu}{q} \cdot \Pr[T] \\
&\geq \Pr[E_1] \cdot \frac{\mu}{q|P|} \cdot \binom{2}{|P|} + \Pr[E_2] \cdot \frac{\mu}{q} \cdot \binom{2}{|P|} \\
&\overset{(1)}{\geq} \binom{2}{|P|} \frac{\mu}{(q|P|)} = \frac{2\mu}{q|P|^2(|P|-1)}
\end{aligned}$$

where (1) holds since $\Pr[E_1] + \Pr[E_2] = 1$.

Hence, S_q breaks the GDH game with probability more than $\binom{2}{|P|}\mu/(q|P|)$ if the attacker distinguishes $Game_1$ from $Game_2$ with probability μ. In particular, if the GDH assumption holds $Game_2$ is indistinguishable from the real setting $Game_1$, and the 1W-AKE security holds. □

The proof for the 1W-anonymity of *Ace* is almost exactly the same as the proof of the 1W-anonymity of *ntor* [GSU12]. Therefore, we refer for the proof to their work and only state the result.

Lemma 53 (*Ace* is 1W-anonymous)**.** *The Ace protocol is 1W-anonymous in the sense of Definition 91.*

8.6. Conclusion

Ace is a novel and provably secure 1W-AKE protocol, and we propose it for use in the next generation of Tor's circuit establishment protocol. Compared to the current key-exchange protocol (*ntor*), *Ace* offers a client-side efficiency improvement of 46% and a server-side efficiency improvement of nearly 19%. Even though *Ace* requires the client to send one additional group element, it does not produce any communication overhead in Tor as *Ace* only occupies some of the unused space in a Tor packet, in the ECC setting. Given that the ECC setting is under consideration for the Tor system, the improved computational efficiency, and the proven security properties make our 1W-AKE an ideal candidate for use in the Tor protocol.

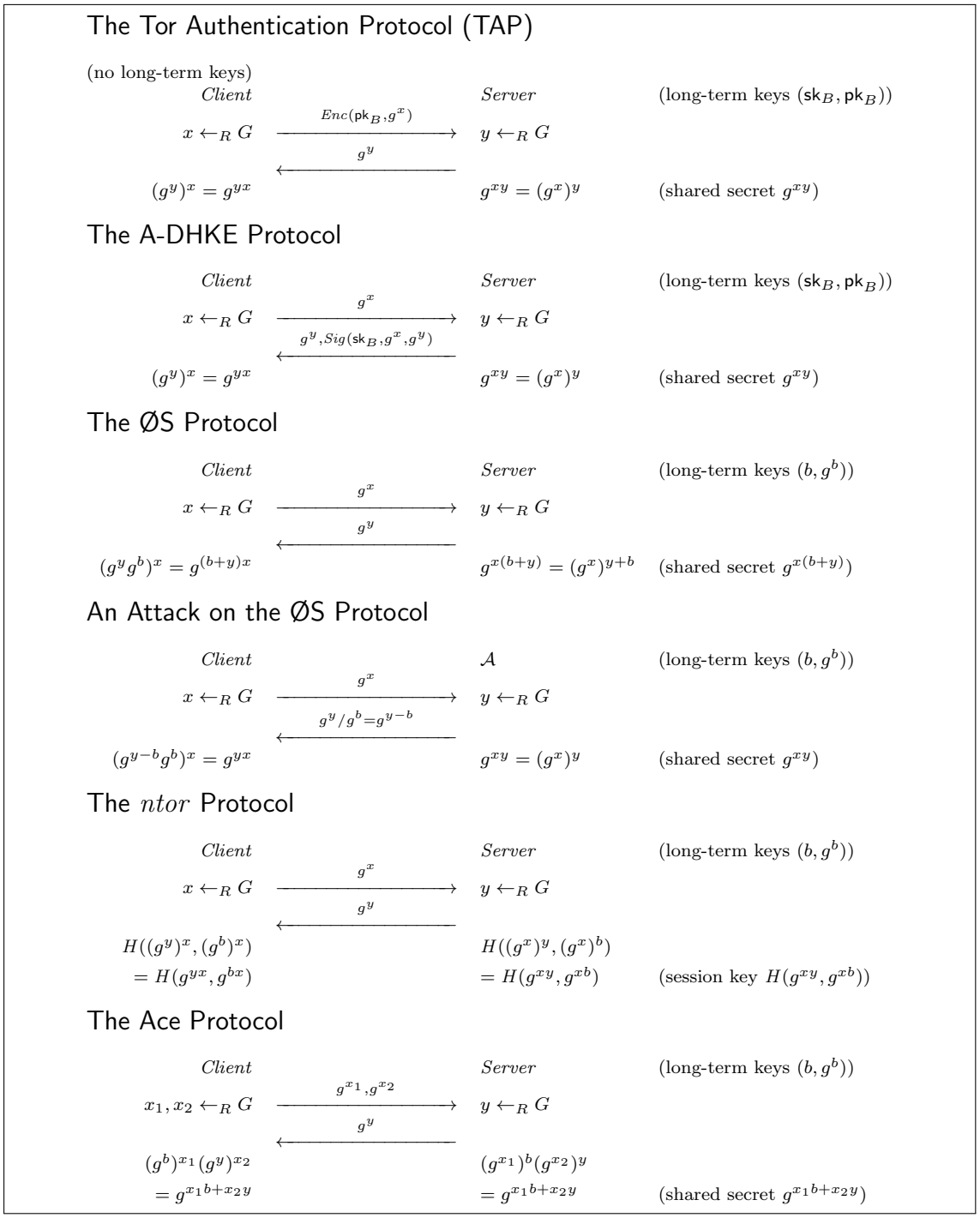

Figure 8.6.: A comparative overview over all discussed protocols: G is the exponent group. For the sake of readability, we neglected the session information used for the key derivation and the key confirmation message.

Chapter 9.

Conclusion

In this thesis we presented work on two kinds of abstractions: symbolic abstractions and ideal functionalities. For symbolic abstractions, we first presented a generalization of the CoSP framework to equivalence properties we extend the CoSP framework to equivalence properties, and we showed that an existing embedding of the applied π-calculus to CoSP can be re-used for uniform bi-processes. On the verification side, as analyses in ProVerif with symbolic length functions often do not terminate, we showed how to combine the recent protocol verifier APTE with ProVerif.

Thereafter, we presented a computationally sound, symbolic abstraction of malleable ZK proofs that is accessible to existing tools for automated verification of security protocols. In particular, we developed an equational theory that captures the semantics of malleable ZK proofs . We proved the computational soundness of our abstraction with respect to trace properties.

Moreover, we devised symbolic abstractions for interactive primitives. As a first step, we devised an abstraction of secure multi-party computations in the applied π-calculus. Based on this abstraction, we proposed a methodology to mechanically analyze the security of cryptographic protocols employing secure multi-party computations. We exemplified the applicability of our framework by analyzing the SIMAP sugar-beet double auction protocol. We finally studied the computational soundness of our abstraction, proving that the analysis of protocols expressed in the applied π-calculus and based on our abstraction provides computational security guarantees.

Interactive cryptographic primitives, such as interactive zero-knowledge proofs or blind signatures, achieve unique security properties that are impossible to achieve for non-interactive primitives, e.g., in an interactive zero-knowledge proofs the verifier can not prove to a third party that the prover issued a convincing proof. Canetti introduced a framework for proving strong, composable security guarantees of interactive primitives, called the UC framework, which has been widely used to prove UC-security of various primitives.

As a second step, we presented a generalization to the large class of UC-secure interactive primitives , such as interactive zero-knowledge proofs and blind signatures, for uniformity. Moreover, our result is cast in the generalized CoSP framework for equivalence properties; hence independent of the calculus. Our result can be combined with and is parametric w.r.t. a given symbolic model.

Next, we identified a general condition under which CS for trace properties implies CS for uniformity of bi-processes, i.e., the class of equivalence properties that ProVerif is able to verify for the applied π-calculus. As a case study, we showed that this general condition holds for a Dolev-Yao model that contains signatures, public-key encryption, and

corresponding length functions. We proved this result in the CoSP framework. Combined with the results from the extension of the CoSP framework, we establish a computationally sound automated verification chain for uniformity of bi-processes in the applied π-calculus that use public-key encryption, signatures, and length functions.

In the area of anonymous communication, we addressed the gap between work on OR protocols and existing OR anonymity analyses. In this work, we address both issues with onion routing by defining a provably secure OR protocol, which is practical for deployment in the next generation Tor network.

We first presented a security definition (an ideal functionality) for the OR methodology in the universal composability (UC) framework. We then determine the exact security properties required for OR cryptographic primitives (onion construction and processing algorithms, and a key exchange protocol) to achieve a provably secure OR protocol. We showed that the currently deployed onion algorithms with slightly strengthened integrity properties can be used in a provably secure OR construction. In the process, we identified the concept of predictably malleable symmetric encryptions, which might be of independent interest and is a stateful variation of HCCA [PR08]. On the other hand, we find the currently deployed key exchange protocol to be inefficient and difficult to analyze and instead show that a recent, significantly more efficient, key exchange protocol can be used in a provably secure OR construction.

In addition, our definition greatly simplifies the process of analyzing OR anonymity metrics. We define and prove forward secrecy for the OR protocol, and realize our (white-box) OR definition from an OR black-box model assumed in a recent anonymity analysis. This realization not only makes the analysis formally applicable to the OR protocol but also identifies the exact adversary and network assumptions made by the black box model. This work has been the foundation for further provable analyses of Tor's anonymity guarantees [BKMM14; BKMMM13; BMM14].

Finally, we present a novel 1W-AKE protocol *Ace* that improves on the computation costs of *ntor*: in numbers, the client has an efficiency improvement of 46% and the server of nearly 19%. As far as communication costs are concerned, our protocol requires a client to send one additional group element to a server, compared to the *ntor* protocol. However, an additional group element easily fits into the 512 bytes fix-sized Tor packets (or cell) in the elliptic curve cryptography (ECC) setting. Consequently, our protocol does not produce a communication overhead in the Tor protocol. Moreover, we prove that our protocol *Ace* constitutes a 1W-AKE. Given that the ECC setting is under consideration for the Tor system, the improved computational efficiency, and the proven security properties make our 1W-AKE an ideal candidate for use in the Tor protocol.

Appendix A.

The Source Code for the Sugar Beet Case Study

```
(*****************************************************************************
 In this file, we model the sugar-beet double auction that has been implemented by
 the SIMAP project in the applied pi calculus.

 We define several predicates, which can be split into three categories. First, we
 have the predicates that are used for the actual verification. The predicate INPUT
 is used to signalize that and which input has been sent, and MCP is uesd to
 characterize the policy. Second, we introduce the arithmetic predicate GE. Third,
 we introduce message predicates, i.e., guards for the quantified; these guards are
 needed for the computational soundness result.
*****************************************************************************)

predicate INPUT(*3*).
predicate MCP(*2*).

predicate GE(*2*).
predicate ADD(*3*).
predicate MULT(*3*).
predicate SUB(*3*).
predicate EQUIV(*2*).

// The following predicates are message predicates, i.e., guards for
// quantified variables.
predicate P_ID_1(*1*).
predicate P_ID_2(*1*).
predicate P_ID_3(*1*).
predicate P_ID_4(*1*).
predicate P_ID_5(*1*).
predicate P_X_1_1(*1*).
predicate P_X_1_2(*1*).
predicate P_X_2_1(*1*).
predicate P_X_2_2(*1*).
predicate P_FLAG_1(*1*).
predicate P_FLAG_2(*1*).
predicate P_SID(*1*).
predicate P_DEC(*1*).
predicate P_MCP(*1*).

(*****************************************************************************
 Types

 In this file, we define three more types upfront: The type T_sid, which is only a
 synonym for the type Un. The type T_flag, which constitutes all integers that are
 either 0 or 1. And, the type T_comparison_result, which carries the properties that
 the output of the SMPC should have.
*****************************************************************************)

type T_sid = s:Un | [*#P_SID(s)*].
type T_cert = <Private, Private, Private, Private, Private, Private, Un, Int>.
type T_flag = x:Int | [*#or(equal(x, string_null(empty())), equal(x, string_one(empty())))*] .
```

```
type T_comparison_result = <bi:Int, bd:Int, bsid:Un>
[*#exists([bx_1_1, bx_1_2, bflag_1, bx_2_1, bx_2_2, bflag_2, bid_4, bid_5,
     bvar_s_1, bvar_s_2, bvar_b_1, bvar_b_2, bvar_b_1_1, bvar_b_2_1, bvar_s, bvar_b,
     bvar_ss_1, bvar_ss_2, bvar_bb_1, bvar_bb_2, bvar_bb_1_1, bvar_bb_2_1, bvar_ss, bvar_bb,
     bone_minus_one, btwo, btwo_minus_one],

    and(and(and(P_X_1_1(bx_1_1), P_X_1_2(bx_1_2)), P_FLAG_1(bflag_1)),
    and(and(and(P_X_2_1(bx_2_1), P_X_2_2(bx_2_2)), P_FLAG_2(bflag_2)),
    and(and(and(and(P_ID_4(bid_4), P_ID_5(bid_5)), P_SID(bsid)),

    not(equal(bid_4, bid_5))),
    and(and(and(
            implies(equal(bflag_1, string_one(empty())), GE(bx_1_1, bx_1_2)),
    implies(equal(bflag_1, string_null(empty())), GE(bx_1_2, bx_1_1))),
    INPUT(bid_4, <bflag_1, bx_1_1, bx_1_2>, bsid)),
    and(and(and(
            implies(equal(bflag_2, string_one(empty())), GE(bx_2_1, bx_2_2)),
    implies(equal(bflag_2, string_null(empty())), GE(bx_2_2, bx_2_1))),
    INPUT(bid_5, <bflag_2, bx_2_1, bx_2_2>, bsid)),

    or(and(EQUIV(bi, string_one(empty())), EQUIV(bd, string_one(empty()))),
       and(MULT(bx_1_1, bflag_1, bvar_s_1),
         and(MULT(bx_2_1, bflag_2, bvar_s_2),
         and(ADD(bvar_s_1, bvar_s_2, bvar_s),
         and(SUB(string_one(empty()), bflag_1, bvar_b_1_1),
         and(MULT(bx_1_1, bvar_b_1_1, bvar_b_1),
         and(SUB(string_one(empty()), bflag_2, bvar_b_2_1),
 and(MULT(bx_2_1, bvar_b_2_1, bvar_b_2),
         and(ADD(bvar_b_1, bvar_b_2, bvar_b),
         and(GE(bvar_s, bvar_b),

         or(and(SUB(string_one(empty()), string_one(empty()), bone_minus_one),
      and(EQUIV(bi, bone_minus_one),
      and(EQUIV(bd, string_null(empty())),
      and(GE(bvar_s, bvar_b),
        not(equal(bvar_s, bvar_b))
            )))),

            and(MULT(bx_1_2, bflag_1, bvar_ss_1),
              and(MULT(bx_2_2, bflag_2, bvar_ss_2),
              and(ADD(bvar_ss_1, bvar_ss_2, bvar_ss),
              and(SUB(string_one(empty()), bflag_1, bvar_bb_1_1),
              and(MULT(bx_1_2, bvar_bb_1_1, bvar_bb_1),
              and(SUB(string_one(empty()), bflag_2, bvar_bb_2_1),
              and(MULT(bx_2_2, bvar_bb_2_1, bvar_bb_2),
              and(ADD(bvar_bb_1, bvar_bb_2, bvar_bb),

      or(and(ADD(string_one(empty()), string_one(empty()), btwo),
           and(EQUIV(bi, btwo),
                   and(EQUIV(bd, string_one(empty())),
           and(GE(bvar_bb, bvar_ss),
               GE(bvar_b, bvar_s)
        )))),


         and(SUB(btwo, string_one(empty()), btwo_minus_one),
           and(EQUIV(bi, btwo_minus_one),
                   and(EQUIV(bd, string_null(empty())),
           and(GE(bvar_ss, bvar_bb),
           and(GE(bvar_b, bvar_s),
               not(equal(bvar_ss, bvar_bb))
        )))))

   )))))))))

 ))))))))))
      )))))
```

```
   )
*]
.

(*****************************************************************************
 Policy

 This process formalizes the properties that the final market clearing price
 should have; namely, the market clearing price (bi) is the maximal price
 such that the demand is greater than or equal to the supply.
*****************************************************************************)

let MCP_Policy =
assume (*#forall([bi, bd, bid_4, bid_5, bx_1_1, bx_1_2, bflag_1, bx_2_1, bx_2_2, bflag_2, bsid,
            bvar_s_1, bvar_s_2, bvar_b_1, bvar_b_2, bvar_b_1_1, bvar_b_2_1, bvar_s, bvar_b,
          bvar_ss_1, bvar_ss_2, bvar_bb_1, bvar_bb_2, bvar_bb_1_1, bvar_bb_2_1, bvar_ss, bvar_bb,
          bone_minus_one, btwo, btwo_minus_one],
    implies(
and(and(and(P_ID_4(bid_4), P_ID_5(bid_5)), and(and(P_MCP(bi), P_DEC(bd)), P_SID(bsid))),
and(and(and(P_X_1_1(bx_1_1), P_X_1_2(bx_1_2)), P_FLAG_1(bflag_1)),
     and(and(and(P_X_2_1(bx_2_1), P_X_2_2(bx_2_2)), P_FLAG_2(bflag_2)),

and(not(equal(bid_4, bid_5)),
and(and(true,
and(and(and(
          implies(equal(bflag_1, one), GE(bx_1_1, bx_1_2)),
        implies(equal(bflag_1, null), GE(bx_1_2, bx_1_1))),
        INPUT(bid_4, <bflag_1, bx_1_1, bx_1_2>, bsid)),
    and(and(
implies(equal(bflag_2, one), GE(bx_2_1, bx_2_2)),
implies(equal(bflag_2, null), GE(bx_2_2, bx_2_1))),
INPUT(bid_5, <bflag_2, bx_2_1, bx_2_2>, bsid)))
),

    or(and(EQUIV(bi, string_one(empty())), EQUIV(bd, string_one(empty()))),
       and(MULT(bx_1_1, bflag_1, bvar_s_1),
         and(MULT(bx_2_1, bflag_2, bvar_s_2),
         and(ADD(bvar_s_1, bvar_s_2, bvar_s),
         and(SUB(string_one(empty()), bflag_1, bvar_b_1_1),
         and(MULT(bx_1_1, bvar_b_1_1, bvar_b_1),
         and(SUB(string_one(empty()), bflag_2, bvar_b_2_1),
 and(MULT(bx_2_1, bvar_b_2_1, bvar_b_2),
         and(ADD(bvar_b_1, bvar_b_2, bvar_b),
         and(GE(bvar_s, bvar_b),

         or(and(SUB(string_one(empty()), string_one(empty()), bone_minus_one),
      and(EQUIV(bi, bone_minus_one),
      and(EQUIV(bd, string_null(empty())),
      and(GE(bvar_s, bvar_b),
        not(equal(bvar_s, bvar_b))
            )))),

            and(MULT(bx_1_2, bflag_1, bvar_ss_1),
              and(MULT(bx_2_2, bflag_2, bvar_ss_2),
              and(ADD(bvar_ss_1, bvar_ss_2, bvar_ss),
              and(SUB(string_one(empty()), bflag_1, bvar_bb_1_1),
              and(MULT(bx_1_2, bvar_bb_1_1, bvar_bb_1),
              and(SUB(string_one(empty()), bflag_2, bvar_bb_2_1),
              and(MULT(bx_2_2, bvar_bb_2_1, bvar_bb_2),
              and(ADD(bvar_bb_1, bvar_bb_2, bvar_bb),

      or(and(ADD(string_one(empty()), string_one(empty()), btwo),
           and(EQUIV(bi, btwo),
                  and(EQUIV(bd, string_one(empty())),
           and(GE(bvar_bb, bvar_ss),
               GE(bvar_b, bvar_s)
        )))),
```

```
         and(SUB(btwo, string_one(empty()), btwo_minus_one),
           and(EQUIV(bi, btwo_minus_one),
                   and(EQUIV(bd, string_null(empty()))),
           and(GE(bvar_ss, bvar_bb),
           and(GE(bvar_b, bvar_s),
               not(equal(bvar_ss, bvar_bb))
         )))))

    )))))))))

 ))))))))))
)))))), MCP(bi, bsid)))
*)
.

(*************************************************************************
 Certification issuer

 The certification issuer is a trusted party that sends (in practice possibly
 in advance) a signature of the id of that party, a list of participants,
 the session id, and the set of prices to each party. The SMPC checks whether
 this signature is valid and whether the information coincides with the actual
 inputs of the parties.
*************************************************************************)

let certification_issuer = (
    new sig_rand(*:Private*);
    let secret_key = sk(<sig_rand, sig_rand, sig_rand, sig_rand, sig_rand, sig_rand, sid, two>)
  (*:SigKey(T_cert)*) in
    let ver_key = vk(<sig_rand, sig_rand, sig_rand, sig_rand, sig_rand, sig_rand, sid, two>)
  (*:VerKey(T_cert)*) in
    (
    out(certification_channel_4, <(*=signature*)sign(<id_4, id_1, id_2, id_3, id_4, id_5, sid, two>,
  secret_key)(*:Signed(T_cert)*),(*=verification_key*)ver_key(*:VerKey(T_cert)*)>
  (*#Red(check(signature, verification_key),
         <id_4, id_1, id_2, id_3, id_4, id_5, sid, two>)
    *)) |
    out(certification_channel_5, <(*=signature*)sign(<id_5, id_1, id_2, id_3, id_4, id_5, sid, two>,
  secret_key)(*:Signed(T_cert)*),(*=verification_key*)ver_key(*:VerKey(T_cert)*)>
  (*#Red(check(signature, verification_key),
         <id_5, id_1, id_2, id_3, id_4, id_5, sid, two>)
    *))
    )
).

(*************************************************************************
 bidder (2 instances)

 The bidder process represents either a sugar-beet producer or a sugar-beet
 buyer. In order to model arbitrary inputs of a certain form, we let the
 inputs be received from a channel input_channel_i of an appropriate type,
 which is refined with the properties that we require from an input.

 This process receives the input over the channel input_channel_(i+3) and
 the certificate over the channel certification_channel_(i+3). Then, it sends
 its input consisting of the received ceritifate and the received input.
 Thereafter, this process receives the result of the SMPC and asserts in case
 the decision has been 0 the market clearing price predicate MCP, i.e., asserts
 that this received price is the maximal price such that the demand is greater
 than or equal to the supply. In case the decision is 1, the process proceeds
 in that it waits for the next output of the SMPC.
*************************************************************************)

let bidder_1 = (
    new input_channel_4(*:Ch(<f:T_flag, x1:Int, x2:Int>
```

```
    [*#
      and(and(GE(x1, null), GE(x2, null)),
      and(and(
  implies(equal(f, one), GE(x1, x2)),
implies(equal(f, null), GE(x2, x1))),
INPUT(id_4, <f, x1, x2>, sid))
      )
    *])*);
    in(input_channel_4, private_input_4);
    let <flag_1, x_1_1, x_1_2> = private_input_4 (*:<f:T_flag, x1:Int, x2:Int>
    [*#
      and(and(GE(x1, null), GE(x2, null)),
      and(and(
  implies(equal(f, one), GE(x1, x2)),
implies(equal(f, null), GE(x2, x1))),
INPUT(id_4, <f, x1, x2>, sid))
      )
    *]*) in
    (
    assume(*#and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1))*)

    |

    in(certification_channel_4, certificate_4);
    out(in_4, <(*=bsid*)sid(*:Un*), <<(*=f*)flag_1(*:Int*), (*=x1*)x_1_1(*:Int*),
               (*=x2*)x_1_2(*:Int*)>(*#
      and(or(equal(f, null), equal(f, one)),
      and(and(and(P_X_1_1(x1), P_X_1_2(x2)), P_FLAG_1(f)),
      and(and(GE(x1, null), GE(x2, null)),
      and(and(
  implies(equal(f, one), GE(x1, x2)),
implies(equal(f, null), GE(x2, x1))),
INPUT(id_4, <f, x1, x2>, sid))
      )))
    *), certificate_4(*:<signature:Signed(T_cert), verification_key:VerKey(T_cert)>[*#
          Red(check(signature, verification_key),
           <id_4, id_1, id_2, id_3, id_4, id_5, sid, two>)
   *]*)>>);
  in(output_channel_4, xoutput);

  let <price, decision, ssid> = xoutput in
  (

  assume (*#P_MCP(price)*) | assume (*#P_DEC(decision)*)

  |

  if decision = null then
  (

  if price = one_minus_one then
  (
      assert (*#MCP(price, ssid)*)
  )

  )
  else
  // if decision = one
  if price = one then
  (
  in(output_channel_4, xoutput_1);
  let <price_1, decision_1, ssid_1> = xoutput_1 in

  (
  assume (*#P_MCP(price_1)*)  | assume (*#P_DEC(decision_1)*)

  |
```

```
  if price_1 = two_minus_one then
  (

    assert (*#MCP(price_1, ssid_1)*)

  )

  else
  let tmp_2 = two in
  if price_1 = tmp_2 then
  (
      assert (*#MCP(price_1, ssid_1)*)
  ))))
)
).

let bidder_2 = (
    new input_channel_5(*:Ch(<f:T_flag, x1:Int, x2:Int>
    [*#
      and(and(GE(x1, null), GE(x2, null)),
      and(and(
  implies(equal(f, one), GE(x1, x2)),
implies(equal(f, null), GE(x2, x1))),
INPUT(id_5, <f, x1, x2>, sid))
      )
    *])*);
    in(input_channel_5, private_input_5);
    let <flag_2, x_2_1, x_2_2> = private_input_5 (*:<f:T_flag, x1:Int, x2:Int>
    [*#
      and(and(GE(x1, null), GE(x2, null)),
      and(and(
  implies(equal(f, one), GE(x1, x2)),
implies(equal(f, null), GE(x2, x1))),
INPUT(id_5, <f, x1, x2>, sid))
      )
    *]*) in
    (
    assume(*#and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2))*)

    |

    in(certification_channel_5, certificate_5);
    out(in_5, <(*=bsid*)sid(*:Un*), <<(*=f*)flag_2(*:Int*), (*=x1*)x_2_1(*:Int*),
  (*=x2*)x_2_2(*:Int*)>(*#
      and(or(equal(f, null), equal(f, one)),
      and(and(and(P_X_2_1(x1), P_X_2_2(x2)), P_FLAG_2(f)),
      and(and(GE(x1, null), GE(x2, null)),
      and(and(
  implies(equal(f, one), GE(x1, x2)),
implies(equal(f, null), GE(x2, x1))),
INPUT(id_5, <f, x1, x2>, sid))
      )))
    *), certificate_5(*:<signature:Signed(T_cert), verification_key:VerKey(T_cert)>[*#
          Red(check(signature, verification_key),
           <id_5, id_1, id_2, id_3, id_4, id_5, sid, two>)
   *]*)>>);
  in(output_channel_5, xoutput);

  let <price, decision, ssid> = xoutput in
  (

  assume (*#P_MCP(price)*) | assume (*#P_DEC(decision)*)

  |

  if decision = null then
```

```
  (

  if price = one_minus_one then
  (
      assert (*#MCP(price, ssid)*)
  )

  )
  else
  // if decision = one
  if price = one then
  (
  in(output_channel_5, xoutput_1);
  let <price_1, decision_1, ssid_1> = xoutput_1 in

  (
  assume (*#P_MCP(price_1)*)  | assume (*#P_DEC(decision_1)*)

  |

  if price_1 = two_minus_one then
  (
    assert (*#MCP(price_1, ssid_1)*)

  )

  else
  if price_1 = two then
  (
      assert (*#MCP(price_1, ssid_1)*)
  ))))
)
).

(*************************************************************************
 computation server (3 instances)

 The process server_i represents the computation servers. The computation
 servers send a start_round message to the ideal Process SMPC and assert upon
 receiving the result the policy if the decision is 0; otherwise it sends the
 command for the computation of the next round. These computation servers are
 introduced in the real implementation for effficiency reasons.
*************************************************************************)
let server_1 =
  out(in_1, <sid(*:T_sid*), start_round(*:Private*)>);
  (
  in(output_channel_1, xoutput);

  let <price, decision, ssid> = xoutput in
  (

  assume (*#P_MCP(price)*) | assume (*#P_DEC(decision)*)

  |

  if decision = null then
  (

  if price = one_minus_one then
  (
      assert (*#MCP(price, ssid)*)
  )

  )
  else
  // if decision = one
  if price = one then
```

```
  (
  out(in_1, <sid(*:T_sid*), start_round(*:Private*)>);
  in(output_channel_1, xoutput_1);
  let <price_1, decision_1, ssid_1> = xoutput_1 in

  (
  assume (*#P_MCP(price_1)*)   | assume (*#P_DEC(decision_1)*)

  |

  if price_1 = two_minus_one then
  (

    assert (*#MCP(price_1, ssid_1)*)

  )

  else
  if price_1 = two then
  (
      assert (*#MCP(price_1, ssid_1)*)
  ))))
)
.

let server_2 =
  out(in_2, <sid(*:T_sid*), start_round(*:Private*)>);
  (
  in(output_channel_2, xoutput);

  let <price, decision, ssid> = xoutput in
  (

  assume (*#P_MCP(price)*)  | assume (*#P_DEC(decision)*)

  |

  if decision = null then
  (

  if price = one_minus_one then
  (
      assert (*#MCP(price, ssid)*)
  )

  )
  else
  // if decision = one
  if price = one then
  (
  out(in_2, <sid(*:T_sid*), start_round(*:Private*)>);
  in(output_channel_2, xoutput_1);
  let <price_1, decision_1, ssid_1> = xoutput_1 in

  (
  assume (*#P_MCP(price_1)*)   | assume (*#P_DEC(decision_1)*)

  |

  if price_1 = two_minus_one then
  (

    assert (*#MCP(price_1, ssid_1)*)

  )

  else
```

```
  if price_1 = two then
  (
      assert (*#MCP(price_1, ssid_1)*)
  ))))
)
.

let server_3 =
  out(in_3, <sid(*:T_sid*), start_round(*:Private*)>);
  (
  in(output_channel_3, xoutput);

  let <price, decision, ssid> = xoutput in
  (

  assume (*#P_MCP(price)*)  | assume (*#P_DEC(decision)*)

  |

  if decision = null then
  (

  if price = one_minus_one then
  (
      assert (*#MCP(price, ssid)*)
  )

  )
  else
  // if decision = one
  if price = one then
  (
  out(in_3, <sid(*:T_sid*), start_round(*:Private*)>);
  in(output_channel_3, xoutput_1);
  let <price_1, decision_1, ssid_1> = xoutput_1 in

  (
  assume (*#P_MCP(price_1)*)   | assume (*#P_DEC(decision_1)*)

  |

  if price_1 = two_minus_one then
  (

    assert (*#MCP(price_1, ssid_1)*)

  )

  else
  if price_1 = two then
  (
      assert (*#MCP(price_1, ssid_1)*)
  ))))
)
.

(*************************************************************************
 SMPC process

 At this point, we define the process SMPC(sidc, adv, in_1, .., in_5, SIMAP),
 where SIMAP denotes the function that characterizes the sugar-beet double
 auction.

 The SIMAP[] context expects in the first round the inputs of the bidding
 parties (bidder_i) and the "start_round" command from the computation
 parties. In the second round it does not expect any inputs from the input
 parties but only a "start_round" command from the computation parties.
```

```
 In each round the SIMAP[] context computes the supply and the demand from
 the input of the bidder parties (bidder_i) and checks whether the supply
 is already greater than the supply, in which case the context sends the
 price decreased by 1 to all parties. If there is not price such that the
 supply is greater than the demand, the context sends the maximal price.
*************************************************************************)

let SMPC =
  in(sidc, sid_1);
  new lin_1 (*:Ch(Private)*);
  new lin_2 (*:Ch(Private)*);
  new lin_3 (*:Ch(Private)*);
  new lin_4 (*:Ch(<<f:Int, x1:Int, x2:Int>
  [*#
    and(or(equal(f, null), equal(f, one)),
    and(and(and(P_X_1_1(x1), P_X_1_2(x2)), P_FLAG_1(f)),
    and(and(GE(x1, null), GE(x2, null)),
    and(and(
  implies(equal(f, one), GE(x1, x2)),
implies(equal(f, null), GE(x2, x1))),
INPUT(id_4, <f, x1, x2>, sid))
    )))
  *], <signature:Signed(T_cert), verification_key:VerKey(T_cert)>[*#
          Red(check(signature, verification_key),
           <id_4, id_1, id_2, id_3, id_4, id_5, sid, two>)
   *]>)*);
  new lin_5 (*:Ch(<<f:Int, x1:Int, x2:Int>
  [*#
    and(or(equal(f, null), equal(f, one)),
    and(and(and(P_X_2_1(x1), P_X_2_2(x2)), P_FLAG_2(f)),
    and(and(GE(x1, null), GE(x2, null)),
    and(and(
  implies(equal(f, one), GE(x1, x2)),
implies(equal(f, null), GE(x2, x1))),
INPUT(id_5, <f, x1, x2>, sid))
    )))
  *], <signature:Signed(T_cert), verification_key:VerKey(T_cert)>[*#
          Red(check(signature, verification_key),
           <id_5, id_1, id_2, id_3, id_4, id_5, sid, two>)
   *]>)*);

  new inloop_1 (*:Ch(Private)*); new inloop_2 (*:Ch(Private)*);
  new inloop_3 (*:Ch(Private)*); new inloop_4 (*:Ch(Private)*);
  new inloop_5 (*:Ch(Private)*);
  (
  // The processes input_1 | .. | input_n
  out(inloop_1, sync) |
  ! in(inloop_1, z); in(in_1, xinput);
    let <sid_tmp_1, x_1> = xinput in out(adv, sid_tmp_1);
    if sid_1 = sid_tmp_1 then out(lin_1, x_1(*:Private*)) else out(inloop_1, sync) |
  out(inloop_2, sync) |
  ! in(inloop_2, z); in(in_2, xinput);
    let <sid_tmp_2, x_2> = xinput in out(adv, sid_tmp_2);
    if sid_1 = sid_tmp_2 then out(lin_2, x_2(*:Private*)) else out(inloop_2, sync) |
  out(inloop_3, sync) |
  ! in(inloop_3, z); in(in_3, xinput);
    let <sid_tmp_3, x_3> = xinput in out(adv, sid_tmp_3);
    if sid_1 = sid_tmp_3 then out(lin_3, x_3(*:Private*)) else out(inloop_3, sync) |
  out(inloop_4, sync) |
  ! in(inloop_4, z); in(in_4, xinput);
    let <sid_tmp_4, x_4> = xinput in out(adv, sid_tmp_4);
    if sid_1 = sid_tmp_4 then out(lin_4, x_4(*:<<f:Int, x1:Int, x2:Int>
    [*#
    and(or(equal(f, null), equal(f, one)),
    and(and(and(P_X_1_1(x1), P_X_1_2(x2)), P_FLAG_1(f)),
    and(and(GE(x1, null), GE(x2, null)),
    and(and(
```

```
  implies(equal(f, one), GE(x1, x2)),
implies(equal(f, null), GE(x2, x1))),
INPUT(id_4, <f, x1, x2>, sid))
    )))
    *], <signature:Signed(T_cert), verification_key:VerKey(T_cert)>[*#
          Red(check(signature, verification_key),
           <id_4, id_1, id_2, id_3, id_4, id_5, sid, two>)
   *]>*)) else out(inloop_4, sync) |
  out(inloop_5, sync) |
  ! in(inloop_5, z); in(in_5, xinput);
    let <sid_tmp_5, x_5> = xinput in out(adv, sid_tmp_5);
    if sid_1 = sid_tmp_5 then out(lin_5, x_5(*:<<f:Int, x1:Int, x2:Int>
    [*#
    and(or(equal(f, null), equal(f, one)),
    and(and(and(P_X_2_1(x1), P_X_2_2(x2)), P_FLAG_2(f)),
    and(and(GE(x1, null), GE(x2, null)),
    and(and(
  implies(equal(f, one), GE(x1, x2)),
implies(equal(f, null), GE(x2, x1))),
INPUT(id_5, <f, x1, x2>, sid))
    )))
    *], <signature:Signed(T_cert), verification_key:VerKey(T_cert)>[*#
          Red(check(signature, verification_key),
           <id_5, id_1, id_2, id_3, id_4, id_5, sid, two>)
   *]>*)) else out(inloop_5, sync) |

  (*****************************************************************************
   From here on the subprocess SIMAP[deliver_1 | .. | deliver_n] begins, where
   SIMAP is the SMPC-suited context that computes the market clearing price and
   deliver_i are the deliver processes of the SMPC process
   SMPC(sidc, adv, in_1, .., in_n, SIMAP) presented in the paper.
   *****************************************************************************)

  in(lin_4, xinput_1);
  in(lin_5, xinput_2);
  in(lin_1, cmd_1);
  in(lin_2, cmd_2);
  in(lin_3, cmd_3);
  if cmd_1 = cmd_2 then
  if cmd_1 = cmd_3 then
  if cmd_1 = start_round then
  let counter = one in
  let <xinput_1_1, xinput_1_2> = xinput_1 in
  let <flag_1, x_1_1, x_1_2> = xinput_1_1 in
  let <sig_4, verif_key_4> = xinput_1_2 in
  let <xinput_2_1, xinput_2_2> = xinput_2 in
  let <flag_2, x_2_1, x_2_2> = xinput_2_1 in
  let <sig_5, verif_key_5> = xinput_2_2 in
  (
  let tmp_4 = check(sig_4, verif_key_4) in
  if tmp_4 = <id_4, id_1, id_2, id_3, id_4, id_5, sid, two> then
  let tmp_5 = check(sig_5, verif_key_5) in
  if tmp_5 = <id_5, id_1, id_2, id_3, id_4, id_5, sid, two> then
  let var_s_1 = mult(x_1_1, flag_1) in
  let var_s_2 = mult(x_2_1, flag_2) in
  let var_s = add(var_s_1, var_s_2) in
  let var_b_1_1 = sub(one, flag_1) in
  let var_b_1 = mult(x_1_1, var_b_1_1) in
  let var_b_2_1 = sub(one, flag_2) in
  let var_b_2 = mult(x_2_1, var_b_2_1) in
  let var_b = add(var_b_1, var_b_2) in
  let z = greatereq(var_b, var_s) in
  if z = var_b then
  let decision = one (*:y:Int|[*#
       and(MULT(x_1_1, flag_1, var_s_1),and(MULT(x_2_1, flag_2, var_s_2),
       and(ADD(var_s_1, var_s_2, var_s),and(SUB(one, flag_1, var_b_1_1),
       and(SUB(one, flag_2, var_b_2_1),and(MULT(x_1_1, var_b_1_1, var_b_1),
```

```
      and(MULT(x_2_1, var_b_2_1, var_b_2),
          and(ADD(var_b_1, var_b_2, var_b),GE(var_b, var_s)))))))))
    *]*) in
  // Here, we are in the case that the demand for price = 1 is greater than the supply.
  // We proceed, but we output decision = 1, i.e., the demand is greater than the
  // supply.
  (
    // At this point, the process deliver_1 | .. | process_n begins.

  out(output_channel_1, <(*=i*)counter(*:Int*), (*=d*)decision(*:Int*), (*=w*)sid(*:Un*)>(*#
    and(and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1)),
    and(and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2)),
    and(and(and(and(P_ID_4(id_4), P_ID_5(id_5)), P_SID(w)),

        not(equal(id_4, id_5))),
    and(EQUIV(i, one),
    and(and(and(implies(equal(flag_1, one), GE(x_1_1, x_1_2)),
implies(equal(flag_1, null), GE(x_1_2, x_1_1))),
INPUT(id_4, <flag_1, x_1_1, x_1_2>, w)),
    and(and(and(implies(equal(flag_2, one), GE(x_2_1, x_2_2)),
implies(equal(flag_2, null), GE(x_2_2, x_2_1))),
INPUT(id_5, <flag_2, x_2_1, x_2_2>, w)),
EQUIV(d, one)
    ))))))
    *));out(inloop_1, sync);0

     |

  out(output_channel_2, <(*=i*)counter(*:Int*), (*=d*)decision(*:Int*), (*=w*)sid(*:Un*)>(*#
    and(and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1)),
    and(and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2)),
    and(and(and(and(P_ID_4(id_4), P_ID_5(id_5)), P_SID(w)),

        not(equal(id_4, id_5))),
    and(EQUIV(i, one),
    and(and(and(implies(equal(flag_1, one), GE(x_1_1, x_1_2)),
implies(equal(flag_1, null), GE(x_1_2, x_1_1))),
INPUT(id_4, <flag_1, x_1_1, x_1_2>, w)),
    and(and(and(implies(equal(flag_2, one), GE(x_2_1, x_2_2)),
implies(equal(flag_2, null), GE(x_2_2, x_2_1))),
INPUT(id_5, <flag_2, x_2_1, x_2_2>, w)),
EQUIV(d, one)
    ))))))
    *));out(inloop_2, sync);0

     |

  out(output_channel_3, <(*=i*)counter(*:Int*), (*=d*)decision(*:Int*), (*=w*)sid(*:Un*)>(*#
    and(and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1)),
    and(and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2)),
    and(and(and(and(P_ID_4(id_4), P_ID_5(id_5)), P_SID(w)),

        not(equal(id_4, id_5))),
    and(EQUIV(i, one),
    and(and(and(implies(equal(flag_1, one), GE(x_1_1, x_1_2)),
implies(equal(flag_1, null), GE(x_1_2, x_1_1))),
INPUT(id_4, <flag_1, x_1_1, x_1_2>, w)),
    and(and(and(implies(equal(flag_2, one), GE(x_2_1, x_2_2)),
implies(equal(flag_2, null), GE(x_2_2, x_2_1))),
INPUT(id_5, <flag_2, x_2_1, x_2_2>, w)),
EQUIV(d, one)
    ))))))
    *));out(inloop_3, sync);0

     |

  out(output_channel_4, <(*=i*)counter(*:Int*), (*=d*)decision(*:Int*), (*=w*)sid(*:Un*)>(*#
```

```
    and(and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1)),
    and(and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2)),
    and(and(and(and(P_ID_4(id_4), P_ID_5(id_5)), P_SID(w)),

        not(equal(id_4, id_5))),
    and(EQUIV(i, one),
    and(and(and(implies(equal(flag_1, one), GE(x_1_1, x_1_2)),
implies(equal(flag_1, null), GE(x_1_2, x_1_1))),
INPUT(id_4, <flag_1, x_1_1, x_1_2>, w)),
    and(and(and(implies(equal(flag_2, one), GE(x_2_1, x_2_2)),
implies(equal(flag_2, null), GE(x_2_2, x_2_1))),
INPUT(id_5, <flag_2, x_2_1, x_2_2>, w)),
EQUIV(d, one)
    ))))))
    *));out(inloop_4, sync);0


     |

  out(output_channel_5, <(*=i*)counter(*:Int*), (*=d*)decision(*:Int*), (*=w*)sid(*:Un*)>(*#
    and(and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1)),
    and(and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2)),
    and(and(and(and(P_ID_4(id_4), P_ID_5(id_5)), P_SID(w)),

        not(equal(id_4, id_5))),
    and(EQUIV(i, one),
    and(and(and(implies(equal(flag_1, one), GE(x_1_1, x_1_2)),
implies(equal(flag_1, null), GE(x_1_2, x_1_1))),
INPUT(id_4, <flag_1, x_1_1, x_1_2>, w)),
    and(and(and(implies(equal(flag_2, one), GE(x_2_1, x_2_2)),
implies(equal(flag_2, null), GE(x_2_2, x_2_1))),
INPUT(id_5, <flag_2, x_2_1, x_2_2>, w)),
EQUIV(d, one)
    ))))))
    *));out(inloop_5, sync);0

    // At this point the process deliver_1 | .. | deliver_n ends.

  |

  in(lin_1, cmd_1);
  in(lin_2, cmd_2);
  in(lin_3, cmd_3);
  in(lin_4, dummy_4);
  in(lin_5, dummy_5);
  if cmd_1 = cmd_2 then
  if cmd_1 = cmd_3 then
  if cmd_1 = start_round then
  let counter_1 = two in
  let var_ss_1 = mult(x_1_2, flag_1) in
  let var_ss_2 = mult(x_2_2, flag_2) in
  let var_ss = add(var_ss_1, var_ss_2) in
  let var_bb_1_1 = sub(one, flag_1) in
  let var_bb_1 = mult(x_1_2, var_bb_1_1) in
  let var_bb_2_1 = sub(one, flag_2) in
  let var_bb_2 = mult(x_2_2, var_bb_2_1) in
  let var_bb = add(var_bb_1, var_bb_2) in
  let z = greatereq(var_bb, var_ss) in
  if z = var_bb then
  let decision = one (*:y:Int|[*#
    and(MULT(x_1_2, flag_1, var_ss_1),and(MULT(x_2_2, flag_2, var_ss_2),
    and(ADD(var_ss_1, var_ss_2, var_ss),and(SUB(one, flag_1, var_bb_1_1),
    and(MULT(x_1_2, var_bb_1_1, var_bb_1), and(SUB(one, flag_2, var_bb_2_1),
    and(MULT(x_2_2, var_bb_2_1, var_bb_2),and(ADD(var_bb_1, var_bb_2, var_bb),GE(var_bb, var_ss)
    ))))))))
    *]*) in
  // The demand is for price = 2 still greater than or equal to the supply:
```

```
  // output mcp = p = 2 and decision = 1
  (
    // At this point, deliver_1 | .. | deliver_n is placed. The context SIMAP has its
    // hole at this point.
    out(output_channel_1, <(*=i*)counter_1(*:Int*), (*=d*)decision(*:Int*), (*=w*)sid(*:Un*)>
    (*#
    and(and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1)),
    and(and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2)),
    and(and(and(and(P_ID_4(id_4), P_ID_5(id_5)), P_SID(w)),

        not(equal(id_4, id_5))),
    and(EQUIV(i, two),
    and(EQUIV(d, one),
    and(and(and(implies(equal(flag_1, one), GE(x_1_1, x_1_2)),
implies(equal(flag_1, null), GE(x_1_2, x_1_1))),
INPUT(id_4, <flag_1, x_1_1, x_1_2>, w)),
    and(and(and(implies(equal(flag_2, one), GE(x_2_1, x_2_2)),
implies(equal(flag_2, null), GE(x_2_2, x_2_1))),
INPUT(id_5, <flag_2, x_2_1, x_2_2>, w)),

    and(MULT(x_1_1, flag_1, var_s_1),and(MULT(x_2_1, flag_2, var_s_2),
    and(ADD(var_s_1, var_s_2, var_s),and(SUB(one, flag_1, var_b_1_1),
    and(MULT(x_1_1, var_b_1_1, var_b_1), and(SUB(one, flag_2, var_b_2_1),
    and(MULT(x_2_1, var_b_2_1, var_b_2),and(ADD(var_b_1, var_b_2, var_b),

    and(MULT(x_1_2, flag_1, var_ss_1),and(MULT(x_2_2, flag_2, var_ss_2),
    and(ADD(var_ss_1, var_ss_2, var_ss),and(SUB(one, flag_1, var_bb_1_1),
    and(MULT(x_1_2, var_bb_1_1, var_bb_1), and(SUB(one, flag_2, var_bb_2_1),
    and(MULT(x_2_2, var_bb_2_1, var_bb_2),and(ADD(var_bb_1, var_bb_2, var_bb),
    and(GE(var_bb, var_ss),GE(var_b, var_s))))))))))))))))))
    )))))))
    *));out(inloop_1, sync);0 //Refinement for comparison result

    |

    out(output_channel_2, <(*=i*)counter_1(*:Int*), (*=d*)decision(*:Int*), (*=w*)sid(*:Un*)>
    (*#
    and(and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1)),
    and(and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2)),
    and(and(and(and(P_ID_4(id_4), P_ID_5(id_5)), P_SID(w)),

        not(equal(id_4, id_5))),
    and(EQUIV(i, two),
    and(EQUIV(d, one),
    and(and(and(implies(equal(flag_1, one), GE(x_1_1, x_1_2)),
implies(equal(flag_1, null), GE(x_1_2, x_1_1))),
INPUT(id_4, <flag_1, x_1_1, x_1_2>, w)),
    and(and(and(implies(equal(flag_2, one), GE(x_2_1, x_2_2)),
implies(equal(flag_2, null), GE(x_2_2, x_2_1))),
INPUT(id_5, <flag_2, x_2_1, x_2_2>, w)),

    and(MULT(x_1_1, flag_1, var_s_1),and(MULT(x_2_1, flag_2, var_s_2),
    and(ADD(var_s_1, var_s_2, var_s),and(SUB(one, flag_1, var_b_1_1),
    and(MULT(x_1_1, var_b_1_1, var_b_1), and(SUB(one, flag_2, var_b_2_1),
    and(MULT(x_2_1, var_b_2_1, var_b_2),and(ADD(var_b_1, var_b_2, var_b),

    and(MULT(x_1_2, flag_1, var_ss_1),and(MULT(x_2_2, flag_2, var_ss_2),
    and(ADD(var_ss_1, var_ss_2, var_ss),and(SUB(one, flag_1, var_bb_1_1),
    and(MULT(x_1_2, var_bb_1_1, var_bb_1), and(SUB(one, flag_2, var_bb_2_1),
    and(MULT(x_2_2, var_bb_2_1, var_bb_2),and(ADD(var_bb_1, var_bb_2, var_bb),
    and(GE(var_bb, var_ss),GE(var_b, var_s))))))))))))))))))
    )))))))
    *));out(inloop_2, sync);0 //Refinement for comparison result


    |
```

```
    out(output_channel_3, <(*=i*)counter_1(*:Int*), (*=d*)decision(*:Int*), (*=w*)sid(*:Un*)>
    (*#
    and(and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1)),
    and(and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2)),
    and(and(and(and(P_ID_4(id_4), P_ID_5(id_5)), P_SID(w)),

        not(equal(id_4, id_5))),
    and(EQUIV(i, two),
    and(EQUIV(d, one),
    and(and(and(implies(equal(flag_1, one), GE(x_1_1, x_1_2)),
implies(equal(flag_1, null), GE(x_1_2, x_1_1))),
INPUT(id_4, <flag_1, x_1_1, x_1_2>, w)),
    and(and(and(implies(equal(flag_2, one), GE(x_2_1, x_2_2)),
implies(equal(flag_2, null), GE(x_2_2, x_2_1))),
INPUT(id_5, <flag_2, x_2_1, x_2_2>, w)),

    and(MULT(x_1_1, flag_1, var_s_1),and(MULT(x_2_1, flag_2, var_s_2),
    and(ADD(var_s_1, var_s_2, var_s),and(SUB(one, flag_1, var_b_1_1),
    and(MULT(x_1_1, var_b_1_1, var_b_1),and(SUB(one, flag_2, var_b_2_1),
    and(MULT(x_2_1, var_b_2_1, var_b_2),and(ADD(var_b_1, var_b_2, var_b),

    and(MULT(x_1_2, flag_1, var_ss_1),and(MULT(x_2_2, flag_2, var_ss_2),
    and(ADD(var_ss_1, var_ss_2, var_ss),and(SUB(one, flag_1, var_bb_1_1),
    and(MULT(x_1_2, var_bb_1_1, var_bb_1), and(SUB(one, flag_2, var_bb_2_1),
    and(MULT(x_2_2, var_bb_2_1, var_bb_2),and(ADD(var_bb_1, var_bb_2, var_bb),
    and(GE(var_bb, var_ss),GE(var_b, var_s))))))))))))))))))
    )))))))
    *));out(inloop_3, sync);0 //Refinement for comparison result

    |

    out(output_channel_4, <(*=i*)counter_1(*:Int*), (*=d*)decision(*:Int*), (*=w*)sid(*:Un*)>
    (*#
    and(and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1)),
    and(and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2)),
    and(and(and(and(P_ID_4(id_4), P_ID_5(id_5)), P_SID(w)),

        not(equal(id_4, id_5))),
    and(EQUIV(i, two),
    and(EQUIV(d, one),
    and(and(and(implies(equal(flag_1, one), GE(x_1_1, x_1_2)),
implies(equal(flag_1, null), GE(x_1_2, x_1_1))),
INPUT(id_4, <flag_1, x_1_1, x_1_2>, w)),
    and(and(and(implies(equal(flag_2, one), GE(x_2_1, x_2_2)),
implies(equal(flag_2, null), GE(x_2_2, x_2_1))),
INPUT(id_5, <flag_2, x_2_1, x_2_2>, w)),

    and(MULT(x_1_1, flag_1, var_s_1),and(MULT(x_2_1, flag_2, var_s_2),
    and(ADD(var_s_1, var_s_2, var_s),and(SUB(one, flag_1, var_b_1_1),
    and(MULT(x_1_1, var_b_1_1, var_b_1), and(SUB(one, flag_2, var_b_2_1),
    and(MULT(x_2_1, var_b_2_1, var_b_2),and(ADD(var_b_1, var_b_2, var_b),

    and(MULT(x_1_2, flag_1, var_ss_1),and(MULT(x_2_2, flag_2, var_ss_2),
    and(ADD(var_ss_1, var_ss_2, var_ss),and(SUB(one, flag_1, var_bb_1_1),
    and(MULT(x_1_2, var_bb_1_1, var_bb_1), and(SUB(one, flag_2, var_bb_2_1),
    and(MULT(x_2_2, var_bb_2_1, var_bb_2),and(ADD(var_bb_1, var_bb_2, var_bb),
    and(GE(var_bb, var_ss),GE(var_b, var_s))))))))))))))))))
    )))))))
    *));out(inloop_4, sync);0 //Refinement for comparison result

    |

    out(output_channel_5, <(*=i*)counter_1(*:Int*), (*=d*)decision(*:Int*), (*=w*)sid(*:Un*)>
    (*#
    and(and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1)),
    and(and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2)),
    and(and(and(and(P_ID_4(id_4), P_ID_5(id_5)), P_SID(w)),
```

```
        not(equal(id_4, id_5))),
    and(EQUIV(i, two),
    and(EQUIV(d, one),
    and(and(and(implies(equal(flag_1, one), GE(x_1_1, x_1_2)),
implies(equal(flag_1, null), GE(x_1_2, x_1_1))),
INPUT(id_4, <flag_1, x_1_1, x_1_2>, w)),
    and(and(and(implies(equal(flag_2, one), GE(x_2_1, x_2_2)),
implies(equal(flag_2, null), GE(x_2_2, x_2_1))),
INPUT(id_5, <flag_2, x_2_1, x_2_2>, w)),

    and(MULT(x_1_1, flag_1, var_s_1),and(MULT(x_2_1, flag_2, var_s_2),
    and(ADD(var_s_1, var_s_2, var_s),and(SUB(one, flag_1, var_b_1_1),
    and(MULT(x_1_1, var_b_1_1, var_b_1), and(SUB(one, flag_2, var_b_2_1),
    and(MULT(x_2_1, var_b_2_1, var_b_2),and(ADD(var_b_1, var_b_2, var_b),

    and(MULT(x_1_2, flag_1, var_ss_1),and(MULT(x_2_2, flag_2, var_ss_2),
    and(ADD(var_ss_1, var_ss_2, var_ss),and(SUB(one, flag_1, var_bb_1_1),
    and(MULT(x_1_2, var_bb_1_1, var_bb_1), and(SUB(one, flag_2, var_bb_2_1),
    and(MULT(x_2_2, var_bb_2_1, var_bb_2),and(ADD(var_bb_1, var_bb_2, var_bb),
    and(GE(var_bb, var_ss),GE(var_b, var_s))))))))))))))))
    )))))))
    *));out(inloop_5, sync);0 //Refinement for comparison result
    )

  else

  // The supply is for price = 2 greater than the demand: output mcp = 1 = price-1 and
  // decision = 0
  let zz = greatereq(var_ss,var_bb) in
  if zz = var_ss then
  let decision = null (*:y:Int|[*#
    and(MULT(x_1_2, flag_1, var_ss_1),and(MULT(x_2_2, flag_2, var_ss_2),
    and(ADD(var_ss_1, var_ss_2, var_ss),and(SUB(one, flag_1, var_bb_1_1),
    and(MULT(x_1_2, var_bb_1_1, var_bb_1), and(SUB(one, flag_2, var_bb_2_1),
    and(MULT(x_2_2, var_bb_2_1, var_bb_2),and(ADD(var_bb_1, var_bb_2, var_bb),GE(var_ss, var_bb)
    ))))))))
    *]*) in
  let counter_3 = sub(counter_1, one) in

(

    // At this point, deliver_1 | .. | deliver_n is placed.
    // The context SIMAP would have a hole here.
    out(output_channel_1, <(*=i*)counter_3(*:Int*), (*=d*)decision(*:Int*), (*=w*)sid(*:Un*)>
    (*#
    and(and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1)),
    and(and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2)),
    and(and(and(and(P_ID_4(id_4), P_ID_5(id_5)), P_SID(w)),

        not(equal(id_4, id_5))),
    and(EQUIV(i, two_minus_one),
    and(EQUIV(d, null),
    and(and(and(implies(equal(flag_1, one), GE(x_1_1, x_1_2)),
implies(equal(flag_1, null), GE(x_1_2, x_1_1))),
INPUT(id_4, <flag_1, x_1_1, x_1_2>, w)),
    and(and(and(implies(equal(flag_2, one), GE(x_2_1, x_2_2)),
implies(equal(flag_2, null), GE(x_2_2, x_2_1))),
INPUT(id_5, <flag_2, x_2_1, x_2_2>, w)),

    and(MULT(x_1_1, flag_1, var_s_1),and(MULT(x_2_1, flag_2, var_s_2),
    and(ADD(var_s_1, var_s_2, var_s),and(SUB(one, flag_1, var_b_1_1),
    and(MULT(x_1_1, var_b_1_1, var_b_1), and(SUB(one, flag_2, var_b_2_1),
    and(MULT(x_2_1, var_b_2_1, var_b_2),and(ADD(var_b_1, var_b_2, var_b),

    and(MULT(x_1_2, flag_1, var_ss_1),and(MULT(x_2_2, flag_2, var_ss_2),
    and(ADD(var_ss_1, var_ss_2, var_ss),and(SUB(one, flag_1, var_bb_1_1),
```

```
    and(MULT(x_1_2, var_bb_1_1, var_bb_1), and(SUB(one, flag_2, var_bb_2_1),
    and(MULT(x_2_2, var_bb_2_1, var_bb_2),and(ADD(var_bb_1, var_bb_2, var_bb),
    and(GE(var_ss, var_bb),and(GE(var_b, var_s),not(equal(var_ss, var_bb))))))))))))))))))))
    )))))))
    *)
);out(inloop_1,sync);0 //Refinement for comparison result

    |
    out(output_channel_2, <(*=i*)counter_3(*:Int*), (*=d*)decision(*:Int*), (*=w*)sid(*:Un*)>
    (*#
    and(and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1)),
    and(and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2)),
    and(and(and(and(P_ID_4(id_4), P_ID_5(id_5)), P_SID(w)),

        not(equal(id_4, id_5))),
    and(EQUIV(i, two_minus_one),
    and(EQUIV(d, null),
    and(and(and(implies(equal(flag_1, one), GE(x_1_1, x_1_2)),
implies(equal(flag_1, null), GE(x_1_2, x_1_1))),
INPUT(id_4, <flag_1, x_1_1, x_1_2>, w)),
    and(and(and(implies(equal(flag_2, one), GE(x_2_1, x_2_2)),
implies(equal(flag_2, null), GE(x_2_2, x_2_1))),
INPUT(id_5, <flag_2, x_2_1, x_2_2>, w)),

    and(MULT(x_1_1, flag_1, var_s_1),and(MULT(x_2_1, flag_2, var_s_2),
    and(ADD(var_s_1, var_s_2, var_s),and(SUB(one, flag_1, var_b_1_1),
    and(MULT(x_1_1, var_b_1_1, var_b_1), and(SUB(one, flag_2, var_b_2_1),
    and(MULT(x_2_1, var_b_2_1, var_b_2),and(ADD(var_b_1, var_b_2, var_b),

    and(MULT(x_1_2, flag_1, var_ss_1),and(MULT(x_2_2, flag_2, var_ss_2),
    and(ADD(var_ss_1, var_ss_2, var_ss),and(SUB(one, flag_1, var_bb_1_1),
    and(MULT(x_1_2, var_bb_1_1, var_bb_1), and(SUB(one, flag_2, var_bb_2_1),
    and(MULT(x_2_2, var_bb_2_1, var_bb_2),and(ADD(var_bb_1, var_bb_2, var_bb),
    and(GE(var_ss, var_bb),and(GE(var_b, var_s),not(equal(var_ss, var_bb))))))))))))))))))))
    )))))))
    *));out(inloop_2,sync);0 //Refinement for comparison result

    |

    out(output_channel_3, <(*=i*)counter_3(*:Int*), (*=d*)decision(*:Int*), (*=w*)sid(*:Un*)>
    (*#
    and(and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1)),
    and(and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2)),
    and(and(and(and(P_ID_4(id_4), P_ID_5(id_5)), P_SID(w)),

        not(equal(id_4, id_5))),
    and(EQUIV(i, two_minus_one),
    and(EQUIV(d, null),
    and(and(and(implies(equal(flag_1, one), GE(x_1_1, x_1_2)),
implies(equal(flag_1, null), GE(x_1_2, x_1_1))),
INPUT(id_4, <flag_1, x_1_1, x_1_2>, w)),
    and(and(and(implies(equal(flag_2, one), GE(x_2_1, x_2_2)),
implies(equal(flag_2, null), GE(x_2_2, x_2_1))),
INPUT(id_5, <flag_2, x_2_1, x_2_2>, w)),

    and(MULT(x_1_1, flag_1, var_s_1),and(MULT(x_2_1, flag_2, var_s_2),
    and(ADD(var_s_1, var_s_2, var_s),and(SUB(one, flag_1, var_b_1_1),
    and(MULT(x_1_1, var_b_1_1, var_b_1), and(SUB(one, flag_2, var_b_2_1),
    and(MULT(x_2_1, var_b_2_1, var_b_2),and(ADD(var_b_1, var_b_2, var_b),

    and(MULT(x_1_2, flag_1, var_ss_1),and(MULT(x_2_2, flag_2, var_ss_2),
    and(ADD(var_ss_1, var_ss_2, var_ss),and(SUB(one, flag_1, var_bb_1_1),
    and(MULT(x_1_2, var_bb_1_1, var_bb_1), and(SUB(one, flag_2, var_bb_2_1),
    and(MULT(x_2_2, var_bb_2_1, var_bb_2),and(ADD(var_bb_1, var_bb_2, var_bb),
    and(GE(var_ss, var_bb),and(GE(var_b, var_s),not(equal(var_ss, var_bb))))))))))))))))))))
    )))))))
    *));out(inloop_3,sync);0 //Refinement for comparison result
```

```
    |

    out(output_channel_4, <(*=i*)counter_3(*:Int*), (*=d*)decision(*:Int*), (*=w*)sid(*:Un*)>
    (*#
    and(and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1)),
    and(and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2)),
    and(and(and(and(P_ID_4(id_4), P_ID_5(id_5)), P_SID(w)),

        not(equal(id_4, id_5))),
    and(EQUIV(i, two_minus_one),
    and(EQUIV(d, null),
    and(and(and(implies(equal(flag_1, one), GE(x_1_1, x_1_2)),
implies(equal(flag_1, null), GE(x_1_2, x_1_1))),
INPUT(id_4, <flag_1, x_1_1, x_1_2>, w)),
    and(and(and(implies(equal(flag_2, one), GE(x_2_1, x_2_2)),
implies(equal(flag_2, null), GE(x_2_2, x_2_1))),
INPUT(id_5, <flag_2, x_2_1, x_2_2>, w)),

    and(MULT(x_1_1, flag_1, var_s_1),and(MULT(x_2_1, flag_2, var_s_2),
    and(ADD(var_s_1, var_s_2, var_s),and(SUB(one, flag_1, var_b_1_1),
    and(MULT(x_1_1, var_b_1_1, var_b_1), and(SUB(one, flag_2, var_b_2_1),
    and(MULT(x_2_1, var_b_2_1, var_b_2),and(ADD(var_b_1, var_b_2, var_b),

    and(MULT(x_1_2, flag_1, var_ss_1),and(MULT(x_2_2, flag_2, var_ss_2),
    and(ADD(var_ss_1, var_ss_2, var_ss),and(SUB(one, flag_1, var_bb_1_1),
    and(MULT(x_1_2, var_bb_1_1, var_bb_1), and(SUB(one, flag_2, var_bb_2_1),
    and(MULT(x_2_2, var_bb_2_1, var_bb_2),and(ADD(var_bb_1, var_bb_2, var_bb),
    and(GE(var_ss, var_bb),and(GE(var_b, var_s),not(equal(var_ss, var_bb))))))))))))))))))
    )))))))
    *));out(inloop_4,sync);0 //Refinement for comparison result

    |

    out(output_channel_5, <(*=i*)counter_3(*:Int*), (*=d*)decision(*:Int*), (*=w*)sid(*:Un*)>
    (*#
    and(and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1)),
    and(and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2)),
    and(and(and(and(P_ID_4(id_4), P_ID_5(id_5)), P_SID(w)),

        not(equal(id_4, id_5))),
    and(EQUIV(i, two_minus_one),
    and(EQUIV(d, null),
    and(and(and(implies(equal(flag_1, one), GE(x_1_1, x_1_2)),
implies(equal(flag_1, null), GE(x_1_2, x_1_1))),
INPUT(id_4, <flag_1, x_1_1, x_1_2>, w)),
    and(and(and(implies(equal(flag_2, one), GE(x_2_1, x_2_2)),
implies(equal(flag_2, null), GE(x_2_2, x_2_1))),
INPUT(id_5, <flag_2, x_2_1, x_2_2>, w)),

    and(MULT(x_1_1, flag_1, var_s_1),and(MULT(x_2_1, flag_2, var_s_2),
    and(ADD(var_s_1, var_s_2, var_s),and(SUB(one, flag_1, var_b_1_1),
    and(MULT(x_1_1, var_b_1_1, var_b_1), and(SUB(one, flag_2, var_b_2_1),
    and(MULT(x_2_1, var_b_2_1, var_b_2),and(ADD(var_b_1, var_b_2, var_b),

    and(MULT(x_1_2, flag_1, var_ss_1),and(MULT(x_2_2, flag_2, var_ss_2),
    and(ADD(var_ss_1, var_ss_2, var_ss),and(SUB(one, flag_1, var_bb_1_1),
    and(MULT(x_1_2, var_bb_1_1, var_bb_1), and(SUB(one, flag_2, var_bb_2_1),
    and(MULT(x_2_2, var_bb_2_1, var_bb_2),and(ADD(var_bb_1, var_bb_2, var_bb),
    and(GE(var_ss, var_bb),and(GE(var_b, var_s),not(equal(var_ss, var_bb))))))))))))))))))
    )))))))
    *));out(inloop_5, sync);0 //Refinement for comparison result
)
    )


  else
```

```
  // The supply is for price = 1 greater than the demand: output mcp = 0 = price-1 and
  // decision = 0
  let z = greatereq(var_s, var_b) in
  if z = var_s then

  let decision = null (*:y:Int|[*#
    and(MULT(x_1_1, flag_1, var_s_1),and(MULT(x_2_1, flag_2, var_s_2),
    and(ADD(var_s_1, var_s_2, var_s),and(SUB(one, flag_1, var_b_1_1),
    and(MULT(x_1_1, var_b_1_1, var_b_1), and(SUB(one, flag_2, var_b_2_1),
    and(MULT(x_2_1, var_b_2_1, var_b_2),and(ADD(var_b_1, var_b_2, var_b),GE(var_s, var_b)
    ))))))))
    *]*) in
  let counter_2 = sub(counter, one) in

     (

    // At this point, deliver_1 | .. | deliver_n is placed.
// The context SIMAP would have a hole here.
    out(output_channel_1, <(*=i*)counter_2(*:Int*), (*=d*)decision(*:Int*), (*=w*)sid_1(*:Un*)>
    (*#
    and(and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1)),
    and(and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2)),
    and(and(and(and(P_ID_4(id_4), P_ID_5(id_5)), P_SID(w)),

        not(equal(id_4, id_5))),
    and(EQUIV(i, one_minus_one),
    and(EQUIV(d, null),
    and(and(and(implies(equal(flag_1, one), GE(x_1_1, x_1_2)),
implies(equal(flag_1, null), GE(x_1_2, x_1_1))),
INPUT(id_4, <flag_1, x_1_1, x_1_2>, w)),
    and(and(and(implies(equal(flag_2, one), GE(x_2_1, x_2_2)),
implies(equal(flag_2, null), GE(x_2_2, x_2_1))),
INPUT(id_5, <flag_2, x_2_1, x_2_2>, w)),

    and(MULT(x_1_1, flag_1, var_s_1),and(MULT(x_2_1, flag_2, var_s_2),
    and(ADD(var_s_1, var_s_2, var_s),and(SUB(one, flag_1, var_b_1_1),
    and(MULT(x_1_1, var_b_1_1, var_b_1), and(SUB(one, flag_2, var_b_2_1),
    and(MULT(x_2_1, var_b_2_1, var_b_2),and(ADD(var_b_1, var_b_2, var_b),

    and(GE(var_s, var_b),not(equal(var_s, var_b))))))))))
    )))))))
    *));out(inloop_1, sync);0 //Refinement for comparison result

|

    out(output_channel_2, <(*=i*)counter_2(*:Int*), (*=d*)decision(*:Int*), (*=w*)sid_1(*:Un*)>
    (*#
    and(and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1)),
    and(and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2)),
    and(and(and(and(P_ID_4(id_4), P_ID_5(id_5)), P_SID(w)),

        not(equal(id_4, id_5))),
    and(EQUIV(i, one_minus_one),
    and(EQUIV(d, null),
    and(and(and(implies(equal(flag_1, one), GE(x_1_1, x_1_2)),
implies(equal(flag_1, null), GE(x_1_2, x_1_1))),
INPUT(id_4, <flag_1, x_1_1, x_1_2>, w)),
    and(and(and(implies(equal(flag_2, one), GE(x_2_1, x_2_2)),
implies(equal(flag_2, null), GE(x_2_2, x_2_1))),
INPUT(id_5, <flag_2, x_2_1, x_2_2>, w)),

    and(MULT(x_1_1, flag_1, var_s_1),and(MULT(x_2_1, flag_2, var_s_2),
    and(ADD(var_s_1, var_s_2, var_s),and(SUB(one, flag_1, var_b_1_1),
    and(MULT(x_1_1, var_b_1_1, var_b_1), and(SUB(one, flag_2, var_b_2_1),
    and(MULT(x_2_1, var_b_2_1, var_b_2),and(ADD(var_b_1, var_b_2, var_b),

    and(GE(var_s, var_b),not(equal(var_s, var_b))))))))))
```

```
    )))))))
    *));out(inloop_2, sync);0 //Refinement for comparison result

|
    out(output_channel_3, <(*=i*)counter_2(*:Int*), (*=d*)decision(*:Int*), (*=w*)sid_1(*:Un*)>
    (*#
    and(and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1)),
    and(and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2)),
    and(and(and(and(P_ID_4(id_4), P_ID_5(id_5)), P_SID(w)),

        not(equal(id_4, id_5))),
    and(EQUIV(i, one_minus_one),
    and(EQUIV(d, null),
    and(and(and(implies(equal(flag_1, one), GE(x_1_1, x_1_2)),
implies(equal(flag_1, null), GE(x_1_2, x_1_1))),
INPUT(id_4, <flag_1, x_1_1, x_1_2>, w)),
    and(and(and(implies(equal(flag_2, one), GE(x_2_1, x_2_2)),
implies(equal(flag_2, null), GE(x_2_2, x_2_1))),
INPUT(id_5, <flag_2, x_2_1, x_2_2>, w)),

    and(MULT(x_1_1, flag_1, var_s_1),and(MULT(x_2_1, flag_2, var_s_2),
    and(ADD(var_s_1, var_s_2, var_s),and(SUB(one, flag_1, var_b_1_1),
    and(MULT(x_1_1, var_b_1_1, var_b_1), and(SUB(one, flag_2, var_b_2_1),
    and(MULT(x_2_1, var_b_2_1, var_b_2),and(ADD(var_b_1, var_b_2, var_b),

    and(GE(var_s, var_b),not(equal(var_s, var_b))))))))))
    )))))))
    *));out(inloop_3, sync);0 //Refinement for comparison result

|
    out(output_channel_4, <(*=i*)counter_2(*:Int*), (*=d*)decision(*:Int*), (*=w*)sid_1(*:Un*)>
    (*#
    and(and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1)),
    and(and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2)),
    and(and(and(and(P_ID_4(id_4), P_ID_5(id_5)), P_SID(w)),

        not(equal(id_4, id_5))),
    and(EQUIV(i, one_minus_one),
    and(EQUIV(d, null),
    and(and(and(implies(equal(flag_1, one), GE(x_1_1, x_1_2)),
implies(equal(flag_1, null), GE(x_1_2, x_1_1))),
INPUT(id_4, <flag_1, x_1_1, x_1_2>, w)),
    and(and(and(implies(equal(flag_2, one), GE(x_2_1, x_2_2)),
implies(equal(flag_2, null), GE(x_2_2, x_2_1))),
INPUT(id_5, <flag_2, x_2_1, x_2_2>, w)),

    and(MULT(x_1_1, flag_1, var_s_1),and(MULT(x_2_1, flag_2, var_s_2),
    and(ADD(var_s_1, var_s_2, var_s),and(SUB(one, flag_1, var_b_1_1),
    and(MULT(x_1_1, var_b_1_1, var_b_1), and(SUB(one, flag_2, var_b_2_1),
    and(MULT(x_2_1, var_b_2_1, var_b_2),and(ADD(var_b_1, var_b_2, var_b),

    and(GE(var_s, var_b),not(equal(var_s, var_b))))))))))
    )))))))
    *));out(inloop_4, sync);0 //Refinement for comparison result

|
    out(output_channel_5, <(*=i*)counter_2(*:Int*), (*=d*)decision(*:Int*), (*=w*)sid_1(*:Un*)>
    (*#
    and(and(and(P_X_1_1(x_1_1), P_X_1_2(x_1_2)), P_FLAG_1(flag_1)),
    and(and(and(P_X_2_1(x_2_1), P_X_2_2(x_2_2)), P_FLAG_2(flag_2)),
    and(and(and(and(P_ID_4(id_4), P_ID_5(id_5)), P_SID(w)),

        not(equal(id_4, id_5))),
    and(EQUIV(i, one_minus_one),
    and(EQUIV(d, null),
    and(and(and(implies(equal(flag_1, one), GE(x_1_1, x_1_2)),
implies(equal(flag_1, null), GE(x_1_2, x_1_1))),
```

```
INPUT(id_4, <flag_1, x_1_1, x_1_2>, w)),
    and(and(and(implies(equal(flag_2, one), GE(x_2_1, x_2_2)),
implies(equal(flag_2, null), GE(x_2_2, x_2_1))),
INPUT(id_5, <flag_2, x_2_1, x_2_2>, w)),

    and(MULT(x_1_1, flag_1, var_s_1),and(MULT(x_2_1, flag_2, var_s_2),
    and(ADD(var_s_1, var_s_2, var_s),and(SUB(one, flag_1, var_b_1_1),
    and(MULT(x_1_1, var_b_1_1, var_b_1), and(SUB(one, flag_2, var_b_2_1),
    and(MULT(x_2_1, var_b_2_1, var_b_2),and(ADD(var_b_1, var_b_2, var_b),

    and(GE(var_s, var_b),not(equal(var_s, var_b))))))))))
    )))))))
    *));out(inloop_5, sync);0 //Refinement for comparison result
     )
  )
).

(*************************************************************************
 Arithmetic theorems

 This process introduces additional theorems about the arithmetic operations.
*************************************************************************)
let assumptions = (
  assume(*#forall([x,y], implies(Red(greatereq(x,y),x), GE(x,y)))*) |
  assume(*#forall([x,y,z], implies(Red(add(x,y),z), ADD(x,y,z)))*) |
  assume(*#forall([x,y,z], implies(Red(mult(x,y),z), MULT(x,y,z)))*) |
  assume(*#forall([x,y,z], implies(Red(sub(x,y),z), SUB(x,y,z)))*) |
  assume(*#forall([x,y,z_1, z_2], implies(and(Red(add(x,y),z_1), Red(add(x,y),z_2)),
                                EQUIV(z_1, z_2)))*) |
    assume(*#forall([x,z_1, z_2], implies(and(Red(idh(x),z_1), Red(idh(x),z_2)),
                                EQUIV(z_1, z_2)))*) |
    assume(*#forall([x,y], implies(equal(x,y), EQUIV(x, y)))*)
).

(*************************************************************************
 final process

 The final process contains all other processes as subprocesses and provides
 and environment. First, it introduces all names and several refinement type
 for channels, which are used by the type system for verifying the policy.
*************************************************************************)
free adv.

process
(
  assumptions |

  new start_round (*:Private*); new sync (*:Private*);
  new id_1 (*:Private*); new id_2 (*:Private*);
  new id_3 (*:Private*); new id_4 (*:Private*);
  new id_5 (*:Private*);
  new sid (*:Un*);
  new sidc (*:Ch(y:Un | [*#and(P_SID(y), equal(y,sid))*])*);
  let null = string_null(empty()) in
  let one = string_one(empty()) in
  let one_minus_one = sub(one, one) in
  let two = add(one, one) in
  let two_minus_one = sub(two, one) in
  (
  assume(*#P_SID(sid)*) | assume(*#P_ID_1(id_1)*) |
  assume(*#P_ID_2(id_2)*) | assume(*#P_ID_3(id_3)*) |
  assume(*#P_ID_4(id_4)*) | assume(*#P_ID_5(id_5)*) |

  assume(*#not(equal(id_1, id_2))*) | assume(*#not(equal(id_1, id_3))*) |
  assume(*#not(equal(id_1, id_4))*) | assume(*#not(equal(id_1, id_5))*) |
  assume(*#not(equal(id_2, id_3))*) | assume(*#not(equal(id_2, id_4))*) |
  assume(*#not(equal(id_2, id_5))*) | assume(*#not(equal(id_3, id_4))*) |
```

```
  assume(*#not(equal(id_3, id_5))*) | assume(*#not(equal(id_4, id_5))*) |

  // The certification issuer channels
  new certification_channel_4(*:Ch(<signature:Signed(T_cert), verification_key:VerKey(T_cert)>[*#
    Red(check(signature, verification_key),
      <id_4, id_1, id_2, id_3, id_4, id_5, sid, two>)
  *])*);
  new certification_channel_5(*:Ch(<signature:Signed(T_cert), verification_key:VerKey(T_cert)>[*#
    Red(check(signature, verification_key),
      <id_5, id_1, id_2, id_3, id_4, id_5, sid, two>)
  *])*);

  // The input channels for the SMPC
  new in_1 (*: Ch(<T_sid, Private>)*);
  new in_2 (*: Ch(<T_sid, Private>)*);
  new in_3 (*: Ch(<T_sid, Private>)*);
  new in_4 (*:Ch(<bsid:Un, <<f:Int, x1:Int, x2:Int>
  [*#
      and(or(equal(f, null), equal(f, one)),
      and(and(and(P_X_1_1(x1), P_X_1_2(x2)), P_FLAG_1(f)),
      and(and(GE(x1, null), GE(x2, null)),
      and(and(
  implies(equal(f, one), GE(x1, x2)),
implies(equal(f, null), GE(x2, x1))),
INPUT(id_4, <f, x1, x2>, sid))
    )))
  *], <signature:Signed(T_cert), verification_key:VerKey(T_cert)>[*#
          Red(check(signature, verification_key),
           <id_4, id_1, id_2, id_3, id_4, id_5, sid, two>)
   *]>>)*);
  new in_5 (*:Ch(<bsid:Un, <<f:Int, x1:Int, x2:Int>
  [*#
    and(or(equal(f, null), equal(f, one)),
    and(and(and(P_X_2_1(x1), P_X_2_2(x2)), P_FLAG_2(f)),
    and(and(GE(x1, null), GE(x2, null)),
    and(and(
  implies(equal(f, one), GE(x1, x2)),
implies(equal(f, null), GE(x2, x1))),
INPUT(id_5, <f, x1, x2>, sid))
    )))
  *], <signature:Signed(T_cert), verification_key:VerKey(T_cert)>[*#
          Red(check(signature, verification_key),
           <id_5, id_1, id_2, id_3, id_4, id_5, sid, two>)
   *]>>)*);

  // The output channel of the SMPC:
  // We separate input and output channels
  // as the typechecker would otherwise be overloaded.
  new output_channel_1 (*:Ch(T_comparison_result)*);
  new output_channel_2 (*:Ch(T_comparison_result)*);
  new output_channel_3 (*:Ch(T_comparison_result)*);
  new output_channel_4 (*:Ch(T_comparison_result)*);
  new output_channel_5 (*:Ch(T_comparison_result)*);

  // Send the session identifier to the process SMPC.
  out(sidc, sid(*:y:Un | [*#and(P_SID(y), equal(y,sid))*]*));
  (

  // The obligatory passive adversary
  !in(adv, input_variable) |

  certification_issuer |
  server_1 |
  server_2 |
  server_3 |
  SMPC |
  bidder_1 |
```

```
  bidder_2 |
  MCP_Policy
  )
  )
).
```

Appendix B.

Bibliography

Research papers on which this thesis is based

[BMM10] M. Backes, M. Maffei, and E. Mohammadi. "Computationally Sound Abstraction and Verification of Secure Multi-Party Computations". In: *Proc. 30th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 352–363.

[BGKM12] M. Backes, I. Goldberg, A. Kate, and E. Mohammadi. "Provably Secure and Practical Onion Routing". In: *Proc. 25th IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society Press, 2012, pp. 369–385.

[BKM12] M. Backes, A. Kate, and E. Mohammadi. "Ace: an efficient key-exchange protocol for onion routing". In: *Proc. 11th annual ACM Workshop on Privacy in the Electronic Society (WPES)*. ACM Press, 2012, pp. 55–64.

[BMR14] M. Backes, E. Mohammadi, and T. Ruffing. "Computational Soundness Results for ProVerif". In: *Proc. 3rd Conference on Principles of Security and Trust (POST)*. Springer, 2014, pp. 42–62.

[BBMMP15] M. Backes, F. Bendun, M. Maffei, E. Mohammadi, and K. Pecina. "A Computationally Sound, Symbolic Abstraction for Malleable Zero-knowledge Proofs". In: *Proc. 28th IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society Press, 2015, pp. 412–480.

[BMR15] M. Backes, E. Mohammadi, and T. Ruffing. "Computational Soundness for Interactive Primitives for Equivalence Properties". In: *Proc. 20th European Symposium on Research in Computer Security (ESORICS)*. Springer, 2015, pp. 125–145.

Other research papers of the author

[BMM14] M. Backes, P. Manoharan, and E. Mohammadi. "TUC: Time-sensitive and Modular Analysis of Anonymous Communication". In: *Proc. 27th IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society Press, 2014, pp. 383–397.

[BKMMM13] M. Backes, A. Kate, P. Manoharan, S. Meiser, and E. Mohammadi. "AnoA: A Framework For Analyzing Anonymous Communication Protocols". In: *Proc. 26th IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society Press, 2013, pp. 163–178.

[BKMM14] M. Backes, A. Kate, S. Meiser, and E. Mohammadi. "(Nothing else) MATor(s): Monitoring the Anonymity of Tor's Path Selection". In: *Proc. 21st ACM Conference on Computer and Communication Security (CCS)*. ACM Press, 2014, pp. 513–524.

[BFM13] M. Backes, D. Fiore, and E. Mohammadi. "Privacy-Preserving Accountable Computation". In: *Proc. 18th European Symposium on Research in Computer Security (ESORICS)*. Springer, 2013, pp. 38–56.

[PMP14] K. Pecina, E. Mohammadi, and C. Pöpper. "Zero-Communication Seed Establishment for Anti-Jamming Techniques". In: *Proc. 1st NDSS Workshop on Security of Emerging Networking Technologies (SENT)*. Internet Society, 2014.

Related Bachelor's & Master's Thesis

[Moh09] E. Mohammadi. *Computational Soundness for Symbolic Zero - Knowledge Proofs Against Active Attackers under Relaxed Assumptions*. Master's Thesis. 2009.

[Ruf12] T. Ruffing. *Computational Soundness of Interactive Primitives*. Bachelor's Thesis. 2012.

Other Papers

[ABW06] M. Abadi, M. Baudet, and B. Warinschi. "Guessing Attacks and the Computational Soundness of Static Equivalence". In: *Proc. 9th International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*. Springer, 2006, pp. 398–412.

[AF01] M. Abadi and C. Fournet. "Mobile Values, New Names, and Secure Communication". In: *Proc. 28th Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2001, pp. 104–115.

[AF06] P. Adão and C. Fournet. "Cryptographically Sound Implementations for Communicating Processes". In: *Proc. 33rd International Colloquium on Automata, Languages and Programming (ICALP)*. Springer, 2006, pp. 83–94.

[AJ01] M. Abadi and J. Jürjens. "Formal Eavesdropping and its Computational Interpretation". In: *Proc. 4th International Symposium on Theoretical Aspects of Computer Software (TACS)*. Springer, 2001, pp. 82–94.

[AP03] S. S. Al-Riyami and K. G. Paterson. "Certificateless Public Key Cryptography". In: *Advances in Cryptology – ASIACRYPT*. Springer, 2003, pp. 452–473.

[App14] Apple Inc. *Apple support: About the security content of iOS 7.0.6*. http://support.apple.com/kb/HT6147. Accessed in September 2014. 2014.

[AR02] M. Abadi and P. Rogaway. "Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption)". In: *Journal of Cryptology* 15.2 (2002), pp. 103–127.

[Ava05] R. M. Avanzi. "The Complexity of Certain Multi-Exponentiation Techniques in Cryptography". In: *Journal of Cryptology* 18.4 (2005), pp. 357–373.

[BAF05] B. Blanchet, M. Abadi, and C. Fournet. "Automated Verification of Selected Equivalences for Security Protocols". In: *Proc. 20th IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society Press, 2005, pp. 331–340.

[BBU13] M. Backes, F. Bendun, and D. Unruh. "Computational Soundness of Symbolic Zero-knowledge Proofs: Weaker Assumptions and Mechanized Verification". In: *Proc. 2nd Conference on Principles of Security and Trust (POST)*. Springer, 2013, pp. 206–225.

[BCCKLS09] M. Belenkiy, J. Camenisch, M. Chase, M. Kohlweiss, A. Lysyanskaya, and H. Shacham. "Randomizable Proofs and Delegatable Anonymous Credentials". In: *Advances in Cryptology – CRYPTO*. Springer, 2009, pp. 108–125.

[BCJSW06] F. Butler, I. Cervesato, A. D. Jaggard, A. Scedrov, and C. Walstad. "Formal Analysis of Kerberos 5". In: *Theoretical Computer Science* 367.1 (2006), pp. 57–87.

[BCK05] M. Baudet, V. Cortier, and S. Kremer. "Computationally Sound Implementations of Equational Theories against Passive Adversaries". In: *Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP)*. Springer, 2005, pp. 652–663.

[BCW13] F. Böhl, V. Cortier, and B. Warinschi. "Deduction Soundness: Prove One, Get Five for Free". In: *Proc. 20th ACM Conference on Computer and Communication Security (CCS)*. ACM Press, 2013, pp. 1261–1272.

[BDLSY11] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. "High-Speed High-Security Signatures". In: *Proc. 13th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 2011, pp. 124–142.

[Ber06] D. J. Bernstein. "Curve25519: New Diffie-Hellman Speed Records". In: *Proc. 9th Conference on Theory and Practice of Public-Key Cryptography (PKC)*. Springer, 2006, pp. 207–228.

[BH04] M. Backes and D. Hofheinz. "How to Break and Repair a Universally Composable Signature Functionality". In: *Proc. 6th Information Security Conference (ISC)*. Springer, 2004, pp. 61–72.

[BHM08a] M. Backes, C. Hritcu, and M. Maffei. "Automated Verification of Remote Electronic Voting Protocols in the Applied Pi-Calculus". In: *Proc. 21st IEEE Symposium on Computer Security Foundations (CSF)*. IEEE Computer Society Press, 2008, pp. 195–209.

[BHM08b] M. Backes, C. Hriţcu, and M. Maffei. “Type-checking Zero-knowledge”. In: *Proc. of the 15th ACM conference on Computer and communications security (CCS)*. ACM Press, 2008, pp. 357–370.

[BHM12] M. Backes, C. Hriţcu, and M. Maffei. “Union and Intersection Types for Secure Protocol Implementations”. In: *Proc. 2011 International Conference on Theory of Security and Applications (TOSCA, now POST)*. Springer, 2012, pp. 1–28.

[BHU09] M. Backes, D. Hofheinz, and D. Unruh. “CoSP: A General Framework for Computational Soundness Proofs”. In: *Proc. 16th ACM Conference on Computer and Communication Security (CCS)*. ACM Press, 2009, pp. 66–78.

[BJP02] M. Backes, C. Jacobi, and B. Pfitzmann. “Deriving Cryptographically Sound Implementations Using Composition and Formally Verified Bisimulation”. In: *Proc. 10th International Symposium of Formal Methods (FME)*. Springer, 2002, pp. 310–329.

[BL06] M. Backes and P. Laud. “Computationally Sound Secrecy Proofs by Mechanized Flow Analysis”. In: *Proc. 13th ACM Conference on Computer and Communications Security (CCS)*. ACM Press, 2006, pp. 370–379.

[Bla01] B. Blanchet. “An Efficient Cryptographic Protocol Verifier Based on Prolog Rules”. In: *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society Press, 2001, pp. 82–96.

[Ble98] D. Bleichenbacher. “Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS”. In: *Advances in Cryptology – CRYPTO*. Springer, 1998, pp. 1–12.

[BLMP10] M. Backes, S. Lorenz, M. Maffei, and K. Pecina. “Anonymous Webs of Trust”. In: *Proc. 10h Privacy Enhancing Technologies Symposium (PETS)*. Springer, 2010, pp. 130–148.

[BMP00] V. Boyko, P. D. MacKenzie, and S. Patel. “Provably Secure Password-Authenticated Key Exchange Using Diffie-Hellman”. In: *Advances in Cryptology – EUROCRYPT*. Springer, 2000, pp. 156–171.

[BMP12] M. Backes, M. Maffei, and K. Pecina. “Automated Synthesis of Privacy-Preserving Distributed Applications”. In: *Proc. 20th Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2012.

[BMU08] M. Backes, M. Maffei, and D. Unruh. “Zero-Knowledge in the Applied Pi-calculus and Automated Verification of the Direct Anonymous Attestation Protocol”. In: *Proc. 29th IEEE Symposium on Security & Privacy (S&P)*. IEEE Computer Society Press, 2008, pp. 158–169.

[BMU10] M. Backes, M. Maffei, and D. Unruh. “Computationally Sound Verification of Source Code”. In: *Proc. 17th ACM Conference on Computer and Communication Security (CCS)*. ACM Press, 2010, pp. 387–398.

[BMU12] M. Backes, A. Malik, and D. Unruh. “Computational Soundness without Protocol Restrictions”. In: *Proc. 19th ACM Conference on Computer and Communication Security (CCS)*. ACM Press, 2012, pp. 699–711.

[BNP08] A. Ben-David, N. Nisan, and B. Pinkas. "FairplayMP: A System for Secure Multi-Party Computation". In: *Proc. of the 15th ACM conference on Computer and communications security (CCS)*. ACM Press, 2008, pp. 257–266.

[Bog+09] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. P. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. I. Schwartzbach, and T. Toft. "Secure Multiparty Computation Goes Live". In: *Proc. 13th Conference on Financial Cryptography and Data Security (FC)*. Springer, 2009, pp. 325–343.

[BP04] M. Backes and B. Pfitzmann. "Symmetric Encryption in a Simulatable Dolev-Yao Style Cryptographic Library". In: *Proc. 17th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society Press, 2004, pp. 204–218.

[BP05] M. Backes and B. Pfitzmann. "Limits of the Cryptographic Realization of Dolev-Yao-style XOR". In: *Proc. 10th European Symposium on Research in Computer Security (ESORICS)*. Springer, 2005, pp. 178–196.

[BPW03a] M. Backes, B. Pfitzmann, and M. Waidner. "A Composable Cryptographic Library with Nested Operations". In: *Proc. 10th ACM Conference on Computer and Communications Security (CCS)*. ACM Press, 2003, pp. 220–230.

[BPW03b] M. Backes, B. Pfitzmann, and M. Waidner. "Symmetric Authentication Within a Simulatable Cryptographic Library". In: *Proc. 8th European Symposium on Research in Computer Security (ESORICS)*. Springer, 2003, pp. 271–290.

[BPW07] M. Backes, B. Pfitzmann, and M. Waidner. "The Reactive Simulatability (RSIM) Framework for Asynchronous Systems". In: *Information and Computation* 205.12 (2007), pp. 1685–1720.

[BU10] M. Backes and D. Unruh. "Computational Soundness of Symbolic Zero-Knowledge Proofs". In: *Journal of Computer Security* 18.6 (2010), pp. 1077–1155.

[Can01] R. Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *Proc. 42nd IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society Press, 2001, pp. 136–145.

[Can04] R. Canetti. "Universally Composable Signatures, Certification and Authorization". In: *Proc. 17th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society Press, 2004, pp. 219–233.

[CC08] H. Comon-Lundh and V. Cortier. "Computational Soundness of Observational Equivalence". In: *Proc. of the 15th ACM conference on Computer and communications security (CCS)*. ACM Press, 2008, pp. 109–118.

[CCD10] V. Cheval, H. Comon-Lundh, and S. Delaune. "Automating Security Analysis: Symbolic Equivalence of Constraint Systems". In: *Proc. 5th International Joint Conference on Automated Reasoning (IJCAR)*. Springer, 2010, pp. 412–426.

[CCD11] V. Cheval, H. Comon-Lundh, and S. Delaune. “Trace Equivalence Decision: Negative Tests and Non-determinism”. In: *Proc. 18th ACM Conference on Computer and Communication Security (CCS)*. ACM Press, 2011, pp. 321–330.

[CCD13] V. Cheval, V. Cortier, and S. Delaune. “Deciding Equivalence-Based Properties Using Constraint Solving”. In: *Theoretical Computer Science* 492 (2013).

[CCP13] V. Cheval, V. Cortier, and A. Plet. “Lengths May Break Privacy – Or How to Check for Equivalences with Length”. In: *Proc. 25th International Conference Computer Aided Verification (CAV)*. Springer, 2013, pp. 708–723.

[CCS12] H. Comon-Lundh, V. Cortier, and G. Scerri. “Security Proof with Dishonest Keys”. In: *Proc. 1st Conference on Principles of Security and Trust (POST)*. Springer, 2012, pp. 149–168.

[CDPW07] R. Canetti, Y. Dodis, R. Pass, and S. Walfish. “Universally Composable Security with Global Setup”. In: *Proc. 4th Theory of Cryptography Conference (TCC)*. Springer, 2007, pp. 61–85.

[CFG09] D. Catalano, D. Fiore, and R. Gennaro. “Certificateless Onion Routing”. In: *Proc. 16th ACM Conference on Computer and Communication Security (CCS)*. ACM Press, 2009, pp. 151–160.

[CGS08] N. Chandran, V. Goyal, and A. Sahai. “New Constructions for UC Secure Computation Using Tamper-Proof Hardware”. In: *Advances in Cryptology – EUROCRYPT*. Springer, 2008, pp. 545–562.

[CH06] R. Canetti and J. Herzog. “Universally Composable Symbolic Analysis of Mutual Authentication and Key Exchange Protocols”. In: *Proc. 3rd Theory of Cryptography Conference (TCC)*. Springer, 2006, pp. 380–403.

[CH11] R. Canetti and J. Herzog. “Universally Composable Symbolic Security Analysis”. In: *Journal of Cryptology* 24.1 (2011), pp. 83–147.

[Cha82] D. Chaum. “Blind Signatures for Untraceable Payments”. In: *Advances in Cryptology – CRYPTO*. Plenum Press, New York, 1982, pp. 199–203.

[Che] V. Cheval. *APTE (Algorihm for Proving Trace Equivalence)*. http://projects.lsv.ens-cachan.fr/APTE/. Accessed in October 2013.

[CHKLN11] J. Camenisch, K. Haralambiev, M. Kohlweiss, J. Lapon, and V. Naessens. “Structure preserving CCA secure encryption and applications”. In: *Advances in Cryptology – ASIACRYPT*. Springer, 2011, pp. 89–106.

[CHKS12] H. Comon-Lundh, M. Hagiya, Y. Kawamoto, and H. Sakurada. “Computational Soundness of Indistinguishability Properties without Computable Parsing”. In: *Proc. 8th International Conference on Information security practice and experience (ISPEC)*. Springer, 2012, pp. 63–79.

[CK11] M. Chase and M. Kohlweiss. *A Domain Transformation for Structure-Preserving Signatures on Group Elements*. Cryptology ePrint Archive, Report 2011/342. 2011.

[CKKW06] V. Cortier, S. Kremer, R. Küsters, and B. Warinschi. "Computationally Sound Symbolic Secrecy in the Presence of Hash Functions". In: *Proc. 26th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. Springer, 2006, pp. 176–187.

[CKLM12] M. Chase, M. Kohlweiss, A. Lysyanskaya, and S. Meiklejohn. "Malleable Proof Systems and Applications". In: *Advances in Cryptology – EUROCRYPT*. Springer, 2012, pp. 281–300.

[CL05] J. Camenisch and A. Lysyanskaya. "A Formal Treatment of Onion Routing". In: *Advances in Cryptology – CRYPTO*. Springer, 2005, pp. 169–187.

[CLOS02] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. "Universally Composable Two-party and Multi-party Secure Computation". In: *Proc. 34th Annual ACM Symposium on Theory of Computing (STOC)*. ACM Press, 2002, pp. 494–503.

[Com08] H. Comon-Lundh. "About Models of Security Protocols". In: *Proc. 28th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2008, pp. 352–356.

[CW05] V. Cortier and B. Warinschi. "Computationally Sound, Automated Proofs for Security Protocols". In: *Proc. 14th European Symposium on Programming (ESOP)*. Springer, 2005, pp. 157–171.

[CW11] V. Cortier and B. Warinschi. "A Composable Computational Soundness Notion". In: *Proc. 18th ACM Conference on Computer and Communication Security (CCS)*. ACM Press, 2011, pp. 63–74.

[CW12] V. Cortier and C. Wiedling. "A Formal Analysis of the Norwegian E-Voting Protocol". In: *Proc. 1st Conference on Principles of Security and Trust (POST)*. Springer, 2012, pp. 109–128.

[DDMRS06] A. Datta, A. Derek, J. Mitchell, A. Ramanathan, and A. Scedrov. "Games and the Impossibility of Realizable Ideal Functionality". In: *Proc. 3rd Theory of Cryptography Conference (TCC)*. Springer, 2006, pp. 360–379.

[DKP09] S. Delaune, S. Kremer, and O. Pereira. "Simulation-Based Security in the Applied Pi Calculus". In: *Proc. 29th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2009, pp. 169–180.

[DKR09] S. Delaune, S. Kremer, and M. Ryan. "Verifying Privacy-Type Properties of Electronic Voting Protocols". In: *Journal of Computer Security* 17.4 (2009), pp. 435–487.

[DKRS11] S. Delaune, S. Kremer, M. D. Ryan, and G. Steel. "Formal Analysis of Protocols Based on TPM State Registers". In: *Proc. 24th IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society Press, 2011, pp. 66–80.

[DM08] R. Dingledine and N. Mathewson. *Tor Protocol Specification*. https://gitweb.torproject.org/torspec.git?a=blob_plain;hb=HEAD;f=tor-spec.txt. Accessed in November 2011. 2008.

[DMS04] R. Dingledine, N. Mathewson, and P. Syverson. "Tor: The Second-Generation Onion Router". In: *Proc. 13th USENIX Security Symposium (USENIX)*. USENIX Association, 2004, pp. 303–320.

[DOW92] W. Diffie, P. C. van Oorschot, and M. J. Wiener. "Authentication and Authenticated Key Exchanges". In: *Designs, Codes and Cryptography* 2.2 (1992), pp. 107–125.

[DR06] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol (Version 1.1), Request for Comments (RFC) 4346*. http://www.ietf.org/rfc/rfc4346.txt. 2006.

[Duc14] P. Ducklin. *Anatomy of a "goto fail" – Apple's SSL bug explained, plus an unofficial patch for OS X!* http://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/. Accessed in October 2014. 2014.

[DY83] D. Dolev and A. C. Yao. "On the Security of Public Key Protocols". In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–208.

[EG83] S. Even and O. Goldreich. "On the Security of Multi-Party Ping-Pong Protocols". In: *Proc. 24th IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society Press, 1983, pp. 34–39.

[FGM07] C. Fournet, A. D. Gordon, and S. Maffeis. "A Type Discipline for Authorization in Distributed Systems". In: *Proc. 20th IEEE Symposium on Computer Security Foundations (CSF)*. IEEE Computer Society Press, 2007, pp. 31–45.

[Fis03] D. Fisher. "Millions of .Net Passport Accounts Put at Risk". In: *eWeek* (2003). (Flaw detected by Muhammad Faisal Rauf Danka.) Accessed in August 2014. URL: http://www.eweek.com/c/a/Security/Millions-of-Net-Passport-Accounts-Put-at-Risk/.

[Fis06] M. Fischlin. "Round-Optimal Composable Blind Signatures in the Common Reference String Model". In: *Advances in Cryptology – CRYPTO*. Springer, 2006, pp. 60–77.

[FJS07a] J. Feigenbaum, A. Johnson, and P. F. Syverson. "A Model of Onion Routing with Provable Anonymity". In: *Proc. 11th Conference on Financial Cryptography and Data Security (FC)*. Springer, 2007, pp. 57–71.

[FJS07b] J. Feigenbaum, A. Johnson, and P. F. Syverson. "Probabilistic Analysis of Onion Routing in a Black-Box Model". In: *Proc. 6th ACM Workshop on Privacy in the Electronic Society (WPES)*. ACM Press, 2007, pp. 1–10.

[FJS11] J. Feigenbaum, A. Johnson, and P. F. Syverson. "Probabilistic Analysis of Onion Routing in a Black-Box Model". In: *ACM Transactions on Information and System Security* 15.14 (2011).

[FP09] G. Fuchsbauer and D. Pointcheval. "Proofs on Encrypted Values in Bilinear Groups and an Application to Anonymity of Signatures". In: *Proc. 3rd International Conference on Pairing-Based Cryptography (Pairing)*. Springer, 2009, pp. 132–149.

[Fuc10] G. Fuchsbauer. “Automorphic Signatures and Applications”. PhD thesis. École normale supérieure, Paris, 2010.

[Gen34] G. Gentzen. “Untersuchungen über das logische Schließen I”. In: *Mathematische Zeitschrift* 39.2 (1934), pp. 176–210.

[GGV08] D. Galindo, F. D. Garcia, and P. Van Rossum. “Computational Soundness of Non-Malleable Commitments”. In: *Proc. 4th International Conference on Information security practice and experience (ISPEC)*. Springer, 2008, pp. 361–376.

[GL01] O. Goldreich and Y. Lindell. “Session-Key Generation Using Human Passwords Only”. In: *Advances in Cryptology – CRYPTO*. Springer, 2001, pp. 408–432.

[GMR89] S. Goldwasser, S. Micali, and C. Rackoff. “The Knowledge Complexity of Interactive Proof Systems”. In: *SIAM Journal on Computing* 18.1 (1989), pp. 186–207.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. “How to Play Any Mental Game – OR – A Completeness Theorem for Protocols with Honest Majority”. In: *Proc. 19th Annual ACM Symposium on Theory of Computing (STOC)*. ACM Press, 1987, pp. 218–229.

[Gol06] I. Goldberg. “On the Security of the Tor Authentication Protocol”. In: *Proc. 6th Workshop on Privacy Enhancing Technologies (PET)*. ACM Press, 2006, pp. 316–331.

[Gre11] A. Greenberg. *How To Protect Yourself Online Like An Arab Revolutionary*. Forbes.com LLC. http://blogs.forbes.com/andygreenberg/2011/03/25/. Accessed in January 2012. 2011.

[GRS96] D. M. Goldschlag, M. Reed, and P. Syverson. “Hiding Routing Information”. In: *Proc. 1st Workshop on Information Hiding*. Springer, 1996, pp. 137–150.

[GRS99] D. M. Goldschlag, M. G. Reed, and P. F. Syverson. “Onion Routing”. In: *Communications of the ACM* 42.2 (1999), pp. 39–41.

[GS08] J. Groth and A. Sahai. “Efficient Non-Interactive Proof Systems for Bilinear Groups”. In: *Advances in Cryptology – EUROCRYPT*. Springer, 2008, pp. 415–432.

[GSU12] I. Goldberg, D. Stebila, and B. Ustaoglu. “Anonymity and One-Way Authentication in Key Exchange Protocols”. In: *Designs, Codes and Cryptography* (2012). Proposal for Tor: https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/ideas/xxx-ntor-handshake.txt, pp. 1–25.

[GTL89] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types*. Cambridge University Press, 1989.

[HLM03] J. Herzog, M. Liskov, and S. Micali. “Plaintext Awareness via Key Registration”. In: *Advances in Cryptology – CRYPTO*. LNCS. Springer, 2003, pp. 548–564.

[Hof11] D. Hofheinz. “Possibility and Impossibility Results for Selective Decommitments”. In: *Journal of Cryptology* 24.3 (2011), pp. 470–516.

[JLM05] R. Janvier, Y. Lakhnech, and L. Mazaré. "Completing the Picture: Soundness of Formal Encryption in the Presence of Active Adversaries". In: *Proc. 14th European Symposium on Programming (ESOP)*. Springer, 2005, pp. 172–185.

[KG10a] A. Kate and I. Goldberg. "Distributed Private-Key Generators for Identity-Based Cryptography". In: *Proc. 7th Conference on Security and Cryptography for Networks (SCN)*. Springer, 2010, pp. 436–453.

[KG10b] A. Kate and I. Goldberg. "Using Sphinx to Improve Onion Routing Circuit Construction". In: *Proc. 14th Conference on Financial Cryptography and Data Security (FC)*. Springer, 2010, pp. 359–366.

[KKW05] D. Kähler, R. Küsters, and T. Wilke. "Deciding Properties of Contract-Signing Protocols". In: *Proc. 22nd Annual Symposium on Theoretical Aspects of Computer Science (STACS)*. Springer, 2005, pp. 158–169.

[KMM94] R. Kemmerer, C. Meadows, and J. Millen. "Three Systems for Cryptographic Protocol Analysis". In: *Journal of Cryptology* 7.2 (1994), pp. 79–130.

[KO12] K. Kurosawa and Y. Ohtaki. "UC-Secure Searchable Symmetric Encryption". In: *Proc. 16th Conference on Financial Cryptography and Data Security (FC)*. Springer, 2012, pp. 285–298.

[KR05] S. Kremer and M. Ryan. "Analysis of an Electronic Voting Protocol in the Applied Pi-Calculus". In: *Proc. 14th European Symposium on Programming (ESOP)*. Springer, 2005, pp. 186–200.

[KTG12] R. Küsters, T. Truderung, and J. Graf. "A Framework for the Cryptographic Verification of Java-like Programs". In: *Proc. 25th IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society Press, 2012, pp. 198–212.

[KZG07] A. Kate, G. M. Zaverucha, and I. Goldberg. "Pairing-Based Onion Routing". In: *Proc. 7th Privacy Enhancing Technologies Symposium (PETS)*. Springer, 2007, pp. 95–112.

[KZG10] A. Kate, G. M. Zaverucha, and I. Goldberg. "Pairing-Based Onion Routing with Improved Forward Secrecy". In: *ACM Transactions on Information and System Security* 13.4 (2010), p. 29.

[Lau01] P. Laud. "Semantics and Program Analysis of Computationally Secure Information Flow". In: *Proc. 10th European Symposium on Programming (ESOP)*. Springer, 2001, pp. 77–91.

[Lau04] P. Laud. "Symmetric Encryption in Automatic Analyses for Confidentiality against Active Adversaries". In: *Proc. 25th IEEE Symposium on Security & Privacy (S&P)*. IEEE Computer Society Press, 2004, pp. 71–85.

[Mat12a] N. Mathewson. *Another key exchange algorithm for extending circuits: alternative to ntor?* The tor-dev mailing list. https://lists.torproject.org/pipermail/tor-dev/2012-August/003901.html. Accessed in August 2012. 2012.

[Mat12b] N. Mathewson. *Talk on Tor*. Rump Session in the 12th Privacy Enhancing Technologies Symposium (PETS). 2012.

[MB09] P. Mittal and N. Borisov. “ShadowWalker: Peer-To-Peer Anonymous Communication Using Redundant Structured Topologies”. In: *Proc. 16th ACM Conference on Computer and Communication Security (CCS)*. ACM Press, 2009, pp. 161–172.

[MIT] MIT Kerberos and Internet trust (MIT-KIT) Consortium. *Official Web Page*. http://www.kerberos.org/. Accessed in June 2014.

[MN96] D. M’Raïhi and D. Naccache. “Batch Exponentiation: A Fast DLP-Based Signature Generation Strategy”. In: *Proc. 3rd ACM Conference on Computer and Communications Security (CCS)*. ACM Press, 1996, pp. 58–61.

[MNPS04] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. “Fairplay – Secure Two-Party Computation System”. In: *Proc. 13th USENIX Security Symposium (USENIX)*. USENIX Association, 2004, pp. 287–302.

[MOV97] A. Menezes, P. V. Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. 1st ed. CRC Press, 1997.

[MP11] M. Maffei and K. Pecina. “Privacy-aware Proof-Carrying Authorization”. In: *Proc. 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*. ACM Press, 2011, Article No. 7.

[MPR13] M. Maffei, K. Pecina, and M. Reinert. “Security and Privacy by Declarative Design”. In: *Proc. 26th IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society Press, 2013, pp. 81–96.

[MSCB13] S. Meier, B. Schmidt, C. J. F. Cremers, and D. A. Basin. “The TAMARIN Prover for the Symbolic Analysis of Security Protocols”. In: *Proc. 25th International Conference Computer Aided Verification (CAV)*. Springer, 2013, pp. 696–701.

[MTHK09] J. McLachlan, A. Tran, N. Hopper, and Y. Kim. “Scalable Onion Routing with Torsk”. In: *Proc. 16th ACM Conference on Computer and Communication Security (CCS)*. ACM Press, 2009, pp. 590–599.

[MVV04] S. Mauw, J. Verschuren, and E. de Vink. “A Formalization of Anonymity and Onion Routing”. In: *Proc. 9th European Symposium on Research in Computer Security (ESORICS)*. Springer, 2004, pp. 109–124.

[MW04] D. Micciancio and B. Warinschi. “Soundness of Formal Encryption in the Presence of Active Adversaries”. In: *Proc. 1st Theory of Cryptography Conference (TCC)*. Springer, 2004, pp. 133–151.

[Nie09] J. D. Nielsen. “Languages for Secure Multiparty Computation and Towards Strongly Typed Macros”. PhD thesis. Department of Computer Science, University of Aarhus, Denmark, 2009.

[NS78] R. Needham and M. Schroeder. “Using Encryption for Authentication in Large Networks of Computers”. In: *Communications of the ACM* 12.21 (1978), pp. 993–999.

[OP01] T. Okamoto and D. Pointcheval. “The Gap-Problems: A New Class of Problems for the Security of Cryptographic Schemes”. In: *Proc. 4th Conference on Theory and Practice of Public-Key Cryptography (PKC)*. Springer, 2001, pp. 104–118.

[PR08] M. Prabhakaran and M. Rosulek. “Homomorphic Encryption with CCA Security”. In: *Proc. 35th International Colloquium on Automata, Languages and Programming (ICALP)*. Springer, 2008, pp. 667–678.

[PRR09] A. Panchenko, S. Richter, and A. Rache. “NISAN: Network Information Service for Anonymization Networks”. In: *Proc. 16th ACM Conference on Computer and Communication Security (CCS)*. ACM Press, 2009, pp. 141–150.

[PW01] B. Pfitzmann and M. Waidner. “A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission”. In: IEEE Computer Society Press, 2001, pp. 184–200.

[Ram29] F. Ramsey. “On a Problem of Formal Logic”. In: *Proc. London Mathematical Society*. 2nd ser. 30 (1929), pp. 338–384.

[RG09] J. Reardon and I. Goldberg. “Improving Tor Using a TCP-over-DTLS Tunnel”. In: *Proc. 18th USENIX Security Symposium (USENIX)*. USENIX Association, 2009, pp. 119–133.

[RSG98] M. Reed, P. Syverson, and D. Goldschlag. “Anonymous Connections and Onion Routing”. In: *IEEE Journal on Selected Areas in Communications* 16.4 (1998), pp. 482–494.

[SBBPW06] C. Sprenger, M. Backes, D. Basin, B. Pfitzmann, and M. Waidner. “Cryptographically Sound Theorem Proving”. In: *Proc. 19th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society Press, 2006, pp. 153–166.

[Shm04] V. Shmatikov. “Probabilistic Analysis of an Anonymity System”. In: *Journal of Computer Security* 12.3-4 (2004), pp. 355–377.

[Sho99] V. Shoup. *On Formal Models for Secure Key Exchange*. Cryptology ePrint Archive, Report 1999/012. Available as Cryptology ePrint Archive, Report 1999/012 http://eprint.iacr.org/1999/012. 1999.

[SMCB12] B. Schmidt, S. Meier, C. J. F. Cremers, and D. A. Basin. “Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties”. In: *Proc. 25th IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society Press, 2012, pp. 78–94.

[Sol01] J. Solinas. *Low-Weight Binary Representations for Pairs of Integers*. Tech. rep. CORR 2001-41. http://cacr.uwaterloo.ca/techreports/2001/corr2001-41.ps. Accessed in June 2012. 2001.

[SSCB14] B. Schmidt, R. Sasse, C. J. F. Cremers, and D. A. Basin. “Automated Verification of Group Key Agreement Protocols”. In: *Proc. 35th IEEE Symposium on Security & Privacy (S&P)*. IEEE Computer Society Press, 2014, pp. 179–194.

[STRL00] P. Syverson, G. Tsudik, M. Reed, and C. Landwehr. "Towards an Analysis of Onion Routing Security". In: *Proc. Workshop on Design Issues in Anonymity and Unobservability (WDIAU)*. Springer, 2000, pp. 96–114.

[Tor03] Tor Project. *Official Web Page*. https://www.torproject.org/. Accessed in November 2011. 2003.

[Unr11] D. Unruh. "Termination-Insensitive Computational Indistinguishability (and Applications to Computational Soundness)". In: *Proc. 24th IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society Press, 2011, pp. 251–265.

[Unr12] D. Unruh. *Programmable Encryption and Key-Dependent Messages*. Tech. rep. 2012. IACR ePrint Archive: 2012/423. URL: http://eprint.iacr.org/2012/423.

[vS07] L. Øverlier and P. Syverson. "Improving Efficiency and Simplicity of Tor Circuit Establishment and Hidden Services". In: *Proc. 7th Privacy Enhancing Technologies Symposium (PETS)*. Springer, 2007, pp. 134–152.

[VSBW13] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. "A Hybrid Architecture for Interactive Verifiable Computation". In: *Proc. 34th IEEE Symposium on Security & Privacy (S&P)*. IEEE Computer Society Press, 2013, pp. 223–237.

[Wik04] D. Wikström. "A Universally Composable Mix-Net". In: *Proc. 1st Theory of Cryptography Conference (TCC)*. Springer, 2004, pp. 317–335.

[Yao82] A. C. Yao. "Protocols for Secure Computations". In: *Proc. 23rd IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society Press, 1982, pp. 160–164.

[Zha09] Y. Zhang. "Effective Attacks in the Tor Authentication Protocol". In: *Proc. 3rd International Conference on Network and System Security (NSS)*. IEEE Computer Society Press, 2009, pp. 81–86.