UNIVERSITÄT
DES
SAARLANDES

Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Trustworthy and Privacy-Preserving Processing of Personal Information

Cryptographic Constructions, Protocols, and Tools

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

von
Kim Rouven Pecina

Saarbrücken, Mai 2015

Tag des Kolloquiums:	10 März 2016
Dekan:	Prof. Dr. Markus Bläser

Prüfungsausschuss:

Vorsitzender:	Prof. Dr. Wolfgang J. Paul
Berichterstattende:	Prof. Dr. Matteo Maffei
	Prof. Dr. Michael Backes
	Dr. Steve Kremer
Akademischer Mitarbeiter:	Dr. Sven Bugiel

Zusammenfassung

Internetservices sammeln viele von Benutzern als sensibel eingestufte Daten, z.B. den Browserverlauf und Emailadressen, oft ohne, dass Benutzer dies merken. Die gesammelten Daten werden zum Personalisieren und zum Geld machen, bspw. durch zielgerichtete Werbung, genutzt.

Die funktionalitätserhaltende Umsetzung moderner Webservices, die die scheinbar unvereinbaren Eigenschaften Vertrauenswürdigkeit und Privacy erfüllen, gestaltet sich als schwierig: Wie können in sozialen Netzwerken nur Kommentare von Freunden zugelassen werden, wenn niemand seine Identität verrät? Wie ist personaliserte, zielgerichtete Werbung möglich, wenn Benutzer ihre Interessen geheim halten?

In dieser Dissertation stellen wir Techniken für die vertrauenswürdige und Privacy-erhaltende Verarbeitung von persönlichen Informationen vor.

Zuerst präsentieren wir eine API für die vertrauenswürdige und Privacy-erhaltende Verbreitung von persönlichen Daten. Die API erlaubt die deklarative Spezifizierung von verteilten Systemen; diese erfüllen anspruchsvolle Sicherheitseigenschaften wie Authorization, Anonymität und Accountability. Mit der API implementieren wir ein anonymes Evaluationssystem, anonyme Webs of Trust und ein sicheres soziales Netzwerk.

Weiterhin stellen wir eine Methodik für das vertrauenswürdige und Privacy-erhaltende Abrufen von Informationen vor. Beispielhaft dafür präsentieren wir ObliviAd, eine Architektur für hoch personalisierte Onlinewerbung, die beweisbar Benutzerprofile schützt.

Abstract

Internet services collect lots of information that users deem highly sensitive such as the browsing history and email addresses, often without users noticing this conduct. The collected information is used for personalizing services and it is monetized, e.g., in the form of targeted advertisements.

Realizing modern web services that maintain their functionality and satisfy the seemingly conflicting properties of trustworthiness and privacy is challenging: in a social network, how to enforce that only friends can post comments, if users are unwilling to reveal their identity? in online behavioral advertising, how to serve personalized ads, if users insist on keeping their interests private?

In this thesis, we propose techniques for the trustworthy and privacy-preserving processing of personal information.

First, we present an API for the trustworthy and privacy-preserving release of personal information. The API enables the declarative specification of distributed systems that satisfy sophisticated security properties, including authorization, anonymity, and accountability. We use this API to implement an anonymous evaluation system, anonymous webs of trust, and a secure social network.

Second, we present a methodology for the trustworthy and privacy-preserving retrieval of information. We exemplify our approach by presenting ObliviAd, an architecture for online behavioral advertising that provably protects user profiles and delivers highly-personalized advertisements.

Background of this Dissertation

This dissertation builds on the following papers. The author contributed to all of these papers as main author as well as to their elaboration.

Chapter 2 builds on the following works:

- Matteo Maffei and Kim Pecina [170].
  Position Paper: Privacy-Aware Proof-Carrying Authorization. In *Proc. ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS'11)*. ACM Digital Library, 2011.

- Michael Backes, Matteo Maffei, and Kim Pecina [30].
  Automated Synthesis of Privacy-Preserving Distributed Applications. In *Proc. Network and Distributed System Security Symposium (NDSS'12)*. Internet Society, 2012.

- Matteo Maffei, Kim Pecina, and Manuel Reinert [171].
  Security and Privacy by Declarative Design. In *Proc. IEEE Symposium on Computer Security Foundations (CSF'13)*, pages 81–96. IEEE Computer Society Press, 2013.

Chapter 3 builds on the following works:

- Michael Backes, Stefan Lorenz, Matteo Maffei, and Kim Pecina [25].
  Anonymous Webs of Trust. In *Proc. Privacy Enhancing Technologies Symposium (PETS'10)*, volume 6205 of *Lecture Notes in Computer Science*, pages 130–148. Springer-Verlag, 2010.

- Michael Backes, Stefan Lorenz, Matteo Maffei, and Kim Pecina [26].
  Brief Announcement: Anonymity and Trust in Distributed Systems. In *Proc. Symposium on Principles of Distributed Computing (PODC'10)*, pages 237–238. ACM Press, 2010.

- Michael Backes, Matteo Maffei, and Kim Pecina [28].
  A Security API for Distributed Social Networks. In *Proc. Network and Distributed System Security Symposium (NDSS'11)*, pages 35–51. Internet Society, 2011.

- Michael Backes, Matteo Maffei, and Kim Pecina [29].
  Brief Announcement: Securing Social Networks. In *Proc. Symposium on Principles of Distributed Computing (PODC'11)*, pages 341–342. ACM Press, 2011.

Chapter 4 builds on the following work:

- Michael Backes, Aniket Kate, Matteo Maffei, and Kim Pecina [23].
  ObliviAd: Provably Secure and Practical Online Behavioral Advertising. In *Proc. IEEE Symposium on Security & Privacy (S&P'12)*, pages 257–271. IEEE Computer Society Press, 2012.

Acknowledgments

I owe a great gratitude to Matteo Maffei. Working with him was an honor and an inspiration. During the last decade, he advised my Bachelor's thesis and this PhD thesis, thus accompanying me for the majority of my academic journey so far. Matteo also became a friend with a lot of understanding for my (sometimes challenging) personality.

Many thanks go to Michael Backes. His enthusiasm both as a teacher and as a researcher has drawn me towards cryptography and information security. He took me in as a Bachelor student and supervised my Bachelor's thesis. He showed me that you need to set high goals and pursue them, and he encouraged me to do the same.

I am very grateful to both Matteo and Michael for agreeing to review this thesis.

I thank all members of the CISPA and the MMCI administration. Without their administrative aid and without the countless, fruitful discussions and continuous support, I could not have mastered this thesis. I particular like to thank all my collaborators during my PhD studies (works with collaborations marked with a * are not contained in this thesis): Michael Backes, Fabian Bendun*, Aniket Kate, Stefan Lorenz, Matteo Maffei, Esfandiar Mohammadi, Pedro Moreno-Sanchez*, Christina Pöpper*, Raphael Reischuk*, and Manuel Reinert.

Thanks go to Stefan Lorenz and Andrea Ney for proof-reading this thesis. Special thanks go to Manuel Reinert for proof-reading this thesis including the appendix. Very special thanks go to my office mate Fabienne Eigner for enduring me during my PhD studies, her cheering me up, having an open ear, and for being a great friend.

For keeping the body fit and in sync with the mind, my gratitude goes to our tennis crew Sebastian Gerling, Milivoj Simeonovski, Manuel Reinert, and Hazem Torfah; I am still amazed that five computer scientists manage to play tennis at 7 in the morning. I also thank Fabian Bendun for demonstrating the RNC on me and, in the process, introducing me to a fantastic sport.

I thank Julian Backes and Stefan Lorenz for the elaborate freedom to finish this thesis during the time I took.

I am grateful to my parents, my family, and friends for their constant support, their patience and their understanding during the last years, especially when my time for them was rare. I particularly thank my girlfriend, fiancée, and soon-to-be wife Andrea Ney during the last years of this thesis: She was and is my motivation and compass when I lose my direction.

Typically, family and close friends constitute the final part of the acknowledgments. To me however, there are three more people that are cornerstones of my path to this thesis. Thanks to Robert Wirth for showing me that computers can also be used for something besides gaming. Back in the days, he gave me a copy of SuSE LinuX 6.1 and introduced me to C programming. Thanks to Walburga Reinert for helping me in dropping my sloppiness (Matteo knows what I mean). Thanks to Nima Zeini-Jahromi for making me join the Graduate School, thus taking the very first step towards a PhD thesis in the first place.

Contents

1 Introduction **1**
1.1 Contribution 2
1.1.1 Trustworthy and privacy-preserving release of personal information. 2
1.1.2 Trustworthy and privacy-preserving retrieval of personal information 4
1.2 Outline of the Thesis 5

Trustworthy and Privacy-Preserving Release of Personal Information **7**

2 Security and Privacy by Declarative Design **9**
2.1 Introduction 9
2.2 Key Ideas 10
2.3 Declarative API 13
2.3.1 Authorization 13
2.3.2 Privacy 15
2.3.3 Controlled Linkability 16
2.3.4 Accountability 17
2.3.5 Identity Escrow 18
2.3.6 Open-endedness and Interoperability 19
2.4 Cryptographic Realization 20
2.4.1 Cryptographic Setup 20
2.4.2 Cryptographic Realization of API Methods 24
2.5 Proofs 27
2.5.1 Cryptographic Proofs of Anonymity and Unlinkablity 27
2.5.2 Type-Based Verification of the API Methods 35
2.6 Implementation and Experiments 40
2.6.1 Experimental Evaluation 41
2.7 Related Work 44

3 Case Studies **51**
3.1 Experimental Setup 51
3.2 tales 52
3.2.1 Design of tales 52

3.2.2 Java Implementation of tales 53
3.2.3 Experimental Evaluation 54
3.3 Anonymous Webs of Trust 56
3.3.1 Designing Anonymous Webs of Trust 57
3.3.2 Implementation of Anonymous Webs of Trust 58
3.3.3 Experimental Evaluation 59
3.3.4 Formal Verification 61
3.3.5 Implementing Sophisticated Trust Measures 64
3.4 A Security API for Distributed Social Networks 65
3.4.1 A Core API for Social Networking 66
3.4.2 Implementation of the Core API 69
3.4.3 Experiments 72
3.4.4 Comparison: Dedicated Implementation vs. Declarative API Implementation. 77

Trustworthy and Privacy-Preserving Retrieval of Personal Information **79**

4 ObliviAd: Provably Secure and Practical Online Behavioral Advertising **81**
4.1 Introduction 81
4.2 Key Ideas 82
4.3 Protocol Overview 85
4.3.1 Adversary Model 85
4.3.2 Preliminaries 85
4.3.3 Cryptographic Assumptions and Requirements 87
4.3.4 Protocol Overview 87
4.4 ORAM Construction 91
4.4.1 ORAM Scheme by Shi et al. 92
4.4.2 Adapted Construction 93
4.5 Performance Analysis 94
4.5.1 Implementation 96
4.5.2 Experiments 96
4.5.3 Discussion 97
4.6 Formal Verification 98
4.6.1 Profile Privacy 98
4.6.2 Profile Unlinkability 99
4.6.3 Billing Correctness 101
4.7 Related Work 102

5 Conclusion and Outlook **105**

Appendices 107

A Well-Typedness of the API Methods 109
A.1 RCF Implementation of the API 109
A.1.1 Preliminaries: RCF Type System 110
A.1.2 API Data Types 114
A.1.3 Strong Types and Typed API Methods 121
A.2 Well-Typedness of the RCF Implementation 174
A.2.1 Type-Checking Auxiliary Functions 197
A.2.2 Type-Checking Main API Methods 202
A.3 Well-Typedness of the ML Implementation 265

B ProVerif Code 269
B.1 Anonymous Webs of Trust 269
B.2 ObliviAd 276

Bibliography 289

List of Figures 311

List of Tables 313

List of Listings 315

1. Introduction

We live in the digital age in which exchanging information between one end of the world and the other is only a click of a button away and takes merely a fraction of a second. Staying in touch with family and friends even across continents, and sharing personal details has never been easier. In fact, giving away such details has become so easy that people tend to overshare personal information, often oblivious to the fact of oversharing and to its consequences [67, 172, 86].

Big Internet companies, among them Google and Facebook, offer highly utilized services [103, 102] that are centered around collecting, assessing, and evaluating information shared by users [126, 108, 88]. For instance, if a user decides to join the largest online social network Facebook [103], she has to register with her name, age, and email address. Once a user is registered, she can enjoy the activities offered by modern social networks such as exchanging messages with her friends and liking new articles. While participating in these activities, her every action is recorded. The data aggregated that way is used to derive frighteningly detailed user profiles, typically used for advertising purposes. Among the most notable traits revealed by the resulting profiles are the political preferences, religious beliefs, and sexual orientation [7].

Awareness of these practices is rising and many Internet users realize the importance and the value of their personal information. In particular, users are beginning to acknowledge the implications on their privacy when data is lost, stolen, or in any other way shared: in a recent survey, a significant fraction of the participants classified personal data such as browsing history, name, demographic data (e.g., age and gender), and email address as highly sensitive [50, 151].

When faced with the decision whether to join a service or not, the strategy for privacy-aware users seems apparent: do not participate. Unfortunately, this strategy does not offer much protection: while casually browsing the web and without using any services, users are constantly tracked and their actions are recorded. For instance, advertisements that are included in many web sites are utilized to record the browsing behavior of users, even across different web sites. The brokers, i.e., the parties that provide these advertisements, use the gathered tracking data to display ads that are personally targeted towards the users' interests and hobbies.

A first intuition may be that since there is no registration of any kind, the tracking data is disconnected from the actual user identities and, consequently, privacy is only marginally affected. An extensive body of research shows that connecting anonymous profiles to users is feasible (e.g., [143, 188, 213, 51, 203, 97, 183, 224]) and, in fact, this tracking mechanism poses serious threats to the privacy of users [158, 159].

To respond to these privacy threats, to protect its citizens, and to reflect the growing desire of the Internet users to protect their privacy, legislation in Europe and the USA have become active: they started to define regulatory frameworks and guides for handling sensitive data and to promote privacy to be an an important cornerstone of IT systems and Internet services [190]. More precisely, the European Commission proposed a general data protection regulation [106] and in the USA the Federal Trade Commission issued recommendations for businesses on how to protect consumer privacy in a digital era [109, 90].

The ultimate goal of all legal regulations is to protect the users' rights and to require service providers to respect users and protect their data. These legal efforts are a crucial and important ingredient but they lack the technological counterparts that help to enforce legal requirements: for instance, a service provider can always choose to actively ignore legal restrictions and misuse data; technical solutions can help to identify malicious service providers or the solutions may prevent service providers from misusing user data in the first place.

There are many reasons and incentives for companies to offer their services in a privacy-preserving way: be it because legislation is expected to set hard requirements on service providers in the future, be it because companies respect their users' concerns and take them seriously, or be it in the light of the big data thefts of the recent years (e.g., [226, 162, 157, 222, 184, 13, 196]) along with the bad record for the affected companies (e.g. [35, 217, 113]).

Finding solutions that offer reliable security is challenging. Devising technical solutions that yield provable guarantees while satisfying the complex and seemingly conflicting requirements of modern web services such as privacy, anonymity, and authorization is even more daunting: how can users participate in Facebook without revealing private information? how can brokers serve targeted and personalized advertisements without collecting privacy-relevant user data? More generally, how to achieve trustworthy and privacy-preserving processing of personal information?

1.1. Contribution

In this thesis, we propose constructions, protocols, and tools that address the problems of the trustworthy and privacy-preserving processing of personal information. We split the general topic of information processing into two parts and treat the release and the retrieval of personal information separately.

1.1.1. Trustworthy and privacy-preserving release of personal information.

Releasing authentic, i.e., trustworthy, information is easy, for instance, by means of a digital signature. A signature, however, explicitly and intentionally reveals the creator of the data. In other words, it does not offer any form of privacy.

Releasing information in a privacy-preserving way is also easy: tools such as TOR [99] enable users to release information in an anonymous, i.e., a privacy-preserving, manner. Since the origin of the data remains anonymous, the data lacks any form of credibility.

Ideally, users would like to release their information such that they remain anonymous (or at least that the data does not point back to them directly) and, at the same time, the authenticity of the released data remains intact.

In the first part of this thesis, we propose a cryptographic solution that combines privacy, anonymity, and authorization in distributed systems. Intuitively, our solution enables users to selectively hide parts of a signed message such as their identity while still convincing the recipient that the messages is signed. More precisely, we combine digital signatures with powerful non-interactive zero-knowledge proofs of knowledge.[1] The resulting systems adopt the strong trust and authorization properties offered by digital signatures and inherit the strong privacy-guarantees achievable by zero-knowledge proofs. For instance, in the scenario of social networks, we can prove statements of the form "one of your friends posted this comment". Notice that this statement guarantees that the post originates from a friend while it keeps the identity of that friend secret.

Additionally to this anonymous authentication, we deploy service-specific pseudonyms. The compelling properties of service-specific pseudonyms are that they are pseudonyms: they hide the identity of its owner, and, at the same time, enable us to link user actions within a service (intra-service linkability) while user actions are unlinkable across different services (inter-service unlinkability). Service-specific pseudonyms are an ideal tool to restrict the amount of actions a user is allowed to take, for instance, to prevent users from submitting multiple anonymous reviews for a single product. Additionally, we develop an identity escrow protocol that, given a trusted third party, allows for revealing the identity of a user. In particular, users are aware if they use messages that can reveal their identity and only the trusted third party can associate a user to a certain action.

We provide a unified access to these cryptographic primitives. We devise a declarative API that harnesses this powerful combination of digital signatures, pseudonyms, identity escrow, and zero-knowledge proofs. The API is expressive enough to enforce sophisticated security properties such as authorization (trustworthiness), privacy, anonymity, controlled linkability, and accountability in complex distributed systems. It is designed to be easy to use and to be accessible even by cryptographic non-experts. In the API, information known to users is represented as logical formulas and the messages exchanged by parties as validity proofs for logical formulas [170]. The resulting systems enjoy the specified authorization properties by construction. Since the API allows users to selectively hide parts of a message, including the identity of the creator of that message, it is a salient tool for the trustworthy and privacy-preserving release of personal information.

We provide cryptographic proofs of the service-specific pseudonym properties, namely, anonymity, intra-service linkability, and inter-service unlinkability, and we show that the escrow mechanism only enables the trusted third party to identify a user. Furthermore,

[1] A zero-knowledge proof combines two seemingly contradictory properties. First, it is a proof of a statement that cannot be forged, that is, 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 besides the bare fact that the proven statement is valid [122]. A non-interactive zero-knowledge proof consists of a single message sent by the prover to the verifier. A proof of knowledge further shows that the prover was in possession of the information kept secret by the zero-knowledge proof.

we derive a security-by-construction guarantee, proving that the authorization policies in declarative specifications are enforced statically by leveraging a state-of-the-art security type system.

We implemented the declarative API in Java. With this implementation, we conducted case studies and we experimentally evaluated the resulting systems to demonstrate the expressiveness and the feasibility of our approach: using the declarative API, we design and implement a toy lecture evaluation system, and we re-implement anonymous webs of trust [25] and a security API for distributed social networks [28]. In previous works, the latter two systems required a complicated, dedicated cryptographic realization. Using the declarative API, we recreate these systems within a few lines of code. This code is easy to understand, even by cryptographic non-experts, and it is secure by construction. We discuss and compare our results to those obtained using the dedicated implementations.

1.1.2. Trustworthy and privacy-preserving retrieval of personal information

It is immediate that releasing information can be a privacy threat. In the same way, retrieving information has the potential to leak highly sensitive details: for instance, a big department store chain in the USA identified pregnant customers by analyzing which products customers looked at and which products they bought [136, 100, 156].

Obtaining information in a privacy-preserving manner is easy: private information retrieval methods [185, 87, 223] enable users to request information while concealing the retrieved data from the data provider. These methods by themselves, however, completely shut down targeted advertisements, since brokers have no information to base the personalization on. The same problem occurs if anonymous browsing solutions such as TOR are used.

In the second part of the thesis, we present ObliviAd, an online behavioral advertising system that enables users to retrieve highly personalized advertisements without revealing their profile. While ObliviAd is specifically designed to mitigate privacy issues that arise when dealing with targeted online advertising, we believe that the underlying technique is general and applicable to many more scenarios where users retrieve data from the Internet. In fact, we successfully applied the principle underlying ObliviAd to protect online money transactions [182].

The distinguishing features of ObliviAd are that brokers (the companies that distribute the advertisements to various web sites) can use their existing infrastructure (e.g., hardware and software) with only minor modifications, widely-used processes such as bid auctioning (only the ad of the advertiser willing to pay the most is shown) can be conducted without major modifications, important properties including click-through rates (a way of measuring the success of an advertisement) can be determined, and, most importantly, ObliviAd provably preserves the users' privacy.

Under the hood, we combine state-of-the-art private information retrieval techniques based on the powerful combination of trusted hardware and oblivious RAMs (ORAM) [223], client-side user profile creation [114], and a powerful billing mechanism. We extend the

ORAM scheme to accommodate requirements of online behavioral advertising. We achieve provable security guarantees without putting any trust assumptions on the broker or any other third party. The hardware-based private information retrieval technique and our billing mechanism cause virtually no computational overhead for users, making it an ideal solution even for mobile devices.

We show that in ObliviAd, no information about the user profiles are leaked (profile privacy) and that retrieved advertisements cannot be associated to a user profile, even if a-priori information about user profiles are available (profile unlinkability); for the billing correctness, we show that an advertiser is charged only if her ad is displayed (or clicked, depending on the payment model). More formally, we model profile privacy and profile unlinkability as an observational equivalence relation and we model the correctness of our billing mechanism as a trace property. Using ProVerif [52, 1], a state-of-the-art automated theorem prover, we show that the aforementioned security and privacy properties hold.

We implemented a prototypical Java-based program that mimics the operations of the trusted hardware to demonstrate the feasibility of our approach.

1.2. Outline of the Thesis

This thesis is separated into two parts. The first part discusses our approach to the trustworthy and privacy-preserving release of personal information and comprises Chapter 2 and Chapter 3. Chapter 2 introduces the declarative API and its key ideas, the cryptographic realization, the cryptographic proofs, and the experimental evaluation. Chapter 3 contains the case studies along with the corresponding code and the experimental evaluation.

The second part of the thesis is dedicated to our approach to the trustworthy and privacy-preserving retrieval of personal information. This part comprises Chapter 4 that describes ObliviAd, a system for practical and provably secure online behavioral advertising.

Finally, Chapter 5 concludes.

Part I
Trustworthy and Privacy-Preserving Release of Personal Information

2. Security and Privacy by Declarative Design

The results presented in this chapter build on the following works:

- Maffei and Pecina [170]: "Position Paper: Privacy-aware Proof-Carrying Authorization"

- Backes, Maffei, and Pecina [30]: "Automated Synthesis of Privacy-Preserving Distributed Applications"

- Maffei, Pecina, and Reinert [171]: "Security and Privacy by Declarative Design"

2.1. Introduction

In this chapter, we introduce a novel framework for specifying systems in a declarative language. The core of the framework comprises a declarative API for data processing and its cryptographic implementation. The API allows the programmer to conveniently specify the overall system architecture and a variety of security requirements such as *authorization*, *privacy*, *controlled linkability*, and *accountability*, while concealing from the programmer the cryptographic details. Developing such a framework is challenging for three fundamental reasons.

Security versus privacy. A generally applicable design methodology for privacy-preserving distributed systems is particularly challenging, since it requires the development of sophisticated and carefully designed cryptographic protocols to reconcile the privacy of users with other seemingly contradictory security requirements, such as authorization policies and accountability, or system functionalities, such as linkability of user actions (for instance, to implement pay-per-usage or access-only-once policies). How to make sure that the principal trying to access a sensitive resource is authorized if this principal is not willing to share any personally identifying information? How to link user actions without jeopardizing the privacy of users? How to hold misbehaving users accountable for their actions without compromising the privacy of honest users?

General applicability and efficiency. The cryptographic realization should guarantee all the aforementioned security requirements. At the same time, it should not put restrictions or assumptions on the structure of the system, e.g., the presence of a trusted third party (TTP), and it should not hamper the system performance, for instance, by requiring additional bootstrapping phases or interactions among parties. Furthermore, the cryptographic framework should allow for open-endedness, that is, the extension of the system with new components, and interoperability, i.e., the sharing of data among them.

Sound and convenient development workflow. Finally, developing a generally applicable and efficient cryptographic infrastructure is not enough. Implementing distributed programs based on advanced cryptographic schemes is highly error-prone, as witnessed by the number of attacks on largely deployed cryptographic protocol implementations (e.g., [191, 55]), and typically requires a strong cryptographic expertise, which may easily go beyond the background of the average programmer. We believe that it is of paramount importance to provide the system developer with programming abstractions that are conveniently integrated in the usual workflow and allow her to concentrate on the system structure and on the desired security properties, ignoring the details of the cryptographic realization.

Outline. The rest of this chapter is organized as follows: Section 2.2 introduces the key ideas underlying the declarative API. Section 2.3 introduces the API methods and their semantics. In Section 2.4, we detail our cryptographic realization of the API methods. Section 2.5 presents the cryptographic proofs for the introduced cryptographic primitives and outlines the authorization property proofs of the API methods. We present our implementation and obtained micro benchmarks in Section 2.6. We discuss related work in Section 2.7.

2.2. Key Ideas

This section provides an overview of the fundamental concepts underlying the declarative API. We start by identifying a suitable digital signature scheme that is powerful enough to express the desired authorization properties. We then proceed with the zero-knowledge scheme that is compatible with the digital signature scheme and supports the anticipated privacy properties. Finally, we shed light on our high-level representation that concentrates on messages and security and privacy properties while concealing the underlying cryptographic details.

Enforcing authenticity: digital signatures. We use digital signatures for enforcing the desired authorization property. Digital signatures are a well-established tool to enforce authorization policies in distributed systems (e.g., [39, 118, 11, 40]). Let m be a message, vk be a verification key, and sig be a digital signature. We write

$$\mathsf{ver}(sig, m, vk)$$

to denote the successful signature verification of sig on m with vk. Since the digital signature cannot be forged, the successful verification shows that the owner of vk has signed message m. We call the pair (m, sig) of message and signature a digital certificate on message m issued by the owner of vk.

We deploy the automorphic[1] signature scheme by Abe et al. [6] to create the certificates. This signature scheme can sign arbitrary tuples of messages but the distinguishing feature is the support of signing verification keys without resorting to any kind of encoding. As we can sign verification keys without encoding, this signature scheme enables a very elegant and powerful symbiosis with zero-knowledge proofs to enforce privacy. In particular, this feature paves the way for efficient zero-knowledge proofs of statements of the form

$$\mathsf{ver}(\mathit{sig}, (m, \mathit{vk}'), \mathit{vk}) \;\wedge\; \mathsf{ver}(\mathit{sig}', m', \mathit{vk}'),$$

i.e., the successful signature verification on a message consisting of the two parts m and vk', the second part vk' is also used as a verification key in a successful signature verification on message m'.

Anticipating the final design, we use verification keys to encode principal identifiers. For instance, one can imagine the above example to model delegation by letting m authorize the owner of vk' to act on behalf of the owner of vk. Consequently, the message m' is treated as if it originated from the owner of vk.

Enforcing privacy and general applicability: zero-knowledge proofs. Zero-knowledge proofs are an established tool to enforce privacy in distributed systems (e.g., [82, 83, 28, 25, 167, 73]). In our case, however, simply applying any zero-knowledge scheme is not enough. Indeed, any zero-knowledge scheme compatible with the chosen digital signatures can be used to enforce privacy properties. We, however, strive for privacy in combination with a general applicability, especially open-endedness and interoperability properties, as well as efficiency. In particular, the requirement for interoperability, i.e., sharing cryptographic material between protocols calls for a malleable zero-knowledge solution.

In typical scenarios, malleability is considered a bug rather than a feature. Intuitively, suppose the following scenario: let p_1 and p_2 be proofs for the statements “I, Alice, own bank account BA” and “Pay \$200 from BA to Eve”, respectively. Even if Eve manages to obtain the two individual proofs, we want to prevent the combination of these two proofs into a single proof of the form $p_1 \wedge p_2$ that authorizes a payment. In our case, however, we need exactly that kind of malleability to enable the desired interoperability property, i.e., to combine proofs from different protocols into one new proof.[2]

The Groth-Sahai zero-knowledge proof scheme [128] perfectly satisfies these requirements. It is expressive enough to prove the necessary statements such as the validity of digital signatures and we can selectively hide parts of the proven statement. For instance,

[1] Automorphic denotes that verification keys are a subset of the message space, i.e., verification keys can be signed as part of a message without any kind of encoding.

[2] We do not run into this problem because we deploy zero-knowledge proofs that are associated to a principal. More precisely, in our scenario, the proven statements would read “Alice says I, Alice, own back account BA” and “Alice says Pay \$200 from BA to Eve”, i.e., the payment was originally authorized by Alice (in particular, the authorization is *not* derived from the combination of the two proofs).

we can show a signature verification of the form

$$\mathsf{ver}(sig, (m, vk'), vk) \ \wedge \ \mathsf{ver}(sig', m', vk'),$$

in zero-knowledge and it is straightforward to selectively hide parts of the statement such as the signatures *sig* and *sig*′. Furthermore, Groth-Sahai proofs are endowed with just the right amount of malleability [42] that enables us to re-randomize a proof: Given a proof p, a user can re-randomize p to yield p'. Then p' shows the same statement but it is unlinkable to p. In particular, re-randomization is also possible, if the user did not compute p but received it during a protocol, a crucial feature for anonymity.

Enabling a convenient development workflow: logic-based programming abstraction. We have identified the necessary cryptographic schemes to enforce the desired security and privacy properties. Building complex distributed systems without any abstraction layer, however, is a daunting task that requires a great amount of expertise that may easily overwhelm programmers and system designers that often lack the cryptographic background. Consequently, the right abstraction is of paramount importance to enable a convenient development workflow.

We choose to describe zero-knowledge statements by means of a logical specification language. For instance, we denote the successful signature verification

$$\mathsf{ver}(sig, (\mathsf{MayAccess}, vk', lab), vk)$$

with the logical formula

$$vk \ \mathsf{says} \ \mathsf{MayAccess}(vk', lab).$$

Here, this signature allows the owner of vk' to enter the lab. This description language is easy to understand: the use of the "says" modality binds logical formulas (here, MayAccess) to principals (here, *vk*) and naturally captures the semantics of digital signatures [11, 41, 118]. Such a logical representation also elegantly captures the hiding of selected parts of a zero-knowledge statement by existential quantification [170]. For instance, the formula that uses a zero-knowledge proof to hide vk' in order to enable the owner of vk' to access the lab anonymously looks as follows:

$$\exists x.\ vk \ \mathsf{says} \ \mathsf{MayAccess}(x, lab).$$

Additionally, this abstraction hides the cryptographic implementation from programmers. For instance, the above says-statement mentions the principal identifier *vk* and the predicate $\mathsf{MayAccess}(vk', lab)$. It hides the fact that the statement is implemented as a digital signature, in particular, *sig*, the signature itself, is not visible.

Controlled linkability of user actions: service-specific pseudonyms. The combination of digital signatures and zero-knowledge proofs enforces authorization and privacy properties. It, however, lacks the possibility to link user actions in a controlled way. We use service-specific pseudonyms (SSPs) to link user actions. Intuitively, an SSP hides the identity of the user and it remains constant throughout a service. For instance, suppose SSP_F

and SSP_R are user U's SSPs for services "Facebook" and "Rating Platform", respectively. The corresponding logical representations look as follows:

$$\mathsf{SSP}(vk_U, \text{“Facebook”}, \mathrm{SSP}_F)$$
$$\mathsf{SSP}(vk_U, \text{“Rating Platform”}, \mathrm{SSP}_R).$$

Since pseudonyms are constant, proofs that reveal the SSP for a given service are linkable within that service. The pseudonyms, however, are unlinkable across different services, i.e., it is not possible to decide whether SSP_F and SSP_R belong to the same user or two different users.

2.3. Declarative API

This section introduces the security-oriented, declarative API for the design of distributed systems. We instantiate the API in ML because of the impressive line of research on analysis techniques for ML-like languages (e.g., [49, 104, 27, 19, 62, 38, 160]). We remark, however, that the API is in principle language-independent and can easily be implemented in any other programming language. In fact, we present a Java implementation in Section 2.6.

Inspired by prior work on information logics for distributed systems [9, 30], the programming abstraction we propose represents the information known to principals as logical formulas and the messages exchanged by parties as validity proofs for logical formulas. The framework is independent of the choice of the logic: we just assume the presence of the "says" modality that binds logical formulas to principals.

Table 2.1 illustrates the methods composing our API, along with the respective functional types. We describe these methods below, classifying them according to the security property they capture.

2.3.1. Authorization

Example 2.1. As a running example, we design a collaborative platform that combines two services. In the first service, a patient receives a certificate from the doctor attesting her visit and including additional information such as the date of the visit and the results of the examination. In the second service, the patient uses this information to evaluate her doctor on a rating platform such as Jameda [142] or Healthgrades [133]. We assume the following authorization policy for the hypothetical rating platform RateYourDoc that allows a patient to evaluate only her treating doctors:

$$\begin{aligned}&\forall Pat, Doc, results, date, opinion.\\&\quad Doc\ \mathsf{says}\ \mathsf{Visit}(Pat, date, results)\\&\quad \wedge\ Pat\ \mathsf{says}\ \mathsf{Rating}(opinion)\\&\qquad \Longrightarrow\ \mathsf{Rated}(Doc, opinion).\end{aligned} \tag{2.1}$$

This authorization policy states that if a patient *Pat* visited the doctor *Doc* on a date *date* with the results *results*, and *Pat* rates *Doc* with *opinion*, then this rating is accepted.

$\mathsf{mkId} : \mathit{string} \to \mathit{uid} * \mathit{uid}_{pub}$	create a fresh pair of identifiers
$\mathsf{mkSays} : x : \mathit{uid} \to f : \mathit{formula} \to \mathit{proof}$	make proof of y says f, $y : \mathit{uid}_{pub}$ corresponds to x
$\mathsf{mk}_{\wedge} : \mathit{proof} * \mathit{proof} \to \mathit{proof}$	make conjunctive proof
$\mathsf{split}_{\wedge} : \mathit{proof} \to \mathit{proof} * \mathit{proof}$	split conjunctive proof
$\mathsf{mk}_{\vee} : \mathit{proof} \to \mathit{formula} \to \mathit{proof}$	make disjunctive proof
$\mathsf{extractForm} : p : \mathit{proof} \to \mathit{formula}$	return description of statement for p
$\mathsf{verify} : p : \mathit{proof} \to f : \mathit{formula} \to \mathit{bool}$	verify that p is a proof of f
$\mathsf{hide} : \mathit{proof} \to \mathit{formula} \to \mathit{proof}$	hide witnesses from a proof (as specified by *formula*)
$\mathsf{rerand} : \mathit{proof} \to \mathit{formula} \to \mathit{proof}$	re-randomize proof (as specified by *formula*)
$\mathsf{mkSSP} : x : \mathit{uid} \to s : \mathit{string} \to \mathit{proof}$	make proof of $\mathsf{SSP}(y, s, \mathit{psd})$, $y : \mathit{uid}_{pub}$ corresponds to x
$\mathsf{mkREL} : f : \mathit{formula} \to \mathit{proof}$	make proof of relation f
$\mathsf{mkEQN} : f : \mathit{formula} \to \mathit{proof}$	make proof of equation f
$\mathsf{mkLM} : x : \mathit{pseudo} \to b : \mathit{string} \to$ $\ell : \mathit{list} \to \mathit{proof}$	make proof of $(x, b) \in \ell$
$\mathsf{mkLNM} : x : \mathit{pseudo} \to \ell : \mathit{list} \to \mathit{proof}$	make proof of $(x, \_) \notin \ell$
$\mathsf{mkIDRev} : \mathit{proof} \to s : \mathit{string} \to \mathit{proof}$	make identity escrow proof for service s

Table 2.1.: High-level API interface functions.

Formulas are encoded as terms of the language (in ML, using data-type constructors): for the sake of readability, here and throughout the remainder of the thesis, we use the standard logical notation. In the first service, the doctor provides a validity proof of his medical license along with personal information $\mathit{PI}_{\mathit{Doc}}$ for the patient *Alice*, vouched for by the hospital *Hosp*, and an attestation of *Alice*'s visit. More precisely, the doctor issues a proof for the formula

$$\mathit{Hosp}\ \mathsf{says}\ \mathsf{IsDoc}(\mathit{Doc}, \mathit{PI}_{\mathit{Doc}}) \wedge \mathit{Doc}\ \mathsf{says}\ \mathsf{Visit}(\mathit{Alice}, \mathit{date}, \mathit{results}).$$

To express her opinion *happy* about the doctor, *Alice* submits a validity proof for the formula $\mathit{Doc}\ \mathsf{says}\ \mathsf{Visit}(\mathit{Alice}, \mathit{date}, \mathit{results}) \wedge \mathit{Pat}\ \mathsf{says}\ \mathsf{Rating}(\mathit{happy})$. This formula satisfies the authorization policy and the reviewing platform can deduce $\mathsf{Rated}(\mathit{Doc}, \mathit{opinion})$. □

Each user u has two identifiers: a private one of type *uid* that is used to refer to the principal executing a certain piece of code, and a public one of type uid_{pub} that is used to refer to other principals. The function mkId takes as input a string such as the name, and returns a pair of private and public identifiers.

The function mkSays x f takes the private identifier x of the user running the code, a formula f, and returns a validity proof for the predicate y says f, where y is the public identifier corresponding to x.

The API provides methods to manipulate proofs, which is crucial for the expressiveness of our framework. The function $\mathsf{mk}_\wedge$ takes as input a proof of f_1 and a proof of f_2, and returns a proof of the conjunction $f_1 \wedge f_2$. This function, as well as the other API functions, raises an exception if the input is not of the expected form (in this case, a pair of validity proofs). Conversely, the function $\mathsf{split}_\wedge$ takes a proof of $f_1 \wedge f_2$, and returns a proof of f_1 and a proof of f_2. The function $\mathsf{mk}_\vee$ takes as input a proof of f_1 and a formula of the form $f_1 \vee f_2$ or $f_2 \vee f_1$, and returns a proof of the specified disjunctive statement. Due to our cryptographic implementation, the construction of a disjunction is only possible if the hide function has not been applied to the input proof yet, i.e., no argument is existentially quantified (see Section 2.4). The function $\mathsf{extractForm}$ takes as input a proof and returns the corresponding formula. Finally, the function verify takes as input a proof and a formula, and checks that the former is a proof of the latter.

Example 2.2. The code for the patient is shown below:

```
let $Pat$ $x_{Pat}$ $y_{Pat}$ $y_{Hosp}$ $y_{Doc}$ $x_{PI_{Doc}}$ $x_{results}$ $x_{date}$ $x_{opinion}$ $x_{addr_{Pat}}$ $x_{addr_{\mathtt{RYD}}}$ =
  let $c$ = listen $x_{addr_{Pat}}$;
  let $y$ = recv $c$;
  if $\mathsf{verify}$ $y$ $\left(\begin{array}{l} y_{Hosp}\ \mathsf{says}\ \mathsf{IsDoc}(y_{Doc}, x_{PI_{Doc}}) \\ \wedge\ y_{Doc}\ \mathsf{says}\ \mathsf{Visit}(y_{Pat}, x_{date}, x_{results}) \end{array}\right)$ then
    let $(pf_{\mathsf{IsDoc}}, pf_{\mathsf{Visit}})$ = $\mathsf{split}_\wedge$ $y$;
    let $pf_s$ = $\mathsf{mkSays}$ $x_{Pat}$ $\mathsf{Rating}(x_{opinion})$;
    let $pf$ = $\mathsf{mk}_\wedge$ $(pf_{\mathsf{Visit}}, pf_s)$;
    connect $c'$ $x_{addr_{\mathtt{RYD}}}$;
    send $pf$ $c'$
```

Listing 2.1: Code for the patient.

The code is self-explanatory: the patient receives a proof for the attestation of her visit from the doctor and constructs the rating proof. She combines the proof of her visit (obtained by splitting the proof received by the doctor apart) and the rating proof in conjunctive form. Finally, she sends the resulting proof to the rating platform. The communication functions such as listen are the standard communication primitives available in any language. □

2.3.2. Privacy

The function hide allows for hiding sensitive arguments, which can be logically captured by existential quantification. This function takes as input a proof p of f and a formula f' obtained from f by existentially quantifying some of the arguments, and it returns a proof of f'.

Example 2.3. The doctor certainly does not want the patient to know her personal information PI_{Doc} included in the certificate. Hence, she sends a proof in which this particular information is hidden. This changes the call to verify:

```
    ...
4   if verify $y$ $\left(\begin{array}{l} \exists w_{PI_{Doc}}. \\ \quad y_{Hosp} \text{ says } \mathsf{IsDoc}(y_{Doc}, w_{PI_{Doc}}) \\ \quad \wedge\ y_{Doc} \text{ says } \mathsf{Visit}(y_{Pat}, x_{date}, x_{results}) \end{array}\right)$ then
    ...
```

Additionally, the patient might desire to submit her evaluation anonymously. She can do so by existentially quantifying her identity, the results, and the date, which is achieved by the following piece of code:

```
    ...
7   let $pf$ = $\mathsf{mk}_\wedge$ $(pf_{\mathsf{Visit}}, pf_s)$;
7e  let $pf'$ = hide $pf$ $\left(\begin{array}{l} \exists w_{Pat}, w_{results}, w_{date}. \\ \quad y_{Doc} \text{ says } \mathsf{Visit}(w_{Pat}, w_{date}, w_{results}) \\ \quad \wedge\ w_{Pat} \text{ says } \mathsf{Rating}(x_{opinion}) \end{array}\right)$;
8   connect $c'$ $x_{addr_{\mathtt{RYD}}}$;
9   send $pf'$ $c'$
```

where *pf* is the proof produced in line 7 of the code shown in Example 2.2. This proof suffices to convince the rating platform of the patient's evaluation for the doctor *Doc* and, from a logical perspective, to entail the predicate Rating(*Doc*, *opinion*). □

2.3.3. Controlled Linkability

The previous example suggests that hiding the identity of users may hinder the enforcement of meaningful authorization policies. For instance, in order to avoid biased results, we would like to make sure that patients cannot submit more than one evaluation. In general, there may be the need for the service provider to link the actions of the users, which should be achieved without making user actions linkable across different services. We rely on service-specific pseudonyms to achieve this goal: each user can create at most one valid SSP per service, which provides intra-service linkability, while her pseudonyms cannot be linked and tracked across different services, which provides inter-service unlinkability. Since SSPs hide the identity of their owner, they can be revealed; a simple comparison suffices to determine whether the user behind a given pseudonym is using a service for the first time or not. Notice that the service structure determines the degree of unlinkability offered to each user: increasing the number of services (e.g., by splitting a service) limits the tracking of user actions and provides stronger unlinkability guarantees.

The function mkSSP takes as input the private user identifier x and the service identifier s, and it returns a proof of the predicate $\mathsf{SSP}(y, s, psd)$, which states that psd is the pseudonym for the public identifier y corresponding to x and the service s.

Example 2.4. We set the service structure so as to reflect doctor specializations. Assume that the doctor who visited the patient is an internist offering the service $x_{Internist}$. Then the patient can extend the proof pf produced in Example 2.2 to accommodate both privacy and linkability requirements as follows:

```
...
let $pf_s$ = mkSays $x_{Pat}$ Rating($x_{opinion}$);
let $pf_{ssp}$ = mkSSP $x_{Pat}$ $x_{Internist}$;
let $s$ = extractForm $pf_{ssp}$;
match $s$ with SSP($y_{Pat}, x_{Internist}, x_{psd}$) $\rightarrow$
  let $pf_\wedge$ = mk$_\wedge$ ($pf_s, pf_{ssp}$);
  let $pf'$ = hide $pf$ $\left(\begin{array}{l} \exists w_{Pat}, w_{results}, w_{date}. \\ \quad x_{Doc} \text{ says Visit}(w_{Pat}, w_{date}, w_{results}) \\ \quad \wedge\ w_{Pat} \text{ says Rating}(x_{opinion}) \\ \quad \wedge\ \text{SSP}(w_{Pat}, x_{Internist}, x_{psd}) \end{array}\right)$
...
```

Notice that the existential quantification binds all occurrences of the patient identifier, including the one in the SSP predicate. The rating platform can discard multiple evaluations by simply checking the pseudonyms conveyed by each proof. □

2.3.4. Accountability

SSPs are designed to prevent the tracking of users across different services. In many applications, however, it is desirable to ban misbehaving users from the whole system or to reward well-behaving ones. We use reputation lists to achieve this kind of accountability requirements without disclosing user identities.

A reputation list binds SSPs to attributes. For the sake of simplicity, we assume that each reputation list refers to a specific service s and contains pairs of the form $(psd, attr)$, where psd is a pseudonym for service s and $attr$ is an attribute. We could easily support lists referring to several services and binding pseudonyms to several attributes, but this would solely complicate the presentation without adding any interesting insight.

The function mkLM takes as input a pseudonym psd, an attribute $attr$, and a reputation list ℓ, and it returns a proof for the formula $(psd, attr) \in \ell$. The function mkLNM takes as input a pseudonym psd and a reputation list ℓ, and it returns a proof for the formula $(psd, \_) \notin \ell$, where _ serves as wildcard. Technically, wildcards are universally quantified: $\forall x.\ (psd, x) \notin \ell$.

The function mkREL takes as input a formula describing a binary arithmetic relation between attributes and returns the corresponding proof. We support arithmetic relations of the form $b\ op\ b'$, with $op \in \{>, \geq, <, \leq, =, \neq\}$.

Similarly, the function mkEQN takes as input a formula describing the computation of a mathematical operation. We support mathematical equations of the form $b_1 = b_2 \; op \; b_3$, with $op \in \{+, -, \cdot\}$.

Example 2.5. We maintain a reputation list for each service (here, doctor specialization). This list contains the pseudonyms of the patients that previously uploaded offensive comments in the associated service. In order to prevent such patients from further participating in evaluation procedures, we require patients to prove that their pseudonyms have not been included in any of such lists. We show below how to extend the proof from Example 2.4. For simplicity, we focus on just one reputation list x_ℓ for the service $x_{Dentist}$. The extension to multiple lists is straightforward.

```
...
let $pf'_{ssp}$ = mkSSP $x_{Pat}$ $x_{Dentist}$;
let $s'$ = extractForm $pf'_{ssp}$
match $s'$ with SSP$(x_{Pat}, x_{Dentist}, x'_{psd})$ $\rightarrow$
   let $pf_{\notin}$ = mkLNM $x'_{psd}$ $x_\ell$;
   let $pf'_{\wedge}$ = mk$_\wedge$(mk$_\wedge$ $(pf_\wedge, pf'_{ssp}), pf_{\notin}$);
   let $pf'$ = hide $pf'_\wedge$ $\left( \begin{array}{l} \exists w_{Pat}, w_{date}, w_{results}, w_{psd'} . \\ \quad x_{Doc} \text{ says } \mathsf{Visit}(w_{Pat}, w_{date}, w_{results}) \\ \quad \wedge \; w_{Pat} \text{ says } \mathsf{Rating}(x_{opinion}) \\ \quad \wedge \; \mathsf{SSP}(w_{Pat}, x_{Internist}, x_{psd}) \\ \quad \wedge \; \mathsf{SSP}(w_{Pat}, x_{Dentist}, w_{psd'}) \\ \quad \wedge \; (w_{psd'}, \_) \notin x_\ell \end{array} \right)$
...
```

The patient pseudonym for $x_{Dentist}$ is existentially quantified, which makes patient evaluations unlinkable across different doctor specializations. □

Finally, we remark that a user can in principle obtain multiple pseudonyms for a service if she registers several user identifiers with the corresponding provider. Notice, however, that the registration phase is not anonymous (see Example 2.1) and the service provider has to willingly register users multiple times.

2.3.5. Identity Escrow

In some scenarios, it is desirable to have a mechanism to reveal the identity of misbehaving users, for instance, if the user severely violated certain regulations or if she even committed a crime. We can achieve that in our framework by means of an identity escrow mechanism.

The user initially contacts the trusted third party EA acting as an escrow agent, which provides the user with a proof of the predicate EA says $\mathsf{EscrowId}(y, r)$, where y is the public identifier of the user and r is a number chosen by EA to identify the user.

The user creates an escrow proof by means of the mkIDRev function. This function takes as input the proof received from EA and the service, and it returns a proof of the predicate $\mathsf{EscrowInfo}(EA, y, r, s, idr)$, where idr is the user's escrow identifier for the service s. Given idr and s, the trusted party EA and only EA can extract the identity of the user. Thus, the user has simply to send a proof of $\exists w, x_r.\ \mathsf{EscrowInfo}(EA, w, x_r, s, idr)$ to the service provider, which hides the user's identity and the value r. Since, similarly to pseudonyms, the escrow identifiers of a user are unlinkable across different services, the identity escrow protocol preserves the inter-service unlinkability of user actions.

We stress that requiring a user action to enable the identity escrow service is an intentional feature of the API: the user has to give her explicit consent to engage in a service in which her anonymity might in principle be compromised.

Example 2.6. We show below how to extend the proof from Example 2.5, assuming that pf_{EA} is the proof that the patient previously received from the rating platform acting as an escrow agent.

```
...
    let $pf'_\wedge$ = $\mathsf{mk}_\wedge(\mathsf{mk}_\wedge\ (pf_\wedge, pf'_{ssp}), pf_{\notin})$;
    let $pf_{escrow}$ = mkIDRev $pf_{EA}$ $x_{Internist}$;
    let $s''$ = extractForm $pf_{escrow}$
    match $s''$ with $w_{EA}$ says $\mathsf{EscrowId}(x_{Pat}, x_r)$ $\rightarrow$
      let $pf''_\wedge$ = $\mathsf{mk}_\wedge\ (pf'_\wedge, pf_{escrow})$;
      let $pf'$ = hide $pf''_\wedge$ $\left(\begin{array}{l} \exists w_{Pat}, w_{date}, w_{results}, w_{psd'}, w_r. \\ \quad x_{Doc}\ \mathsf{says}\ \mathsf{Visit}(w_{Pat}, w_{date}, w_{results}) \\ \quad \wedge\ w_{Pat}\ \mathsf{says}\ \mathsf{Rating}(x_{opinion}) \\ \quad \wedge\ \mathsf{SSP}(w_{Pat}, x_{Internist}, x_{psd}) \\ \quad \wedge\ \mathsf{SSP}(w_{Pat}, x_{Dentist}, w_{psd'}) \\ \quad \wedge\ (w_{psd'}, \_) \notin x_\ell \\ \quad \wedge\ \mathsf{EscrowInfo}(w_{EA}, w_{Pat}, w_r, x_{Internist}, x_{idr}) \end{array}\right)$
...
```

□

2.3.6. Open-endedness and Interoperability

We have already demonstrated that the API is well-suited for open-ended applications by extending the client code to accommodate the new functionality added to the protocol. We finally remark that the API is also well-suited for the development of interoperable systems, that is, systems that can be extended with services sharing resources and interoperating with each other.

Example 2.7. We introduce an online pharmacy service (e.g., Medco [178]) that delivers medicines on request. To this end, the patient appends the order to the doctor's attestation of her visit, hiding the doctor's identity. Formally, she combines the validity proof issued by the doctor at her visit with the request to buy medication from the online pharmacy. She sends the resulting validity proof of the following formula to the pharmacy:

$$\begin{array}{l}\exists w_{Doc}, w_{PI_{Doc}}.\\ \quad x_{Hosp} \textsf{ says } \textsf{IsDoc}(w_{Doc}, w_{PI_{Doc}})\\ \quad \wedge\ w_{Doc} \textsf{ says } \textsf{Visit}(x_{Pat}, x_{date}, x_{results})\\ \quad \wedge\ x_{Pat} \textsf{ says } \textsf{Buy}(\mathit{medicine}).\end{array}$$

2.4. Cryptographic Realization

This section details the cryptographic realization of the API methods. We start by setting-out the cryptographic building blocks in Section 2.4.1 and we describe the concrete cryptographic implementation of the zero-knowledge proofs deployed in the API methods in Section 2.4.2.

2.4.1. Cryptographic Setup

We cryptographically implement private user identifiers as handles to the corresponding signing keys. The keys themselves are not accessible by the interface and, thus, are invisible to the programmer. In particular, programmers cannot accidentally leak signing keys. The storage medium for the signing key is chosen depending on the security requirements: signing keys can be stored in files protected by the operating system or, to achieve better security guarantees, in cryptographic devices capable of computing digital signatures (e.g., cryptographic coprocessors [139, 23]).

Public user identifiers are realized as verification keys and we rely on a public-key infrastructure (PKI) to bind users to their key; the running example in Section 2.3 implements a decentralized PKI that resembles webs of trust, where the hospital vouches for the doctor, who in turn vouches for the user.

In the following, we detail the cryptographic constructions used in the implementation of our API.

Security parameter. In cryptography, the security of a system is virtually always relative to a security parameter η.[3] Depending on the cryptographic primitives, the security parameter influences the group size in the case of asymmetric schemes, the key size in the case of symmetric cryptographic schemes, or the length of the output in the case of hash functions. Current recommendations stipulate that a security parameter of at least 112 bits is used [208]. Throughout the rest of the thesis, we use η to denote the security parameter.

[3] A notable exception is the one-time pad which is unconditionally secure [198].

2.4.1.1. Elliptic Curves with a Bilinear Map

Elliptic curves have proven to be a highly versatile tool in cryptography. Firstly, they allow for significantly smaller group sizes when compared to classical cryptographic groups [147]; secondly, elliptic curves often have bilinear maps that have given rise to many sophisticated cryptographic constructions. In fact, all known and cryptographically relevant bilinear maps operate on elliptic curves. Since most of the cryptographic schemes used in the cryptographic realization of the API methods require a bilinear map, elliptic curves are a crucial building block.

A bilinear map e is a function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ that takes as input two values from the groups $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively, and returns a value in the group $\mathbb{G}_T$. For our particular setup, we require that $|\mathbb{G}_1| = |\mathbb{G}_2| = |\mathbb{G}_T| = p$, for some large prime p, and that e is a type III pairing [116], i.e., $\mathbb{G}_1 \neq \mathbb{G}_2$ and there is no efficiently computable homomorphism between $\mathbb{G}_1$ and $\mathbb{G}_2$. From e, we require that

1. e is efficiently computable;
2. e is bilinear (linear in both arguments), i.e., the condition

   $$\begin{aligned}&\forall a, b \in \mathbb{Z}_p, \mathcal{X} \in \mathbb{G}_1, \mathcal{Y} \in \mathbb{G}_2 :\\ &\quad e(a\mathcal{X}, b\mathcal{Y}) = e(\mathcal{X}, b\mathcal{Y})^a = e(a\mathcal{X}, \mathcal{Y})^b\end{aligned}$$

   holds;
3. e is non-degenerate, i.e., if $\langle \mathcal{G} \rangle = \mathbb{G}_1$ and $\langle \mathcal{H} \rangle = \mathbb{G}_2$, then $\langle e(\mathcal{G}, \mathcal{H}) \rangle = \mathbb{G}_T$, where $\langle x \rangle$ denotes the group generated by x.

We instantiate the elliptic curves with bilinear map with MNT curves [181].

Notation. In cryptography, group operations are typically written in multiplicative form g^x. For historical reasons, we write elliptic curve operations in additive form $x\mathcal{G}$. More precisely, we use the convention that operations on the elliptic curve groups $\mathbb{G}_1$ and $\mathbb{G}_2$ are written in additive form; operations in the target group $\mathbb{G}_T$ are written in multiplicative form.

Here and throughout the rest of this thesis, we let $\mathcal{G}$ and $\mathcal{H}$ denote the distinguished generators of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively. Furthermore, we let p denote a large prime, calligraphic uppercase letters ($\mathcal{G}$, $\mathcal{H}$, ...) denote elliptic curve elements, lowercase letters denote elements from $\mathbb{Z}_p$.

2.4.1.2. Commitments

Commitments are an essential building block for the Groth-Sahai zero-knowledge proof scheme. Intuitively, a commitment is the digital equivalent of a message in a closed envelope lying on top of a table. The creator of the message cannot change it and no one can look inside until it is opened.

More formally, a principal commits to a value x by applying the randomized commitment function to obtain a commitment C_x on x along with the so-called opening information O. Opening C_x requires the opening information O, and C_x itself. In our case, the opening information is the committed value x and the randomness r used in the commitment.

We use ElGamal encryptions as commitments in order to obtain proofs of knowledge. A proof of knowledge is formalized by a knowledge extractor [121] that, given a zero-knowledge proof, can extract the witnesses hidden by a zero-knowledge proof. Since instances of ElGamal encryptions [105] naturally have a decryption key, this key allows a knowledge extractor to open all commitments and to extract all values used in a zero-knowledge proof, including the hidden ones. As such, the security of the commitments relies on the decisional Diffie-Hellman (DDH) problem. To achieve security in our setting, we should have a security parameter of at least $\eta = 112$ [208].

Throughout the remainder of this thesis, we let C_x denote a commitment to value x and $[\![C]\!]$ denote the value committed to in C, i.e., $[\![C_x]\!] = x$.

2.4.1.3. Groth-Sahai Zero-Knowledge Proof Scheme

Groth-Sahai proofs are non-interactive zero-knowledge proofs of knowledge,[4] which capture relations among committed values that involve elliptic curve operations and bilinear map applications. For instance, the equation

$$[\![C_x]\!] \cdot [\![C_{\mathcal{G}}]\!] = [\![C_{\mathcal{H}}]\!]$$

states that the value committed to in C_x multiplied by the value committed to in $C_{\mathcal{G}}$ equals the value committed to in $C_{\mathcal{H}}$, where $c \cdot \mathcal{V}$ denotes the scalar multiplication of c by $\mathcal{V}$. In general, Groth-Sahai proofs fulfill only the weaker notion of witness-indistinguishability [122]. Our equations, however, are of a special form for which Groth-Sahai proofs are also zero-knowledge [128].

A Groth-Sahai proof on its own solely states that some values contained inside commitments satisfy a given equation. The expressive power of the Groth-Sahai scheme stems from the capability to selectively reveal and hide values occurring in these equations. For instance, if the proof for the equation above contains the opening information for $C_{\mathcal{G}}$ and $C_{\mathcal{H}}$, then this proof shows the knowledge of the discrete logarithm x of $\mathcal{H}$ to the basis $\mathcal{G}$, keeping the discrete logarithm x hidden. Naturally, values can be hidden by removing the respective opening information from a proof. The zero-knowledge property ensures that no information about the hidden witnesses can be learned by the verifier, which faithfully captures the privacy property expressed by existential quantification.

Since Groth-Sahai proofs show the validity of a set of equations, concatenating two proofs shows the validity of the union of the equations proven by the two individual proofs; separating the set of proven equations creates two proofs, each showing the validity of its share of the equations. Realizing a logical disjunction is significantly more challenging since such a proof must hide which branch is valid. We use arithmetization techniques [128] but the prover must have all values appearing in the proof at her disposal. This explains why function $\mathsf{mk}_{\vee}$ succeeds only if the proof passed as input has not previously been processed

[4] Technically, the non-interactive property of Groth-Sahai proofs relies on a common reference string [121].

by the function hide (see Section 2.3.1). Finally, we mention that the Groth-Sahai scheme relies on a common reference string (CRS). We assume a global, trustworthy CRS. Such a CRS can be created by a TTP or by a distributed community effort, for instance, using secure multiparty computation schemes.

The Groth-Sahai proof system can be setup to rely on different assumptions. We use the instantiation based on the symmetric external Diffie-Hellman assumption (SXDH) [128], that is, the DDH problem is intractable in $\mathbb{G}_1$ and $\mathbb{G}_2$ (evidence that MNT curves satisfy the SXDH assumptions are given, for instance, by Ballard et al. [36] and Ateniese et al. [16]). This assumption justified the necessity of a type-III map as otherwise, the DDH problem would be trivially solvable in $\mathbb{G}_1$ [117]. As for the commitment schemes, a security parameter of at least $\eta = 112$ is recommended [208].

2.4.1.4. Automorphic Signature Scheme

We use the automorphic digital signature scheme proposed by Abe et al. [6]. This scheme is highly efficient and allows us to sign verification keys without encoding them. As previously mentioned, this is crucial to obtain efficient zero-knowledge proofs.

A verification key is a tuple of the form $vk = (x\mathcal{G}, x\mathcal{H})$, where $sk := x \in_{\mathsf{R}} \mathbb{Z}_p$ is the randomly-chosen signing key corresponding to vk. Here and throughout the remainder of the thesis, we use the notation $e \in_{\mathsf{R}} S$ to denote that element e is chosen uniformly at random from the set S. We write $\mathsf{sign}(m)_{sk_I}$ to denote the signature on message m with I's signing key sk_I. Given a verification key $vk = (x\mathcal{G}, x\mathcal{H})$, the first component $x\mathcal{G}$ is used as part of a signed message and the second component $x\mathcal{H}$ is used as a verification key in a signature verification. If both components occur simultaneously, they are connected by proving that

$$e(x\mathcal{G}, \mathcal{H}) = e(\mathcal{G}, x\mathcal{H}),$$

that is, both components belong together because they are associated to the same secret key. Furthermore, the scheme is fully compatible with the Groth-Sahai proof system: we write

$$\mathsf{ver}([\![C_{sig}]\!], [\![C_m]\!], [\![C_{vk}]\!])$$

to denote a zero-knowledge proof showing that the value committed to in C_{sig} is a signature on the value committed to in C_m, which can be verified using the verification key committed to in C_{vk} [6]. This proof realizes a proof for the formula $[\![C_{vk}]\!]$ says $[\![C_m]\!]$ and can be fine-tuned to open any of these commitments, revealing the respective values. Notice that the cryptographic as well as the logical notation hide the proofs for connecting $x\mathcal{G}$ and $x\mathcal{H}$. We keep these implicit throughout the remainder of the thesis.

The digital signature scheme by Abe et al. is existentially unforgeable under chosen-message attacks [124], the standard notion of security for signature schemes. Its security relies on the q-ADH-SDH assumption [6] and the AWF-CDH assumption. The AWF-CDH assumption is implied by the SXDH assumption (see [6], Lemma 1). As a consequence, we need a security parameter η that at least renders the SXDH problem infeasible, i.e., $\eta \geq 112$ [208].

2.4.1.5. Hashing into $\mathbb{G}_1$.

There are several ways to define hash functions h that map arbitrary strings into the group $\mathbb{G}_1$. The most straightforward way is to use any ordinary hash function such as $\mathsf{SHA\text{-}256}$ and let $h(x) := \mathsf{SHA\text{-}256}(x) \cdot \mathcal{G}$. The drawback of this method is that it reveals the discrete logarithm of the hash value with respect to $\mathcal{G}$, which will break the security of service-specific pseudonyms. There are several schemes in the literature (e.g., [53, 197, 140]) that solve this problem by encoding points directly on the curve. We use the method by Icart [140] as this scheme enjoys properties such as one-wayness and collision-resistance.

We idealize the hash function and assume the random oracle model [111, 47], that is, we assume that hash functions output a truly random string but answer consistently with previous queries.

2.4.1.6. Service-Specific Pseudonyms.

Service-specific pseudonyms are a cryptographic primitive that is compatible with the Groth-Sahai proof scheme. An SSP is computed from a service description S and a signing key. More precisely, the owner of verification key $vk = x \cdot \mathcal{G}$ computes her pseudonym psd for the service S as follows: $psd := x \cdot \mathcal{S}$ where $\mathcal{S} := h(S)$. The hash function h is as described above. In particular, the discrete logarithm of $\mathcal{S}$ to the basis $\mathcal{G}$ is unknown. The zero-knowledge proof for service-specific pseudonyms then shows the validity of the two equations

$$[\![C_x]\!] \cdot [\![C_{\mathcal{G}}]\!] = [\![C_{vk}]\!] \ \wedge \ [\![C_x]\!] \cdot [\![C_{\mathcal{S}}]\!] = [\![C_{psd}]\!].$$

The left conjunct shows the well-formedness of the verification key. The right conjunct computes the service-specific pseudonym in zero-knowledge, using the same commitment for the signing key in both proofs. This proof shows the validity of the formula $\mathsf{SSP}(vk, \mathcal{S}, psd)$. Notice that the signing key x is essential for the creation of the proof but does not occur in the logical description. We stipulate that this proof always keeps the signing key x hidden and always reveals $\mathcal{G}$. In a proof comprising more than one pseudonym, the left conjunct needs to be shown only once since it is the same for all of the user's pseudonyms.

The security of SSPs relies on the DDH assumption in $\mathbb{G}_1$ and the random oracle model. The DDH assumption is implied by the SXDH assumption. Consequently, we need a security parameter of at least $\eta = 112$ [208].

2.4.2. Cryptographic Realization of API Methods

We describe the concrete cryptographic realization of all API methods using the constructions and building blocks introduced in Section 2.4.1.

Proving binary relations. Proving binary relations in zero-knowledge is a well-studied problem and several approaches that are compatible with the Groth-Sahai zero-knowledge proof scheme exist. Proofs of equality are natively supported by the Groth-Sahai proof system and inequality proofs are well known (see, e.g., Bangerter et al. [37], §4.6). Respectively, we denote these proofs by

$$[\![C]\!] = [\![D]\!] \text{ and } [\![C]\!] \neq [\![D]\!].$$

For arithmetic relations $\mathsf{op} \in \{<, \leq, \geq, >\}$, we follow the approach proposed by Meiklejohn [179] to implement the zero-knowledge proofs $[\![C_{s_1}]\!] \ \mathsf{op} \ [\![C_{s_2}]\!]$.

The construction of Meiklejohn requires the strong non-degeneracy from the bilinear map e, i.e., $e(\mathcal{X}, \mathcal{Y}) = 0$ if and only if $\mathcal{X} = \mathcal{O}$ or if $\mathcal{Y} = \mathcal{O}$, whereas the definition of bilinear map requires that $e(\mathcal{X}, \mathcal{Y}) \neq 0$ for all generators $\mathcal{X}$ and $\mathcal{Y}$ of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively. In the setting where $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ are prime-order groups, however, every non-zero element is a generator and the non-degeneracy property implies the strong non-degeneracy requirement.

Proving mathematical operations. Proving mathematical operations in zero-knowledge is natively supported by the Groth-Sahai proof scheme. We denote these proofs by

$$[\![C]\!] = [\![D_1]\!] \ \mathsf{op} \ [\![D_2]\!]$$

for arithmetic operations $\mathsf{op} \in \{+, -, \cdot\}$ and the elements C, D_1, and D_2 from $\mathbb{Z}_r$. Since $\mathbb{Z}_r$ is finite, numbers can grow large enough to be affected by the computation modulo the group order. For instance, let $p = 11$ and let $[\![D_1]\!] = [\![D_2]\!] = 6$. Then $[\![D_1]\!] + [\![D_2]\!] = 1$. However, in many application scenarios, numbers will remain small enough and can be treated as if they were integers.

Proving the ownership of a pseudonym. We already discussed above how SSPs are cryptographically realized. We use the notation

$$\mathrm{SSP}([\![C_x]\!], [\![C_{vk}]\!], [\![C_{\mathcal{S}}]\!]) = [\![C_{psd}]\!]$$

as shorthand for

$$[\![C_x]\!] \cdot [\![C_{\mathcal{G}}]\!] = [\![C_{vk}]\!] \ \wedge \ [\![C_x]\!] \cdot [\![C_{\mathcal{S}}]\!] = [\![C_{psd}]\!].$$

Proving list non-membership. For proving $(psd, \_) \notin L$, given a list $L = (psd_1, attr_1), \ldots, (psd_\ell, attr_\ell)$, we show that psd is different from all pseudonyms in L:

$$\mathrm{SSP}([\![C_x]\!], [\![C_{vk}]\!], [\![C_{\mathcal{S}}]\!]) = [\![C_{psd}]\!] \ \wedge \ \textstyle\bigwedge_{i=1}^{\ell} [\![C_{psd}]\!] \neq [\![C_{psd_i}]\!].$$

Proving list membership. The proof of list membership assumes the list administrator's signatures $\mathsf{sign}(psd_i, attr_i, tag)_{sk_{Admin}}$ on each of the individual list elements $(psd_i, attr_i)$, where tag uniquely identifies the list L. We exploit this particular list representation to make the list membership proof independent of the list size: we show the existence of a signature that belongs to the list without revealing the signature itself nor the pseudonym it signs. This construction closely resembles the signature-based set membership proof by Camenisch et al. [70], which we extend in order to prove statements of the form $(x, \_) \in L$ as opposed to $x \in L$. Specifically, a proof for $(psd, attr) \in L$ shows the validity of the formula

$$\mathsf{ver}([\![C_s]\!], ([\![C_{psd}]\!], [\![C_{attr}]\!], [\![C_{tag}]\!]), [\![C_{vk}]\!]).$$

We stipulate that this proof always reveals the tag *tag* to show that the $(psd, attr)$ pair indeed belongs to the list L.

As reputation lists are dynamic objects that change over time, one has to be careful in the choice of the tag uniquely identifying the list. For instance, if the tag were the hash value of the list, a change in the list would require re-signing all individual elements. Therefore, we propose to use a combination of a list description and an epoch number as tags. For instance, the list for a service S would be tagged "List for service S, epoch 2" for the second epoch. Thus, adding elements does not require any re-signing, since only the newly added entries must be signed. Only removing elements causes an increase of the epoch numbers and requires the list administrator to re-sign all elements.

Identity escrow. The identity escrow proof exploits the idea of the service-specific pseudonyms. Since SSPs are designed to protect the identity of the users, however, we have to modify the protocol and add an extra piece of information to enable an escrow agent EA to reveal the user's identity. More precisely, the user obtains from the EA a random value r and a signature $s := \mathsf{sign}(\mathcal{R})_{sk_{EA}}$ on the escrow value $\mathcal{R} := r \cdot vk$ where vk is the user's verification key. We use this value to compute the escrow information $idr := r \cdot \mathcal{S}$ for service $\mathcal{S}$. The user has to prove the following statement:

$$\begin{aligned}&\mathsf{ver}([\![C_s]\!], [\![C_{\mathcal{R}}]\!], [\![C_{vk_{EA}}]\!])\\ \wedge\ &[\![C_r]\!] \cdot [\![C_{vk}]\!] = [\![C_{\mathcal{R}}]\!]\\ \wedge\ &[\![C_r]\!] \cdot [\![C_{\mathcal{S}}]\!] = [\![C_{idr}]\!].\end{aligned}$$

We stipulate that r is never revealed.

Akin to SSPs, this proof does not reveal the identity of the user. The EA, however, knows all the random value-user pairs, in our case r and vk. If the EA is informed of a cogent reason to reveal the identity of the user associated with the escrow information idr for service $\mathcal{S}$, the EA can successively try all stored random values r' to check whether $r' \cdot \mathcal{S} = idr$; eventually, $r' = r$ and the user is identified.

For some scenarios, assuming the presence of a trusted third party is not possible. It is possible to decrease the trust assumption from a TTP via secure multiparty computation schemes (e.g., [48, 74]). Furthermore, the escrow agents can be distributed to offer better trust guarantees. In the latter case, the user has to obtain a random value and the corresponding signature from each of the escrow agents and combine the random values into one. Identity escrow then requires the collaboration of all the escrow agents. They may use a secure multi-party computation to compute the joint value r without revealing one's share to other parties.

The security of escrow identifiers relies on the DDH assumption in $\mathbb{G}_1$ and the random oracle model. The DDH assumption is implied by the SXDH assumption.

2.5. Proofs

Dedicated to the API proofs, this section is organized in two parts: the first part proves the anonymity and (un-)linkability properties of service-specific pseudonyms and the identity escrow protocol in a cryptographic setting; the second part proves the soundness result for the API methods in a symbolic setting.

The proofs for the cryptographic building blocks rely on hardness assumptions. More precisely, we show that breaking any of the claimed security properties requires operations that are computationally infeasible. The soundness proofs for the API methods rely upon a symbolic abstraction of the cryptographic primitives. Intuitively, the soundness results shows that whenever a zero-knowledge proof for a formula successfully verifies, then that formula holds. The different proof techniques are necessary because the computational proofs show properties for single cryptographic building blocks. The individual security of these building blocks, however, does not imply the overall security of the API methods. For instance, one could imagine that the malleability properties of the zero-knowledge scheme enables an unforeseen interleavings of API methods that breaks the desired soundness result.

We start with the computational proofs for the cryptographic building blocks.

2.5.1. Cryptographic Proofs of Anonymity and Unlinkablity

In this section, we prove the uniqueness, the unlinkability, and the anonymity properties of pseudonyms and escrow identifiers. We start with the SSP properties and then proceed to the properties of the escrow identifiers.

2.5.1.1. Uniqueness, Unlinkability, and Anonymity of Service-Specific Pseudonyms

Uniqueness of pseudonyms. In the following, we use function Y to define the uniqueness property. Intuitively, $Y_{\mathcal{X}}(\mathcal{A}, \mathcal{B})$ extracts the discrete logarithm x of $\mathcal{A}$ to the basis $\mathcal{X}$ and returns $x\mathcal{B}$. Notice that the value $Y_{\mathcal{G}}(vk, \mathcal{S})$ corresponds to the SSP of the owner of vk and the service $\mathcal{S}$, and the value $Y_{vk}(\mathcal{R}, \mathcal{S})$ corresponds to the escrow identifier of the owner of vk and the service $\mathcal{S}$.

Definition 2.1. *We define the function* $Y_{\mathcal{X}} : \mathbb{G}_1 \times \mathbb{G}_1 \to \mathbb{G}_1$ *as* $Y_{\mathcal{X}}(x\mathcal{X}, \mathcal{Y}) \mapsto x\mathcal{Y}$.

In general, $Y_{\mathcal{X}}$ cannot be efficiently computed and is only used for defining properties of SSPs and escrow identifiers. In fact, the DDH assumption implies that the discrete logarithm cannot be extracted and used as suggested by function Y.

To prove the uniqueness result, we first state basic facts about the distribution of hash values and of secret signing keys. The following proposition holds since the output of the (random oracle) hash function h and the signing key are uniformly random values from a set that is exponentially large in the security parameter η (see Section 2.4 and Abe et al. [6]).

Proposition 2.1. *The following probabilities are negligible in η:*

- *The output of the hash function $h : \{0,1\}^* \to \mathbb{G}_1$ is $\mathcal{O}$, the neutral element of the group operation of $\mathbb{G}_1$.*
- *The output of the hash function $h : \{0,1\}^* \to \mathbb{G}_1$ coincides for polynomially (in η) many different inputs.*
- *A signing key x is 0.*
- *A pseudonym $psd := x\mathcal{S}$ is $\mathcal{O}$.*
- *Two signing keys from a set of polynomially (in η) many coincide.*

We can now proceed to prove the uniqueness theorem. First, we formally define the uniqueness for SSPs using the Y function.

Definition 2.2 (Uniqueness of service-specific pseudonyms)**.** *We say that service-specific pseudonyms are* unique *if and only if, the following conditions hold with overwhelming probability:*

1. *for any service $\mathcal{S}$ and two honestly-generated verification keys vk_1 and vk_2, $Y_{\mathcal{G}}(vk_1, \mathcal{S}) \neq Y_{\mathcal{G}}(vk_2, \mathcal{S})$,*
2. *for any verification key vk and service $\mathcal{S}$, $Y_{\mathcal{G}}(vk, \mathcal{S})$ is a unique value,*
3. *for any two different service descriptions* S_1 *and* S_2 *and verification key vk, $Y_{\mathcal{G}}(vk, h(\mathrm{S}_1)) \neq Y_{\mathcal{G}}(vk, h(\mathrm{S}_2))$.*

Theorem 2.1 (Uniqueness of SSP)**.** *In the random oracle model service-specific pseudonyms and escrow identifiers are unique.*

Proof. First, we note that the signing key x is chosen randomly from the set $\mathbb{Z}_p$.

Condition 1: for any service $\mathcal{S}$ and two verification keys vk_1 and vk_2, $Y_{\mathcal{G}}(vk_1, \mathcal{S}) = Y_{\mathcal{G}}(vk_2, \mathcal{S})$ if and only if $\mathcal{S} = \mathcal{O}$, the only non-generator of $\mathbb{G}_1$, or $vk_1 = vk_2$; the two verification keys coincide if and only if the two corresponding, honestly-chosen signing keys coincide. These two events happen only with negligible probability by Proposition 2.1.

Condition 2: follows immediately since Y is a deterministic function.

Condition 3: for any verification key and two different service descriptions S_1 and S_2, $Y_{\mathcal{G}}(vk, h(\mathrm{S}_1)) = Y_{\mathcal{G}}(vk, h(\mathrm{S}_2))$ if and only if h maps S_1 and S_2 to the same hash value or the signing key is 0. These two events happen only with negligible probability by Proposition 2.1.

□

Anonymity of pseudonyms (intra-service unlinkability). We now prove the theorem asserting that service-specific pseudonyms preserve the anonymity of users. We begin by stating our definition of anonymity.

Definition 2.3 (Pseudonym-based Anonymity)**.** *A set of k pseudonyms $\{psd_1 := x\mathcal{S}_1, \dots, psd_k := x\mathcal{S}_k\}$ for k services (as constructed in Section 2.4) provides anonymity if and only if, given a set $\{vk_1, \dots, vk_m\}$ of m verification keys, any polynomially-bounded attacker can determine which verification key was used to compute $psd_1, \dots, psd_k$ with probability at most $\frac{1}{m} + \mu$, where μ is negligible in η.*

Intuitively, pseudonyms provide anonymity if pure guessing essentially is as good as an attacker that tries to determine which verification key vk from the set M was used for computing $psd_1, \dots, psd_k$.

We now work our way towards the main theorem. The proof is a reduction against the decisional Diffie-Hellman (DDH) problem. For the sake of completeness, we give all the necessary definitions.

Definition 2.4 (DDH and DDH Advantage)**.** *Given the tuple $(\mathcal{G}, x\mathcal{G}, y\mathcal{G}, \mathcal{C})$, where $\langle\mathcal{G}\rangle = \mathbb{G}_1$ is a generator of $\mathbb{G}_1$, and $x, y \in \mathbb{Z}_p$ are randomly chosen, the DDH problem is to decide whether $\mathcal{C} = xy\mathcal{G}$.*

The advantage of a DDH attacker $\mathbb{B}$ is defined as

$$\begin{aligned}&\mathrm{Adv}^{\mathrm{DDH}}(\mathbb{B}) = \\ &\quad |\mathsf{Pr}[1 \leftarrow \mathbb{B}(1^n, \mathcal{G}, x\mathcal{G}, y\mathcal{G}, xy\mathcal{G}) \mid b = 1] - \mathsf{Pr}[1 \leftarrow \mathbb{B}(1^n, \mathcal{G}, x\mathcal{G}, y\mathcal{G}, z\mathcal{G}) \mid b = 0]|\end{aligned}$$

where z is a random value in $\mathbb{G}_1$ and b is randomly chosen from $\{0, 1\}$.

Intuitively, the advantage of a DDH attacker states how much better than pure guessing the attacker performs.

Assumption 2.1 (Hardness of DDH)**.** *For all polynomially-bounded attackers $\mathbb{B}$, the advantage $\mathrm{Adv}^{\mathrm{DDH}}(\mathbb{B})$ is negligible in η.*

Reviewing the construction of service-specific pseudonyms, we see that the values $(\mathcal{G}, vk, \mathcal{S}, psd)$ form a valid Diffie-Hellman tuple since $vk = x\mathcal{G}$ for a random x, $\mathcal{S} = r\mathcal{G}$ for a random r, and $psd = x\mathcal{S} = xr\mathcal{G}$. We now state and prove our main theorem about service-specific pseudonyms.

Theorem 2.2 (Anonymity of Service-Specific Pseudonyms)**.** *In the random oracle model and under the DDH assumption, service-specific pseudonyms (as constructed in Section 2.4) provide anonymity.*

Proof. The proof is a reduction against DDH. Figure 2.2 visualizes the steps of this reduction proof. Intuitively, the set $\{vk_1, \dots, vk_k\}$ consists of the verification keys obtained by a service provider during the registration of k principals. Suppose there is an attacker $\mathbb{A}$ that, on input $(\mathcal{G}, \{vk_1, \dots, vk_m\}, (\mathcal{S}_1, \dots, \mathcal{S}_k), (psd_1, \dots, psd_k))$, outputs ℓ such that vk_ℓ and $psd_1, \dots, psd_k$ are associated with probability $1/m + \mu$ where μ is non-negligible. From this attacker, we construct an attacker $\mathbb{B}$ that breaks the decisional Diffie-Hellman problem with non-negligible probability.

The DDH challenger $\mathbb{C}$ uniformly at random draws a bit $b \in_{\mathsf{R}} \{0,1\}$. If $b = 1$, $\mathbb{C}$ generates a valid DDH tuple, if $b = 0$, $\mathbb{C}$ generates a fake DDH tuple, that is, a tuple where $\mathcal{C} = z\mathcal{G}$ for $z \in_{\mathsf{R}} \mathbb{Z}_p$. The resulting tuple is sent to attacker $\mathbb{B}$.

Given that DDH challenge $(\mathcal{G}, x\mathcal{G}, y\mathcal{G}, \mathcal{C})$, attacker $\mathbb{B}$ must give a perfect simulation to attacker $\mathbb{A}$, so that $\mathbb{A}$ cannot differentiate between a normal challenge and a challenge constructed by $\mathbb{B}$. We note that the value $\mathcal{S}$ in our service-specific pseudonym construction is a value that is indistinguishable from a random value in $\mathbb{G}_1$: it is the output of a random oracle and hence, its discrete logarithm r with respect to $\mathcal{G}$ is also indistinguishable from a random number in $\mathbb{Z}_p$. Hence, a computationally bounded attacker cannot notice the difference and $\mathcal{S}$ matches with $y\mathcal{G}$. Furthermore, we note that verification keys are constructed exactly as $x\mathcal{G}$.

$\mathbb{B}$ chooses $\ell' \in_{\mathsf{R}} \{1, \dots, m\}$ and generates m random verification keys $vk_i \in \mathbb{G}_1$. Next, $\mathbb{B}$ randomly draws $s_i \in_{\mathsf{R}} \mathbb{Z}_p$ for $i \in \{2, \dots, k\}$. Since we are in a set with prime-order groups, every service and every pseudonym is a generator of the whole group $\mathbb{G}_1$ (except for $\mathcal{O}$, which occurs only with negligible probability, see Proposition 2.1). Therefore, given a service $\mathcal{S}$ and a corresponding pseudonym $psd := x \cdot \mathcal{S}$, multiplying both with a random value s yields $s \cdot \mathcal{S}$ and $s \cdot psd = x \cdot (s \cdot \mathcal{S})$. The products form another random service and the corresponding (random) pseudonym, justifying $\mathbb{B}$'s action to draw random numbers and multiply them in the following call to $\mathbb{A}$.

$\mathbb{A}(\{vk_1, \dots, vk_{\ell'-1}, x\mathcal{G}, vk_{\ell'+1}, \dots, vk_m\}, \mathcal{G}, (y\mathcal{G}, ys_2\mathcal{G}, \dots, ys_k\mathcal{G}), (\mathcal{C}, s_2\mathcal{C}, \dots, s_k\mathcal{C}))$ is called by $\mathbb{B}$. In turn, $\mathbb{B}$ receives ℓ as answer. $\mathbb{B}$ returns 1 if and only if $\ell = \ell'$ where 1 denotes $\mathbb{B}$'s decision that $c = xy\mathcal{G}$. In the following calculation, we let $z \in_{\mathsf{R}} \mathbb{Z}_p$.

$$
\begin{aligned}
& \mathrm{Adv}^{\mathrm{DDH}}(\mathbb{B}) \\
= \; & |\mathsf{Pr}[1 \leftarrow \mathbb{B}(\mathcal{G}, x\mathcal{G}, y\mathcal{G}, xy\mathcal{G}) \mid b = 1] - \mathsf{Pr}[1 \leftarrow \mathbb{B}(\mathcal{G}, x\mathcal{G}, y\mathcal{G}, z\mathcal{G}) \mid b = 0]| \\
\overset{(1)}{=} \; & \left| \begin{array}{l}
\mathsf{Pr}[vk_i \in_{\mathsf{R}} \mathbb{G}_1, \ell' \in_{\mathsf{R}} \{1, \dots, m\}, s_i \in_{\mathsf{R}} \mathbb{Z}_p : \\
\qquad \ell \leftarrow \mathbb{A}(\{vk_1, \dots, vk_{\ell'-1}, x\mathcal{G}, vk_{\ell'+1}, \dots, vk_m\}, \mathcal{G}, (y\mathcal{G}, ys_2\mathcal{G}, \dots, ys_k\mathcal{G}), \\
\qquad\qquad (xy\mathcal{G}, s_2xy\mathcal{G}, \dots, s_kxy\mathcal{G})) \;\wedge\; \ell = \ell'] \\
-\mathsf{Pr}[vk_i \in_{\mathsf{R}} \mathbb{G}_1, \ell' \in_{\mathsf{R}} \{1, \dots, m\}, s_i \in_{\mathsf{R}} \mathbb{Z}_p : \\
\qquad \ell \leftarrow \mathbb{A}(\{vk_1, \dots, vk_{\ell'-1}, x\mathcal{G}, vk_{\ell'+1}, \dots, vk_m\}, \mathcal{G}, (y\mathcal{G}, ys_2\mathcal{G}, \dots, ys_k\mathcal{G}), \\
\qquad\qquad (z\mathcal{G}, s_2z\mathcal{G}, \dots, s_kz\mathcal{G})) \wedge \ell = \ell']
\end{array} \right| \\
\overset{(2)}{=} \; & \left| \left(\frac{1}{m} + \mu \right) - \frac{1}{m} \right| = \mu
\end{aligned}
$$

For equality (1), we substitute the attacker $\mathbb{B}$ with its definition. Equality (2) holds for the following reason: the left part of the difference holds as we give a perfect simulation for attacker $\mathbb{A}$ who, by assumption, can associate the verification key with the pseudonym with probability $1/m + \mu$, where μ is non-negligible. For the right part of the difference, there are only uniformly random values involved. More precisely, $\mathcal{G}$, $x\mathcal{G}$, $y\mathcal{G}$, and $z\mathcal{G}$ are values that are chosen independently and uniformly at random. As there is no structure that can be used to correlate the values, the best that $\mathbb{A}$ can do is to return some number from $\{1, \dots, k\}$. This number hits ℓ' with probability $1/m$ (since ℓ' was chosen uniformly

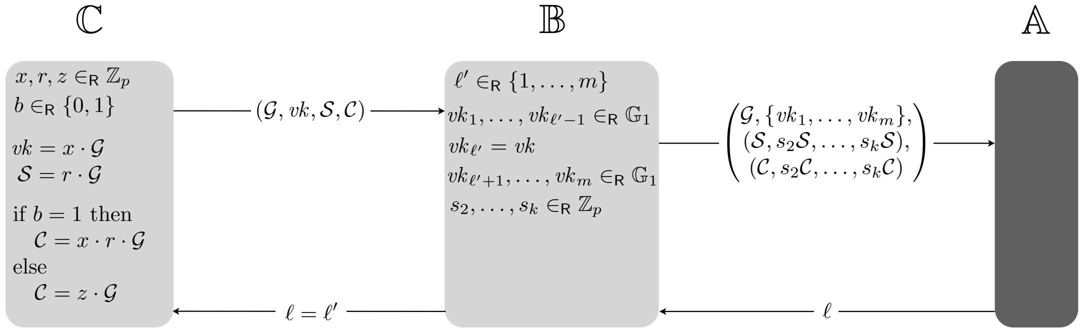

$\mathbb{C}$
$\mathbb{B}$
$\mathbb{A}$
$x, r, z \in_{\mathsf{R}} \mathbb{Z}_p$
$b \in_{\mathsf{R}} \{0,1\}$
$vk = x \cdot \mathcal{G}$
$\mathcal{S} = r \cdot \mathcal{G}$
if $b = 1$ then
$\mathcal{C} = x \cdot r \cdot \mathcal{G}$
else
$\mathcal{C} = z \cdot \mathcal{G}$
$(\mathcal{G}, vk, \mathcal{S}, \mathcal{C})$
$\ell' \in_{\mathsf{R}} \{1, \ldots, m\}$
$vk_1, \ldots, vk_{\ell'-1} \in_{\mathsf{R}} \mathbb{G}_1$
$vk_{\ell'} = vk$
$vk_{\ell'+1}, \ldots, vk_m \in_{\mathsf{R}} \mathbb{G}_1$
$s_2, \ldots, s_k \in_{\mathsf{R}} \mathbb{Z}_p$
$\begin{pmatrix} \mathcal{G}, \{vk_1, \ldots, vk_m\}, \\ (\mathcal{S}, s_2\mathcal{S}, \ldots, s_k\mathcal{S}), \\ (\mathcal{C}, s_2\mathcal{C}, \ldots, s_k\mathcal{C}) \end{pmatrix}$
$\ell = \ell'$
ℓ

Figure 2.2.: Proof overview of Theorem 2.2.

at random). Consequently, $\mathbb{B}$ can use $\mathbb{A}$ to break the DDH problem with non-negligible probability. This contradicts our intractability assumption and we conclude that no such attacker $\mathbb{A}$ can exist and our protocol provides anonymity. □

Unlinkability across services (inter-service unlinkability). The last property of service-specific pseudonyms is unlinkability across services, i.e., it is computationally infeasible to associate two pseudonyms from the same user but for different services with each other. We first define the desired security property and proceed directly with proving that our construction satisfies that definition.

Definition 2.5 (Pseudonym-based Unlinkability across Services)**.** *We say pseudonyms (as constructed in Section 2.4) are unlinkable across services if and only if given a verification key* $vk := x\mathcal{G}$*, an associated pseudonym* $psd_1 := x\mathcal{S}_1$ *for service* $\mathcal{S}_1$*, and a pseudonym* psd_2 *for service* $\mathcal{S}_2 \neq \mathcal{S}_1$*, it is computationally infeasible to decide whether* $psd_2 = x\mathcal{S}_2$*, that is, to decide whether the two pseudonyms belong to the same user or not.*

Theorem 2.3 (Unlinkability of Pseudonyms across Services)**.** *In the random oracle model and under the DDH assumptions, pseudonyms (as constructed in Section 2.4) are unlinkable across services.*

Proof. Suppose $\mathbb{A}$ is an attacker that takes as input a tuple of the form $(\mathcal{G}, vk, \mathcal{S}_1, psd_1 := x_1\mathcal{S}_1, \mathcal{S}_2, psd_2 := x_2\mathcal{S}_2)$ where $\mathcal{S}_1 \neq \mathcal{S}_2$, and decides whether $x_1 = x_2$ with a probability $1/2 + \mu$, where μ is non-negligible; $\mathbb{A}$ outputs 1 to denote that $x_1 = x_2$ and 0 to denote that $x_1 \neq x_2$. We use this attacker to construct attacker $\mathbb{B}$ against the decisional Diffie-Hellman assumption.

The DDH challenger randomly chooses $b \in_{\mathsf{R}} \{0,1\}$. If $b = 1$, the challenger produces a valid DDH tuple $(\mathcal{G}, x\mathcal{G}, y\mathcal{G}, z\mathcal{G})$ for uniformly random values x and y, where $z = x \cdot y$. If $b = 0$, z is randomly chosen. We construct attacker $\mathbb{B}$ that uses $\mathbb{A}$ to solve the given DDH challenge.

First, $\mathbb{B}$ chooses $r \in_{\mathsf{R}} \{1, \ldots, |\mathbb{G}_1|\}$, sets $vk := x\mathcal{G}$, $\mathcal{S}_1 := r\mathcal{G}$, $psd_1 := r \cdot vk$, $\mathcal{S}_2 := y\mathcal{G}$, $psd_2 := z\mathcal{G}$, calls $\mathbb{A}(\mathcal{G}, vk, \mathcal{S}_1, psd_1, \mathcal{S}_2, psd_2)$, and answers the challenge with $\mathbb{A}$'s answer. We observe that $\mathcal{S}_1$ is indistinguishable from the output of a (random oracle) hash function and that psd_1 is the pseudonym associated to vk and $\mathcal{S}_1$. Notice that $psd_2 = x\mathcal{S}_2$ if and only if $z = x \cdot y$.

Let us now compute the success probability of our constructed adversary against the DDH challenge, where we let z denote a random value.

$$\begin{aligned}
& \mathrm{Adv}^{\mathrm{DDH}}(\mathbb{B}) \\
= \; & |\mathsf{Pr}[1 \leftarrow \mathbb{B}(\mathcal{G}, x\mathcal{G}, y\mathcal{G}, xy\mathcal{G}) \mid b = 1] - \mathsf{Pr}[1 \leftarrow \mathbb{B}(\mathcal{G}, x\mathcal{G}, y\mathcal{G}, z\mathcal{G}) \mid b = 0]| \\
\overset{(1)}{=} \; & \left| \begin{array}{l} \mathsf{Pr}[r \in_{\mathsf{R}} \{1, \ldots, |\mathbb{G}_1|\} : 1 \leftarrow \mathbb{A}(\mathcal{G}, x\mathcal{G}, r\mathcal{G}, rx\mathcal{G}, y\mathcal{G}, xy\mathcal{G}) \mid b = 1] \\ -\mathsf{Pr}[r \in_{\mathsf{R}} \{1, \ldots, |\mathbb{G}_1|\} : 1 \leftarrow \mathbb{A}(\mathcal{G}, x\mathcal{G}, r\mathcal{G}, rx\mathcal{G}, y\mathcal{G}, z\mathcal{G}) \mid b = 0] \end{array} \right| \\
\overset{(2)}{=} \; & \left| \begin{array}{l} \mathsf{Pr}[r \in_{\mathsf{R}} \{1, \ldots, |\mathbb{G}_1|\} : 1 \leftarrow \mathbb{A}(\mathcal{G}, x\mathcal{G}, r\mathcal{G}, rx\mathcal{G}, y\mathcal{G}, xy\mathcal{G}) \mid b = 1] \\ -(1 - \mathsf{Pr}[r \in_{\mathsf{R}} \{1, \ldots, |\mathbb{G}_1|\} : 0 \leftarrow \mathbb{A}(\mathcal{G}, x\mathcal{G}, r\mathcal{G}, rx\mathcal{G}, y\mathcal{G}, z\mathcal{G}) \mid b = 0])| \end{array} \right| \\
\overset{(3)}{=} \; & |\tfrac{1}{2} + \mu - (1 - (\tfrac{1}{2} + \mu))| = 2\mu
\end{aligned}$$

In step (1), we substitute attacker $\mathbb{B}$ with its definition, and in step (2), we consider that if $\mathbb{A}$ decides whether $z = x \cdot y$, then $\mathbb{A}$ also decides whether $z \neq x \cdot y$. In step (3), we substitute $\mathbb{A}$ with its success probability. Thus, attacker $\mathbb{B}$ can use $\mathbb{A}$ to break the decisional Diffie-Hellman challenge with non-negligible probability, which violates our assumption. We conclude that no attacker $\mathbb{A}$ exists. □

2.5.1.2. Uniqueness, Unlinkability, and Anonymity of Escrow Identifiers

Analogous to the proofs for service-specific pseudonyms, we show that escrow identifiers are unique, preserve the anonymity of users, and that they are unlinkable across services. The cryptographic construction strongly resembles that of service-specific pseudonyms but the amount of information available to outside parties is different and requires different proofs. We use most of the notation introduced above and begin by stating the uniqueness property.

Definition 2.6 (Uniqueness of escrow identifiers)**.** *We say that escrow identifier are unique if and only if, the following conditions hold with overwhelming probability:*

1. *for any service $\mathcal{S}$, two verification keys vk_1 and vk_2, and two honestly-generated escrow values $\mathcal{R}_1$ and $\mathcal{R}_2$, $Y_{vk_1}(\mathcal{R}_1, \mathcal{S}) \neq Y_{vk_2}(\mathcal{R}_2, \mathcal{S})$,*

2. *for any verification key vk, corresponding escrow value $\mathcal{R}$, and service $\mathcal{S}$, the escrow identifier $Y_{vk}(\mathcal{R}, \mathcal{S})$ is a unique value,*

3. *for any two different service descriptions S_1 and S_2 and verification key vk and corresponding escrow value $\mathcal{R}$, $Y_{vk}(\mathcal{R}, h(\mathrm{S}_1)) \neq Y_{vk}(\mathcal{R}, h(\mathrm{S}_2))$.*

Theorem 2.4 (Uniqueness of Escrow Identifiers)**.** *In the random oracle model, escrow identifiers are unique.*

Proof. The proof is analogous to that of Theorem 2.1, where the secret x is replaced by the value r that is randomly chosen by the trusted third party EA.

□

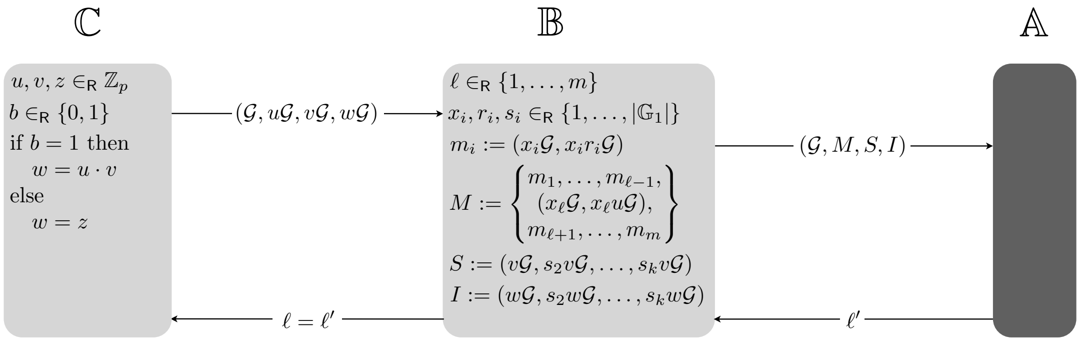

$\mathbb{C}$
$\mathbb{B}$
$\mathbb{A}$
$u, v, z \in_{\mathsf{R}} \mathbb{Z}_p$
$b \in_{\mathsf{R}} \{0,1\}$
if $b = 1$ then
$w = u \cdot v$
else
$w = z$
$(\mathcal{G}, u\mathcal{G}, v\mathcal{G}, w\mathcal{G})$
$\ell \in_{\mathsf{R}} \{1, \dots, m\}$
$x_i, r_i, s_i \in_{\mathsf{R}} \{1, \dots, |\mathbb{G}_1|\}$
$m_i := (x_i\mathcal{G}, x_i r_i \mathcal{G})$
$M := \left\{ \begin{matrix} m_1, \dots, m_{\ell-1}, \\ (x_\ell\mathcal{G}, x_\ell u\mathcal{G}), \\ m_{\ell+1}, \dots, m_m \end{matrix} \right\}$
$S := (v\mathcal{G}, s_2 v\mathcal{G}, \dots, s_k v\mathcal{G})$
$I := (w\mathcal{G}, s_2 w\mathcal{G}, \dots, s_k w\mathcal{G})$
$(\mathcal{G}, M, S, I)$
ℓ'
$\ell = \ell'$

Figure 2.3.: Proof overview of Theorem 2.3.

Definition 2.7 (Escrow-Identifier-based Anonymity)**.** *Let $M = \{(vk_1, \mathcal{R}_1 := r_1 vk_1), \dots, (vk_m, \mathcal{R}_m := r_m vk_m)\}$ be a set of verification keys and their corresponding random values (induced by the signing process of the EA). We say that escrow identifiers (as constructed in Section 2.4) provide anonymity if and only if, given k services $S := (\mathcal{S}_1, \dots, \mathcal{S}_k)$ and k corresponding escrow identifiers $I := (idr_1 := r_i\mathcal{S}_1, \dots, idr_k := r_i\mathcal{S}_k)$, it is computationally infeasible to decide which $(vk, \mathcal{R})$ pair is associated with the services and escrow identifier (that is, which i is such that $\mathcal{R} = r_i vk$).*

The following theorem states that escrow identifiers preserve the anonymity of users.

Theorem 2.5 (Anonymity for Escrow Identifiers)**.** *In the random oracle model and under the DDH assumptions, escrow identifiers (as constructed in Section 2.4) provide anonymity.*

Proof. We reduce this problem against the decisional Diffie-Hellman problem. Suppose we are given an attacker $\mathbb{A}$ that breaks the anonymity of escrow identifiers, that is, given M, S, and I as in Definition 2.7, $\mathbb{A}$ outputs the correct i with probability greater than $1/m + \mu$ where μ is non-negligible in the security parameter η. From this attacker, we construct attacker $\mathbb{B}$ that breaks the DDH problem.

We are given the tuple $(\mathcal{G}, u\mathcal{G}, v\mathcal{G}, w\mathcal{G})$ and we are to decide whether $w = u \cdot v$. First we draw x_i, r_i, and s_i uniformly at random from the set of exponents $\{1, \dots, |\mathbb{G}_1|\}$, and we draw $\ell \in_{\mathsf{R}} \{1, \dots, m\}$. Let $M := \{(x_1\mathcal{G}, x_1 r_1\mathcal{G}), \dots, (x_{\ell-1}\mathcal{G}, x_{\ell-1} r_{\ell-1}\mathcal{G}), (x_\ell\mathcal{G}, x_\ell u\mathcal{G}), (x_{\ell+1}\mathcal{G}, x_{\ell+1} r_{\ell+1}\mathcal{G}), \dots, (x_m\mathcal{G}, x_m r_m\mathcal{G})\}$, let $S := (v\mathcal{G}, s_2 v\mathcal{G}, \dots, s_k v\mathcal{G})$, and let $I := (w\mathcal{G}, s_2 w\mathcal{G}, \dots, s_k w\mathcal{G})$. We run $\mathbb{A}(\mathcal{G}, M, S, I)$. The set M, S, and I now have exactly the shape of M, S, and I as described in Definition 2.7, respectively, if $w = u \cdot v$. If $\mathbb{A}$ can figure out ℓ, then our attacker $\mathbb{B}$ can break the given DDH challenge.

Let us now compute the advantage of our construction $\mathbb{B}$ against the given DDH challenge:

$$\begin{aligned}
& \mathrm{Adv}^{\mathrm{DDH}}(\mathbb{B}) \\
= \;& |\mathsf{Pr}[1 \leftarrow \mathbb{B}(\mathcal{G}, u\mathcal{G}, v\mathcal{G}, uv\mathcal{G}) \mid b = 1] - \mathsf{Pr}[1 \leftarrow \mathbb{B}(\mathcal{G}, u\mathcal{G}, v\mathcal{G}, w\mathcal{G}) \mid b = 0]| \\
\overset{(1)}{=} \;& \left| \begin{array}{l} \mathsf{Pr}[x_i, r_i, s_i \in_{\mathsf{R}} \{1, \ldots, |\mathbb{G}_1|\} : \\ \qquad \ell \in_{\mathsf{R}} \{1, \ldots, m\} : \ell' \leftarrow \mathbb{A}(\mathcal{G}, M, S, I) \mid \ell = \ell' \wedge b = 1] \\ -\mathsf{Pr}[x_i, r_i, s_i \in_{\mathsf{R}} \{1, \ldots, |\mathbb{G}_1|\} : \\ \qquad \ell \in_{\mathsf{R}} \{1, \ldots, m\} : \ell' \leftarrow \mathbb{A}(\mathcal{G}, M, S, I) \mid \ell = \ell' \wedge b = 0] \end{array} \right| \\
\overset{(2)}{=} \;& |\tfrac{1}{m} + \mu - \tfrac{1}{m}| = \mu
\end{aligned}$$

Equality (1) holds as we substituted $\mathbb{B}$ by its definition. For equality (2), the left part holds by assumption, $\mathbb{A}$ succeeds with a probability non-negligibly higher than $1/m$. For the right part, however, the DDH challenge consists of 4 uniformly random values that are not correlated in any way and the best that $\mathbb{A}$ can do is to output some number k. With probability $1/m$, this number coincides with ℓ. Since μ is non-negligible, it contradicts our assumption that DDH is computationally intractable. We conclude that no such attacker $\mathbb{A}$ exists. □

Due to the very close construction of escrow identifiers and SSPs, the definition for unlinkability of escrow-identifiers across services is also close to the definition of unlinkability of SSPs across services.

Definition 2.8 (Escrow-identifier-based Unlinkability across Services)**.** *We say escrow identifiers (as constructed in Section 2.4) are unlinkable across services if and only if given a verification key* $vk := x\mathcal{G}$*, an associated escrow value* $\mathcal{R} := r \cdot vk$*, an escrow identifier* $idr_1 := r\mathcal{S}_1$ *for service* $\mathcal{S}_1$*, and an escrow identifier* idr_2 *for service* $\mathcal{S}_2 \neq \mathcal{S}_1$*, it is computationally infeasible to decide whether* $idr_2 = x\mathcal{S}_2$*, that is, to decide whether the two escrow identifiers belong to the same user or not.*

The following theorem states the unlinkability of escrow identifiers across services.

Theorem 2.6 (Unlinkability of Escrow Identifiers across Services)**.** *In the random oracle model and under the DDH assumptions, escrow identifiers (as constructed in Section 2.4) are unlinkable across services.*

Proof. Suppose $\mathbb{A}$ is an attacker that takes as input a tuple of the form $(\mathcal{D}, vk, \mathcal{R} := r \cdot vk, \mathcal{S}_1, idr_1 := r\mathcal{S}_1, \mathcal{S}_2, idr_2)$ where $\mathcal{S}_1 \neq \mathcal{S}_2$, and decides whether $idr_2 = t \cdot \mathcal{S}_2$ with a probability $1/2 + \mu$, where μ is non-negligible; $\mathbb{A}$ outputs 1 to denote that $idr_2 = r\mathcal{S}_2$ and 0 to denote that $idr_2 \neq r\mathcal{S}_2$. We use this attacker to construct attacker $\mathbb{B}$ against the decisional Diffie-Hellman assumption.

The DDH challenger randomly chooses $b \in_{\mathsf{R}} \{0, 1\}$. If $b = 1$, the challenger produces a valid DDH tuple $(\mathcal{G}, x\mathcal{G}, y\mathcal{G}, z\mathcal{G})$ for uniformly random values x and y, where $z = x \cdot y$. If $b = 0$, z is randomly chosen. We construct attacker $\mathbb{B}$ that uses $\mathbb{A}$ to solve the given DDH challenge.

First, $\mathbb{B}$ chooses $d, r \in_{\mathsf{R}} \{1, \ldots, |\mathbb{G}_1|\}$ and sets $\mathcal{D} := d\mathcal{G}$, $vk := \mathcal{G}$, $\mathcal{R} := x\mathcal{G}$, $\mathcal{S}_2 := y\mathcal{G}$, $idr_2 := z\mathcal{G}$, $\mathcal{S}_1 := r\mathcal{D}$, and $idr_1 := r \cdot d \cdot \mathcal{R}$ and calls $\mathbb{A}(\mathcal{D}, vk, \mathcal{S}_1, idr_1, \mathcal{S}_2, idr_2)$ and answers the challenge with $\mathbb{A}$'s answer. We observe that we can derive vk from $\mathcal{D}$ by setting $vk = d^{-1}\mathcal{D}$. Therefore, vk is of the correct distribution and forms an honestly-generated verification key (since d is chosen uniformly at random from a prime-order group, d^{-1} exists and is a uniformly random value), $\mathcal{S}_1 = r\mathcal{D}$ and $idr_1 = r \cdot d \cdot \mathcal{R} = r \cdot d \cdot x\mathcal{G} = rx\mathcal{D} = x\mathcal{S}_1$, that is, $\mathcal{S}_1$ and idr_1 have the correct form and distribution (since r is chosen uniformly at random) of a service and an escrow identifier. Notice that $idr_2 = x\mathcal{S}_2$ if and only if $z = x \cdot y$.

Let us now compute the success probability of our constructed adversary against the DDH challenge, where we let z denote a random value.

$$
\begin{aligned}
&\mathrm{Adv}^{\mathrm{DDH}}(\mathbb{B}) \\
&= |\mathsf{Pr}[1 \leftarrow \mathbb{B}(\mathcal{G}, x\mathcal{G}, y\mathcal{G}, xy\mathcal{G}) \mid b = 1] - \mathsf{Pr}[1 \leftarrow \mathbb{B}(\mathcal{G}, x\mathcal{G}, y\mathcal{G}, z\mathcal{G}) \mid b = 0]| \\
&\overset{(1)}{=} \left| \begin{array}{l} \mathsf{Pr}[d, r \in_{\mathsf{R}} \{1, \ldots, |\mathbb{G}_1|\} : 1 \leftarrow \mathbb{A}(d\mathcal{G}, \mathcal{G}, x\mathcal{G}, r \cdot d \cdot \mathcal{G}, r \cdot d \cdot x\mathcal{G}, y\mathcal{G}, xy\mathcal{G}) \mid b = 1] \\ -\mathsf{Pr}[d, r \in_{\mathsf{R}} \{1, \ldots, |\mathbb{G}_1|\} : 1 \leftarrow \mathbb{A}(d\mathcal{G}, \mathcal{G}, x\mathcal{G}, r \cdot d \cdot \mathcal{G}, r \cdot d \cdot x\mathcal{G}, y\mathcal{G}, z\mathcal{G}) \mid b = 0] \end{array} \right| \\
&\overset{(2)}{=} \left| \begin{array}{l} \mathsf{Pr}[d, r \in_{\mathsf{R}} \{1, \ldots, |\mathbb{G}_1|\} : 1 \leftarrow \mathbb{A}(d\mathcal{G}, \mathcal{G}, x\mathcal{G}, r \cdot d \cdot \mathcal{G}, r \cdot d \cdot x\mathcal{G}, y\mathcal{G}, xy\mathcal{G}) \mid b = 1] \\ -(1 - \mathsf{Pr}[d, r \in_{\mathsf{R}} \{1, \ldots, |\mathbb{G}_1|\} : 0 \leftarrow \mathbb{A}(d\mathcal{G}, \mathcal{G}, x\mathcal{G}, r \cdot d \cdot \mathcal{G}, r \cdot d \cdot x\mathcal{G}, y\mathcal{G}, z\mathcal{G}) \mid b = 0]) \end{array} \right| \\
&\overset{(3)}{=} |\tfrac{1}{2} + \mu - (1 - (\tfrac{1}{2} + \mu))| = 2\mu
\end{aligned}
$$

In step (1), we substitute attacker $\mathbb{B}$ with its definition; in step (2), we consider that if $\mathbb{A}$ decides if $z = xy$, then $\mathbb{A}$ also decides if $z \neq xy$. In step (3), we substitute $\mathbb{A}$ with its success probability. Thus, attacker $\mathbb{B}$ can use $\mathbb{A}$ to break the decisional Diffie-Hellman challenge with non-negligible probability, which violates our assumption. We conclude that no attacker $\mathbb{A}$ exists. □

2.5.2. Type-Based Verification of the API Methods

This section formally proves that the cryptographic implementation enforces the authorization policies specified by the programmer. This is of paramount importance in our setting to make sure that the malleability of zero-knowledge proofs does not constitute an attack surface. Intuitively, we aim at showing that whenever a principal successfully verifies a validity proof for formula F, then F holds true. First, we formally define what it means for a logical formula to hold true (Section 2.5.2.1). We then show how to symbolically encode the semantics of malleable zero-knowledge proofs (Section 2.5.2.2). This encoding allows us to leverage F7 [49], a state-of-the-art type checker for verifying security properties on the source code of cryptographic implementations (Section 2.5.2.3). We obtain security by construction guarantees (Section 2.5.2.4): using the API suffices to enforce the desired authorization policies.

2.5.2.1. Authorization Policies

Following a well-established methodology for the specification and static analysis of authorization policies in a distributed setting, we decorate the code with assumptions and assertions [49]: *assumptions* introduce logical formulas which are assumed to hold at a given point, while *assertions* specify logical formulas which are expected to follow from the previously introduced (active) assumptions.

Authorization policies (for instance, Equation 2.1) are explicitly assumed in the system. Furthermore, we place an assumption within the implementation of the mkSays method, reflecting the intention of the user to introduce a new logical formula in the system: for instance, let us recall Example 2.2 from Section 2.3.1:

```
  ...
4 if verify $y$ $\left(\begin{array}{l} x_{Hosp} \text{ says } \mathsf{IsDoc}(x_{Doc}, x_{PI_{Doc}}) \\ \wedge\ x_{Doc} \text{ says } \mathsf{Visit}(x_{Pat}, x_{date}, x_{results}) \end{array}\right)$ then
5   let $(pf_{\mathsf{IsDoc}}, pf_{\mathsf{Visit}})$ = split$_\wedge$ $y$;
6   let $pf_s$ = mkSays $y_{Pat}$ Rating$(x_{opinion})$;
  ...
```

There, executing the mkSays method on line 6 introduces an assumption of the form

$$\mathsf{assume}\ y_{Pat}\ \mathsf{says}\ \mathsf{Eval}(x_{opinion}).$$

Finally, assertions are placed immediately after each call to the verify method: for instance, the call to the verify method on line 4 of Example 2.2 is followed by

$$\mathsf{assert}\ \left(\begin{array}{l} x_{Hosp}\ \mathsf{says}\ \mathsf{IsDoc}(x_{Doc}, x_{PI_{Doc}}) \\ \wedge\ x_{Doc}\ \mathsf{says}\ \mathsf{Visit}(x_{Pat}, x_{date}, x_{results}) \end{array}\right)$$

2.5.2.2. Symbolic Cryptography

As usual in the static analysis of cryptographic protocol implementations, we rely on a symbolic abstraction of cryptographic primitives that captures their ideal behavior.

Prior work showed how standard cryptographic primitives such as encryptions and signatures [49] as well as non-malleable zero-knowledge proofs [27] can be faithfully modeled using a sealing-based technique [148]. The advantage of sealing-based techniques is that they are purely based on standard language constructs. Sealing-based abstractions are well-suited for verification purposes because they do not require any extension or modification of the programming language.

In a nutshell, a seal comprises two functions: (*i*) a sealing function that takes as input a message, stores this message in a secret list along with a fresh handle, and returns this handle; (*ii*) an unsealing function that takes as input a handle, scans the secret list in search for the associated message, and returns that message. The fundamental insight is that the only way to extract a sealed value is via the unsealing function. Sealing-based abstractions of encryptions and signatures have been proven computationally sound [33], that is, security results verified on these abstractions carry over to the actual cryptographic implementation.

Previous sealing-based abstractions for non-malleable zero-knowledge proofs [27] use one seal per proven statement. The sealing and unsealing functions can only be accessed by the functions to create and verify proofs. Since the number of proven statements in a protocol is finite in the non-malleable setting, the number of seals is finite as well. In a malleable setting, however, this approach yields an unbounded number of seals since proofs can be arbitrarily combined. We therefore devise a finite sealing-based library for malleable zero-knowledge proofs: We model malleable zero-knowledge proofs using one seal for the proofs themselves and one seal to model the commitments used inside zero-knowledge proofs.

The seal for zero-knowledge proofs stores the proven statement and a random value; the random value corresponds to the randomness used in the generation of the computational zero-knowledge proof and the fresh handle corresponds to the zero-knowledge proof. The sealing and unsealing functions are only used inside the functions to create and verify zero-knowledge proofs, respectively.

The seal for commitments stores in its secret list the committed values and the randomness used in the commitment; the fresh handle corresponds to the commitment. Only the sealing function to compute commitments is public.

The proof creation function takes the formula to be proven as input (say, $a \leq b$), creates the commitments (C_a and C_b), and passes the zero-knowledge statement ($[\![C_a]\!] \leq [\![C_b]\!]$) to the sealing function, which outputs the zero-knowledge proof. The verification function takes as input the proof along with the zero-knowledge statement, internally opens the commitments, and executes the zero-knowledge statement on the witnesses to check its validity. The functions to manipulate zero-knowledge proofs (for instance, splitting of logical conjunctions) are straightforwardly implemented by using the sealing and unsealing functions for zero-knowledge proofs and for commitments.

2.5.2.3. Typed Interface

Type systems (and static analysis techniques in general [52, 24]), proved successful in the certification of formal security guarantees for cryptographic protocols [52, 64, 127, 63, 22, 65, 21, 66, 112, 61] and implementations thereof [49, 104, 62]. We type-check the symbolic implementation of our API using F7 [49], a type-checker for authorization policies. This type-checker works on RCF, a refined and concurrent λ-calculus that can be used to reason about a large fragment of ML and of Java by encoding. In F7, the universal type *unit* describes values without a security import, that is, values of type *unit* can be passed to and received from the attacker. All types used in the API interface (see Table 2.1) are encoded as *unit*. Signing keys are the only confidential data but, as previously discussed, they are not exported by the API. The interface types coincide with the ones shown in Table 2.1, except for the type of the verify method. This method is given a refinement type of the form:

$$\begin{aligned}&\mathsf{verify}_F : \mathit{proof} \to y : \mathit{formula} \to \\ &\qquad \{z : \mathit{bool} \mid \forall \widetilde{x}.\ y = \underline{F} \wedge z = \mathsf{true} \implies F\}\end{aligned}$$

Intuitively, a value v has type $\{x : T \mid F\}$ if v has type T and, additionally, the logical formula $F\{v/x\}$ (i.e., F where every occurrence of x is replaced by v) is entailed by the active assumptions. The type of verify ensures that if the returned value z is true and the formula y passed as input is the ML encoding $\underline{F}$ (defined in Appendix A.1.2) of the logical formula F, then the formula F is entailed at run-time by the currently active assumptions. In other words, malleable zero-knowledge proofs constitute a sound implementation of our logic-based data processing API.

Well-formedness of formulas. It is interesting to observe that not all proofs are meaningful. For instance, suppose that a principal receives a proof of the following formula:

$$\exists x.\ x \ \mathsf{says}\ \mathsf{Eval}(ev, course) \tag{2.4}$$

We would be tempted to let this principal entail $\exists x.\ x\ \mathsf{says}\ \mathsf{Eval}(ev, course)$. The proof for this formula, however, does not reveal the identity of the person issuing the statement, nor is there any information about the origin of the creator of the proof. In fact, this proof might have been constructed by an attacker, using a fresh key-pair and, therefore, the formula $\exists x.\ x\ \mathsf{says}\ \mathsf{Eval}(ev, course)$ is not necessarily entailed by the formulas proved by principals of the system. Notice that we assume that the principals of the system are honest, i.e., they issue signatures to witness the validity of the corresponding logical predicates. We cannot, of course, assume the same for the attacker.

We stipulate that principals only use public identifiers that belong to principals of the system (as opposed to the attacker identity) in their formulas. We call these identifiers *trustworthy*. Checking whether an identifier that occurs in a zero-knowledge proof is trustworthy is subtle. The idea is that an identifier is considered trustworthy if either it is revealed by the proof and known to belong to a principal of the system, or, recursively, it is endorsed by a trustworthy public identifier. For instance, the formula depicted in Equation 2.4 does not guarantee that the existentially quantified public identifier x is trustworthy. Conversely, let us recall Example 2.4 from Section 2.3.3:

...

$$\texttt{let}\ \ pf' \ \texttt{=}\ \mathsf{hide}\ \ pf \begin{pmatrix} \exists w_{Pat}, w_{results}, w_{date}. & \\ \quad x_{Doc}\ \ \mathsf{says}\ \ \mathsf{Visit}(w_{Pat}, w_{date}, w_{results}) & \\ \quad \land\ \ w_{Pat}\ \ \mathsf{says}\ \ \mathsf{Rating}(x_{opinion}) & \end{pmatrix}\ \texttt{;}$$

...

There, the identifier of the patient is endorsed by the doctor and, therefore, is trustworthy. Hence, the proof pf' justifies the corresponding formula.

In a nutshell, a formula is well-formed if it ensures that all private identifiers are trustworthy. Despite the simplicity of this intuition, the formal definition has to consider a number of complications, including the presence of logical disjunctions in the statement. For instance, the statement

$$\exists x_1.\ x_1\ \mathsf{says}\ F\ \lor\ Doc\ \mathsf{says}\ F$$

is not well-formed, since we do not know which of the two disjuncts holds true. The idea is to transform a statement in disjunctive form and then to check that all identifiers in each sequence of conjunctions are registered.

We formalize the notion of trustworthiness for keys in Appendix A.2. Here and throughout the rest of the chapter, we stipulate that all formulas are well-formed.

2.5.2.4. Soundness Result

The soundness result shows security by construction for the API methods. In other words, the programmer does not have to type-check her code. Instead, the usage of our API suffices to yield security guarantees by construction.

We first formalize the notion of safety for authorization policies and then proceed to the soundness results. The complete source code along with the proofs and elaborate explanations are given in Appendix A.

Intuitively, safety states that assertions never fail at run-time, even in the presence of an opponent.

Definition 2.9 (Safety, Opponent, and Robust Safety [49]). *A program P is safe if and only if, in all executions of P, all assertions are entailed by the current assumptions.*

A program B is an opponent if and only if B contains no assertions and the only type occurring in B is unit.

A program P is robustly safe if and only if the application $B\ P$ is safe for all opponents B.

The type system establishes judgments of the form $E \vdash P : T$ for some typing environment E, program P, and type T. Intuitively, E tracks the types of variables in scope. The following theorem states that well-typed programs are robustly safe.

Theorem 2.7 (Safety by Typing [49]). *If $\emptyset \vdash P : T$, then P is safe. If $\emptyset \vdash P : unit$, then P is robustly safe.*

Finally, the soundness theorem states that well-typed programs that are linked to the typed API implementation are robustly safe. This definition uses the typed API interface E_{API} (see Table A.4) and the typed API implementation P_{API} (see Appendix A).

Theorem 2.8 (Soundness). *If $E_{\mathsf{API}} \vdash P : unit$, then $P_{\mathsf{API}}; P$ is robustly safe.*

Notice that Theorem 2.8 only applies to well-typed programs P. In the following, we formally demonstrate that the soundness result depends only on the well-typing of the API implementation and not on the program using the API.

This result is based on the opponent typability lemma [49]. Opponent typability captures our intuition that programs that only use values of type *unit* and do not use assertions are safe. In the API, the only values whose type is different from *unit* is the verification function. Therefore, we construct a verification wrapper function $\mathsf{verify}'_{\underline{F}}$. This function calls the corresponding verification function $\mathsf{verify}_{\underline{F}}$. If the result is true, the wrapper function immediately executes the corresponding assertion $\mathsf{assert}\ F$. We state the wrapper function in Appendix A.3.

We proved that $E_{\mathsf{API}} \vdash \mathsf{verify}'_F : \mathit{unit}$. Furthermore, as previously mentioned, all types except for verify_F in E_{API} are encoded as *unit*. We can then define a variant of our typed API, named E'_{API}, in which the verify_F method is replaced by verify'_F. We call P'_{API} the corresponding implementation. E'_{API} is refinement-free and only exports *unit* types, which allows us to prove the following theorem. Intuitively, this theorem states that user programs linked to P'_{API} are robustly safe.

Theorem 2.9 (Security by Construction)**.** *Let P be an assertion-free program such that unit is the only type occurring therein and the free names and free variables $\mathit{fnfv}(P) \subseteq E'_{\mathsf{API}}$ (that is, P only uses functions exported by E'_{API}). Then $P'_{\mathsf{API}}; P$ is robustly safe.*

2.6. Implementation and Experiments

We implemented the API methods as a Java library on top of the the jPBC library [94], the PBC library [168], and the peloba ZK Library [20]. The jPBC library implements elliptic curve operations and the corresponding bilinear map. It supports a pure Java implementation as well as an implementation that uses the native PBC library to speedup the mathematical computations. The peloba ZK Library works on top of the jPBC library and implements the equations provable with the Groth-Sahai proof system (see Section 2.4).

Our implementation uses the peloba ZK Library with the jPBC library running the native PBC library for better performance. Furthermore, it exploits the multi-core architecture of modern CPUs: to get a good trade-off between implementation work and efficiency advantages, the multi-threading is situated at the level of Groth-Sahai equations. Thus, we compute Groth-Sahai equations in parallel but we do not multi-thread the individual mathematical operations. The library is freely available [187].

Implementing an efficient and secure re-randomization mechanism. Groth-Sahai proofs allow for re-randomizing commitments contained in a proof and the proof itself [42], that is, we can change and adapt both the randomness contained in the commitments and in the proof. Re-randomization is crucial, for instance, to enforce unlinkability and anonymity properties. Since re-randomization operations are computationally expensive, we tried to reduce the number of these operations without sacrificing security.

In general, there are two approaches to implement API methods: a transaction-like system and an immediate-effect system. In a transaction-like system, a user starts a transaction and begins queuing proof manipulation steps, for instance, adding a pseudonym and hiding an identity. When she is finished, she closes the transaction and the API executes the necessary API methods including the re-randomization steps. In an immediate-effect system, every modification to a proof is executed immediately, even if the resulting proof is only intermediate and will be modified further.

We implemented the immediate-effect system. This approach spares users the necessity to open and close transactions, and it yields access to intermediate proofs. In particular, these proofs are useful in an open-ended setting, where they can be used as cryptographic material in a different protocol. More precisely, we use the following strategy when dealing with re-randomization: when computing the logical conjunction of two proofs, the API only re-randomizes commitments so that all equal, non-hidden values are represented by one single commitment; when hiding parts of a proof, the API completely re-randomizes the resulting proof. This prevents users from accidentally revealing their identity: original proof and proofs derived by hiding operations are always unlinkable.

Cryptographic setup. The implementation is parametric in the security parameter and the number of threads to use. We use MNT elliptic curves with security parameter $\eta = 112\,\text{bits}$, $\eta = 128\,\text{bits}$, and $\eta = 256\,\text{bits}$; NIST recommendations [208] deem 112 bit security parameters secure until the year 2030, larger key sizes are expected to hold even longer. To derive these security parameters, let the group sizes of $\mathbb{G}_1$ and $\mathbb{G}_2$ be approximately $2^{2\eta}$. With MNT curves, this corresponds to $\eta = 112$. To achieve a security parameter beyond that, different curve types are preferable over MNT curves [115].

Since we are not aware of an off-the-shelf implementation of these curves, we use MNT curves to estimate the running times of higher security parameters. We stress, however, that the times obtained for $\eta = 128$ and $\eta = 256$ are to be understood as lower bounds.

2.6.1. Experimental Evaluation

We conduct an experimental evaluation of the Java implementation to demonstrate the feasibility of our approach. We measure the proof generation time, proof verification time, and the proof size for the concrete implementation of SSP proofs, list membership and non-membership proofs, and the identity escrow protocol. For the list membership and non-membership proofs, we fix the total number of list elements to 1000, which we distribute over various amounts of lists. We evaluate the identity escrow protocol for various security parameters and we determine how many escrow identifiers an escrow agent can check per second. We report the average results of 1000 runs on a computer with an Intel Xeon E5645 six core processor with 2.4 GHz and hyper-threading (i.e., the processor can handle 12 threads at the same time), and 4 GB of RAM. Notice that although this computer has great multi-tasking capabilities, the proofs tested here are too small to show the potential of parallel computation. Anticipating the experimental evaluation of the case studies in Chapter 3, we show the impact of the multi-threaded application. Also, the case studies highlight the impact of re-randomizing parts of the proof.

Discussion. In the following, all quantities refer to a 112 bit security parameter.

Figure 2.5 depicts the results for SSP proofs. Proof generation as well as proof verification are highly efficient and take 76 ms and 67 ms, respectively. The proof size is 2.6 KB.

Figure 2.6 shows that the list non-membership proof is practical, even for long lists. We vary the number of lists since users have to recompute their pseudonym in zero-knowledge for every list. The number of lists, however, plays only a small role as the proof is dominated

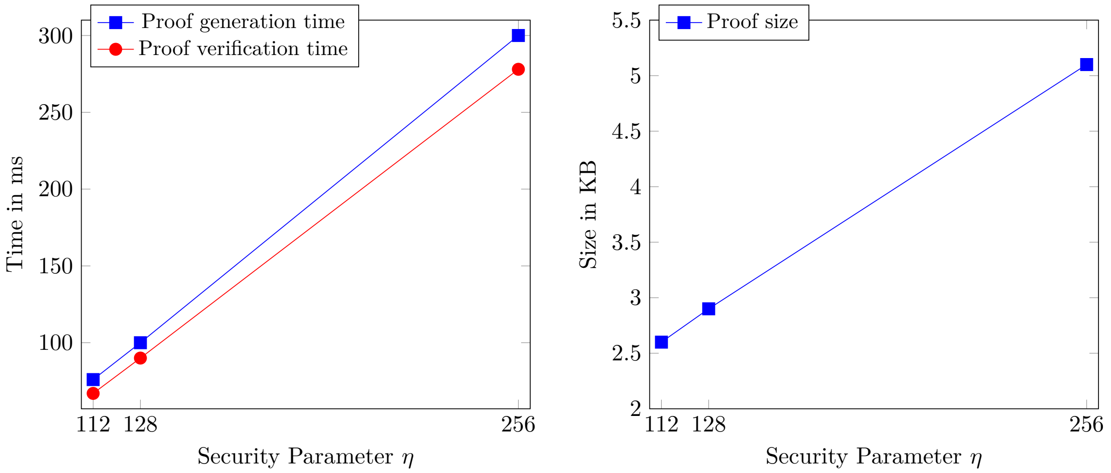


Figure 2.5.: The results for the computation of a service-specific pseudonym.

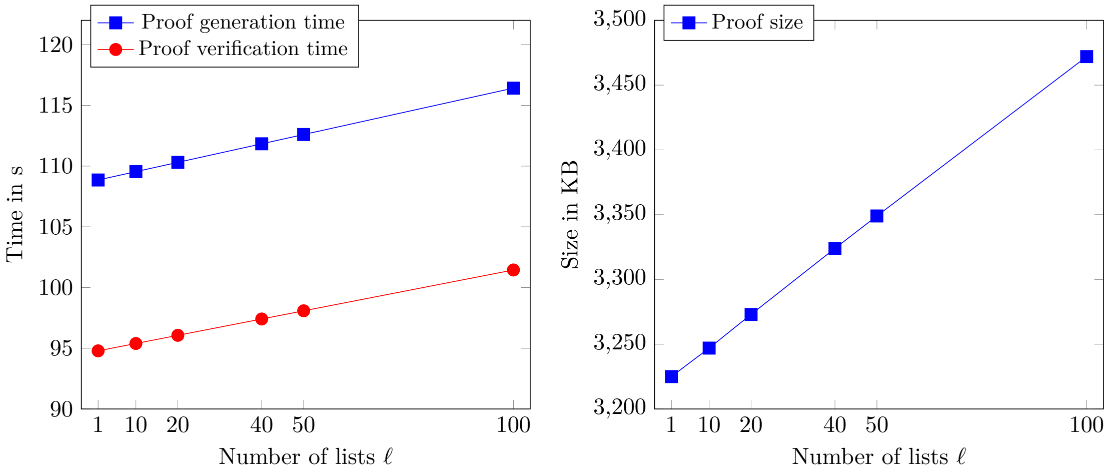


Figure 2.6.: The results for the non-membership proof for ℓ lists, a total number of 1000 elements distributed over the lists, and a security parameter of $\eta = 112$ bits.

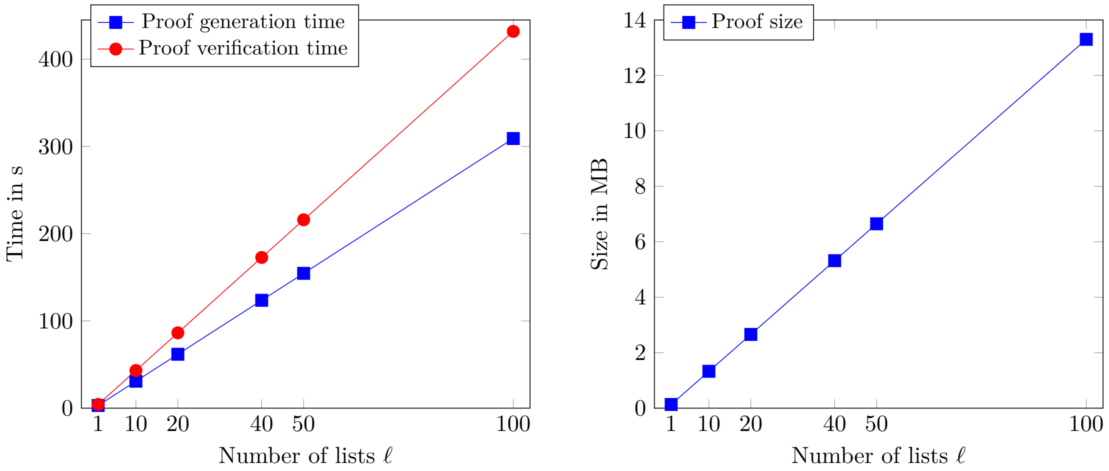


Figure 2.7.: The results for the membership proof for ℓ lists, a total number of 1000 elements distributed over the lists, and a security parameter of $\eta = 112$ bits.

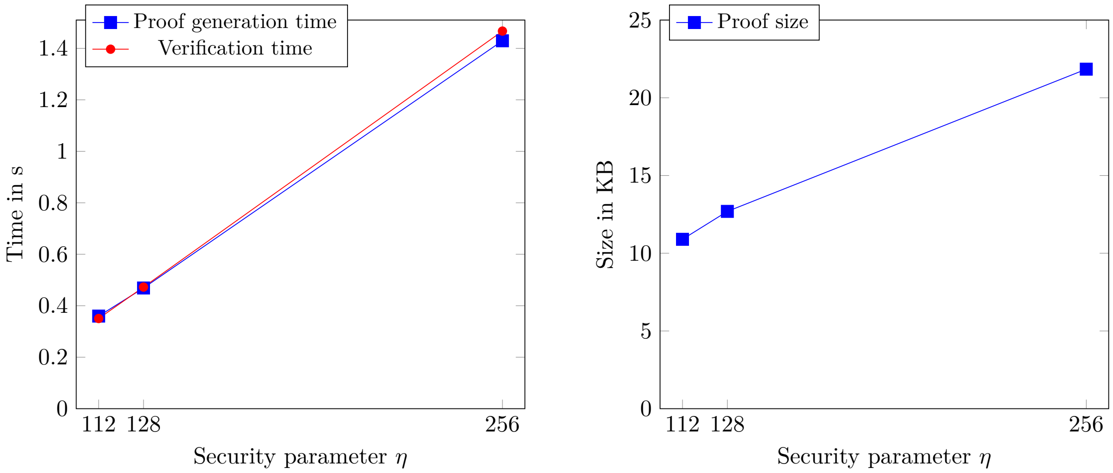


Figure 2.8.: The results for the identity escrow protocol.

by the computations for the list elements: the proof for one list with 1, 000 elements takes 109 s and the proof for 100 lists with a total of 1, 000 elements takes 116.5 s. The proof size varies between 3.2 MB for 1 list and 3.4 MB for 100 lists.

Figure 2.7 presents the results for the list membership proof. As expected, the proof for a single list is very efficient as it is independent of the size of the list. Creating a proof for many lists, however, is more expensive, since signatures on message tuples are computationally burdensome. The proof for one list and 1, 000 elements takes 3 s and the proof for 100 lists with a total of 1, 000 elements takes 309.1 s. The proof size varies between 133 KB for 1 list and 13.3 MB for 100 lists. We believe that these numbers do not undermine the practicality of our approach: typical users only participate in a small number of services and therefore are only confronted with a small number of list membership proofs.

As shown in Figure 2.8, the identity escrow proof constitutes only a small computational burden for the prover and the verifier: the proof takes 360 ms to generate, requires 350 ms to verify, and is 10.8 KB in size. The computation of the *EA* consists only of scalar multiplications and equality tests. These are extremely efficient and the *EA* can perform more than 10, 000 of the necessary computations per second on a single core. Furthermore, these computations can be conducted in parallel.

In many proof systems, generating a proof takes longer than verifying a proof (e.g. [28, 25]). The Groth-Sahai setting is a notable difference to this rule of thumb. The technical reason is that the proof generation relies on computations in $\mathbb{G}_1$ and $\mathbb{G}_2$, the proof verification relies mostly on the bilinear map that maps elements into an extension field. Since $\mathbb{G}_1$ and $\mathbb{G}_2$ are six times smaller than the extension field (in the case of our MNT curves), the proof computation is faster than the proof verification. Nonetheless, the experiments show that the proof computation often takes longer than the verification. The reason is that computing a proof consists of many steps, for instance, the computation of digital signatures, the actual computation of the zero-knowledge proofs, and re-randomization steps. As a consequence, proof generation is only faster if the number of these steps is small. This is the case for the list membership proof (see Figure 2.7), where the proof computation only involves a signature computation and its transformation into a zero-knowledge proof.

2.7. Related Work

We discuss work related to the declarative API and its cryptographic realization. We survey various alternative cryptographic realizations and detail their advantages and disadvantages. Finally, we relate schemes and systems including anonymous credentials and declarative languages to the API.

Σ-protocols. Σ-protocols are *interactive* protocols [93] that are highly efficient and, when properly used, have many desirable properties [91]. A Σ-protocol is an atomic zero-knowledge proof, for instance, for proving knowledge of a discrete logarithm [195] and proving the equality of two discrete logarithms [73]. Larger Σ-protocols are built on top of these atomic protocols. The most notable properties of Σ-protocols are their efficiency and the range of provable statements [72, 73, 25, 28]. In particular, virtually all of these protocols can be arbitrarily combined to prove their logical conjunctions and logical disjunctions [91].

Despite these properties, Σ-protocols suffer from shortcomings with respect to flexibility. First of all, Σ-protocols are interactive proofs that cannot be used as a credential. Intuitively, for a proof to be convincing, the verifier must contribute randomness. Consequently, a forwarded proof transcript is not convincing since the verifier did not contribute randomness to the proof.

Σ-protocol based zero-knowledge proofs can be made non-interactive by applying the Fiat-Shamir heuristic [111]. This heuristic relies on the random oracle model. A random oracle is a function that, upon a query, outputs true randomness and answers queries consistently with previous answers. The random oracle is used to derive the random input expected from the verifier. Since also the prover can obtain true randomness without interacting with the verifier, the proof becomes non-interactive. This technique is highly efficient, but it has one far-reaching consequence:

Using a hash function to substitute the random input from the verifier yields non-malleable proofs, i.e., proofs cannot be changed in any way. Consequently, it is impossible to selectively hide parts of the statement or to apply re-randomization: this would change the input to the hash function and in turn require re-computation of a given proof. In particular, re-computation requires knowledge of all witnesses, which renders the Fiat-Shamir heuristic impractical for our scenario. As elaborated in Section 2.4, the Groth-Sahai proof system overcomes these shortcomings and supports the selective hiding of parts of the statement and the re-randomization of any commitment as well as the zero-knowledge proof itself, even without any knowledge of the involved witnesses.

Group signatures and ring signatures. Group signatures (e.g., [17, 54, 85, 152]) allow a member of a group to sign a message on behalf of the group. They often rely on a group manager to distribute keys and, in case of a dispute, reveal the signer of a message. The presence of a trusted third party that can reveal the identity of a signer is the most noticeable difference from our approach, where a third party (the escrow agent) is only needed to reveal the identity of misbehaving users. Furthermore, group signatures tend to increase the key management overhead since every group requires a different set of keys, whereas in our system, each user needs just one key-pair.

Ring signatures (e.g., [193, 58, 218]) allow a user to sign a message m on behalf of a set of users: the user gathers all the verification keys of the users in the set (including her own key) and uses the ring signature to create a signature on m [193]. The verifier of this signature will only learn the members of the set and that a member of the set created the signature but not which one. Consequently, the use of ring signatures requires that it is public knowledge which principal uses which service. Otherwise, the set of principals for one particular service cannot be assembled. This poses serious privacy issues and even prohibits the specification of certain systems: for instance, in decentralized social networks, users want to hide their friend list (see Section 3.4) which is not possible using ring signatures. Users take the role of service providers and their friends constitute the set of users using that service, which must be public to construct a ring signature.

Identity-based and attribute-based signatures. The advent of elliptic curve cryptography and bilinear maps has given rise to identity-based cryptography [220, 225, 135] and attribute-based cryptography [163, 4]. In identity-based cryptography, the identity of a user (e.g., the name of the user) is used as a public key. In attribute-based signatures, a signature does not yield the identity of the signer but that a certain attribute (e.g., "is-a-doctor") is allotted to the signer.

For our application scenario, it is crucial that verification keys can be signed and that signatures and, in particular, signatures on verification keys are efficiently provable in zero-knowledge. We are not aware of any automorphic signature scheme based on identity-based or attribute-based cryptography.

Pseudonyms. Chaum [82] initiated the research on pseudonyms and since then many schemes have been introduced (e.g., [194, 167, 28, 173, 43, 12, 68, 221, 56]). Many schemes do not consider the notion of service (e.g., [167, 28, 43, 194]), they incorporate a compulsory trusted third party (e.g., [68, 221]), they do not enforce the uniqueness property (e.g., [167, 28, 43]), or they do not support any form of authorization policy unless the pseudonym owner is fully disclosed (e.g., [194]). In the following, we discuss four recent pseudonym schemes closely related to our concept of service-specific pseudonyms.

Martucci et al. [173] use a TTP only to register the real identity. After the registration, users generate pseudonyms on their own using a non-interactive publicly verifiable variant of a special signature scheme and then self-certify them by means of anonymous credentials and group signatures. A pseudonym is unique within a given context and a user is linkable for actions performed within this context. The compulsory presence of a trusted third party is a fundamental difference from the pseudonym system considered in this thesis: here, the presence of a TTP is optional and only needed to reveal the identity of misbehaving users.

Brands et al. [56] use a central authority to register users in a system: they receive a fixed number of pseudonyms that are used to register with a service provider, one pseudonym for every available service. Should a user misbehave, she can be completely revoked from the system but identity escrow is not possible. The central authority, the fixed number of services, and the absence of an identity escrow protocol significantly differentiate their work from ours.

Service-specific pseudonyms coincide with the concept of domain pseudonyms from Identity Mixer cryptographic library (idemix) [141], scope-exclusive pseudonyms from the attributed-based credentials for trust project (ABC4Trust) [4], and pseudonyms used in U-Prove [101]. The flexibility of our cryptographic setup based on Groth-Sahai proofs and the identity escrow protocol are the most prominent differences.

Accumulators. Accumulators store an arbitrary number of values and are generally equipped with efficient membership and non-membership proofs, that is, proofs of whether a value is stored in an accumulator or not. While accumulators seem to be ideal for implementing reputation lists, incorporating them into the existing framework requires encoding pseudonyms into a special form that is compatible with the accumulator. Proving this encoding in zero-knowledge, however, makes the overall protocol very inefficient, outweighing the gains of accumulators over reputation lists.

For instance, there exist accumulators for numbers in $\mathbb{Z}_n$ (e.g., [212]). The Groth-Sahai scheme only supports quadratic equations and proving the computation of an SSP S^x in zero-knowledge takes time linear in the security parameter since we would have to resort to the square-and-multiply algorithm. The resulting computational overhead each time a user uses a pseudonym is significant. Other schemes that work on elliptic curves directly (e.g., [71, 77]) require a symmetric bilinear map, or they work in an RSA-like system (e.g., [77]), requiring composite-order groups. These requirements are incompatible with automorphic signatures.

Set (non-)membership proofs. Set membership and set non-membership proofs are efficient protocols to convince a verifier that a prover is in possession of a value that is contained in a set or that is not contained in a set without revealing the value to the verifier [70, 78, 180, 153]. Our protocols are trivial set membership and set non-membership protocols that fit our cryptographic setting because they rely on digital signatures and zero-knowledge proofs only. Naturally, it is possible to adapt the cryptographic setting to incorporate more sophisticated and efficient set (non-) membership proofs.

Anonymous credential systems. We compare our work to the line of research on anonymous credential systems that support anonymous and delegatable authentication. All the following protocols, apart from the scheme by Belenkiy et al. [42], rely on Σ-protocols and, as a consequence lack the flexibility to selectively hide individual parts of the proven statement. This limitation is prohibitive for the design of open-ended systems. For instance, the protocol in Example 2.7 cannot be implemented using Σ-protocols, since it requires the hiding of the doctor's identity and parts of the signed message from a given proof. Our work instead relies on the Groth-Sahai zero-knowledge proof system that is flexible, general, and, in particular, efficient enough to selectively hide and reveal any given part of the proven statement. Furthermore, our work supports an optional TTP-based identity-escrow functionality, which is offered by neither of the systems mentioned below.

The direct anonymous attestation (DAA) protocol [59] offers a pseudonymous-attestation functionality, which allows users to authenticate their trusted platform module (TPM) with a service provider using a pseudonym, derived from the TPM's secret value (chosen by an external party) and a base value chosen by the resource provider, yielding the notion of service. The TPM's secret value is signed by a third party, called the issuer. In this work, we do not require trusted hardware and a trusted third party is only needed if identity escrow is desired.

From the recently-proposed Nymble systems (e.g., [145, 60, 211, 18, 212, 134, 164]), BLACR [18] is the most expressive and efficient. In BLACR, users generate fresh private keys that get authenticated by a group manager. Users use their keys to generate tickets that are revealed to service providers. Service providers can blacklist or whitelist these tickets and assign scores to them. Users traverse all lists, adding up the scores in zero-knowledge and revealing the final result to the service provider who can use this result to allow or deny access. The complexity of such proofs is linear in the size and number of lists. For monotonically increasing lists, the user can ask the service provider for a token certifying her reputation for the current list, allowing the user to prove her reputation only for the subsequent part of the list for future requests. In our framework, every membership proof is independent of the list size (see Section 2.4) and our construction is fully distributed, does not involve any group manager, and supports a much larger class of authorization policies, which may depend on (possibly anonymous) certificates released by any party of the system.

The delegatable anonymous credential scheme by Belenkiy et al. [42] is based on the Groth-Sahai proof system. There, a root authority issues anonymous credentials that can further be delegated. Delegatable credentials indicate the root authority and they reveal how often they have been delegated. For instance, in Example 2.1, the doctor has a level-1 credential and the patient has a level-2 credential, both rooted at the hospital. Although based on Groth-Sahai proofs, their scheme is not open-ended because the root is unalterably anchored in every credential and proofs originating from different root authorities cannot be combined. Additionally, it is not possible to change the root authority without re-issuing all delegated credentials, e.g., when the doctor switches to another hospital.

Security-oriented, declarative languages. The seminal works by Abadi et al. [161, 2] on access control in distributed systems paved the way for the development of a number of authorization logics and languages [41, 144, 118, 79, 215], which all rely on digital signatures to implement logical formulas based on the says modality. Maffei and Pecina extended this line of research with the concept of privacy-aware proof carrying authorization [170], showing how to cryptographically realize existential quantification by zero-knowledge proofs.

Building on that work, Backes et al. [30] have devised a framework for automatically deriving cryptographic implementations from a logic-based declarative specification language derived from evidential DKAL [9]. In their work, the programmer has to supply a logical derivation that is compiled piece by piece into executable code. In our framework, the high-level declarative API is directly embedded into the programming language, which allows programmers to devise systems without switching to an external logic-based language and to conveniently access the data exchanged in the protocol. Furthermore, besides authorization and privacy, our framework supports controlled linkability, accountability, and identity escrow.

G2C [31] is a goal-driven specification language for distributed applications capable of expressing secrecy, access control, and anonymity properties. These properties are enforced using broadcast encryption schemes and group signatures and the cryptographic details are automatically generated by a compiler. This compiler generates cryptographic protocol descriptions as opposed to executable implementations. Furthermore, the protocols are not open-ended and extending them often requires the re-generation of the whole system from scratch.

Encryption schemes. Our framework supports security as well as privacy properties. In the framework, messages are created in the form of zero-knowledge proofs and we preserve the privacy of sensitive information by selectively hiding parts of the message inside the secret witnesses of a zero-knowledge proof. The resulting proof can be used as a credential to gain access to resources or to log-in to a system. Naturally, the question arises how to preserve the privacy of those credentials themselves.

The necessary encryption mechanism is intentionally not part of the API methods. Intuitively, the reason is that a public-key encryption does not yield any useful logical information since everybody can create ciphertexts. Since the encryption is decoupled from the API, we can use any off-the-shelf, state-of-the-art encryption scheme to protect the privacy of zero-knowledge proofs.

3. Case Studies

The results presented in this chapter build on the following works:

- Backes, Lorenz, Maffei, and Pecina [25]: “Anonymous Webs of Trust”
- Backes, Lorenz, Maffei, and Pecina [26]: “Brief Announcement: Anonymity and Trust in Distributed Systems”
- Backes, Maffei, and Pecina [28]: “A Security API for Distributed Social Networks”
- Backes, Maffei, and Pecina [29]: “Brief Announcement: Securing Social Networks”

We conducted three case studies to show the expressiveness and applicability of the declarative API (introduced in Chapter 2) to real-world applications.

The first case study demonstrates the usefulness of the API: we let a programmer without prior experience in the declarative, logic-based system design devise and implement tales, an anonymous lecture evaluation system. In the second and the third case study, we re-design, re-implement, and improve previous work on anonymous webs of trust [25] and distributed social networks [28].

The remarkable result is that the API enables us to easily and conveniently implement the systems cryptographically within only a few days. For the two previous works on anonymous webs of trust and distributed social networks, a significant effort was necessary to develop a cryptographic realization in the first place.

3.1. Experimental Setup

Since the experimental setup for all three case studies is identical, we describe the setup here; the case studies solely report the results for the pure computation time[1] followed by a discussion. The experiments all use the high-level library as described in Section 2.6. We used a computer that is equipped with 8 CPUs[2] with hyper threading clocked at 2.40 GHz and 32 GB of RAM. We report the average of 100 runs for a security parameter $\eta = 112$, $\eta = 128$, and $\eta = 256$ bits. To stress the scalability of the implementation on modern computer architectures, we explicitly highlight the number of threads used during the proof generation and verification. To test the impact of the CPU clock speed on the computation, we also conducted several experiments with 8 threads on a computer with 32 CPUs[3] with

[1] This time excludes pre-computation steps such as drawing random values.

[2] Intel Xeon CPU E5-2665

[3] Intel Xeon CPU E5-4650L

hyper threading that are clocked at 2.6 GHz and equipped with 768 GB of RAM. The number of threads used by the implementation is restricted programatically.

3.2. tales

To determine how easy it is to use the API, we developed tales,[4] an anonymous lecture evaluation system. In tales, professors can register students to a lecture and registered students may hand in lecture evaluations anonymously. Naturally, evaluations and students must not be linkable but students are only allowed to submit at most one evaluation.

We provided a programmer[5] with the API and only briefly explained the available methods and their purpose. The API turned out to be widely self-explanatory; we received only a few questions, mostly to confirm that all necessary parts of a message were hidden in order to achieve the privacy and anonymity properties expected of the lecture evaluation system. Devising and implementing tales took approximately two weeks. Out of these two weeks, roughly two days were effectively spent on implementing the core functionality. The remainder of the time was used to design and to internationalize the user interface. tales is freely available online for downloading and testing [165].

3.2.1. Design of tales

Since the requirements are straightforward and we already described in Section 2.3 how to achieve these properties, we only briefly discuss the design of tales.

Lecture registration. The professor *Prof* registers a student *Stud* for her lecture *Security* by issuing a proof for the formula Prof says $\mathsf{Registered}(\mathit{Stud}, \mathit{Security})$. The professor forwards this proof to the student, who in turn can create her evaluation.

Evaluating a lecture. The student *Stud* receives the professor's registration proof. She appends a proof for the formula Stud says $\mathit{evaluate}(\mathit{Security}, \mathit{fb})$ to give her feedback *fb* to the professor. Additionally, she creates a service-specific pseudonym $\mathsf{SSP}(\mathit{Stud}, \mathit{Security}, \mathit{psd})$ for the lecture; this pseudonym prevents multiple submissions even though she hides her identity in the next step before submitting the feedback.

Anonymizing the evaluation. At this point, the student has assembled the following proof:

$$\begin{aligned} &\mathit{Prof}\ \mathsf{says}\ \mathsf{Registered}(\mathit{Stud}, \mathit{Security}) \\ \wedge\ &\mathit{Stud}\ \mathsf{says}\ \mathit{evaluate}(\mathit{Security}, \mathit{fb}) \\ \wedge\ &\mathsf{SSP}(\mathit{Stud}, \mathit{Security}, \mathit{psd}). \end{aligned}$$

Since her identity is the only information that would reveal *Stud* as the originator of this evaluation, she existentially quantifies her identity and turns this proof into an anonymous

[4] tales stands for **t**he **a**nonymous **l**ecture **e**valuation **s**ystem.

[5] Special thanks to Stefan Lorenz for participating in this case study.

evaluation:

$$
\begin{aligned}
&\exists x.\\
&\quad \mathit{Prof}\ \textsf{says}\ \textsf{Registered}(x, \mathit{Security})\\
&\quad \wedge\ x\ \textsf{says}\ \mathit{evaluate}(\mathit{Security}, \mathit{fb})\\
&\quad \wedge\ \textsf{SSP}(x, \mathit{Security}, \mathit{psd}).
\end{aligned}
$$

3.2.2. Java Implementation of tales

In the following paragraphs, we describe the calls to the Java implementation of the library. Each API method has its counterpart in the Java implementation. The most notable difference is that variable-sized arguments are always passed as arrays, here denoted by square brackets $[a_1, \ldots, a_n]$ (technically, these are realized as an instance of the Java class `ArrayList<Object>`).

We detail the calls to the API and leave the Java-related code abstract (e.g., we write <read from file> rather than showing the actual Java calls to read a file from disk and we do not show the construction of the arrays).

Lecture registration. The name of the student and the name of the lecture are denoted by the Java variables `studId` and `lectureName`, respectively. The API to create the registration proof looks as follows:

```
Formula regFormula = new Says(profId, "registered", [studId,
    lectureName]);
Proof regProof = new Proof(regFormula);
```

The professor sends this proof by her favorite means of transportation, for instance, via a TLS-secured connection.

Lecture evaluation. After importing this proof, the student can evaluate the lecture by creating the proof for the evaluation and the service-specific pseudonym. The student evaluation is denoted by the Java variable `fb`:

```
Proof proof = <read from file>;
Formula eval = new Says(studId, "evaluate", [lectureName, fb
    ]);
proof.append(eval);
Formula ssp = new SSP(studId, lectureName);
proof.append(ssp);
```

The `append` method of the proof objects corresponds to the logical conjunction.

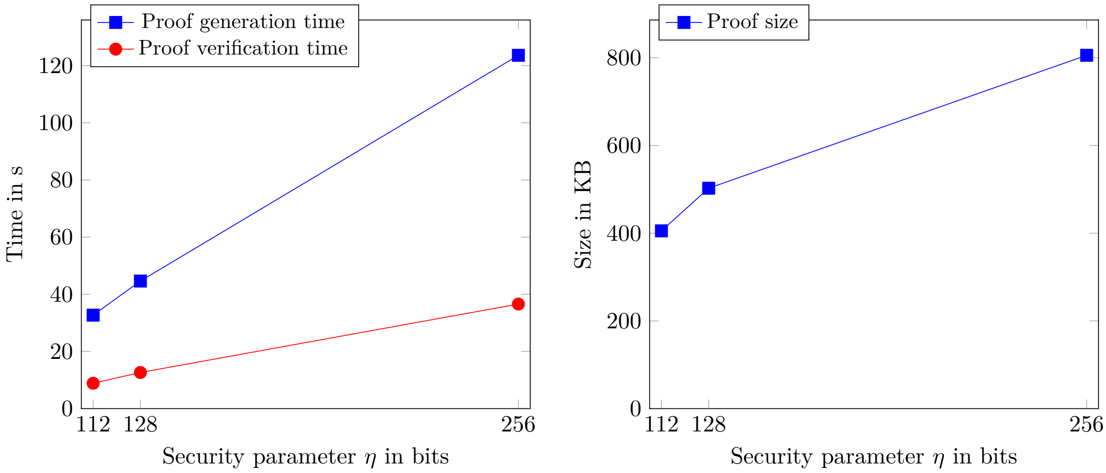


Figure 3.1.: The results for anonymous evaluation proof for various security parameters. The number of threads is fixed to 8.

Anonymizing the evaluation. Finally, the student needs to hide her identity from the proof.

```
proof.hide(studId);
```

This `hide` method call ensures that all occurrences of the identity are hidden. Technically, this call also ensures that the commitments and the proof itself are appropriately re-randomized.

To not trivially break anonymity, the student must use an anonymous and secure connection to submit the evaluation to the professor, e.g., by using TLS over TOR.

3.2.3. Experimental Evaluation

We fixed the number of threads to 8 and we experimentally evaluated the time needed by students to transform a registration proof into an anonymous evaluation proof, the time it takes the professor to verify such a proof, and the size of such a proof. The results are reported in Figure 3.1. Furthermore, we fixed the security parameter to $\eta = 112$ bits and we evaluated the effect of multi-threading on the proof generation and the proof verification; Figure 3.2 reports these results. Finally, Figure 3.3 evaluates the effect of the CPU clock speed on the experimental results.

Discussion. As depicted by Figure 3.1, creating a proof for the anonymous evaluation requires between 32.7 s and 123.6 s, the verification is significantly faster and varies between 8.8 s and 36.6 s for a security parameter of $\eta = 112$ bits and $\eta = 256$ bits, respectively. A large portion of the proof generation time, however, is spent on re-randomizing. For instance, the 32.7 seconds for the smallest security parameter consists of 29.4 s of re-randomization. The reason is that adding the service-specific pseudonym to the proof

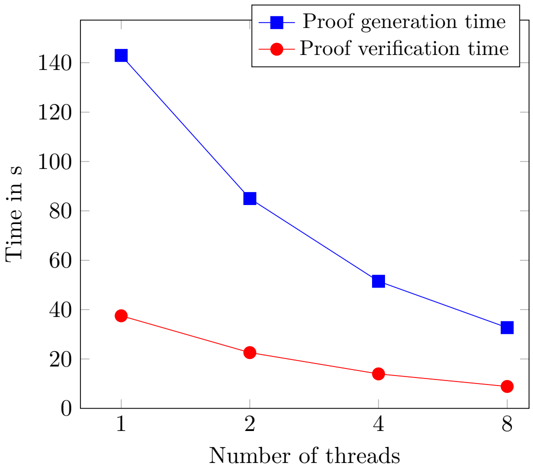


Figure 3.2.: The results for anonymous evaluation proof for various numbers of concurrent threads. The security parameter is fixed to $\eta = 112$ bits.

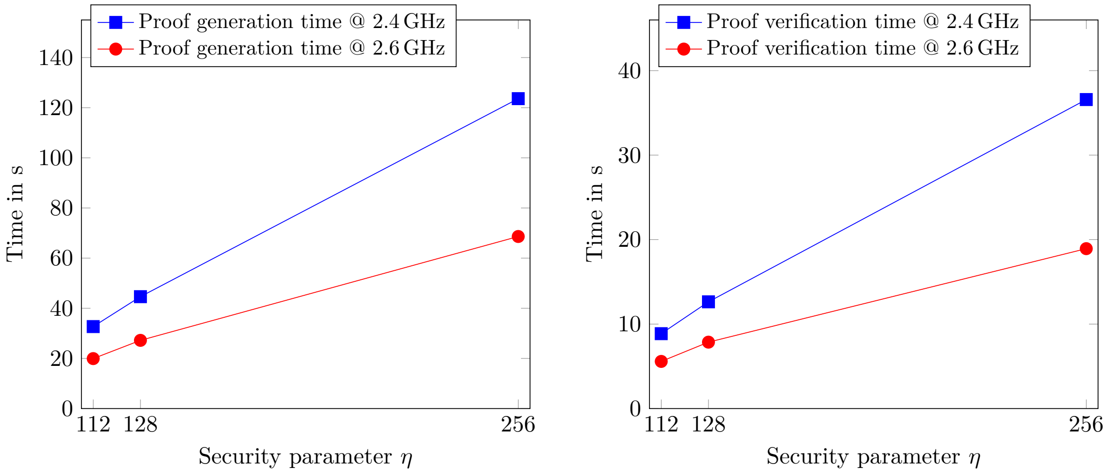


Figure 3.3.: The results for anonymous evaluation proof using different CPUs with different clock speed. The number of threads is fixed to 8.

as well as the hide operation both trigger re-randomization steps. The proof size varies between 405 KB and 805 KB.

As expected, increasing the security parameter causes a linear-logarithmic increase in the computation time (it may appear linear because of the compressed scaling of the y-axis versus the elaborate scaling of the x-axis). Although the computations are not fast enough for daily use, we believe they are fast enough for handing in anonymous evaluations several times a year. Also the proof size of roughly 500 KB does not pose a problem thanks to the broadband internet connections that are becoming increasingly available.

Figure 3.2 shows the impact of multi-threading on the computation times. As expected, the time decreases. Overhead such as thread management and thread communication prevent the performance to scale inverse-proportionally. In other words, using n threads does not divide the total time required by n.

Figure 3.3 shows the impact of a faster CPU with the fixed number of 8 threads. As expected, the CPU with the faster cores outperform the slower CPU. This effect is even more prominent with larger security parameters: larger security parameters are computationally more involved and push overhead such as thread management into the background.

3.3. Anonymous Webs of Trust

Over the last years, the Internet has evolved into the premium forum for freely disseminating and collecting data, information, and opinions. Often, information providers want to keep their true identity hidden: for instance, some may want to present their opinions anonymously to avoid associations with their race, ethnic background, or other sensitive characteristics. The ability to anonymously exchange information, and hence the inability of users to identify the information providers and to determine their credibility, raises serious concerns about the reliability of exchanged information. Ideally, one would like to have a mechanism for assigning trust levels to users, allowing them to anonymously exchange data and, at the same time, certifying the trust level of the information provider.

Webs of trust. Webs of trust (WOT) constitute a well-established alternative to centralized public key infrastructures (PKI) such as those incorporated in browsers and operating systems (e.g., VeriSign [216]). The role of a PKI is to bind public keys to their owner, the reason being that public keys by themselves are only related to their corresponding secret keys but not to their owners. Hence, using a public key to encrypt a sensitive message can only guarantee privacy if this public key in fact belongs to the intended recipient. Similarly, signatures guarantee the authenticity of a message only if the verification key is bound to some well-known identity. In a WOT, there is no central authority but each participant decides which public keys she considers trustworthy. This trust is expressed by signing the public keys that are considered authentic along with a set of user and key attributes (e.g., user name and key expiration date). These certificates along with the signed public key and user attributes are publicly stored on so-called key servers; everybody with access to the key servers can participate in the WOT. Furthermore, these certificates can be chained in order to express longer trust relationships: For instance, the certificate chain

$$\mathsf{sign}(vk_1, attr_1)_{sk_2}, \mathsf{sign}(vk_2, attr_2)_{sk_3}$$

says that the owner of vk_3 has certified the binding between the public key vk_2 and the set $attr_2$ of attributes, and the owner of vk_2 has certified the binding between the public key vk_1 and the set $attr_1$ of attributes. After receiving a signature on message m that can be verified using vk_1, the owner of vk_3 knows that m comes from a user bound to the attributes $attr_1$ of trust level 2; initially, we define the trust level provided by a certificate chain as the number of chained elements. In Section 3.3.5, we consider a more sophisticated trust measure. Hence for authenticating a message in the context of a WOT, the sender needs to search the key servers to find a chain of certificates starting with a certificate released by the intended recipient and ending with a certificate for the sender's key.

3.3.1. Designing Anonymous Webs of Trust

Anonymous webs of trust are an extension of WOT that preserve the privacy of users while offering strong authenticity guarantees. For instance, the owner of vk_1 might want to prove the existence of the certificate chain $\mathsf{sign}(vk_1, attr_1)_{sk_2}, \mathsf{sign}(vk_2, attr_2)_{sk_3}$ in order to authenticate a message m with the owner of sk_3, without revealing any information about the keys and attributes involved in this certificate chain but proving to be a user of trust level 2.

Loosely speaking, the owner of vk_1 would like to prove a statement of the form "there exist certificates C_1, C_2, a signature S, keys K_1, K_2, and attributes A_1, A_2 such that *(i)* C_1 is a certificate for (K_1, A_1) that can be verified with key K_2 *(ii)* C_2 is a certificate for (K_2, A_2) that can be verified with key vk_3, and *(iii)* S is a signature on m that can be verified with K_1". This statement reveals only the length of the chain, i.e., the trust level of the sender, the authenticated message, and the public key of the intended recipient, without saying anything about the other keys, the certificates, and the attributes involved in the certificate chain.

In this way, the sender achieves a high degree of anonymity. In some circumstances, however, hiding all the attributes in the certificate chain may not be desirable since these attributes may capture trust properties that the receiver may want to check. For instance, each certificate could contain a number describing to what extent the signer trusts the signed key (as in the trust signatures of the OpenPGP standard [69]). The other extreme, namely revealing all the attributes, may leak too much information about the sender. To remedy this trade-off between anonymity and trust, one could reveal the average of the trust attributes along the chain, or some more sophisticated trust measure, without disclosing the individual attributes.

3.3.2. Implementation of Anonymous Webs of Trust

Using the API, we implement anonymous webs of trust via formulas of the form

$$vk_U \ \mathsf{says}\ \mathsf{trusted}(vk_{U_2}, attr_{U_2}).$$

Recall that public keys serve as user identifiers. Instead of uploading certificates along with the signed keys to the key servers, the corresponding proof is uploaded.

Since trust into public keys cannot always be established directly, we use *certificate chains.*

Definition 3.1 (Certificate Chain). *A certificate chain from $(vk_1, attr_1)$ to $(vk_\ell, attr_\ell)$ is a sequence of certificates $\mathcal{C} = (cert_1, \ldots, cert_{\ell-1})$ of length $\ell - 1$, where $cert_i = \mathsf{sign}(vk_{i+1}, attr_{i+1})_{sk_i}$ and $\ell \geq 2$. We say that $(vk_\ell, attr_\ell)$ has* trust level *$\ell - 1$. We assume to know the binding between sk_1 and $(vk_1, attr_1)$, which can be captured by an additional self-generated certificate $\mathsf{sign}(vk_1, attr_1)_{sk_1}$.*

In order to authenticate a message m with the owner of vk_1, the owner of vk_ℓ has to retrieve a certificate chain from vk_1 to vk_ℓ and create a proof p for the formula

$$vk_1 \ \mathsf{says}\ \mathsf{trusted}(vk_2, attr_2) \wedge \cdots \wedge vk_{\ell-1} \ \mathsf{says}\ \mathsf{trusted}(vk_\ell, attr_\ell) \wedge vk_\ell \ \mathsf{says}\ \mathsf{msg}(m)$$

using the following code, where the values p_i are retrieved from the key servers and correspond to proofs for formulas vk_i says $\mathsf{trusted}(vk_{i+1}, attr_{i+1})$, respectively:

```
let $p'_2$ = $\mathsf{mk}_\wedge$ $(p_1, p_2)$;
let $p'_3$ = $\mathsf{mk}_\wedge$ $(p'_2, p_3)$;
let $p'_4$ = $\mathsf{mk}_\wedge$ $(p'_3, p_4)$;
  ⋮
let $p'_{\ell-1}$ = $\mathsf{mk}_\wedge$ $(p'_{\ell-2}, p_{\ell-1})$;
```

The logical conjunction $p'_{\ell-1}$ constitutes a validity proof for the corresponding certificate chain.

Using certificate chains, we can transmit authenticated messages from the user whose key vk_ℓ comprises the end of the certificate chain to the user who signed the first element. We use the predicate $\mathsf{msg}(m)$, which encapsulates the authenticated message m.

```
let $p_m$ = mkSays $sk_\ell$ $\mathsf{msg}(m)$;
let $p$ = $\mathsf{mk}_\wedge$ $(p'_{\ell-1}, p_m)$;
```

Since the proof should not reveal the user identities, we weaken this statement by existentially quantifying over all secret witnesses:

$$\mathsf{let}\ p' = \mathsf{hide}\ p \left(\begin{array}{l} \exists \widetilde{x}, \widetilde{y}.\ vk_1 \ \mathsf{says}\ \mathsf{trusted}(x_2, y_2) \\ \qquad \vdots \\ \qquad \wedge\ x_{\ell-1} \ \mathsf{says}\ \mathsf{trusted}(x_\ell, y_\ell) \\ \qquad \wedge\ x_\ell \ \mathsf{says}\ \mathsf{msg}(m) \end{array} \right)$$

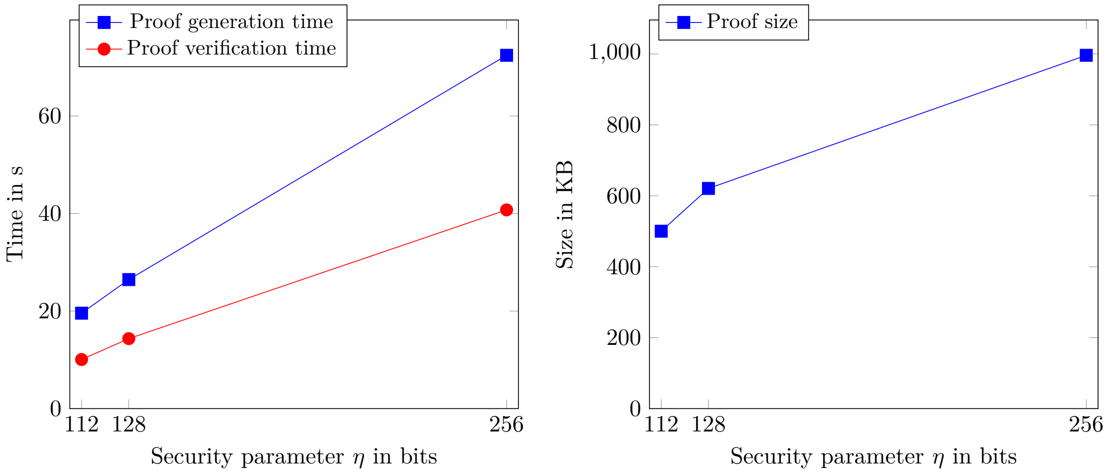


Figure 3.4.: The results for anonymously authenticating a message via a certificate chain of length 2 using various security parameters. The number of threads is fixed to 8.

using the expected API calls.

The proof p' shows that the formula only reveals the public key vk_1 of the intended recipient, the authenticated message m, and the length of the chain (that is, the trust level of the sender); p' is sent to the verifier, who, after successful verification, authenticates message m as coming from a principal of level $\ell - 1$.

3.3.3. Experimental Evaluation

We fixed the number of threads to 8 and we experimentally evaluated the time needed to create a proof for authenticating a message via a certificate chain of length 2. The results are reported in Figure 3.4. Furthermore, we fixed the security parameter to $\eta = 112$ bits and we evaluated the effect of multi-threading on the proof generation and the proof verification times. Figure 3.5 reports the results.

Discussion. As depicted by Figure 3.4, creating a proof for anonymously authenticating a message over a chain of length 2 requires between 19.6 s and 72.4 s, the verification times varies between 10.1 s and 40.7 s for a security parameter of for $\eta = 112$ bits and $\eta = 256$ bits, respectively. As for the tales case study, a large portion of the proof generation time is spent on re-randomizing the proof parts that are hidden. In particular, the non-optimized API implementation re-randomizes the complete proof for every hide operation. Since this proof requires three hide operations, namely for the two principals and the message, the fraction spent on re-randomizing is even larger. The proof size varies between 500 KB and 996.3 KB.

Increasing the security parameter increases the computation time but due to the scaling of the y-axis versus the scaling of the x-axis. As for the tales case study, the computation costs are too high for online use, but they are feasible if used for certain

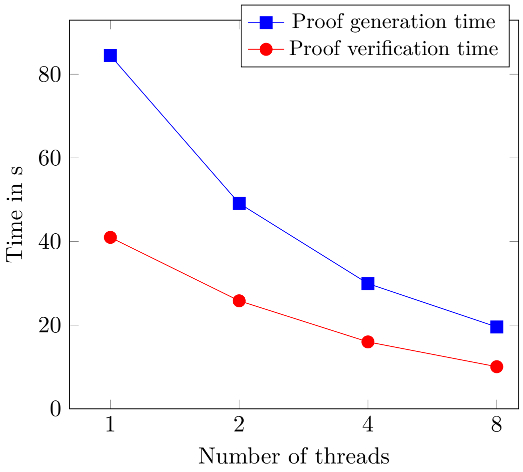


Figure 3.5.: The results for anonymously authenticating a message via a certificate chain of length 2 using various number of threads. The security parameter is fixed to $\eta = 112$ bits.

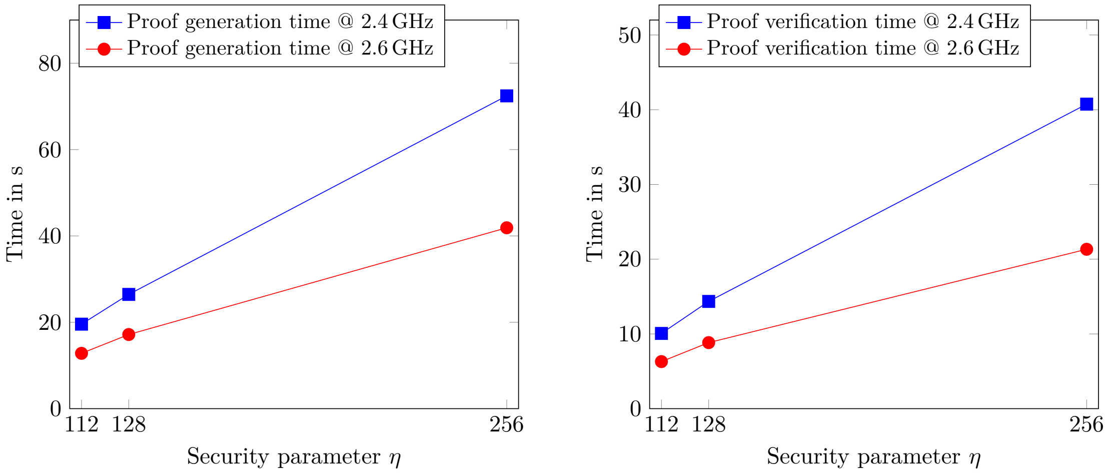


Figure 3.6.: The results for the anonymous authentication of a message via a certificate chain of length 2 using different CPUs with different clock speed. The number of threads is fixed to 8.

scenarios. The proof size of 500 KB does not pose a problem due to the widely available broadband internet connections today.

Figure 3.5 shows the impact of multi-threading on the computation times. The time decreases but computational overhead such as thread management and inter-thread communication prevent the optimal performance gain.

Figure 3.6 shows the impact of a faster CPU with the fixed number of 8 threads. As expected, the CPU with the faster cores outperform the slower CPU. This effect is even more prominent with larger security parameters that demand more mathematical operations.

Comparison: dedicated implementation vs. declarative API implementation. In previous work, Backes et al. [25] implemented anonymous webs of trust using a dedicated cryptographic setup based on Σ-protocols. As discussed in Section 2.7, statements for which Σ-protocols exist are generally more efficient than the respective proof based on the Groth-Sahai zero-knowledge scheme. For the statements of the form "I know a signature on a verification key that is used to verify a signature", i.e., statements necessary in webs of trust, we are not aware of an efficient Σ-protocol.[6] Intuitively, the reason is that it is extraordinarily expensive to draw a connection between a verification key signed as part of the message and the same verification key used as key to verify a signature. Technically, this connection requires a non-standard exponentiation proof [73] that shows that a committed value is the result of computing the exponentiation of a committed value to the power of a committed value. This proof re-computes in zero-knowledge the square-and-multiply algorithm. This algorithm branches on every exponent bit and does an exponentiation or a multiplication followed by an exponentiation, depending on whether the exponent bit was 0 or 1, respectively. The corresponding zero-knowledge proof, however, cannot mimic this branching because the resulting branching pattern reveals the secret exponent. Intuitively, the zero-knowledge proof mitigates this by always computing both branches and hiding which result is used for the next computation step. The additional computation overhead renders this proof impractical.

At this point, we would expect a comparison between the running time of the dedicated, Σ-protocol based implementation and the implementation based on the declarative API. Using the dedicated implementation for a small security parameter of $\eta = 80$ bits, computing a proof takes several days. The experiments for a security parameter of $\eta = 112$ bits, let alone an even larger one did not terminate in a feasible amount of time.

3.3.4. Formal Verification

Implementations based on the declarative API guarantee that the specified authorization properties hold. It is important, however, to verify that the protocol as a whole guarantees the desired anonymity properties to exclude unintended protocol interleavings or logical protocol errors. We conducted a formal security analysis by modeling our protocol in the applied pi-calculus [3], formalizing the anonymity property as an observational equivalence

[6] Here, automorphic signatures show their strength: proving such statements causes virtually no overhead.

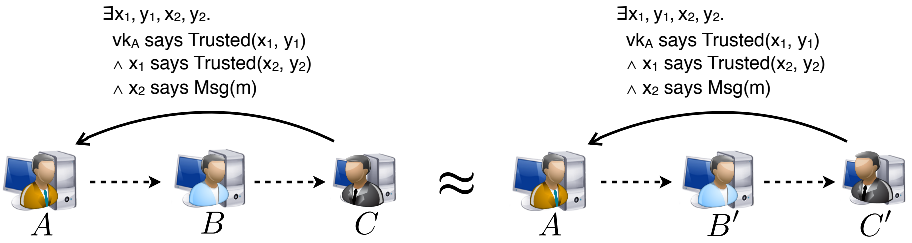

∃x₁, y₁, x₂, y₂.
vk_A says Trusted(x₁, y₁)
∧ x₁ says Trusted(x₂, y₂)
∧ x₂ says Msg(m)
A
B
C
≈
∃x₁, y₁, x₂, y₂.
vk_A says Trusted(x₁, y₁)
∧ x₁ says Trusted(x₂, y₂)
∧ x₂ says Msg(m)
A
B′
C′

Figure 3.7.: Anonymity game.

relation, and verifying our model with ProVerif [52, 1], a state-of-the-art automated theorem prover that provides security proofs for an unbounded number of protocol sessions. We model zero-knowledge proofs following the approach described in Chapter 2. For easing the presentation, in this section we focus on certificate chains without attributes. We discuss the ProVerif model in more detail below. The scripts used in the analysis are reported in Section B.1.

Attacker model. In our analysis, we consider a standard symbolic Dolev-Yao active attacker who dictates the certificates released by each party (i.e., the attacker controls the web of trust), the certificate chains proven in zero-knowledge, and the proofs received by each verifier.

Verification of anonymity. Intuitively, we formalize the anonymity property as a cryptographic game where two principals act in a web of trust set up by the attacker and one of them authenticates by proving in a certificate chain chosen by the attacker. If the attacker cannot guess which of the two principals generated the corresponding proof, then the protocol guarantees anonymity. Our model includes an arbitrary number of honest and compromised parties as well as the two (honest) principals engaging in the anonymity game.

The anonymity game is defined by two distinct processes that are replicated (that is, spawned an unbounded number of times) and in parallel composition (i.e., concurrently executed). In the first process, each of the two principals releases certificates as dictated by the attacker. Since the attacker controls also the certificates released by the other parties in the system, both honest and compromised ones, the attacker controls the topology of the whole web of trust. In the second process, the two principals receive two (possibly different) certificate chains from the attacker. If both certificate chains are valid and of the same length, we non-deterministically choose one of the two principals C and C', and we let it output the corresponding proof. The observational equivalence relation $\approx$ (see Figure 3.7) says that the attacker should not be able to determine whether model $\mathcal{M}_1^{\mathsf{Anon}}$ in which C outputs the proof or $\mathcal{M}_2^{\mathsf{Anon}}$ in which C' outputs the proof is being executed.

Theorem 3.1 (Anonymity)**.** *For the two processes $\mathcal{M}_1^{\mathsf{Anon}}$ and $\mathcal{M}_2^{\mathsf{Anon}}$, the observational equivalence relation $\mathcal{M}_1^{\mathsf{Anon}} \approx \mathcal{M}_2^{\mathsf{Anon}}$ holds true.*

Proof. Automatically proven using ProVerif. The scripts are given in Appendix B. □

Discussion. Malleable zero-knowledge models are conceptually simple and often yield more compact and cleaner models than non-malleable counterparts. The reason is that malleability requires only the modeling of the atomic proofs occurring in the protocol. Complex proofs can be assembled from the atomic proofs, exploiting the malleable nature. In the non-malleable case, proofs cannot be combined and all used constellations of atomic proofs occurring in the protocol have to be explicitly considered in the model. This conceptual simplicity becomes apparent in the anonymous webs of trust case because the protocol relies solely on signature verifications. Using the malleability property, these verifications are stringed-together to prove signature chains. The malleable nature, however, causes problems in the termination behaviour of automated verification techniques.

Malleable proofs are designed to be separable and arbitrarily combinable. While this makes them a perfect match for open-ended and interoperable systems, the sheer number of possible combinations causes problems for the automated verification techniques underlying ProVerif. It requires careful design to devise a faithful malleable zero-knowledge model that yields termination, especially because the model contains re-randomization operations.

In general, creating a sound symbolical model of re-randomization is impossible [214]. Intuitively, the reason is that randomness can cancel each other out (using inverse elements in the respective mathematical groups), which cannot be captured symbolically. The insight is that we require only properties that have a sound model. For instance, we assume that honest protocol participants always use a true random value in the re-randomization process, i.e., values are never chosen in order to cancel each other out.

Furthermore, the model must allow honest participants and the attacker to re-randomize any part of the proof, even if they did not create the proof and the corresponding part is hidden. A first approach is to let re-randomization replace randomness in a proof, i.e., the proof itself and the contained commitments, with a given value. Since commitments can be opened once the randomness is known, such an approach cannot provide any privacy properties. Another idea is to symbolically combine the randomness using a constructor. While functional, this approach effectively yields non-termination due to the sheer number of possible combinations. In particular, the corresponding equational theory must consider the commutativity of the underlying mathematical operation in order to be sound: neglecting commutativity yields concrete attacks that exploit this property but that are not captured symbolically.

We solve the two aforementioned problems by introducing for every component of the zero-knowledge proof two different random values: one that can only be modified by honest protocol participants and one that can be arbitrarily modified by the attacker. If a protocol participant applies re-randomization, we replace the corresponding “honest randomness”; if the attacker applies re-randomization, we replace the corresponding “attacker randomness”. At first, this method seems to enable protocol participants and the attacker to selectively replace and in turn learn the randomness used in a proof. Naturally, honest protocol participants do not exploit this fact. For the malicious case, the attacker cannot touch the “honest randomness”. Consequently, the attacker can only know the complete randomness (honest and attacker randomness), if the randomness was revealed in the first place.

Still, re-randomization has a huge impact on the verification process, even with all

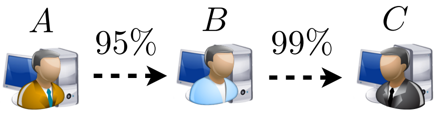


Figure 3.8.: Single chain annotated with trust values.

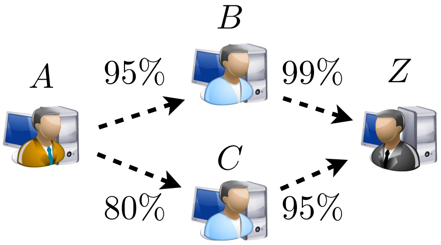


Figure 3.9.: Web of trust with multiple paths.

these insights and tweaks: the model without re-randomization terminates within a few minutes while the model with re-randomization requires several days.

3.3.5. Implementing Sophisticated Trust Measures

Since the API is very expressive and allows for selectively hiding parts of the proven statement, it is easy to implement the extensions envisioned by Backes et al. [25]. For instance, it is straightforward to create a proof for certificate chains with complex attributes.

In the following, we use complex attributes to show how to accommodate sophisticated trust measures (e.g., [174, 5, 146, 76, 75, 14, 138]). Specifically, we focus on the trust measure proposed by Caronni [76]. The examples in this section are intentionally borrowed from Caronni in order to show the applicability of the API to existing trust models. Let us consider the web of trust in Figure 3.8. As shown by the weight of the two links, the trust of B in C is higher than the trust of A in B. Caronni's trust measure is based on the multiplication of the trust values of the individual links. Therefore the trust degree provided by the chain between A and C is $95\% \cdot 99\% = 94.05\%$.

The trust degree is embedded into the attributes of the respective signatures: $attr_B := (t_1, attr'_B)$ and $attr_C := (t_2, attr'_C)$ where $t_1 := 95$ and $t_2 := 99$ are the trust values taken from Figure 3.8, and $attr'_B$ and $attr'_C$ are further attributes such as an expiration date. More precisely, the used predicates are:

$$vk_A \ \mathsf{says}\ \mathsf{trusted}(vk_B, t_1, attr'_B) \qquad vk_B \ \mathsf{says}\ \mathsf{trusted}(vk_C, t_2, attr'_C)$$

In addition to proving the validity of the certificate chain, we additionally show the formula

$$t = t_1 \cdot t_2$$

using the mkEQN method. Naturally, we can hide any value of this equation. Notice that the result t of this proof equals $9,405$, because division is not supported. The recipient of the proof has to manually divide this number by $10,000$ to obtain the expected result in percent. In general, the recipient has to divide the result by 100^ℓ where ℓ equals the number of chain elements.

Often, revealing the exact trust value t severely threatens the users' anonymity: the exact trust value limits the number of possible chains or even uniquely identifies the used chains and consequently identifies the users involved in the chain. We solve this problem by showing a lower (or upper) bound with the API method mkREL. For instance, in addition to the validity proof of certificate chains, we show the formula

$$\begin{aligned} & t = t_1 \cdot t_2 \\ \wedge\ & 9000 < t \end{aligned}$$

where t, t_1, and t_2 are hidden and only the lower bound $9,000$ is revealed. This formula convinces the recipient that the trust value is larger than 90%, thus strengthening the anonymity of the involved users.[7]

Exploiting the ability to prove logical conjunctions and inequalities, we can deal with even more complex scenarios. Consider the graph in Figure 3.9: Z has to show that there exist two distinct paths from A to Z. The total trust degree is computed as

$$1 - (1 - 95\% \cdot 99\%) \cdot (1 - 80\% \cdot 95\%) \approx 98.6\%. \tag{3.10}$$

More precisely, we prove the following formula

$$\begin{aligned} & t_1 = t_{AB} \cdot t_{B_Z} \\ \wedge\ & t_2 = t_{AC} \cdot t_{CZ} \\ \wedge\ & t'_1 = 100^2 - t_1 \\ \wedge\ & t'_2 = 100^2 - t_2 \\ \wedge\ & t_3 = t'_1 \cdot t'_2 \\ \wedge\ & t = 100^4 - t_3 \\ \wedge\ & vk_B \neq vk_C \end{aligned}$$

where $t_{I,J}$ corresponds to the trust value from I to J (extracted from the respective formula vk_I says trusted$(vk_J, t_{IJ}, attr'_J)$); the values 100^2 and 100^4 stem from the fact that we cannot divide and need to adjust Equation 3.10 accordingly. In this proof, the inequality proof $vk_B \neq vk_C$ is of particular importance to prevent a malicious user from reusing the same certificate chain over and over to boost the trust value: using the same chain n times yields a trust value of $1 - (1 - x)^n$ that approaches the highest possible trust value of 1 as n grows and $x > 0$.

3.4. A Security API for Distributed Social Networks

Over the last years, online social networks (OSNs) have become the natural means to get in touch with people and to engage in a number of social activities, such as sharing information, exchanging opinions, organizing events, and publishing advertisements. The new dimensions

[7] As noted by Backes et al. [25], Section 4, it is possible to hide the exact length of a certificate chain. The same idea can be applied using the API.

$\mathcal{M}$	::=		masks
	$\mid$	p	pseudonym
	$\mid$	$\mathcal{R}$	social relation
op	::=		operations
	$r \mid w \mid rw$		
ACL	::=		access control list
	$(\mathcal{M},op)$::ACL $\mid$ [ ]		

M	::=	Register (J,p_J)
	$\mid$	getHandles $(\mathcal{M}_J)$
	$\mid$	getResource $(\mathcal{M}_J,hdl(res))$
	$\mid$	putResource $(\mathcal{M}_J,hdl(res),\ res')$
	$\mid$	getFriends $(\mathcal{M}_J)$
	$\mid$	IndirectRegister $(\mathcal{M}_J,K)$

We let I, J, and K range over principals. We write $hdl(res)$ to denote the handle of the resource *res*.

Table 3.1.: Grammar of access control lists.

of social interaction and the opportunities deriving from these novel functionalities tend to push into the background the impressive leakage of personal information (e.g., religious beliefs, political opinions, and sexual orientations) and the consequent threats to users' privacy.

3.4.1. A Core API for Social Networking

This section describes a security API for social networking, which includes methods to establish social relationships as well as to upload and download resources (for instance, pictures and videos). The goal is not to specify a fully-fledged API but to focus on a concise set of methods, which suffice to encode the most prominent features of modern social networks.

A central feature of the social network API is that social links are kept secret and principals can engage in social activities (e.g., post a comment or retrieve a picture) without disclosing their identities. In particular, the API protects the social relations of a user, even if her server is compromised and all her secret information is disclosed.

We could not keep social relations private if access control lists revealed the identity of the principals with read and write capabilities: an attacker compromising the server of principal I would be able to read the access control lists stored therein and immediately learn the identity of I's friends. For this reason, access control lists are defined on *masks* (see Table 3.1), which are ranged over by $\mathcal{M}$ and consist of either a *pseudonym* p or a *social relation* $\mathcal{R}$. The idea is that a user J communicates its pseudonym p_J while establishing social relations: J is the only user that can use this pseudonym and only the users whom J registered with know the link between p_J and J (the pseudonym itself does not reveal any information about the owner's identity, see Section 2.5). We do not impose constraints on the usage of pseudonyms: B can decide to always use the same pseudonym such that friends can track all its activities or to use different pseudonyms to become unlinkable. The social relation $\mathcal{R}$ is simply a tag characterizing a certain social relation. An access control list consists of a list of pairs, whose first component is a mask and the second component is an operation (e.g., r, w, and rw). For instance, $[(p_J, rw), (\mathit{friends}, r)]$ is an access control list specifying that the associated resource can be read and written by the principal with pseudonym p_J and it can additionally be read by the principals in the *friends* relation.

The protocols comprising the social network API are designed to protect the social relations and the anonymity of principals against external observers and against attackers that compromise the servers running the API. A dishonest principal can of course reveal its social relations but an attacker that observes the network traffic or breaks into an API server should not be able to learn them. The key idea to achieve this strong anonymity property is that principals can get and post resources by simply revealing their pseudonym or by proving to be in a certain social relation with the resource provider. Since the cryptographic protocols rely on our declarative security API (see Chapter 2), the messages exchanged between protocol parties are zero-knowledge proofs that do not require any secret input to the verification process; only the access control lists are required to grant access or reject a request. As expected, the requester has to prove to be associated to a certain pseudonym or to be in a certain social relation with the resource provider. This procedure requires the knowledge of some information about the social relations that is, however, only needed when a principal goes online and wants to authenticate. Hence, this data does not need to be stored on a server (it can be stored, for example, on a secure portable device) and it is not leaked in case of server compromise.

Note that an ACL does *not* reveal enough structure about the social graph to apply de-anonymization techniques [183]. An access control list typically consists of social relations that do not reveal any structural information on the social graph. Should a user decide to mainly use pseudonyms, "padding" the ACL with fake pseudonyms (e.g., stipulating that social relatives register only fresh pseudonyms and adding fake pseudonyms until all resources have a fixed number, for instance, 1,000, of pseudonyms associated with them) suffices to hide the actual structure of the social graph and to render de-anonymization techniques inapplicable. Such a blinding technique has no consequence for the requester and only causes a negligible computational overhead for the resource provider. Moreover, since all authorization credentials, i.e., proofs for formulas such as I says $\mathsf{friend}(J)$, and the pseudonym-user bindings are stored on a well-hidden external device, even complete access to a number of servers does not reveal any social relation.

We now describe the methods composing the API, which are summarized in Table 3.1. We write $I.M$ to denote the method M exported by I's API.

$\mathcal{R} \leftarrow I.\mathsf{Register}(J, p_J)$: This method takes as input the identifier J of the principal that wants to establish a social relationship with I and a pseudonym p_J created by J (see Section 2.4). This method returns a tag $\mathcal{R}$ chosen by I to characterize the social relation (e.g., a string such as "friends"). The cryptographic realization ensures that the caller corresponds to the identifier specified in the argument. The pseudonym p_J can be used by I to allow user-based access to J when setting up its access control list. Although I is able to link p_J to J's identity, the pseudonym itself does not reveal anything about J's identity, which is crucial to achieve anonymity. In particular, even if I's server is compromised, the access control list is leaked, and the incoming authentication requests are monitored, the anonymity guarantees still hold.

$\mathit{hdl}(\mathit{res}_1), \ldots, \mathit{hdl}(\mathit{res}_n) \leftarrow I.\mathsf{getHandles}(\mathcal{M}_J)$: This method takes as input the caller's mask

$\mathcal{M}_J$ and returns the handles to I's resources. A handle identifies and describes the resource without disclosing it.[8] Handles are passed to the other methods to specify the resource of interest, as discussed below. Since this method takes as input the caller's mask, J has the option to reveal its pseudonym or to stay anonymous by just proving to be in a certain social relation with I, depending on whether I accepts anonymous requests or reveals its handles only to non-anonymous requesters.

$res \leftarrow I.\mathsf{getResource}(\mathcal{M}_J, hdl(res))$: This method takes as input the caller's mask $\mathcal{M}_J$ and the handle $hdl(res)$ of the requested resource. If $\mathcal{M}_J$ is given read access to res in the corresponding access control list, getResource returns the resource res.

$ack \leftarrow I.\mathsf{putResource}(\mathcal{M}_J, hdl(res), res')$: This method takes as input the caller's mask $\mathcal{M}_J$, the handle $hdl(res)$ of the resource to modify, and the new content res'. This method encodes the methods used by J to post comments on I's wall, to upload pictures, and so on. Typically, messages and pictures are appended while other resources such as the profile picture are replaced. Since the result of this method depends on the specific social network and on the kind of resource, we intentionally leave the behavior unspecified.

$\mathcal{R} \leftarrow I.\mathsf{IndirectRegister}(\mathcal{R}_{IJ}, p_J, K)$: This method allows users to establish indirect social relations (e.g., "friend of a friend"). It takes as input the social relation $\mathcal{R}_{IJ}$ between I and J, a pseudonym p_J chosen by J and the identifier K of the principal which J is interested in establishing an indirect relation with. Notice that J must be in a direct social relation with I, and I has to be in a direct social relation with K. The idea is that if K accepts indirect relations, J can ask I to establish an indirect relation with K on her behalf. This method returns a tag $\mathcal{R}$ that describes the newly established social relation between K and J.

This method could in principle be cascaded to establish indirect relationships of arbitrary degree (e.g., "friend of a friend of a friend" relations). Since such relations are not used in practice, we do not consider them here.

$K_1, \ldots, K_n \leftarrow I.\mathsf{getFriends}(\mathcal{M}_J)$: This method takes as input the caller's mask $\mathcal{M}_J$ and returns the list of I's friends that accept indirect relations. Notice that J must be in a direct social relation with I.

The API additionally comprises standard functions to deal with access control lists (e.g., creation and modification). Since these operations are local, they do not need any cryptographic infrastructure and, for the sake of readability, we omit them throughout this chapter.

[8] For the sake of generality, we do not specify the format of handles: for instance, one can use thumbnails as handles for pictures and URI-style descriptions for text documents.

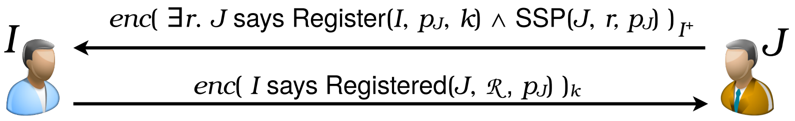

I
J
enc(∃r. J says Register(I, p_J, k) ∧ SSP(J, r, p_J))_I+
enc(I says Registered(J, ℛ, p_J))_k

Figure 3.11.: Protocol Register: user J registers with user I.

3.4.2. Implementation of the Core API

We now describe the cryptographic protocols implementing the social network API using the declarative security API described in Chapter 2. Since the social network API comprises complete cryptographic protocols, they require asymmetric and symmetric encryption schemes, which are not included in the declarative security API. We briefly discuss their application in the social network API.

Symmetric and asymmetric encryptions. The social network API is used to exchange messages between users. To protect these messages from unsolicited eavesdropping, a natural solution is to encrypt all exchanged messages with the public key of the intended recipient. In our scenario, however, we want to keep the identity of the sender of a message hidden, while enabling the recipient of a message to reply to the sender in private. Therefore, we also include symmetric encryption: the sender of a message includes a symmetric session key and the recipient can use this key to reply to the sender without knowing the identity of the sender. In the following, we write $enc(m)_k$ to denote the symmetric encryption of the message m with the key k; we write $enc(m)_{I+}$ to denote the asymmetric encryption of message m with I's public key.

Implementing the core API. We describe the declarative implementation of the core API methods.

$\mathcal{R} \leftarrow I.\mathsf{Register}(J, p_J)$: The protocol is depicted in Figure 3.11. J starts the registration procedure by encrypting a proof for the formula

$$\begin{aligned}&\exists r.\\ &\quad J \text{ says } \mathsf{Register}(I, p_J, k)\\ &\quad \wedge\ \mathsf{SSP}(J, r, p_J)\end{aligned}$$

with the asymmetric public key of J. The corresponding proof attests J's wish to be registered with I using the pseudonym p_J. The randomly chosen value r (corresponding to the service in the computation of the service-specific pseudonym) and the fresh (symmetric) session key k is used in the response. The intended recipient's identity prevents I to reuse this proof and to impersonate J.

I replies by encrypting a proof for the formula

$$I \text{ says } \mathsf{Registered}(J, \mathcal{R}, p_J)$$

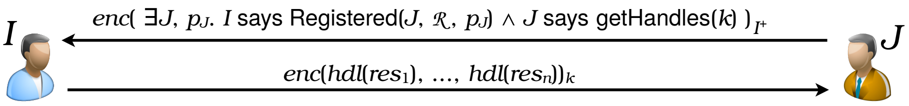
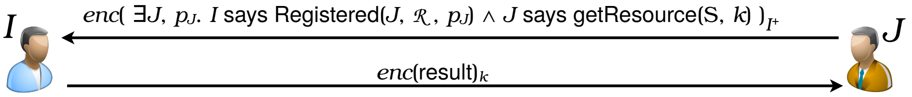

I
enc(∃J, p_J. I says Registered(J, R, p_J) ∧ J says getHandles(k))_{I+}
J
enc(hdl(res_1), ..., hdl(res_n))_k

Figure 3.12.: Protocol for getHandles: user J anonymously requests user I's handles.


I
enc(∃J, p_J. I says Registered(J, R, p_J) ∧ J says getResource(S, k))_{I+}
J
enc(result)_k

For the putResource and the getFriends protocol, the transmitted predicate changes accordingly. Furthermore, for the getResource protocol, S corresponds to a handle obtained by getHandles; for the putResource, S corresponds to a handle and a new resource; for the getFriends protocol, S is empty. For every protocol, the general placeholder result contains the expected return value.

Figure 3.13.: Protocols for getResource, putResource, and getFriends: user J issues a request to user I.

with the transmitted session key k for J. This proof witnesses the social relation of I towards J and also acknowledges that p_J is now recognized by I. This proof is the basis for all future authentication requests from J to I.

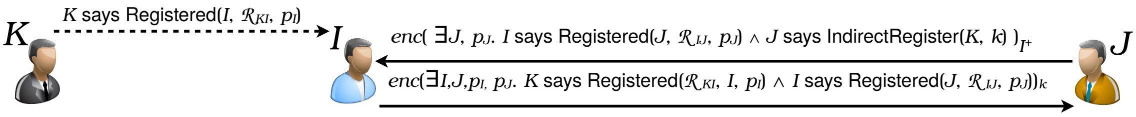
We remark that social relations are unidirectional but they can straightforwardly be made bidirectional by running twice the registration protocol. Furthermore, two users can repeat this procedure with a different random value for r to register different pseudonyms with one another. It is also straightforward to modify the registration procedure and all subsequent protocols to only use the social relation or to include more authentication modalities.

$hdl(res_1), \ldots, hdl(res_n) \leftarrow I.\mathsf{getHandles}(\mathcal{M}_J)$: The protocol is depicted in Figure 3.12. J sends a proof in encrypted form to authenticate with I. We provide three authentication modalities, namely, *pseudonymous authentication*, *relation authentication*, and *anonymous authentication*. In the pseudonymous authentication, J creates a proof for the formula

$$\begin{aligned}&\exists J, \mathcal{R}, r.\\ &\quad I \text{ says } \mathsf{Registered}(J, \mathcal{R}, p_J)\\ &\quad \wedge\ \mathsf{SSP}(J, r, p_J)\\ &\quad \wedge\ J \text{ says } \mathsf{getHandles}(k)\end{aligned}$$

to show that the pseudonym p_J is acknowledged by I and that it belongs to the requester J. In the relational authentication, J creates a proof for the formula

$$\begin{aligned}&\exists J, p_I.\\ &\quad I \text{ says } \mathsf{Registered}(J, \mathcal{R}, p_J)\\ &\quad \wedge\ J \text{ says } \mathsf{getHandles}(k).\end{aligned}$$


K
K says Registered(I, $\mathcal{R}_{KI}$, p_I)
I
enc(∃J, p_J. I says Registered(J, $\mathcal{R}_{IJ}$, p_J) ∧ J says IndirectRegister(K, k))$_{I^+}$
enc(∃I,J,p_I, p_J. K says Registered($\mathcal{R}_{KI}$, I, p_I) ∧ I says Registered(J, $\mathcal{R}_{IJ}$, p_J))$_k$
J

The dotted line denotes the registration process between K and I that must have occurred at some point before the indirect registration method can succeed.

Figure 3.14.: Protocol for IndirectRegister: user J anonymously issues an indirect registration request to user I. After the successful completion of the protocol, J is in an indirect social relation with user K.

and for the anonymous authentication, J creates a proof for the formula

$$\begin{aligned}&\exists J, \mathcal{R}, p_I.\\ &\quad I \text{ says } \mathsf{Registered}(J, \mathcal{R}, p_J)\\ &\quad \wedge\ J \text{ says } \mathsf{getHandles}(k).\end{aligned}$$

We apply a re-randomization step to ensure that the proof parts taken from the registration procedure cannot be linked to J.

Naturally, one could also use the proof for the pseudonymous authentication and hide the pseudonym to achieve an anonymous authentication. Since the proof for the pseudonymous authentication contains more proof elements, the variant shown above is computationally more efficient. The fresh session key k to be used in the response is attached as to the request predicate getHandles. Upon receiving the proof for either formula, I verifies it, checks which resource handles the prover has the permission to read, and sends them to the prover encrypted with the session key received in the first message.

A variant of this protocol is used to implement the getResource, putResource, getFriends methods, shown in Figure 3.13. The only difference is that the corresponding arguments (e.g., $hdl(res)$ in the case of getResource) are additional arguments to the respective predicate (denoted as S in the picture) and the message encrypted in the response is in general the result of the method call (denoted as result in the figure).

The three authentication modalities give different anonymity guarantees and their usage depends on the required service (or resource) and on the access control list. For instance, if a certain resource res is protected by the access control list $[(p_J, rw), (\mathit{friends}, r)]$, then J can run the relation authentication protocol to read res but B has to run the pseudonymous authentication protocol, thus revealing its identity to I, to write on res. In general, there is a trade-off between the restrictiveness of access control lists and the anonymity of requesters.

$\mathcal{R}_{KJ} \leftarrow I.\mathsf{IndirectRegister}(\mathcal{R}_{IJ}, p_J, K)$: The protocol is depicted in Figure 3.14. J sends the formula

$$\begin{aligned}&\exists J, p_J.\\ &\quad I \text{ says } \mathsf{Registered}(J, \mathcal{R}_{IJ}, p_J)\\ &\quad \wedge\ J \text{ says } \mathsf{IndirectRegister}(K, k).\end{aligned}$$

to I, proving to have a social relation with I and requesting an indirect relation with K. As usual, the session key k to be used in the response from I is an argument to the IndirectRegister predicate. I verifies the proof and responds with the formula

$$\begin{aligned}&\exists I, J, p_I, p_J.\\ &\quad K \text{ says } \mathsf{Registered}(I, \mathcal{R}_{KI}, p_I)\\ &\quad \wedge\ I \text{ says } \mathsf{Registered}(J, \mathcal{R}_{IJ}, p_J)\end{aligned}$$

encrypted with the session key k. To break any connection between this proof, K, and I, I can re-randomize all components of the proof except for the hidden identity J of the requester. Intuitively, if J's identity in the proof is not re-randomized, then J can remove the existential quantification on it. Technically, the reason is that since the commitments corresponding to the identity J of the requester did not change, the opening information can be used to unhide the identity. Finally, after removing the existential quantification, J obtains a proof for the formula

$$\begin{aligned}&\exists I, p_I, p_J.\\ &\quad K \text{ says } \mathsf{Registered}(I, \mathcal{R}_{KI}, p_I)\\ &\quad \wedge\ I \text{ says } \mathsf{Registered}(J, \mathcal{R}_{IJ}, p_J)\end{aligned}$$

Notice that since the identity of J is revealed in this formula, it is easy to attach any request such as getHandles to the formula. Since the social relations $\mathcal{R}_{KI}$ and $\mathcal{R}_{IJ}$, K can deduce the relation between her and J. For instance, if $\mathcal{R}_{KI}$ equals "colleague" and $\mathcal{R}_{IJ}$ equals "friend", the proof obtained at the end by J classifies J as a "friend-of-a-colleague" to K.

In order to allow for fine-grained access control policies, we additionally enable a similar protocol in which I authenticates with K by revealing its pseudonym p_I instead of the social relation $\mathcal{R}_{KI}$. The variants of this protocol allow K to define a more precise access control list, built on relations of the form "friends-of-p_I".

Concerning the anonymity guarantees provided by this protocol, I does not learn the identity of J, and K learns neither the identity of I nor the one of J. In particular, K does not actively participate in this protocol; the only requirement is that K previously registered I. Finally, since future requests are encrypted with a session key and sent directly from J to K, I can neither read let alone modify messages sent by K to I.

3.4.3. Experiments

We have experimentally evaluated the proofs used in the protocol of the social network API. More precisely, we used several security parameters to evaluate the proofs used in the registration protocol, the resource request protocols (specifically, the getHandles-protocol) and the indirect registration protocol. For all these proofs, we fixed the number of threads to 8 and we have determined the time needed to create a proof, to verify a proof, and the size of the proof. Furthermore, for the resource request and the indirect registration

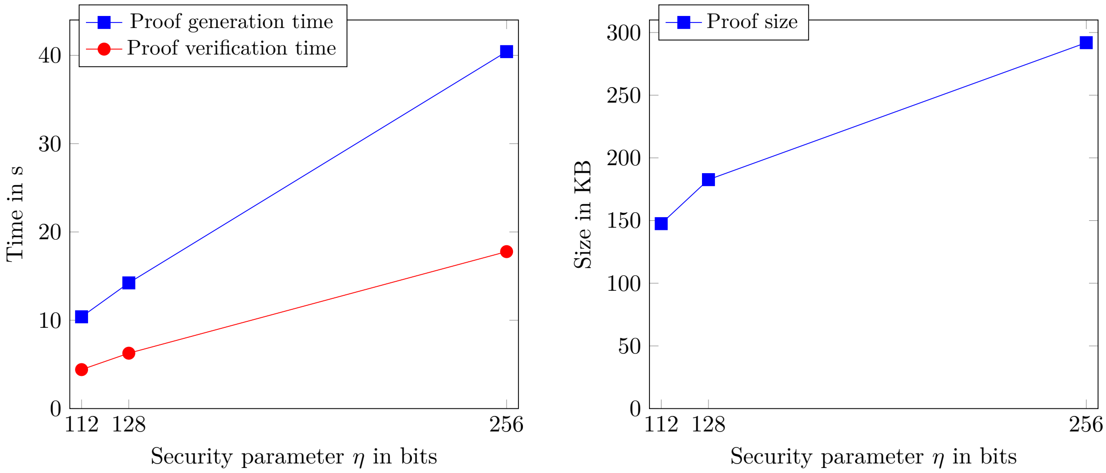


Figure 3.15.: The results for proof used to initiate the registration protocol. The number of threads is fixed to 8.

protocol, we have fixed the security parameter to $\eta = 112$ and used various amounts of threads to show how these computation times scale with the number of used cores. The results are reported in Figure 3.15–3.22.

Discussion. We structure the discussion based on the different proofs:

- Figure 3.15 shows the results for the friend request proof. The proof generation time varies between 10.4 s and 40.4 s, the proof verification time between 4.4 s and 17.7 s, and the proof size between 147.5 KB and 291.9 KB for a security parameter $\eta = 112$ bits and $\eta = 256$ bits, respectively.

- Figure 3.16 shows the results for the friend registration proof, i.e., the response to the friend request proof. The proof generation time varies between 2.8 s and 10.3 s, the proof verification time between 4 s and 16.1 s, and the proof size between 137.3 KB and 271.8 KB for a security parameter $\eta = 112$ bits and $\eta = 256$ bits, respectively.

  Since this proof consists only of one signature, i.e., there are no re-randomization steps, the proof generation is faster than the proof verification.

- We evaluate the getHandles and the indirect registration protocol more thoroughly since they deploy the most complex and involved proofs. Figure 3.17, Figure 3.18, and Figure 3.19 show the obtained results. The proof generation time varies between 15 s and 57.3 s, the verification time between 7.5 s and 30.4 s, and the proof size between 336.1 KB and 668.7 KB for a security parameter $\eta = 112$ bits and $\eta = 256$ bits, respectively.

- The indirect registration protocol is arguably the most complex protocol occurring in the social network API. It requires input from three different parties and exploits

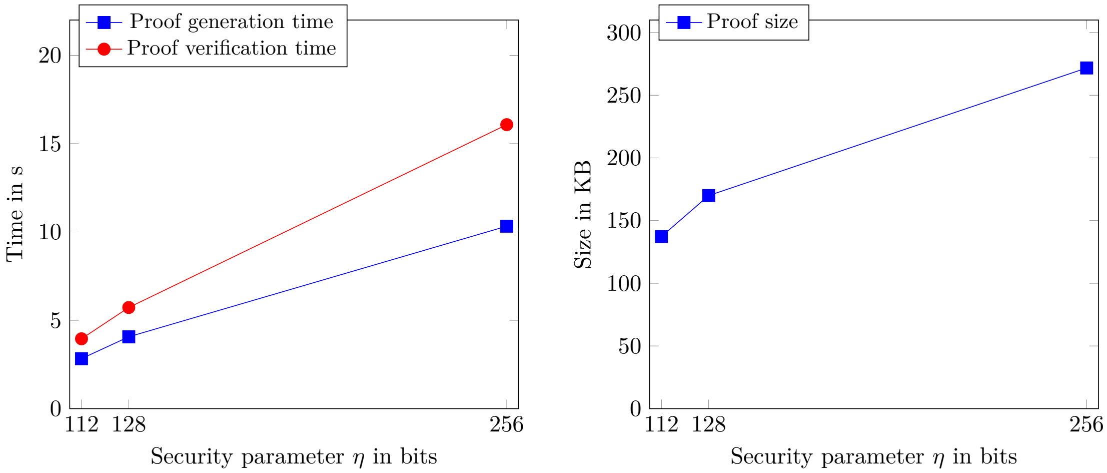


Figure 3.16.: The results for registration proof used to register a protocol participant. The number of threads is fixed to 8.

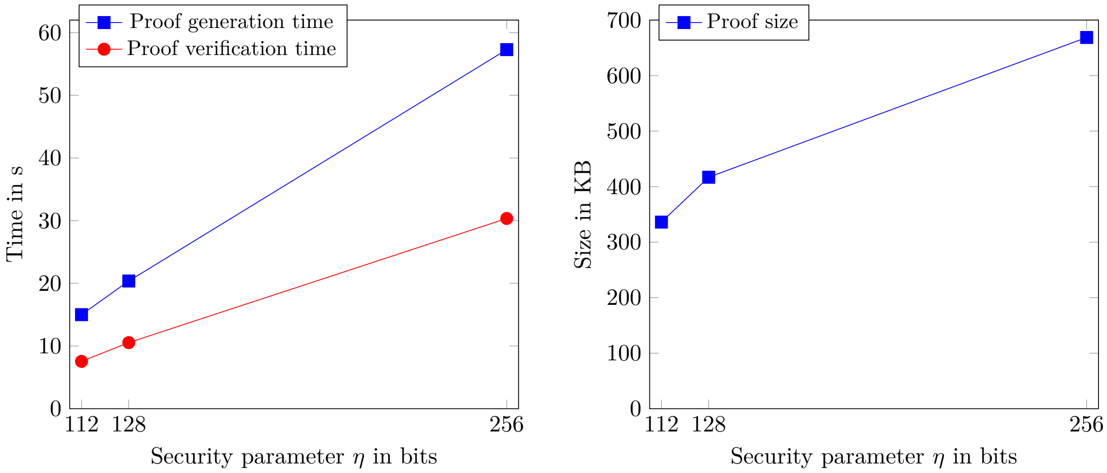


Figure 3.17.: The results for proof used in the getHandles protocol. The number of threads is fixed to 8.

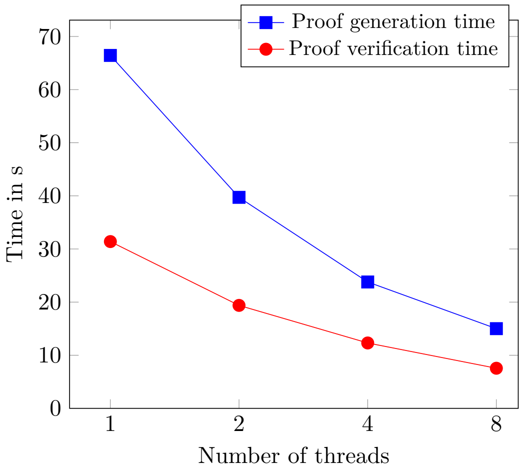


Figure 3.18.: The results for the proof used in the getHandles protocol using various number of threads. The security parameter is fixed to $\eta = 112$ bits.

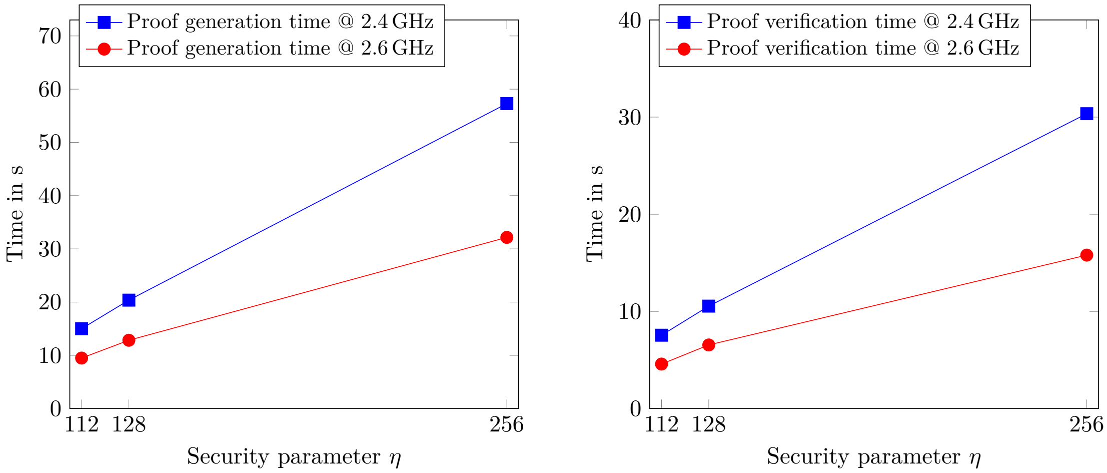


Figure 3.19.: The results for the proof used to initiate the getHandles protocol using different CPUs with different clock speed. The number of threads is fixed to 8.

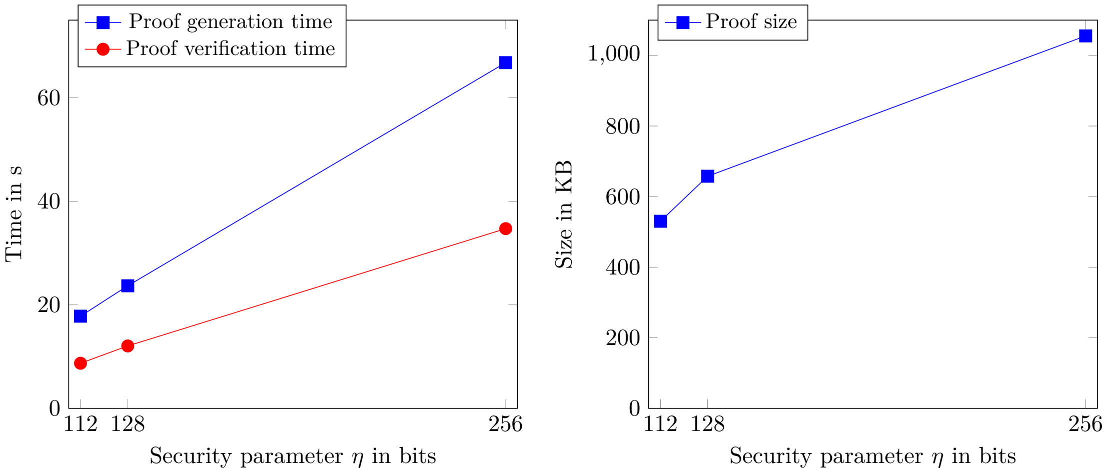


Figure 3.20.: The results for proof used to initiate the indirect registration protocol. The number of threads is fixed to 8.

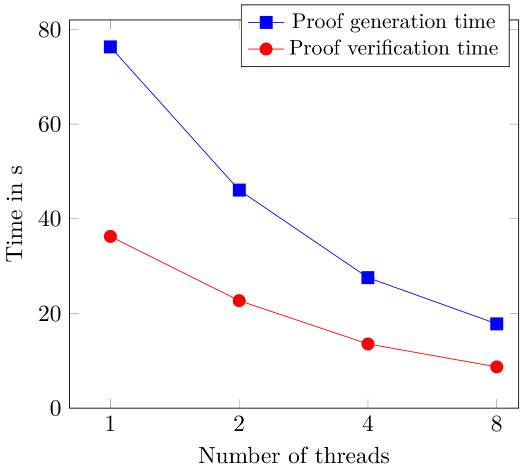


Figure 3.21.: The results for the proof used to initiate the indirect registration protocol using various number of threads. The security parameter is fixed to $\eta = 112$ bits.

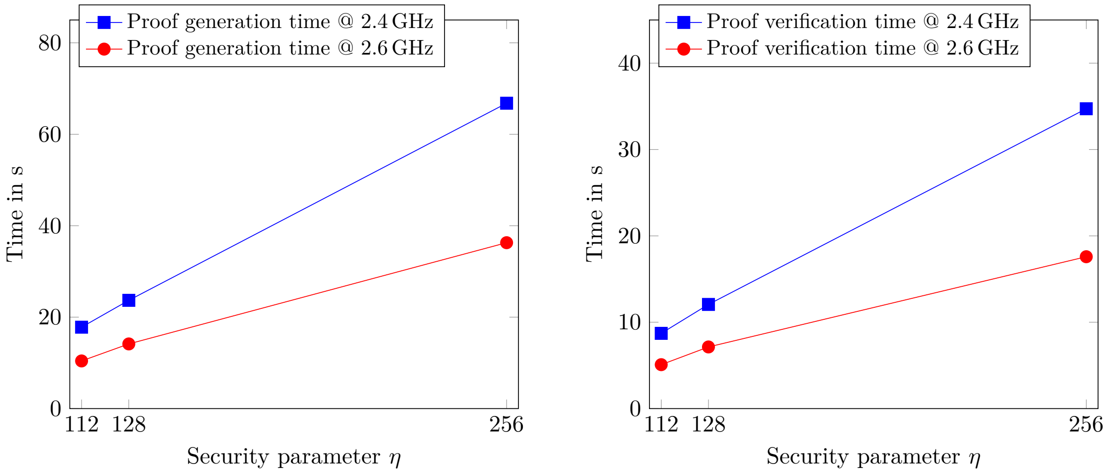


Figure 3.22.: The results for the proof used to initiate the indirect registration protocol using different CPUs with different clock speed. The number of threads is fixed to 8.

all properties that are offered by the underlying zero-knowledge proof scheme. The obtained results are depicted by Figure 3.20, Figure 3.21, and Figure 3.22. The proof generation time varies between 17.8 s and 66.8 s, the proof verification time between 8.7 s and 34.7 s, and the proof size between 530.1 KB and 1055.3 KB for a security parameter $\eta = 112$ bits and $\eta = 256$ bits, respectively.

As for all case studies above, increasing the security parameter causes a linear-logarithmic increase in the computational costs. There are certain scenarios where the incurred costs are bearable for the users, but for the way social networks are used today, the time required to create and verify proofs is too long. Since social networks are also excessively used from mobile devices such as cell phones, the proof size can also become a limiting factor.

Using a different cryptographic implementation, Backes et al. [28] achieve more competitive experimental results. We compare their results with ours below.

3.4.4. Comparison: Dedicated Implementation vs. Declarative API Implementation.

In general, there is a trade-off between a high-level, API-based implementation and a dedicated implementation that is tailored for a given application. For instance, our API enables a programmer to specify sophisticated authorization properties using a high-level, declarative language. At the same time, this deprives programmers from the opportunity to exploit application-specific optimizations.

Efficiency. In previous work, Backes et al. [25] implemented the social network API using a dedicated cryptographic setup based on Σ-protocols. In particular, their signature scheme is equipped with highly-efficient Σ-protocols that outperform our general implementation.

For instance, creating and verifying a proof for an anonymous request of a resource takes less than 580 ms and 410 ms, respectively. The proof size is below 30 KB.

Devising this dedicated implementation was a significant effort and implementing it took several weeks. Furthermore, achieving provable authorization properties required several design iterations. Using the declarative API, the implementation of the social network API took only one day and the authorization properties hold by construction.

Formal verification of anonymity properties. The dedicated implementation based on Σ-protocols relies on non-malleable zero-knowledge proofs. For non-malleable proofs, efficient proof techniques that are compatible with ProVerif exist [32]. Since the declarative API is based on malleable zero-knowledge proofs, such techniques cannot be applied.

We modeled the malleable social network zero-knowledge proofs as we have for the anonymous webs of trust case study (see Section 3.3.4). The malleable nature and the different proofs used in the various protocols, however, prevented ProVerif from terminating during our experiments. This even was the case without the equational theory for the re-randomization of zero-knowledge proofs, which, in our experience with the anonymous webs of trust case study, causes the largest workload for the automated theorem prover. We leave it as future work to devise a faithful zero-knowledge model that allows us to prove anonymity properties for large malleable zero-knowledge models.

Part II
Trustworthy and Privacy-Preserving Retrieval of Personal Information

4. ObliviAd: Provably Secure and Practical Online Behavioral Advertising

The results presented in this chapter build on the following work:

- Backes, Kate, Maffei, and Pecina [23]: "ObliviAd: Provably Secure and Practical Online Behavioral Advertising"

4.1. Introduction

In this section, we introduce ObliviAd, an architecture for practical and privacy-preserving online behavioral advertising. The core of ObliviAd is a mechanism that enables users to retrieve advertisements in a privacy-preserving way. More precisely, only the user learns which advertisement she downloaded. The overall construction provably guarantees *profile privacy* and *profile unlinkability*, two newly defined properties that are specific to online behavioral advertising. Furthermore, the ObliviAd architecture guarantees the correctness of the billing process inherent to the advertising business.

The results in this chapter are clearly targeted towards online behavioral advertising and how to retrieve advertisements in a privacy-preserving manner. Nonetheless, we are convinced that the underlying techniques and building blocks are applicable to other domains and, in general, facilitate the privacy-preserving retrieval of personal information.

Online behavioral advertising. Today, a large majority of online services garner most of their revenues through advertising instead of directly charging their clientele for the usage. Online advertisements, however, are not effective unless they are targeted to the right audience. As a result, online advertisements are no longer shown in a scattershot way but rather targeted to reach a clientele that is selected according to various traits such as demographics, previously visited urls, the current web page, information stored in cookies, and, in general, any kind of observed behavior.

An online behavioral advertising (OBA) system consists of four principal players: the *advertisers*, *brokers*, *publishers*, and *clients*. Advertisers want the ads for their products to reach plausible clients (e.g., car manufacturers want to inform those people interested in buying a car about a new model) and they are willing to pay for this service. Publishers (e.g., newspapers and blogs) are willing to place ads on their webpages, but they expect to be payed in return. Brokers (e.g., Google and Yahoo) mediate between advertisers, publishers, and clients: the advertisers send their ads and bids to the broker, who then distributes them to the publishers' webpages, which are in turn viewed by the clients.

The client may click on an ad if she finds it to be relevant to her needs or interests. For every ad viewed in the *pay-per-view* (PPV) advertising model or clicked in the *pay-per-click* (PPC) advertising model, the advertiser pays the broker who in turn pays the publisher. The PPC, being a pay-for-performance model, receives more attention in the industry. OBA protocols are divided in two phases: the *distribution phase*, in which the broker distributes the ads, and the *tallying phase*, in which the broker computes the number of viewed (or clicked) ads for billing purposes.

In existing OBA systems, publishers embed a link for the broker on their webpages. The sole purpose of these broker links is user tracking. When a user views the webpage, the user's browser contacts the broker's servers, which enables the broker to track the user across all partnering publishers. The broker then runs its own algorithm over the tracked data to decide which ad to present on the publisher's page. This tracking practice poses a significant threat to the privacy of users as research has shown that it is often easy to link tracked information with an individual's personally identifying information (PII) [158, 159].

Outline. The rest of this chapter is organized as follows. Section 4.2 introduces the key ideas underlying ObliviAd and defines the privacy and system goals of non-tracking behavioral advertising. Section 4.3 presents an overview of the ObliviAd architecture. In Section 4.4, we describe the cryptographic constructions and systems underlying ObliviAd. We analyze the performance of our system in Section 4.5. In Section 4.6, we conduct a formal security analysis. Section 4.7 discusses the related work.

4.2. Key Ideas

Building an OBA system that has the potential to meet approval from brokers must satisfy several criteria. Meeting them individually is easy but achieving a system that combines all the properties mentioned below is a challenging task. For instance, we cannot expect brokers to change their established OBA systems on a large scale; yet, we have to significantly improve the privacy properties of users.

Practicality and reusability of established broker infrastructure and techniques. We design ObliviAd such that the broker infrastructure can mostly be reused. More precisely, brokers will need to add to their existing infrastructure special hardware that enables the trustworthy and privacy-preserving distribution of advertisements. This, however, is the only change to the overall system. Every other point in the system such as the bidding and ad selection remain unchanged. Furthermore, ObliviAd supports fraud prevention such as click-fraud. The key insight is that we obtain privacy without relying on anonymous channels. Consequently, all techniques that detect fraudulent behavior without relying on the advertisement itself but rather on the click behavior work virtually out of the box.

Overall, ObliviAd does not introduce significant computational overhead or financial costs to either the users or the broker. We avoid fancy cryptography at the user's end as we want our solution to work even for web browsers on mobile devices. ObliviAd's design is scalable, since brokers can utilize multiple secure coprocessors, thus improving the performance without altering the privacy properties of our system. We believe that the privacy guarantees that the broker may demonstrate to the privacy-conscious user base make the investment into ObliviAd a worthwhile venture.

Private information retrieval. Users want to retrieve targeted advertisements from brokers in a privacy-preserving manner. A natural candidate for this task is a private information retrieval (PIR) technique. Following the approach proposed by Williams and Sion [223], we implement our PIR protocol using a secure coprocessor (*SC*). *SC*s such as the 4765 cryptographic coprocessor by IBM [139] are independent computers that run on a PC extension card. They are protected by many hardware mechanisms so that it is not possible to extract information stored on the card, for instance, via cold-boot attacks [132]. They run a dedicated operating system and are generally able to execute arbitrary code. Hardware-based PIR constitutes the first practically usable PIR construction and additionally offloads most of the computational costs to the hardware. In particular, it is feasible for users to access the PIR from mobile devices such as cell phones.

*SC*s offer a remote code attestation procedure that allows clients to verify which code is being executed [201]. This prevents brokers from changing the code run by the *SC* to leak information. In a nutshell, this remote attestation mechanism ensures the privacy of data as well as the integrity of the computation, even if the broker is malicious.

Hardware-based PIR relies on *oblivious RAM* (ORAM) that enables the *SC* to query advertisements from the broker's advertisement database without revealing which advertisement was retrieved. While fetching an ad, the user sends her profile in encrypted form to the *SC* that resides on the broker side. The *SC* runs an ORAM protocol to retrieve the candidate advertisements. The broker may learn the identity of the user but not the user profile. The *SC* selects the advertisement that best fits the user profile according to the algorithm specified by the broker.

We build on a state-of-the-art ORAM protocol [199], to prevent the broker from learning any information about the selected advertisement. Since ORAM emulates memory, ORAM protocols can only handle one entry per keyword, that is, only one stored piece of information per memory address. We modify the ORAM scheme to handle multiple entries per keyword, i.e., multiple, different advertisements can be stored and retrieved for a single keyword, and we prove our modifications secure. The advertisement is finally shipped in encrypted form to the user along with a fresh electronic token.

Token-based billing process. Every advertisement that is delivered to a user also contains an electronic token, i.e., a signed piece of data comprising a sequential timestamp and the symmetrically encrypted advertisement id. The creation of such tokens causes only a minimal computational overhead. As soon as the ad is clicked or viewed, depending on the business model, the token is sent back to the broker. After gathering a sufficiently large number of tokens, the broker passes the tokens to the *SC*, which decrypts, mixes, and finally publishes them in order to charge advertisers. The tokens themselves do not reveal for which advertisement they were issued and mixing prevents the final output from being linked to the input tokens. The timestamps prevent brokers from having the *SC* conduct a decryption and mix process twice with a set of tokens that differ only in a few places and consequently identify the advertisements seen by clients.

Privacy and system goals. Our privacy-preserving OBA system meets the following privacy goals [192]:

Profile Privacy. The broker cannot associate any unit of learned information (e.g., clicked ads) with any user PII (including the network address).[1]

Profile Unlinkability. The broker cannot associate separate units of learned information with a single client.

The former property is analogous to vote privacy in the setting of electronic voting, i.e., the impossibility of associating a vote in the final tally with a specific voter. The latter property prevents a broker from building up a user profile and then associating it with a known user by using externally gathered information. For instance, even if the broker knows that two users have seen two ads each and the four ads comprise two car ads and two football ads, the broker does not know whether the two car-ads were seen by the same user or not, i.e., whether or not the two users have disjoint interests.

In principle, both privacy goals can be trivially satisfied by any anonymous browsing solution [209, 10]. Existing anonymity networks, however, have two drawbacks: they do not provide adequate performance (ads should be displayed almost instantaneously) and they make users unaccountable [57], implications of which are not acceptable to the ad industry.

Besides the properties above, we additionally satisfy the following system properties.

Client-side Fraud Detection. The likelihood of detection of clients' malicious behaviors should not decrease as compared to existing systems.

Click Success Measures. Computations of success measures such as click-through rate [110] or click-probability [192] should be possible on the broker's or the client's side.

Performance. Privacy-preserving mechanisms should not hamper the system performance and the auction mechanism should achieve close-to-ideal ranking of ads.

[1] Reznichenko, Guha, and Francis [192] term this property anonymity. We intentionally use profile privacy since users need not be anonymous in ObliviAd.

4.3. Protocol Overview

We first define our attacker model that we intend to secure our OBA system (Section 4.3.1) and we introduce the cryptographic concepts that are deployed by ObliviAd (Section 4.3.2). We then describe the cryptographic assumptions and requirements of the system (Section 4.3.3). Finally, we present a high-level protocol description and discuss the most important properties of the ObliviAd architecture (Section 4.3.4).

4.3.1. Adversary Model

We assume an active adversary with read and write capabilities on the public network, on the SC-to-database bus, and on the database itself. The adversary exercises full control over the broker[2] and the publisher can issue arbitrary requests to the secure coprocessor, obtain the respective response, and observe the resulting operations on the database. The trusted computing base is limited to the management of key material within the SC, i.e., we assume that secret keys are not leaked. The integrity of the code executed by the SC can be enforced, since modern secure coprocessors offer a remote code attestation procedure [201] that gives clients the ability to verify that the SC is executing a specific code. In our architecture, this server-side code is made public for peer scrutiny.

Unlike other privacy-preserving advertising systems [129] where brokers are assumed to be honest-but-curious, we do not make any assumptions about them. We also allow the attacker to arbitrarily corrupt or create client principals and act on their behalf.

In the case of the user clicking on the retrieved ad, we have to assume that the advertiser and the brokers are not colluding; profile privacy is otherwise impossible without using an external anonymity solution such as Tor [209] or Anonymizer [10], since the client reveals her identity to the broker when retrieving an advertisement and clicking on it reveals her identity along with the retrieved advertisement, i.e., her profile, to the advertiser.

4.3.2. Preliminaries

Digital signatures and encryption schemes. Our construction requires an existentially unforgeable digital signature scheme [124] and an authenticated encryption scheme (for instance, INT-CTXT and IND-CPA secure [46] or IND-CCA2 secure [44, 92]).

We do not rely on any particular digital signature or encryption scheme. In fact, our construction is fully parametric in these two cryptographic primitives, as long as they satisfy the respective security definitions.

[2] Since users cannot select which broker will deliver their ads, it is perfectly reasonable to consider the broker to be a malicious (rather than honest but curious) party. In fact, users are usually not even aware of the identity of the broker that is serving them ads.

Oblivious RAM (ORAM). Oblivious RAM was originally devised to protect the access pattern of software on the local memory and thus to prevent the reverse engineering of that software [120]. The observation is that encryption by itself prevents an attacker from learning the content of any memory cell but monitoring how memory is accessed and modified still leaks a great amount of sensitive information.

In the ORAM model, the processor executing a program is considered a black box, i.e., it is impossible to observe the processor's internal state, internal storage, and internal operations. The external storage and the bus connecting that storage with the processor are observable and ORAM schemes use sophisticated combinations of data structures and cryptographic operations to mask their access pattern on the external storage. In our construction, the ORAM scheme is running on a secure coprocessor (SC), which enforces the black-box characteristics. Clients contact the black box via a secure channel (e.g., TLS) to prevent an attacker from obtaining any information on the requested operation. The ORAM storage is organized as a data structure such that every entry contains a keyword kw and a payload (i.e., an ad and possibly additional information in our case). ORAM schemes export two methods, namely $\mathsf{Read}(kw)$ and $\mathsf{Write}(kw, ad)$. The former returns the list of ads associated with kw and, for access privacy reasons, removes the corresponding entries from the data structure; the latter adds the entry (kw, ad) to the data structure.

Private information retrieval. Private information retrieval schemes allow a client to access a database stored on a server, while hiding the query and the resulting answer from the database [87]. ObliviAd uses a PIR scheme to allow the client to download relevant ads from the broker, without the broker learning any information about such ads or the user profile. Following the approach by Williams and Sion [223] we implement a PIR scheme using ORAM over a secure coprocessor. However, as we discuss in Section 4.4 below, the previously mentioned ORAM constructions are not useful for OBA systems. Our construction instead builds on the ORAM protocol recently developed by Shi et al. [199]. We modify this scheme to fit our needs. In particular, we require the ORAM to run on a SC and to associate multiple ads with single keywords.

This solution is also well-suited for clients with only little computational power such as cell phones and netbooks, because the main work (i.e., the cryptographic operations) is performed by the SC, which has dedicated cryptographic hardware and resides on the broker's side; the client merely has to establish a secure connection to the SC, send the query, and receive the result.

Electronic tokens. Intuitively, electronic tokens are the digital equivalent of real-world money, i.e., it is impossible or, at least, computationally infeasible to fake them; a token by itself reveals neither its spender nor what it was spent on; and double-spending a token is detectable. In our construction, electronic tokens enforce the correct billing of the advertiser, while preventing brokers from tracking the respective user.

Electronic tokens may resemble electronic coins [84, 81]. Our tokens, however, are purely based on highly efficient symmetric encryption and digital signature schemes. Intuitively, a token contains a timestamp to prevent double-spending and an identifier that associates this token with the corresponding advertisement; the encryption keeps the identifier, i.e., the user profile, private, the signature prevents forgeries.

Mixing. The concept of mixing was introduced by Chaum [80]. Here, we use it to prevent the attacker from learning the correlation between the content of electronic tokens and the respective users. Specifically, the broker provides the *SC* with a set of (symmetrically) encrypted tokens containing the ad identifiers. The *SC* decrypts those tokens on behalf of the broker. It also randomly permutes (or mixes) the resulting ad identifiers in order to maintain profile privacy and profile unlinkability.

4.3.3. Cryptographic Assumptions and Requirements

We assume a publicly verifiable binding between the *SC* and its public key. Such a binding is easily possible with a standard public key infrastructure (PKI) such as VeriSign [216]. Using this binding, the client software can establish authenticated and encrypted TLS connections with the *SC*.

Our broker-side code must be executed in a trusted environment, and it requires a rich set of operations, e.g., file I/O, data structure management, TLS connections, digital signatures, and authenticated encryptions; thus, we need a programmable *SC* [202]. Furthermore, to guarantee that an *SC* is executing a correct program, we also expect a remote attestation capability from the *SC* [201].

4.3.4. Protocol Overview

We now overview the cryptographic protocol underlying ObliviAd.

1. The advertisers initiate the protocol by uploading their ads ad, the corresponding keywords kws, and other information (e.g., bids) to the broker's server. The broker forwards these triples to the *SC* along with unique ad identifiers. The *SC* includes these tuples in the ORAM structure, i.e., it stores tuples containing the advertisement ad, the corresponding identifier $\mathcal{M}_{ad}$, and one keyword kw_{ad} in encrypted form on the broker's server. Notice that for every advertisement-keyword pair, there is a distinct triple in the ORAM structure. Publishers that are interested in showing ads on their webpages must also register with the broker.

2. When a user visits a publisher's webpage containing an ad box, the broker's client program on the user machine is invoked. This program maintains the user's profile in the form of keywords kws_U based on the user's online behavioral history and sends those keywords to the *SC* over a secure and (server-side) authenticated channel.

3. The *SC* then searches the ORAM structure for kws_U, collects the resulting ads, and selects a subset according to the ranking algorithm specified by the broker, which usually takes into account a number of factors, such as bids, click-probabilities, and so on. Finally, the *SC* attaches an electronic token to each selected ad for the

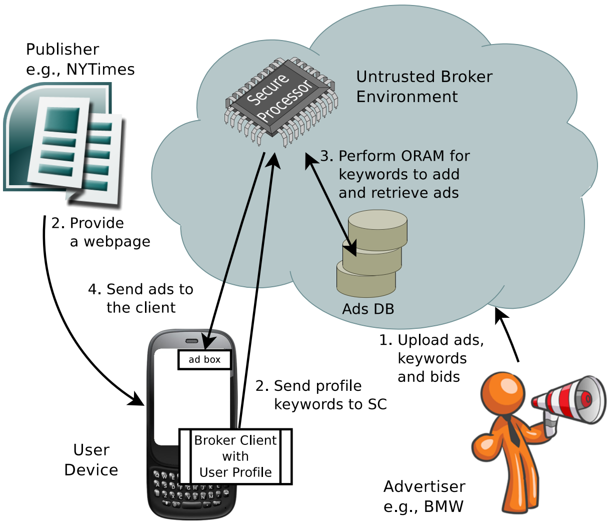

Publisher
e.g., NYTimes
Secure Processor
Untrusted Broker Environment
3. Perform ORAM for keywords to add and retrieve ads
2. Provide a webpage
4. Send ads to the client
Ads DB
1. Upload ads, keywords and bids
ad box
2. Send profile keywords to SC
Broker Client with User Profile
User Device
Advertiser e.g., BMW

Figure 4.1.: Distribution phase.

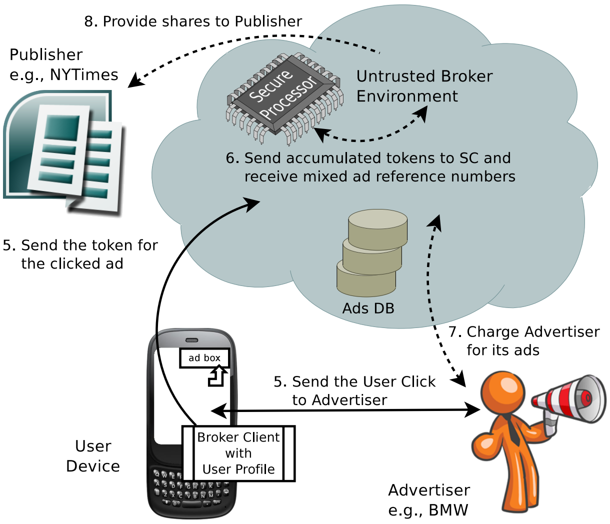

8. Provide shares to Publisher
Publisher
e.g., NYTimes
Secure Processor
Untrusted Broker Environment
6. Send accumulated tokens to SC and receive mixed ad reference numbers
5. Send the token for the clicked ad
Ads DB
7. Charge Advertiser for its ads
ad box
5. Send the User Click to Advertiser
Broker Client with User Profile
User Device
Advertiser e.g., BMW

Figure 4.2.: Tallying phase.

future accounting. The electronic token for the advertisement ad is of the form $sig(enc(\mathcal{M}_{ad}, kw_{ad})_k, t)_{sk_{SC}}$, that is, it consists of a digital signature produced by SC on (i) the ciphertext obtained by encrypting the ad's identifier $\mathcal{M}_{ad}$ and the corresponding keyword kw_{ad} with a symmetric key k, which is chosen by the SC and kept secret, and on (ii) a timestamp t (or, alternatively, on an increasing number).

4. Once the SC has finished its processing, it sends the retrieved ads and associated tokens to the client software over the secure and authenticated channel. The client software then presents a selection of these ads to the user.

5. When the presented ad is viewed by the user in the PPV model or is clicked in the PPC model, the client software sends back the token to the broker server.

6. Following the mixing methodology, the broker server accumulates the tokens over a predefined billing period and sends the set of accumulated tokens to the *SC*. The *SC* removes duplicates (i.e., tokens with the same timestamp), removes the tokens with timestamps outside of the current billing period, decrypts the ads in the remaining tokens, and publishes a random permutation thereof.

7. The broker then distributes these identifiers to the corresponding advertisers and charges them accordingly. The ad keywords retrieved from the tokens may be used for improving future auctions, e.g., for further click-through analysis.

8. Finally, the broker provides revenue shares to publishers.

4.3.4.1. Discussion

Role of the client. Similarly to other privacy preserving OBA architectures (e.g., Adnostic [210] and Privad [129]), we assume that user profiles, encoded as sets of keywords, are created and managed in the broker's client software, which is envisioned as a browser extension on the user's device. The client monitors user behavior (that is, the user's browsing, ads viewed and clicked and so on) in order to create and maintain the user profile. We assume that the client is not compromised and specifically does not leak the keywords or the ads to the broker. Although our system is flexible in the choice and implementation of the client, privacy preserving browser-based mining of core interests can be performed by using, for instance, the recently introduced RePriv platform [114].

Role of the *SC*. The secure coprocessor establishes a secure program execution environment on the server. The *SC*'s public key is certified by a PKI to ensure that the user is indeed communicating with the *SC*; a remote code attestation procedure [201] ensures the user that the correct program is running.

Privacy of the user profile. User profiles are transferred to the *SC* in encrypted form over secured channels, and are therefore not visible to any third party. The ORAM architecture on the broker's side ensures that not even the selection of ads leaks any information about user profiles to the broker. The broker may learn which electronic token was processed by which user, since we do not assume anonymous channels. Still, the broker cannot learn which electronic token corresponds to which ad, thanks to the mixing performed by the *SC*. It is interesting to observe that the degree of privacy of user profiles is determined by the number of electronic tokens that are provided by non-compromised clients in the respective mixing procedure, given that the tally of the ads must be made public for billing purposes. If all other electronic tokens are provided by the attacker, the user profile privacy cannot be guaranteed. This is reminiscent of electronic elections, where the privacy of the user vote cannot be guaranteed if all other voters are under the control of the attacker, given that the final tally is public [96].

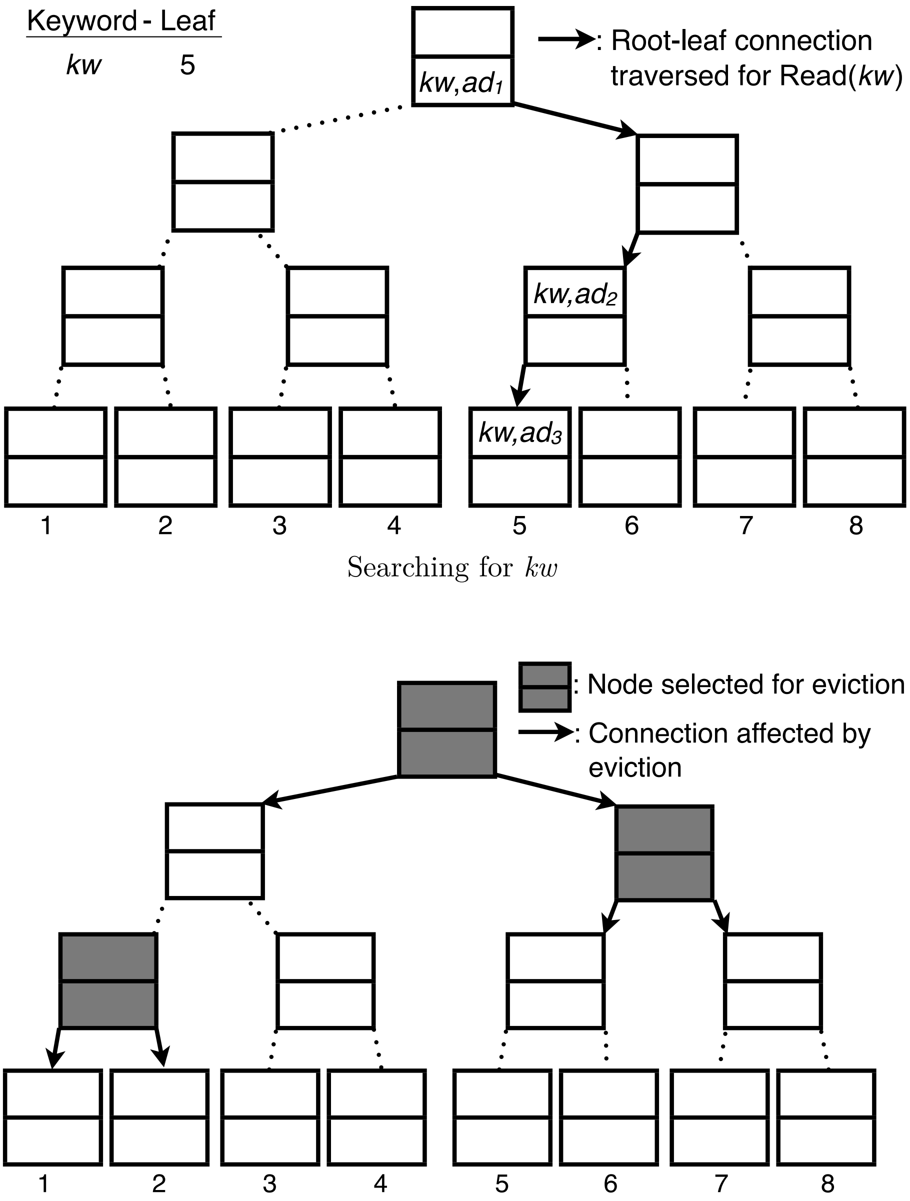
Notice that a malicious broker could in principle derive a particular user profile by allowing only the tokens of that honest client to reach the SC. The resulting bill would reveal the user profile, thus breaking the desired privacy property. For protecting client profiles unconditionally, the usage of anonymous channels is indispensable, a solution we do not advocate due to its computational cost and network delay. Typical brokers, however, behave rationally, i.e., their primary goal is to excel in commerce rather than to identify users at all costs. Intuitively, our scheme protects the privacy of the user profile against rationally-behaving brokers: excluding user tokens from the tally leads to a significant monetary loss, since the timestamp mechanism prevents those tokens from being counted in the next tallying periods.

Profile unlinkability. Not only is the broker unable to learn which user has seen which ad, she cannot even learn whether or not two or more ads were seen by the same user. This property is enforced by (*i*) the structure of the electronic tokens, which are unlinkable and do not reveal any information about user profiles, and (*ii*) the mixing, which breaks the correlation between the list of tallied ads and the list of received tokens. Breaking this correlation is crucial since the attacker may learn the correlation between tokens and clients by looking at the traffic on the non-anonymous communication channel between clients and SC.

Billing correctness. The timestamp mechanism also serves the purpose of ensuring the correctness of the billing process, since each ad cannot be counted more than once in the final tally and it is counted only if the client has forwarded the electronic token to the broker, i.e., the user has viewed (or clicked) the ad.

Click-related information. Without using an anonymous browsing solution, which we do not want to adopt for efficiency reasons and for the sake of click-fraud detection (see below), it is impossible to prevent the advertiser from learning which user clicked which ad. In practice, the broker may try to collude with the advertiser to obtain this information. If OBA is the only goal of the broker, however, there is no motivation for the broker to determine which user clicked a particular ad. The information required to hold auctions, such as the click-through rate or the click-probability of an ad, can be derived by the broker from the tally produced by the SC.

Click-fraud detection. In our design the interaction pattern between clients and brokers remains almost unchanged and the broker is still notified when the client clicks on an ad, although she does not know which ad was clicked. Thus, real-time click-fraud detection mechanisms, which typically monitor the click-ratio of each user, continue to work. Offline detection mechanisms are expected to continue to work as well, since they are typically enforced on the advertiser side and, in our architecture, advertisers know who clicked their ads.


Keyword - Leaf
kw 5
kw,ad1
: Root-leaf connection traversed for Read(kw)
kw,ad2
kw,ad3
1
2
3
4
5
6
7
8
: Node selected for eviction
: Connection affected by eviction
1
2
3
4
5
6
7
8

Searching for kw

Eviction with $\nu = 1$

Figure 4.3.: ORAM Operations for a bucket size of 2 and a tree depth of 4.

4.4. ORAM Construction

We adopt the ORAM scheme by Shi et al. [199], which we modify to fit the OBA setting. We first review their basic construction in Section 4.4.1 and we discuss our modifications in Section 4.4.2.

4.4.1. ORAM Scheme by Shi et al.

Shi et al. propose three different constructions. In this work, we adopt their "basic construction with trivial buckets". The complete database is stored as a binary tree. Every node in the binary tree is a bucket ORAM, i.e., an array of entries. Each entry consists of a keyword kw and a payload in encrypted form. An authenticated encryption scheme is used to prevent an active attacker from learning and modifying entries in an unobservable way.

Initially, the bucket is filled with dummy entries. The ORAM scheme manipulates buckets via the $\mathsf{ReadAndRemove}(kw, b)$ and $\mathsf{Add}(b, e)$ operations. Both traverse the complete bucket b, reading and decrypting every entry e_{old}, and replacing e_{old} with a new entry e_{new}. In case of $\mathsf{ReadAndRemove}(kw, b)$, this new entry is either a re-randomization of e_{old} if the keyword does not match e_{old}, or an encryption of a dummy entry otherwise. In case of $\mathsf{Add}(b, e)$, this new entry is either e if e_{old} is a dummy entry and e has not been stored yet, or a re-randomization of e_{old} otherwise. The IND-CPA indistinguishability property of the encryption scheme ensures that the adversary cannot differentiate between a dummy entry and an entry comprising a kw and a payload.

The ORAM construction maintains the invariant that the entries for every keyword kw are located on a unique path from the root node to a leaf ℓ assigned to that keyword. This keyword-leaf (kw, ℓ) assignment is securely maintained in the ORAM (in the SC in our case).

To read the entry for a keyword kw, the Read operation (see Section 4.3.2) traverses the complete path from the root node to the leaf assigned to kw, searching for a bucket containing kw (see Figure 4.3). The entry for kw is removed from that bucket using $\mathsf{ReadAndRemove}$ bucket operations and stored in the ORAM program. Finally, the program assigns a new randomly chosen leaf ℓ' to the keyword kw, and moves the retrieved entry from its internal memory to the root node using the Add bucket operation. Any subsequent query for the same keyword kw is indistinguishable from queries to other keywords as the search paths are randomly distributed in the tree.

To write a new entry for the keyword kw, the Write operation first retrieves the entry stored for that keyword and drops it. This enforces that at most one entry per keyword is present in the database. The new entry is added to the root node.

If we keep on adding entries to the root bucket, it will eventually overflow. To prevent that, a background *eviction* process continuously moves entries from the root towards their designated leaves. After every query and on every tree level (starting from the root towards the leaves), a constant number ν of buckets is randomly chosen and evicted (a bucket can be chosen multiple times during one eviction phase). One entry $(kw, payload)$ in a chosen bucket $\mathcal{N}$ is removed and written to the next bucket on the path towards the leaf ℓ assigned to kw; the other child bucket of $\mathcal{N}$ is re-randomized; if $\mathcal{N}$ contains only dummy elements, both of its child buckets are re-randomized. To make this operation oblivious, the order is fixed and the left child is always processed before the right child. Figure 4.3 provides an example of the eviction process for $\nu = 1$. Notice that it is not necessary for the security of the scheme to perform an eviction after every ORAM operation and, in fact, eviction is only necessary to prevent bucket overflows[3] and to guarantee the performance of the scheme, although the eviction process must, of course, be performed in an oblivious way.

We now state the security property for ORAM schemes.

Definition 4.1 (ORAM security [199]). *A data request sequence $\vec{x}$ is an operation-argument tuple sequence $\vec{x} = ((op_1, arg_1), \ldots, (op_\ell, arg_\ell))$ where $arg_i = kw$ if $op_i = \mathsf{Read}$ and $arg_j = (kw, ad)$ if $op_j = \mathsf{Write}$ for keyword kw and data ad. We let $ops(\vec{x}) := (op_1, \ldots, op_\ell)$ and $\mathcal{A}(\vec{x})$ denote the access pattern resulting for the execution of the data request sequence $\vec{x}$.*

An ORAM construction is secure if and only if for every two arbitrary data request sequences $\vec{x}$ and $\vec{y}$ such that $ops(\vec{x}) = ops(\vec{y})$, the access patterns $\mathcal{A}(\vec{x})$ and $\mathcal{A}(\vec{y})$ are computationally indistinguishable.

The ORAM scheme by Shi et al. is secure according to that definition.

Theorem 4.1 (ORAM properties [199]). *The data structure by Shi et al. is a secure oblivious RAM with an $\mathcal{O}(\log^2 N)$ worst-case and average-case time complexity and program storage of $\mathcal{O}(K \log N)$ size, where N is the number of entries in the database and K is the number of keywords.*

4.4.2. Adapted Construction

In OBA, there can be multiple ads for every *kw*. It is therefore natural to have multiple entries for a keyword *kw* in our database, and we need to obtain all of them while searching (i.e., executing the Read operation) for a keyword *kw*. Note that the ability to store and retrieve multiple elements per keyword is not definitional to ORAM. In fact, ORAM designs based on the so-called square-root solution [123, 125] cannot retrieve more than one entry per query from their ORAM architecture. In contrast, the construction by Shi et al. can retrieve multiple entries associated with a *kw*, which, together with its good worst-case complexity, is the reason we chose it as ORAM scheme. We now explain how the ORAM data structure described above can handle multiple entries per keyword without hampering the access privacy.

In the ORAM construction, it is possible to retrieve (Read) all the entries for a specific keyword while the *SC* goes through the complete path (from the root to the leaf) assigned to the keyword: we modify the ReadAndRemove operation to store these entries in the ORAM memory and removing from traversed buckets. The subsequent Write operation stores the retrieved entries back into the root bucket. We stress that these modifications do not affect the cryptographic operations performed in the query processing, but only the amount of retrieved and written data.

Like the original ReadAndRemove (resp. Add) operation, the adapted ReadAndRemove (resp. Add) operation also modifies the complete bucket; thus, this operation remains secure as the IND-CPA property of the encryption scheme prevents the attacker from learning the

[3] Should a bucket, however, overflow at some point, it leaks information. Such overflows, therefore should be prevented by regularly evicting the tree.

number of entries which have been replaced by (resp. have replaced) dummy entries. As a result, there is no difference in the adversarial view of ORAM from a privacy perspective between the original ORAM construction and our variant.

One further important difference is that the ORAM client is replaced by a SC in our setting, as we use ORAM to efficiently perform PIR. As a result, during an ORAM Read operation, we cache the retrieved entries inside the SC internal memory before sending the response to the client and putting (via the Add operation) the retrieved entries into the root node bucket.

Let K be the number of keywords in the system. Let N be the number of keyword-advertisement entries $(kw, \mathcal{M}_{ad}, ad)$ to be stored in the database, where every entry is of size B bits. Note that N is generally greater than the number of ads as there can be multiple keywords *kws* attached to an *ad*. Therefore, the database size D is equal to $N \cdot B$, which we expect to be in the order of several GB or even a couple of TB for some brokers. Further, we expect the server storage n to be larger than D due to the overhead imposed by the ORAM construction. We also expect the SC to have an internal storage of size $m = \Omega(K \log N)$, which we use to store a mapping between keywords and leaves: we need to associate K keywords with leaves and we need $\mathcal{O}(\log N)$ space to describe a leaf.

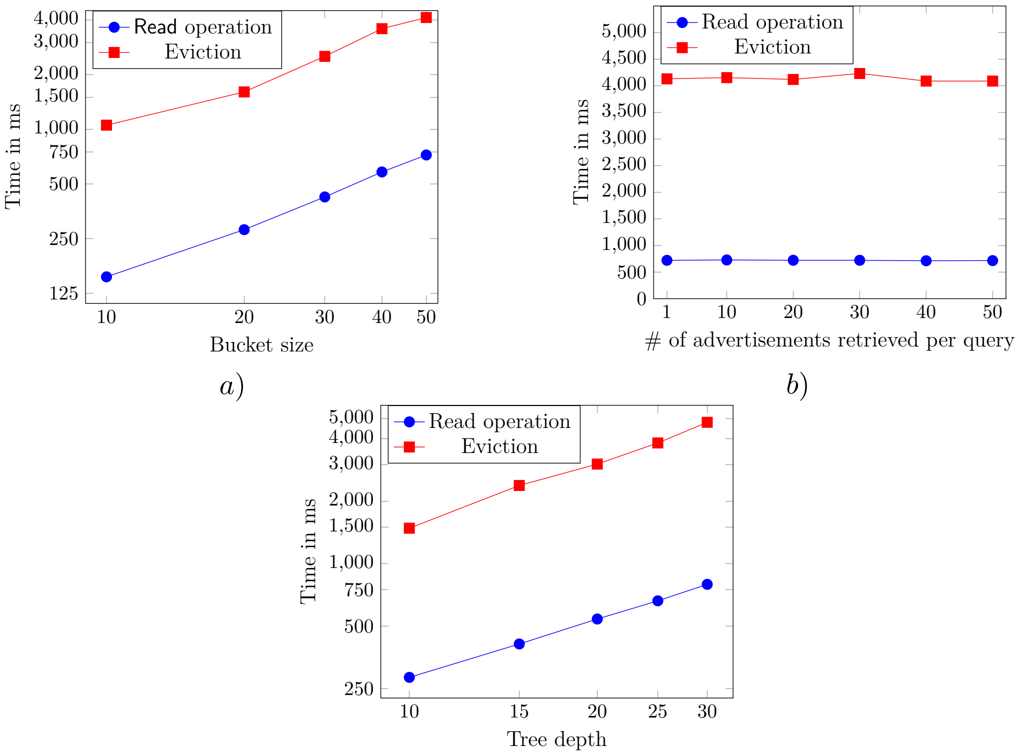
The tree contains $\mathcal{O}(N)$ nodes and we let the ORAM buckets be of size $\mathcal{O}(\log^2 N)$, implicitly bounding the maximum number of entries per keyword to $\mathcal{O}(\log N)$. As a result, our Read and Evict operations take $\mathcal{O}(\log^3 N)$ time, while the write operation takes $\mathcal{O}(\log^2 N)$ time. Intuitively, Read and Evict operate on each level of the tree (from the root to the leaves) a constant number of times, while Write operates on the root only.

Theorem 4.2 (Properties of the adapted ORAM scheme)**.** *The data structure by Shi et al. with the modifications detailed above is a secure oblivious RAM with an $\mathcal{O}(\log^3 N)$ worst-case and average-case time complexity and an $\mathcal{O}(K \log(N))$ SC storage requirement.*

Proof (Sketch). The ORAM property follows from the ORAM property of the original scheme [199]. The required decryption and encryption operations are performed also in the original version and our modifications are not distinguishable for the attacker thanks to the IND-CPA property of the encryption scheme. □

4.5. Performance Analysis

In this section, we describe the implementation and evaluate the practicality of our ORAM construction, which dominates the computational cost of our solution. We also suggest some optimizations based on our analysis and discuss other important system factors of our solution.



a)

b)

c)

Figure 4.4.: Results of our microbenchmark. For experiments *a*) and *b*), the tree depth is fixed to 24. For experiment *c*), the bucket size is fixed to 30.

4.5.1. Implementation

We have developed a prototype to demonstrate the feasibility of our construction. Our implementation is a single-threaded application, comprising approximately 1200 lines of Java code, which performs the operations that in a concrete implementation of our system would be performed by the *SC*.

In our implementation, we assume that the ranking and selection of ads is conducted inside the *SC* [189]; note that it would, in principle, be possible to shift these actions to the client side, if required [192]. We implemented buckets as arrays. The array size is determined at compile-time. Every array slot holds an advertisement and the corresponding keyword. Typically, buckets are only a few MB in size and fit easily into the *SC* memory. Therefore, we encrypt at the bucket level and not at the slot level. Consequently, if the advertisements differ greatly in size, we must apply a padding before encrypting a bucket.

4.5.2. Experiments

All experiments were conducted on a commodity PC with an Intel i5 quad-core processor with 3.3 GHz and 8 GB RAM. The hard drive has a speed of 7200 RPM and a cache of 16 MB. We implement the authenticated encryption scheme with AES encryption and HMAC. The cryptographic implementation was provided by the standard SunJCE Provider. To get consistent and comparable results, we set the advertisement size to 20 KB and fix the tree depth to 24,[4] and report the average of 100 repetitions of the following experiments:

- We measure the impact of the bucket size (in terms of array slots) on the overall performance of our system. Figure 4.4 *a*) displays the time required to read an advertisement from the database and the time required for the eviction process for a bucket size varying between 10 and 50 entries; the tree depth remains unchanged at 25.

  For a reasonable bucket size of 30, we are able to read advertisements for a keyword in 424 ms. Even for a large bucket of size 50, we only require 750 ms where more than 85% of the time was spent on the cryptographic operations.

  The eviction process takes longer (in between 1 s and 4.1 s, depending on the bucket size) but is performed after the reply is sent and, therefore, not experienced by the user.

- We measure how the number of ads retrieved in a single query influences the system performance. We fix the bucket size to 50 (the tree depth remains unmodified) and we store various amounts of advertisements for a single keyword inside the ORAM. Figure 4.4 *b*) depicts the time required to read all the advertisements for the given keyword and the time spent by the eviction process. The results show that the number of ads does not affect the retrieval time.

[4] A tree with depth 24 stores more than 16 million advertisements. Given an average advertisement size of 20 KB, the database stores over 300 GB of advertisement data.

- We show the scalability of our approach and fix the bucket size to 30 and let the depth of the tree vary from 10 to 30 entries, i.e., we let the number of advertisements stored in the tree vary from one million to over one billion. Figure 4.4 *c*) depicts the obtained timings; the experienced delay increases linearly from 280 ms for one million advertisements to 780 ms for one billion advertisements.

4.5.3. Discussion

Impact on the user-experienced delay. The experiments show that the Read operation requires up to 750 ms and the Evict operation requires up to 4.2 s. As the eviction is not necessary for achieving security, a client does not have to wait for the eviction process to finish. We can deliver the retrieved ad as soon as the Read process has terminated, increasing the overall delay of our system only by the amount of time required by a Read operation.

Our experimental results show that on a commodity PC with the cryptographic operations performed in software, our implementation requires, depending on the bucket size, between 150 ms and 728 ms. More than 85% of that time is spent in the various cryptographic routines. A dedicated hardware implementation as available in an *SC* will further decrease the time required to retrieve an advertisement and increase the performance of our system.

We use a secure and authenticated link between user devices and the *SC* to privately download the ads. Comprehensive studies [89] show that this delay is negligible compared to the ORAM-induced delay and, as *SC* CPU speeds increases, this delay will drop even further. The generation of an electronic token takes only 1.4 ms using RSA signatures and also constitutes a negligible overhead in comparison with the ORAM computations. Since the final tallying among the broker, publishers, and advertisers remains virtually identical to the existing system, the overall experienced user delay is dominated by the delay induced by the ORAM scheme.

Other system delays. We determined that individual token verification operations take only 0.08 ms for RSA signatures. In addition, batch verification techniques [45] can further improve the overall verification performance during the tallying phase. Thus, our billing system can quickly process large amounts of electronic tokens.

Replication and concurrency. The bottleneck of our construction is the *SC* fetching an advertisement from the database. Replications of the database and resulting concurrent computations can significantly improve the performance. It is possible for the broker to employ multiple *SC* units so that each of them maintains its own replicated copy of the database and caters to a different set of users in a completely parallel fashion. This replication does not affect the profile privacy of the users as their network addresses are known to the broker anyway. Realizing such a replication is harder to achieve in anonymity-based solutions as opposed to privacy preserving OBA (e.g., Privad).

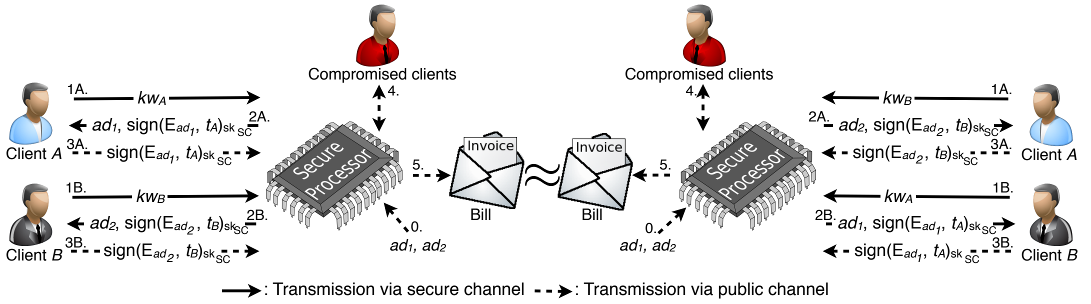

Compromised clients
4.
1A.
kw_A
Client A
2A.
ad_1, sign(E_{ad_1}, t_A)$_{sk_{SC}}$
3A.
sign(E_{ad_1}, t_A)$_{sk_{SC}}$
1B.
kw_B
Client B
2B.
ad_2, sign(E_{ad_2}, t_B)$_{sk_{SC}}$
3B.
sign(E_{ad_2}, t_B)$_{sk_{SC}}$
Secure Processor
0.
ad_1, ad_2
5.
Invoice
Bill
≈
Invoice
Bill
5.
0.
ad_1, ad_2
Secure Processor
Compromised clients
4.
1A.
kw_B
2A.
ad_2, sign(E_{ad_2}, t_B)$_{sk_{SC}}$
3A.
sign(E_{ad_2}, t_B)$_{sk_{SC}}$
Client A
1B.
kw_A
2B.
ad_1, sign(E_{ad_1}, t_A)$_{sk_{SC}}$
3B.
sign(E_{ad_1}, t_A)$_{sk_{SC}}$
Client B
: Transmission via secure channel
: Transmission via public channel

Figure 4.5.: Overview of the observational equivalence relation for profile privacy. The left side of the picture corresponds to P and the right side to Q.

It is also possible to let multiple *SC* units operate on the same ORAM storage since all ORAM operations consist of bucket-level operations, since they are independent from each other. We just have to let all *SC* units share the same key material and enforce that at most one *SC* unit operates on a single bucket. Consequently, the only concurrency issue that needs to be taken care of is that all collaborating *SC* chips maintain the same copy of the keyword-leaf assignment. In that respect, the only blocking operation is the Add operation at the root node (as it changes the keyword-leaf assignment). At the cost of using memory in the *SC* chip, it is possible to postpone these Add operations, thus improving the performance. Finally, we mention that the eviction procedure is also highly parallelizable since it operates on distinct buckets.

4.6. Formal Verification

We conduct a formal security analysis of our system. Although the deployed cryptographic primitives are secure by themselves, we must ensure the absence of flaws in the protocol design, e.g., unintended or attacker-driven interleavings of concurrently executed protocol sessions that break the security properties. To exclude such flaws and to establish a security proof, we model our protocol in the applied-pi calculus [3]. We formalize privacy properties as observational equivalence relations between processes and correctness properties as trace properties. The verification is automatically conducted using ProVerif [52]. The ProVerif scripts used in the analysis can be found in Appendix B.2.

4.6.1. Profile Privacy

We verify that an attacker cannot obtain any information about client profiles, even when in full control of the advertiser, the publisher, and the broker. We model this property as an indistinguishability game denoted by ≈ between two processes, as depicted in Figure 4.5. Here and throughout the rest of this chapter, we let E_{ad} denote the symmetric encryption $enc(\mathcal{M}_{ad}, kw_{ad})_k$ of the ad identifier $\mathcal{M}_{ad}$ along with the ad keyword kw_{ad}. In the first process P (left-hand side), A and B's profiles consist of the keywords kw_A and kw_B, respectively. In the second process Q (right-hand side), the two profiles are swapped. If the processes P and Q are observationally equivalent, written $\mathsf{P} \approx \mathsf{Q}$, then the attacker cannot learn which profile belongs to which client. We assume a very pessimistic setting where the attacker has the control over arbitrarily many clients and knows the two profiles kw_A and kw_B. More precisely, our game works as follows:

0. The attacker chooses two advertisements ad_A and ad_B and stores them in the SC (this corresponds to the broker filling her ORAM database via the SC). These two advertisements match the two profiles kw_A and kw_B, respectively.

1A./1B. Client A and Client B send their profile to the SC via a secure channel.

2A./2B. The SC sends back the response, comprising the advertisement that best matches the received profile along with the accompanying token.

3A./3B. The two clients publish their token on a public channel.

4. Corrupted clients, i.e., clients acting exclusively on behalf of the attacker, can arbitrarily interact with the SC.

5. After collecting the two honest clients' tokens, and possibly other tokens from compromised clients, the SC initiates the accounting process. The SC verifies the signatures in the tokens, verifies the timestamps, decrypts the ad identifiers, and publishes a permutation thereof, which constitutes the bill.

The attacker has full control over scheduling decisions, e.g., the actions of client A, client B, and compromised clients can be interleaved in any order, with the natural constraint that client A and client B follow the protocol, i.e., their respective actions are executed in the right order.

Theorem 4.3 (Profile Privacy). *The observational equivalence relation* $\mathsf{P} \approx \mathsf{Q}$ *holds true.*

Proof. Automatically proven using ProVerif. □

4.6.2. Profile Unlinkability

When verifying the privacy of the client profiles, we assume the worst case scenario, i.e., the attacker knows the client profiles. We now analyze a different property, namely, profile unlinkability. In this scenario, the attacker does not know the client profiles. We verify that it is impossible for an attacker to deduce any information about client profiles by observing the tokens sent by that client and the final tally, even when in full control of the advertiser, the publisher, and the broker. We model this property as an indistinguishability game between two processes P (left-hand side) and Q (right-hand side), as depicted in Figure 4.6. In both processes, A and B's profiles consist of the keyword kw_A and kw_B, respectively. The game obeys the following steps:

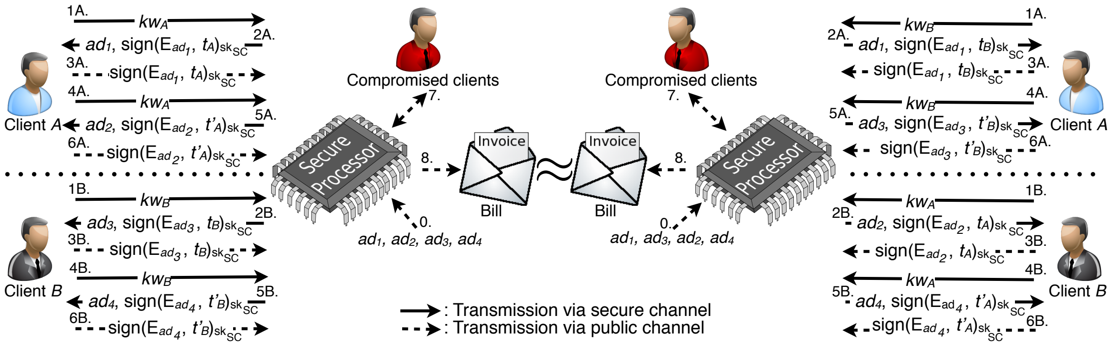


Figure 4.6.: Overview of the observational equivalence relation for profile unlinkability. The left side of the picture corresponds to P and the right side to Q.

0. The attacker chooses four advertisements ad_1, ad_2, ad_3, and ad_4, and stores them in the SC (this corresponds to the broker filling her ORAM database via the SC). In process P, advertisements ad_1 and ad_3 are the best-matching advertisements for kw_A and kw_B, and ad_2 and ad_4 are the second-best-matching advertisements for kw_A and kw_B, respectively.[5] Notice that we assume the worst-case scenario, i.e., the two client profiles kw_A and kw_B are disjoint and, thus, easier to be distinguished. In process Q, ad_2 and ad_3 are swapped.

1A./1B to 3A./3B. Client A and B both send their keyword to the SC via a secured channel and receive back the best-matching advertisements (i.e., ad_1 and ad_3 in P and ad_1 and ad_2 in Q), and the corresponding tokens. Following the protocol, the tokens are immediately sent to the SC.

4A./4B to 6A./6B. Both clients perform the same steps as above. The returned advertisements, however, are the second-best-matching ones (i.e., ad_2 and ad_4 in P and ad_3 and ad_4 in Q).

7. Compromised clients can arbitrarily interact with the SC.

8. After collecting the four honest clients' tokens, and possibly other tokens from compromised clients, the SC initiates the accounting process. The SC verifies the signatures on the tokens, verifies the time stamps, decrypts the ad identifiers, and publishes a permutation thereof, which constitutes the bill.

As in the game for profile privacy, the attacker has full control over scheduling decisions and the above described game steps can be interleaved in any order, as long as the actions of client A and client B follow the protocol. If the processes P and Q are observationally equivalent, then the profiles are unlinkable, i.e., the adversary cannot determine which entries in the final tally were caused by which profile. For instance, if the final tally contains two entries for cars and two for sports, it is impossible to say if each client is interested only in one of the two topics, or if the two clients are both interested in sports and in cars.

[5] We recall that the SC ranks the advertisements according to each user profile.

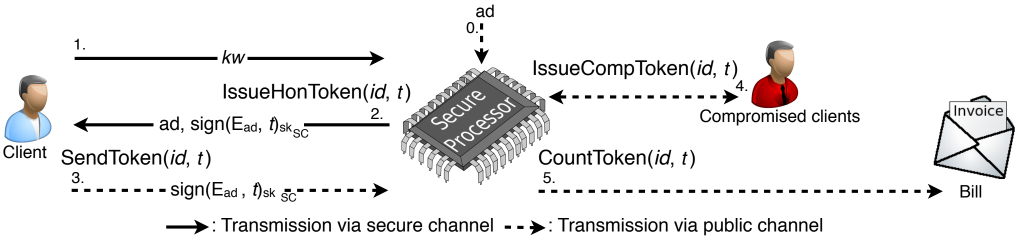


Figure 4.7.: The process P, annotated with logical predicates, used in the verification of the trace property in Equation 4.8.

Theorem 4.4 (Profile Unlinkability)**.** *The observational equivalence relation* $\mathsf{P} \approx \mathsf{Q}$ *holds true.*

Proof. Automatically proven using ProVerif. □

4.6.3. Billing Correctness

A fundamental goal of our system is the correctness of the billing process. Brokers expect to be reimbursed for their services and the advertiser is only willing to pay for advertisements that have been seen or clicked on. A first property we expect is *non-reusability of tokens* (each token is counted at most once). A second property is *billing fairness*, i.e., whenever a token is counted, then the corresponding ad was really clicked (or viewed) by the user. We formalize and verify these two properties in a strong adversarial model, in which the attacker has the control over arbitrarily many corrupted clients. Therefore, we distinguish whether a token $sig(\mathsf{E}_{ad}, t)_{sk_{SC}}$ is issued for an honest client, annotated with the predicate $\mathsf{IssueHonToken}(\mathcal{M}_{ad}, t)$, or for a compromised client, marked with the predicate $\mathsf{IssueCompToken}(\mathcal{M}_{ad}, t)$. Additionally, we decorate the point in the protocol where an honest client views the ad and sends her token with $\mathsf{SendToken}(\mathcal{M}_{ad}, t)$ and the point where the SC counts a token with $\mathsf{CountToken}(\mathcal{M}_{ad}, t)$. Notice that we cannot decorate compromised clients, since they run arbitrary code under the control of the attacker. Figure 4.7 depicts the process P annotated with these predicates. We want to verify that in all protocol executions each CountToken predicate is preceded either by a distinct SendToken predicate, which is in turn preceded by a distinct IssueHonToken predicate (honest clients), or by a distinct IssueCompToken predicate (compromised clients). This kind of properties are known as injective agreement [166]. The billing correctness property can be formalized in ProVerif notation as follows:

$$\begin{aligned} &\mathsf{CountToken}(\mathcal{M}_{ad}, t) ==>_1 \\ &\quad (\mathsf{SendToken}(\mathcal{M}_{ad}, t) ==>_1 \mathsf{IssueHonToken}(\mathcal{M}_{ad}, t)) \\ &\qquad \vee\ \mathsf{IssueCompToken}(\mathcal{M}_{ad}, t) \end{aligned} \tag{4.8}$$

where $P_1 ==>_1 P_2$ denotes the requirement that each predicate P_1 must be preceded by a distinct predicate P_2 in all protocol executions. The above property says that tokens are never counted more than once. For honest clients, we also know that whenever a token is counted, then the corresponding ad has been viewed. Compromised clients cannot be decorated with events, reflecting the fact that a compromised client might forward the token to the broker behind the scenes, i.e., without the user actually having viewed the ad.

Theorem 4.5. *The trace property stated in Equation 4.8 holds true in all possible execution traces of the process* P*.*

Proof. Automatically proven using ProVerif. □

4.7. Related Work

We discuss work and projects related to online behavioral advertising. We start with proposals and opt-out solutions that depend on the cooperation of brokers to a certain degree. We then focus on anonymous browsing solutions, PIR schemes, and ORAM constructions. Finally, we proceed by comparing ObliviAd with other OBA systems.

Consumer group proposals, initiatives, and regulatory reformations. There have been a number of proposals for addressing these privacy concerns. Consumer groups have suggested that the behavioral advertising industry undergo regulatory reforms, with the goal of enhancing users' privacy and data protection by constraining the role and the activities of third-parties such as advertisers and brokers [90, 154]. However, the effectiveness of such a reform is questionable, since it is hard to prevent third parties from passively gathering PII (personally identifying information) and later destroying any traces in the case of a future investigation [155]. Initiatives like Do Not Track [175] enable users to opt out of tracking by analytics services, advertising networks, and social platforms. These initiatives do not solve the core problem, however, as they significantly hamper the economic model of free online services that depend solely on advertising revenue.

Anonymity solutions. Anonymous browsing solutions such as Tor [209] can enforce the desired privacy goals. In fact, truly anonymous browsing prevents any kind of tracking and naturally prevents profile building. Anonymous browsing solutions, however, make detection of client-side fraud (e.g., click fraud[6]) infeasible. Moreover, they are not considered scalable enough to support the existing user base [177]. As a consequence, they do not offer a scalable solution that simultaneously satisfies the financial goals of web services and the privacy requirements of users.

ORAM schemes. Originally introduced by Goldreich and Ostrovsky [123], oblivious RAM and oblivious techniques in general received a growing interest in the research community (e.g., [169, 204, 199]); several highly efficient ORAM schemes have been introduced [206, 205] that superseed the ORAM scheme described in ObliviAd. Since the ORAM scheme in ObliviAd is a modular building block, it is suited to accommodate new and improved schemes. The only requirement that the ORAM scheme must satisfy is the possibility to store and retrieve multiple entries per keyword.

[6] Click-fraud consists of users or bots clicking on ads in order to drive up a given advertiser's costs, a publisher's revenue, the click-through rate of an advertisement, and so on.

PIR schemes. There are many PIR schemes that work in different situations. For instance, PIR can be achieved in either an information-theoretic or a computational setting, or in a single-server or a multiple-server setting (e.g. [186, 119, 15, 87]). The trivial single-server PIR solution is to download the whole database, which is infeasible given mobile devices and the massive amount of advertisements. Other information-theoretic PIR schemes require multiple non-colluding database servers. The non-collusion assumption is certainly inappropriate for broker servers. Computational PIR, instead, relies on cryptographic assumptions, is suitable also for a single database server, and thus better fits our setting. According to recent analyses, however, none of the existing computational PIR schemes significantly outperforms the trivial solution of downloading the whole database [200, 185]. For this reason, we follow the approach proposed by Wang et al. [219, 98]. Hardware-based solutions turn out to be orders of magnitude faster than the other computational PIR solutions. In particular the recent scientific interest in efficient ORAM schemes significantly improved the performance of hardware-based PIR.

OBA systems. Given the significance of the privacy-preserving OBA problem to the masses, the privacy enhancing technologies (PETs) research community is showing a growing interest in this problem [130, 129, 149, 150, 114, 210, 107]. Instead of blocking online behavioral advertising [175] or trying to obscure the user profile [137], these PETs are meant to be practical privacy preserving alternatives that the advertising industry should also find attractive. The solutions proposed so far, however, fall short of providing an adequate degree of privacy and a satisfactory performance at the same time.

Juels [149] was the first to explore the notion of targeted yet privacy-preserving advertising and the first to suggest the usage of client-side proxies to manage user profiles, a concept used by almost all privacy-preserving advertising systems today. Like ObliviAd, Juels utilizes a PIR scheme for privacy-preserving distribution of ads. That PIR scheme, however, is impractical for a real time use and, in particular, makes it impossible to retrieve ads on-the-fly. Furthermore, that work does not secure the tallying phase, in which the broker computes the number of ads viewed by clients for billing purposes. Finally, provable anonymity is achieved against a threshold adversary, which we believe does not model real life brokers accurately. In contrast, we use a highly efficient PIR solution based on a secure coprocessor, which enables us to retrieve ads on-the-fly. We secure the tallying phase using electronic tokens that are mixed by the SC in order to preserve the privacy of the client profiles and we model the broker as a malicious party.

Privad [130, 129] presents a complete system for privacy-preserving targeted advertising. Along with the usual client software and broker entities, Privad introduces a *reference monitor* that watches the client software to ensure that no information is sent by the client to the broker through a covert channel, and a *dealer* that works as an anonymizing proxy between the user and the broker. Similar to our system, the Privad client builds a user profile and requests relevant ads from the broker by sending the profile information to the broker system, with the fundamental difference that every communication in Privad is proxied by the dealer. Further, in Privad, any communication between the client and the broker is encrypted with the broker's public key or an agreed session key so that the dealer cannot see the profile information or the downloaded, viewed or clicked ads.

Privad achieves profile privacy through the anonymity of the client. In order to facilitate client-side fraud detection, Privad does not employ a reliable anonymity solution like Tor [209]; it instead asks for a privacy-preserving proxy (the dealer) that is assumed to be *honest-but-curious* and, specifically, to not collude with the broker. For detecting client-side fraud, however, the dealer needs to satisfy the advertiser's and the broker's demands. Realizing such a party seems to be a difficult problem to solve from the administrative, legal, and technological perspective. Even if realized, the dealer is highly susceptible to traffic analysis attacks and may also introduce a single-point-of-failure in the advertising system. In our solution, we avoid such an entity and completely eliminate the corresponding trust requirement by introducing a broker-side (unbribable) trusted hardware. As a result, we achieve profile privacy without enforcing user anonymity.

Adnostic [210] provides a completely different approach to privacy-preserving OBA: when a user visits a webpage that contains a slot for advertisements, a privacy-protecting client software obtains a small set of ads that are randomly chosen by the broker. The software then selects the most appropriate ad based on the user profile and shows it to the user. For the viewed ads, Adnostic computes homomorphic encryptions and zero-knowledge proofs on the user devices and sends them to the broker. In turn, the broker uses the encryptions and zero-knowledge proofs to reliably settle the accounts with the publishers and the advertisers without knowing who viewed which ad. A fundamental assumption in Adnostic is that the advertisers and the broker will always collaborate; thus, ad-clicks are treated the same as in current ad networks, where the client reports clicks directly to the broker.

Unlike ObliviAd and Privad, Adnostic does not hide users' web browsing or clicking behavior from the broker, which makes their privacy goals considerably weaker. Further, as ads are sent without any behavioral targeting and only in small sets, it seems that this approach would be more comparable to the old scattershot approach than today's modern OBA methods. Further, although their accounting solution is novel and cryptographically interesting, it does not provide any information about the viewed ads. This information is needed for conducting future ad-auctions and click-through analysis; thus, the quality of OBA will not improve, even over a longer period. Finally, our solution is significantly more efficient on both the client's and the broker's end: Adnostic uses fairly expensive public-key homomorphic encryptions and zero knowledge proofs, while we, instead, rely on inexpensive symmetric-key encryptions.

An interesting work in the field of OBA is RePriv [114], an architecture that provides an ideal solution for realizing the client-side software required by virtually all privacy-preserving OBA systems, including ours. Reznichenko, Guha and Francis [192] have recently proposed a solution for running advertising auctions that leverage user profiles for ad ranking without compromising their privacy. We observe that their auction design can easily be incorporated in our system.

5. Conclusion and Outlook

In this thesis, we presented solutions for the trustworthy and privacy-preserving processing of personal information. We divided this comprehensive problem into the release and the retrieval of information.

Trustworthy and privacy-preserving release of personal information. We presented a framework for the declarative design of distributed systems, which supports a wide range of security properties, including authorization policies, privacy, controlled linkability, and accountability. The core component of the framework is a declarative API that exports primitives for data processing. The programming abstraction represents the information known to principals as logical formulas and the messages exchanged by parties as validity proofs for logical formulas. The cryptographic implementation relies on a powerful combination of digital signatures, non-interactive zero-knowledge proofs of knowledge, service-specific pseudonyms, and reputation lists. Our framework constitutes an ideal plugin for proof-carrying authorization infrastructures [144, 118, 19, 170, 30].

We showed how to leverage an existing security type system for ML to statically enforce authorization policies in declarative specifications and we proved that these policies are enforced by the cryptographic implementation. In particular, the authorization policies specified via the API methods hold by construction. We also proved the security of the cryptographic constructions introduced in this thesis (namely, service-specific pseudonyms and the identity escrow protocol).

We conducted several case studies to show the feasibility and the applicability of our declarative API: we designed and implemented tales (The Anonymous Lecture Evaluation System), and we re-designed, re-implemented, and improved previous works on anonymous webs of trust [25] and on a security API for distributed social networks [28]. For the latter two case studies, we were able to conveniently and quickly derive a cryptographic implementation that offered more functionality than the dedicated cryptographic implementations of the corresponding previous works.

Trustworthy and privacy-preserving retrieval of personal information. We presented a methodology for the privacy-preserving retrieval of personal information that relies on secure hardware and oblivious RAM. We exemplify our approach and devised ObliviAd, an architecture for online behavioral advertising.

We showed that it is possible to achieve strong privacy properties while retaining practicality in current online behavioral advertisement models. In particular, our solution introduces only minor changes to the current broker infrastructure, namely, the installation of secure coprocessors on the servers of the brokers. The current business and system models also require only very minor adaptations. Our solution utilizes PIR technology, which we implemented using secure coprocessors and an efficient ORAM construction. This technology allows clients to retrieve advertisements that best match their behavioral profile without the broker learning any personal information. At the same time, our architecture allows brokers to learn any non-personal information that they may find useful for improving their business model (e.g., click-through rates, statistics for click-fraud detection, etc.).

We formalized the two fundamental privacy goals profile privacy and profile unlinkability as observational equivalence relations and the billing correctness property as a trace property. These are the first formal security definitions for OBA. We used ProVerif, an automated cryptographic protocol verifier, to formally establish them on ObliviAd. An experimental evaluation demonstrates the feasibility of our approach.

Outlook. The API presented in the first part of this thesis supports a rich set of properties such as authorization in distributed systems. There are many more properties that are desirable to have, in particular in distributed systems. For instance, electronic voting schemes typically strive for properties such as coercion-resistance and receipt-freeness [95]. The presented API may be able to express some of these properties but it lacks a first-class support within the declarative language.

Programs that use our security API have security-by-construction guarantees while anonymity properties are still proven by hand. Obtaining anonymity-by-construction guarantees would remove the necessity to prove privacy properties separately. There are techniques capable of proving privacy properties (e.g. [104, 52]). It is conceivable that they can be used to achieve by-construction-style guarantees, albeit protocols that provide anonymity properties are not generally composable.[1]

We have presented ObliviAd as an example of how to realize the privacy-preserving retrieval of personal information. We have successfully demonstrated that the underlying technique is also powerful and flexible enough to be applied to other scenarios such as privacy-preserving payment systems [182]. There are many questions that impose themselves: for instance, using this approach, is there a general way to transform any protocol (of a certain form or class) to improve the offered privacy?

Our technique relies on a complex piece of hardware. Recently, CPU manufacturers have started to incorporate special instruction sets that enable code attestation [8, 176]. While this approach does not yield obliviousness, it may still be sufficient to yield provable guarantees, possibly against a weaker attacker model.

[1] Consider two protocol P_1 and P_2 for the same service s. In P_1, a pseudonym psd_1 either originated from *Alice* or *Bob*, and in P_2, a pseudonym psd_2 either originated from *Bob* or *Charlie*. If the pseudonyms coincide, *Bob* is the owner of the pseudonym, otherwise *Alice* is the owner of psd_1 and *Charlie* is the owner of psd_2.

Appendices

A. Well-Typedness of the API Methods

This chapter details the implementation of the API in RCF and the proof of well-typedness. RCF is a λ-calculus that is very well suited for verification purposes and it is expressive enough to encode programming languages such as ML and Java. Appendix A.1 starts with the implementation of the API methods in RCF. Appendix A.2 leverages the state-of-the-art F7 type-checker [49] to obtain the well-formedness of the RCF implementation.

A.1. RCF Implementation of the API

This section contains all the details of the symbolic, sealing-based [148] RCF API implementation. First, we introduce the necessary machinery including the RCF types. As suggested by Bengtson et al. [49], we use the standard ML notation to keep the code readable.

In the remainder of this section, we use the following convention: we write $\mathcal{F}^e$ to denote an elementary formula, i.e., a formula for says-statements, SSP-statements and so on without conjunctions and disjunction. We write $\mathcal{F}^\wedge$ to denote formulas that may contain conjunctions of elementary formulas but do not contain disjunctions, we write $\mathcal{F}^\vee$ to denote formulas that contain disjunctions in disjunctive normal form, and we write $\mathcal{F}$ to denote arbitrary formulas built from conjunction, disjunction, and elementary formulas. The difference between a formula $\mathcal{F}^\vee$ and $\mathcal{F}$ is that formula $\mathcal{F}^\vee$ is stated in disjunctive normal form, i.e., formulas $\mathcal{F}^\vee$ are of the form

$$\mathcal{F}^\vee := \bigvee_{i=1}^{n} \mathcal{F}_i^\wedge$$

where the $\mathcal{F}_i^\wedge$ in turn are of the form

$$\mathcal{F}_i^\wedge := \bigwedge_{j=1}^{m} \mathcal{F}_j^e$$

for some n and m. It is a well-known result that every logical formula can be written in disjunctive normal form. The distinction between $\mathcal{F}^e$, $\mathcal{F}^\wedge$, and $\mathcal{F}^\vee$ is crucial for the proofs.

$H, T, U, V ::=$	type
$\quad unit$	$\quad$ unit type
$\quad \{x : T \mid \mathcal{F}^{\vee}\}$	$\quad$ refinement type (scope of x is $\mathcal{F}^{\vee}$)
$\quad \Pi x : T.\, U$	$\quad$ dependent function type (scope of x is U)
$\quad \Sigma x : T.\, U$	$\quad$ dependent pair type (scope of x is U)
$\quad T + U$	$\quad$ disjoint sum type
$\quad \mu\alpha.\, T$	$\quad$ iso-recursive type (scope of α is T)
$\quad \alpha$	$\quad$ type variable
$\{\mathcal{F}^{\vee}\} \triangleq \{\_ : unit \mid \mathcal{F}^{\vee}\}$	ok type
$bool \triangleq unit + unit$	Boolean type

Table A.1.: RCF syntax of types.

A.1.1. Preliminaries: RCF Type System

This section briefly reviews the RCF type system components, namely, the types, subtyping and kinding. Bengtson et al. [49] describe all of these concepts in detail.

Typing environments. Type-checking and, in general, all judgments of a type system are always conducted relative to a typing environment E. This typing environment E keeps track of bound variables, logical formulas, and so on. For instance, suppose we are given a typing environment E and we are to type-check the following code borrowed from Example 2.2:

```
let pf_s = mkSays x_Pat Rating(x_opinion);
```

After type-checking this line, the extended typing environment E' inherits all information from E and also records that pf_s is a variable of type *proof*. More precisely, $E' := E, pf_s : proof$. How the entries affect the type-checking process depends on the currently proven judgment. In RCF, the typing environment is monotonously increasing, i.e., there is no judgment that removes entries from the environment. In the following description, we assume that E “fits” the current context; we formalize the meaning of “fits” and typing environment in Appendix A.2.

RCF types. We give a brief overview of the RCF type system and the types that appear in our API. RCF is a security type system that statically enforces authorization policies on distributed systems. As such, the types it supports are not *integer* or *string*. Rather, a type in RCF intuitively determines whom a value might originate from and to whom it can be sent. If RCF determines that a values comes from a trustworthy source, this value conveys information that helps to enforce authorization policies. For instance, if we can determine that a message was sent from a trustworthy source that the value can convey information that helps to enforce authorization policies.

We overview the syntax of RCF types in Table A.1. The only basic type is the universal type *unit*. A value $v : unit$ does not convey any additional information besides the value itself. In general, the RCF type *unit* captures all concrete untrusted values, i.e., values can be sent over and received from the Internet such as Boolean values, principal identifiers, strings and so on. The name *unit* is a little misleading since the RCF type *unit* is populated by a plethora of values and does not correspond to the *unit* type in programming languages such as OCaml and F$^{\#}$; in these two language, the empty tuple () is the only value of type *unit*.

Based on the basic type *unit*, more complicated RCF types are constructed as follows:

Refinement types $\{x : T \mid \mathcal{F}\}$:

Refinement types determine the type of a given value and additionally transport a logical formula that may depend on the value itself. More precisely, a value v of type $\{x : T \mid \mathcal{F}\}$ is first of all a value of type T and additionally, the formula $\mathcal{F}\{v/x\}$ holds, i.e., the formula $\mathcal{F}$ where every occurrence of x is replaced by v (the formula $\mathcal{F}$ need not contain x). These refinement types are a salient tool to convey logical predicates from one principal in the system to another.

For instance, let us reconsider the doctor evaluation system from Section 2.3: if the patient can give v type *unit* and additionally deduce the predicate $\mathsf{Rating}(v)$, then she can give the value v the type $\{x : \mathit{unit} \mid \mathsf{Rating}(x)\}$. The patient sends v to the evaluation platform. If the evaluation platform can assign v the refinement type $\{x : \mathit{unit} \mid \mathsf{Rating}(x)\}$, then in the context of the evaluation platform, $v : \mathit{unit}$ and additionally, the logical predicate $\mathsf{Rating}(v)$ holds.

Dependent functions $\Pi x : T.\, U$

and

Dependent pairs $\Sigma x : T.\, U$:

RCF uses standard dependent functions $\Pi x : T.\, U$ and dependent pairs $\Sigma x : T.\, U$. If x does not occur in the type U, i.e., the function or the pair is not dependent on the first component, then the dependent function and the dependent pair corresponds to the usual function and pair types as used in OCaml and F$^{\#}$. If x occurs in U, then the return value of a function or the second component of a tuple depends on the first component.

For instance, we use dependent functions to specify the type of the proof verification function

$$\mathsf{verify}_{\mathcal{F}^{\vee}} : \mathit{proof} \to f : \mathit{formula} \to \{z : \mathit{bool} \mid \forall \widetilde{x}.\, f = \underline{\mathcal{F}^{\vee}} \wedge z = \mathsf{true} \implies \mathcal{F}^{\vee}\}.$$

It is a function that takes as input a proof p and a formula f where f is the encoding of $\mathcal{F}^{\vee}$, and returns a Boolean value z. Additionally, the formula

$$\forall \widetilde{x}.\, f = \underline{\mathcal{F}^{\vee}} \wedge z = \mathsf{true} \implies \mathcal{F}^{\vee}$$

holds for the return value z. Here and throughout the rest of this thesis, we write $\underline{\mathcal{F}}$ to denote the ML encoding of the logical formula $\mathcal{F}$.

Disjoint sum types $T + U$:

Disjoint sum types $T + U$ are used to encode the usual ML data types. For instance, common ML data types such as $\mathsf{datatype}\ \mathit{bool} = \mathsf{true} \mid \mathsf{false}$ are encoded as a disjoint sum type. More precisely, the type $T + U$ is constructed by the type constructors inl and inr; inl takes as input a value of type T (i.e., the left side) and returns a value of type $T + U$ and the type constructor inr takes as input a value of type U (i.e., the right side) and returns a value of type $T + U$. In the following, we encode $\mathit{bool} \triangleq \mathit{unit} + \mathit{unit}$ where we define $\mathsf{true} := \mathsf{inr}()$ and $\mathsf{false} := \mathsf{inl}()$. We use the notation $\mathsf{inl}\langle T, T + U\rangle$ (resp. $\mathsf{inr}\langle U, T + U\rangle$) to denote that this use of the type constructor inl (resp. inr) takes type T (resp. U) and returns type $T + U$.

Technically, all sum types are built from the $\mathsf{inl}\langle T,\ T + U\rangle$ and $\mathsf{inr}\langle U,\ T + U\rangle$ type constructors, i.e., more complicated types have to be encoded using only these two type constructors. For instance, the type $T + U + V = T + T'$ where $T' := U + V$. As suggested by Bengtson et al. [49], we assume a unique encoding and use arbitrary data type constructors as syntactic sugar and we will use the usual ML data type notation.

Iso-recursive types $\mu\alpha.\ T$:

Iso-recursive types are constructed with the fold type constructor and allow for defining recursive data structures such as lists. The core idea of iso-recursive types $T := \mu\alpha.\ U$ is that the type variable α that occurs in U can be replaced with T.

For instance, the usual ML list $\mathsf{datatype}\ T\ \mathit{list} = \mathsf{NIL} \mid \mathsf{Cons\ of}\ T * (T\ \mathit{list})$ for values of type T is defined in RCF as $T\ \mathit{list} \triangleq \mu\alpha.\ \mathit{unit} + T * \alpha$. Initially, lists are constructed from a value $\mathsf{NIL} : \mathit{unit}$ by applying the constructor $\mathsf{fold}\langle \mathit{unit} + T\{(\mu\alpha.\ \mathit{unit} + T * \alpha)/\alpha\},\ \mu\alpha.\ \mathit{unit} + T * \alpha\rangle$ to the value $\mathsf{inl\ NIL}$. Unfolding $T\ \mathit{list} \triangleq \mu\alpha.\ \mathit{unit} + T * \alpha$ yields $\mathit{unit} + T * (\mu\alpha.\ \mathit{unit} + T * \alpha)$; the unfolded type is a disjoint sum and can be matched to check if the list is empty (left case) or whether the list has a head and a tail (right case).

For a more formal and exhaustive discussion, we refer to Bengtson et al. [49] and the book by Gunter [131].

Concretely, we use iso-recursion to specify the types for our signature scheme. Intuitively, we verify signatures on verification keys (which encode principal identifiers) and therefore, the type of a verification key must be able to describe verification keys. As for the disjoint sum types, we use the standard ML notation as syntactic sugar to describe iso-recursive types.

We use the abbreviation $\{\mathcal{F}\} := \{\_ \mid \mathcal{F}\}$ to denote that a refinement type where the variable (denoted by the anonymous variable “_”) does not occur in the formula.

Polymorphic types. RCF does not support polymorphic types. Since in a program, only a finite number of types can occur, it is possible to encode this polymorphism by instantiating the type variable as needed [49]. For instance, suppose a program uses the polymorphic function $\mathsf{fail}\langle\alpha\rangle : \mathit{unit} \to \alpha$ once with the return type $\alpha := T$ and once with the return type $\alpha := U$, for some types T and U; $\langle\alpha\rangle$ denotes that α is universally quantified. This can be translated into two functions $\mathsf{fail}^{T} : \mathit{unit} \to T$ and $\mathsf{fail}^{U} : \mathit{unit} \to U$ that instantiate α with T and U, respectively. Since an implementation has only finitely many occurrences of polymorphic functions, the translation is also finite and well-founded. We use the same convention for polymorphic data types and we write $\mathsf{datatype}\ \alpha\ T = U$ to denote the data type T that is parameterized in α; the scope of α is U.

Names, restrictions, and channels. Channels model communication media such as the Internet. For instance, the Internet is modeled as $a \updownarrow \mathit{unit}$, i.e., a channel a used for sending and receiving values of type *unit*. Technically, channels are the only names in RCF. Since channels do not surface in the API, we only mention them here for the sake of completeness. Technically, they are used to implement references that, in turn, are used to implement the sealing mechanism.

Subtyping and kinding. Type systems without the possibility to compare types by subtyping are simple but they severely lack expressiveness. For instance, a channel $a \updownarrow \mathit{unit}$ for values of type *unit* could not be used to transmit values of type $v : \{x : \mathit{unit} \mid \mathcal{F}\}$, even though v is a value of type *unit*. RCF solves this restriction by means of an elaborate subtyping mechanism.

Subtyping in RCF relies on two concepts: standard subtyping purely based on types and a mechanism based on kinding. For instance, standard subtyping allows us to deduce that values of type $\{x : \mathit{unit} \mid \mathcal{F}\}$, are also of type *unit*. We write $E \vdash T <: U$ to denote that type T is a subtype of U under typing environment E, i.e., E proves that a value of type T can be used in place of a value of type U. Intuitively, the kinding mechanism decides subtyping based on whether a type is of kind public, i.e., it can be sent to the attacker, or whether a type is of kind tainted, i.e., it can originate from the attacker. We write $E \vdash T :: \nu$, i.e., E proves type T to be of kind ν.

$$\textsc{Sub Public Tainted}\quad \frac{E \vdash T :: \mathsf{pub} \qquad E \vdash U :: \mathsf{tnt}}{E \vdash T <: U}$$

Kinding interacts with the standard subtyping via the rule Sub Public Tainted. The kinding mechanism is necessary as pure type-based subtyping cannot decide whether a functional type $T \to U$ is a subtype of *unit*. This is necessary since only values that can be given type *unit* can be sent over an untrusted channel such as the Internet. The problem occurs while subtyping the functional type: the subtyping rule Sub Fun for function types only relates two function types but *unit* is not a functional type.

$$\textsc{Sub Fun}\quad \frac{E \vdash T' <: T \qquad E \vdash U <: U'}{E \vdash (\Pi x : T.\, U) <: (\Pi x : T'.\, U')}$$

The kinding relation, however, can decide $E \vdash T \to U <: \mathit{unit}$ if the function f of type

$T \to U$ can be given kind pub (public), i.e., f can be given to the attacker.

Kind Fun

$$\frac{E \vdash T :: \overline{\nu} \qquad E, x : T \vdash (\Pi x : T.\, U) :: \nu}{E \vdash (\Pi x : T.\, U) :: \nu}$$

Intuitively, it is safe to pass a function to the attacker, if the attacker cannot extract non-public information from it. Indeed, Kind Fun formalizes this intuition. It states that a function type is public if the arguments are tainted (i.e., the attacker controls the input to the function) and the output is public.

A.1.2. API Data Types

In this section, we describe the data types used in the API implementation in an ML-style language.

Basic data types. We start by describing the basic data types used by the API. We use self-explanatory names such as *commitment* to make the API as easily accessible as possible.

Type *random*:
The type *random* describes random numbers.

Type *bitstring*:
The type *bitstring* describes raw data that can, for instance, be directly sent over or received from the network. We also use it to denote integers. In the latter case, an implementation would interpret the bit string as an encoded integer, for instance, in two's complement representation.

Type *signature*:
The type *signature* denotes digital signatures.

Type *commitment*:
The type *commitment* denotes cryptographic commitments.

Type *zero-knowledge*:
A Groth-Sahai proof consists of three components: the commitments of the values used in the proof, possible opening information for selected commitments, and the cryptographic zero-knowledge proof. The type *zero-knowledge* describes the cryptographic zero-knowledge proof.

Type *pseudo*:
The type *pseudo* describes service-specific pseudonyms.

Type *string*:
The type *string* denotes ordinary strings. In the API, they are used to describe the services of SSPs.

$$\textsf{datatype } \alpha\ \mathit{list} = \textsf{Cons of } \alpha * \alpha\ \mathit{list} \mid \textsf{NIL}$$

$$\begin{array}{ll}
\textsf{datatype } \alpha\ \mathit{RevHid} = & \qquad \textsf{datatype } \alpha\ \mathit{option} = \\
\quad \mid \textsf{Revealed of } \alpha & \qquad \quad \mid \textsf{Some of } \alpha \\
\quad \mid \textsf{Hidden of } \mathit{bitstring} & \qquad \quad \mid \textsf{None}
\end{array}$$

$$\begin{array}{ll}
 & \mathit{predicate}^P ::= \\
\mathit{predicate}^F ::= & \quad \mid P_1^P \textsf{ of } \mathit{commitment} * (T_1^1 * \mathit{random})\ \mathit{option} * \\
\quad \mid P_1^F \textsf{ of } T_1^1 * \cdots * T_{n_1}^1 & \qquad \cdots * \mathit{commitment} * (T_{n_1}^1 * \mathit{random})\ \mathit{option} \\
\quad \mid \cdots & \quad \mid \cdots \\
\quad \mid P_m^F \textsf{ of } T_1^m * \cdots * T_{n_m}^m & \quad \mid P_m^P \textsf{ of } \mathit{commitment} * (T_1^m * \mathit{random})\ \mathit{option} * \\
 & \qquad \cdots * \mathit{commitment} * (T_{n_m}^m * \mathit{random})\ \mathit{option}
\end{array}$$

where $T_i^j \in \{\mathit{bitstring\ RevHid}, \mathit{uid}_{\mathit{pub}}\ \mathit{RevHid}\}$

$$\begin{array}{l}
\textsf{datatype } \mathit{formula} = \\
\quad \mid \textsf{Says of } (z : \mathit{uid}_{\mathit{pub}}\ \mathit{RevHid}) * \mathit{predicate}^F \\
\quad \mid \textsf{SSP of } (z : \mathit{uid}_{\mathit{pub}}\ \mathit{RevHid}) * (s : \mathit{string\ RevHid}) * (\mathit{psd} : \mathit{pseudo\ RevHid}) \\
\quad \mid \textsf{REL of } (x : \mathit{bitstring\ RevHid})\ * (\mathit{op} : \mathit{string}) * (y : \mathit{bitstring\ RevHid}) \\
\quad \mid \textsf{EQN of } (x : \mathit{bitstring\ RevHid}) * (\mathit{op} : \mathit{string}) * (y : \mathit{bitstring\ RevHid}) * \\
\qquad (z : \mathit{bitstring\ RevHid}) \\
\quad \mid \textsf{LM of } (x : \mathit{pseudo\ RevHid}) * (b : \mathit{bitstring\ RevHid}) * (\ell : (\mathit{pseudo} * \mathit{bitstring})\ \mathit{list}) \\
\quad \mid \textsf{LNM of } (x : \mathit{pseudo\ RevHid}) * \\
\qquad (\ell : (\mathit{pseudo} * \mathit{bitstring})\ \mathit{list}) \\
\quad \mid \textsf{EscrowInfo of } (z : \mathit{uid}_{\mathit{pub}}) * (x : \mathit{uid}_{\mathit{pub}}\ \mathit{RevHid}) * \\
\qquad (R : \mathit{bitstring\ RevHid}) * (s : \mathit{string\ RevHid}) * (\mathit{idr} : \mathit{bitstring\ RevHid}) \\
\quad \mid \textsf{And of } (f_1 : \mathit{formula}) * (f_2 : \mathit{formula}) \\
\quad \mid \textsf{Or of } (f_1 : \mathit{formula}) * (f_2 : \mathit{formula})
\end{array}$$

Nullary type constructors C denote C of *unit*.

Table A.2.: ML data type declarations.

Type $\mathit{uid}_{\mathit{pub}}$:

The type $\mathit{uid}_{\mathit{pub}}$ describes public verification keys.

Type *uid*:

Instead of letting users handle secret cryptographic keys directly, we stipulate the usage of handles. Since these handles do not contain the actual key, they can be leaked without compromising the corresponding key. The type *uid* denotes such handles.

Constructed data types. We use the basic types to construct the types that describe the input to the API methods. Table A.2 and Table A.3 depict the constructed data types in an ML-style notation. In the following, we briefly describe each of them.

datatype $\mathit{statement} =$

$\mid \mathsf{Says_p}$ of $(c_z : \mathit{commitment}) * ((z : \mathit{uid}_{\mathit{pub}}) * (r_z : \mathit{random}))\ \mathit{option} *$
$\quad (c_{\mathit{sig}} : \mathit{commitment}) * ((\mathit{sig} : \mathit{signature}) * (r_{\mathit{sig}} : \mathit{random}))\ \mathit{option} * \mathit{predicate}^P$

$\mid \mathsf{SSP_p}$ of $(c_z : \mathit{commitment}) * ((z : \mathit{uid}_{\mathit{pub}}) * (r_z : \mathit{random}))\ \mathit{option} *$
$\quad (c_s : \mathit{commitment}) * ((s : \mathit{string}) * (r_s : \mathit{random}))\ \mathit{option} *$
$\quad (c_{\mathit{psd}} : \mathit{commitment}) *$
$\quad ((\mathit{psd} : \mathit{pseudo}) * (r_{\mathit{psd}} : \mathit{random}))\ \mathit{option} * (c_x : \mathit{commitment})$

$\mid \mathsf{REL_p}$ of $(c_x : \mathit{commitment}) * ((x : \mathit{bitstring}) * (r_x : \mathit{random}))\ \mathit{option} *$
$\quad (\mathit{op} : \mathit{string}) *$
$\quad (c_y : \mathit{commitment}) * ((y : \mathit{bitstring}) * (r_y : \mathit{random}))\ \mathit{option}$

$\mid \mathsf{EQN_p}$ of $(c_x : \mathit{commitment}) * ((x : \mathit{bitstring\ RevHid}) * (r_x : \mathit{random}))\ \mathit{option} *$
$\quad (\mathit{op} : \mathit{string}) *$
$\quad (c_y : \mathit{commitment}) * ((y : \mathit{bitstring\ RevHid}) * (r_y : \mathit{random}))\ \mathit{option} *$
$\quad (c_z : \mathit{commitment}) * ((z : \mathit{bitstring\ RevHid}) * (r_z : \mathit{random}))\ \mathit{option}$

$\mid \mathsf{LM_p}$ of $(c_x : \mathit{commitment}) * ((x : \mathit{pseudo}) * (r_x : \mathit{random}))\ \mathit{option} *$
$\quad (c_b : \mathit{commitment}) * ((b : \mathit{bitstring}) * (r_b : \mathit{random}))\ \mathit{option} *$
$\quad (\ell : (\mathit{pseudo} * \mathit{bitstring})\ \mathit{list})$

$\mid \mathsf{LNM_p}$ of $(c_x : \mathit{commitment}) * ((x : \mathit{pseudo}) * (r_x : \mathit{random}))\ \mathit{option} *$
$\quad (\ell : (\mathit{pseudo} * \mathit{bitstring})\ \mathit{list})$

$\mid \mathsf{EscrowInfo_p}$ of $(z : \mathit{uid}_{\mathit{pub}}) *$
$\quad (c_x : \mathit{commitment}) * ((x : \mathit{uid}_{\mathit{pub}}) * (r_x : \mathit{random}))\ \mathit{option} *$
$\quad (c_R : \mathit{commitment}) * ((R : \mathit{bitstring}) * (r_R : \mathit{random}))\ \mathit{option} *$
$\quad (c_s : \mathit{commitment}) * ((s : \mathit{string}) * (r_s : \mathit{random}))\ \mathit{option} *$
$\quad (c_{\mathit{idr}} : \mathit{commitment}) * ((\mathit{idr} : \mathit{pseudo}) * (r_{\mathit{idr}} : \mathit{random}))\ \mathit{option} *$
$\quad (c_r : \mathit{commitment})$

$\mid \mathsf{And_p}$ of $(p_1 : \mathit{statement}) * (p_2 : \mathit{statement})$

$\mid \mathsf{Or_p}$ of $(p_1 : \mathit{statement}) * (p_2 : \mathit{statement})$

datatype $\mathit{proof} =$
$\quad \mathsf{ZK}$ of $(\mathit{zkv} : \mathit{zero\text{-}knowledge}) * (\mathit{stm} : \mathit{statement})$

Nullary type constructors C denote C of *unit*.

Table A.3.: ML data type declarations continued.

Type α *RevHid*:

The type α *RevHid* describes values that occur in the ML encoding of formulas. The values can be either revealed, denoted by the type constructor Revealed x, or they can be hidden, denoted by Hidden z.

The argument z of the Hidden type constructor acts as a positional index and is used to establish the equality among hidden values. For instance, consider the formula Revealed vk says $\mathit{Good}^F(\mathsf{Revealed}\ m, \mathsf{Revealed}\ m)$; the superscript ^{F} denotes that this is a formula, see below. A proof for this formula can be changed into a proof for the formula Revealed vk says $\mathit{Good}^F(\mathsf{Hidden}\ 1, \mathsf{Hidden}\ 1)$, i.e., the first and the second argument to the predicate *Good* are the same, indicated by the same index. We stress that the converse is not true, since this proof would also verify for the formula Revealed vk says $\mathit{Good}^F(\mathsf{Hidden}\ 1, \mathsf{Hidden}\ 2)$, i.e., different arguments to the type constructor Hidden do not imply that the hidden values are different.

Type α *option*:

The usual *option* type is used to capture whether a proof contains the opening information to a commitment or whether the opening information has been removed. In the former case, the constructor Some holds the opening information, in the latter case, the constructor None denotes the removal of the opening information.

We now describe the types *formula* for encoding logical formulas and type *statement* for describing zero-knowledge proof statements. Every value v that occurs in a logical formula must be encoded into type *formula* and in type *statement*. The canonical encoding into type *formula* uses the *RevHid*, i.e., v can either be revealed or hidden. The type *statement* has to closely match the requirements of zero-knowledge proofs. Consequently, it contains for every value v a commitment c_v : *commitment* to v and opening information (v, r_v) to the commitment. The opening information are in turn described by an option type to capture that they can be removed from a proof. Additionally, type *statement* contains the cryptographic material used in the zero-knowledge proofs. This material is also represented by commitments and opening information.

Since all values that are contained in type *formula* are also contained in type *statement*, we only describe type *formula* and highlight the additional information contained inside of zero-knowledge proof statements.

Types $\mathit{predicate}^F$ and $\mathit{predicate}^P$:

$$\begin{array}{l}
\mathit{predicate}^F ::= \\
\quad \mid P_1^F \ \mathsf{of}\ T_1^1 * \cdots * T_{n_1}^1 \\
\quad \mid \cdots \\
\quad \mid P_m^F \ \mathsf{of}\ T_1^m * \cdots * T_{n_m}^m
\end{array}$$

$$\begin{array}{l}
\mathit{predicate}^P ::= \\
\quad \mid P_1^P \ \mathsf{of}\ \mathit{commitment} * (T_1^1 * \mathit{random})\ \mathit{option} * \\
\qquad\qquad \cdots * \mathit{commitment} * (T_{n_1}^1 * \mathit{random})\ \mathit{option} \\
\quad \mid \cdots \\
\quad \mid P_m^P \ \mathsf{of}\ \mathit{commitment} * (T_1^m * \mathit{random})\ \mathit{option} * \\
\qquad\qquad \cdots * \mathit{commitment} * (T_{n_m}^m * \mathit{random})\ \mathit{option}
\end{array}$$

where $T_i^j \in \{\mathit{bitstring}\ \mathit{RevHid}, \mathit{uid}_{pub}\ \mathit{RevHid}\}$

The types $\mathit{predicate}^F$ and $\mathit{predicate}^P$ describe logical predicates that are used in combination with the says modality. The type $\mathit{predicate}^F$ represents predicates inside of formulas, indicated by the superscript ^{F}, and the type $\mathit{predicate}^P$ represents predicates inside of zero-knowledge proof statements, indicated by the superscript ^{P}.

We use the convention that the name of the type constructor coincides with that of the logical predicate; the predicate name of the logical formula does not have a superscript. For instance, the logical predicate $\mathit{Good}(m)$ is represented by $\mathit{Good}^F(\mathsf{Revealed}(m)) : \mathit{predicate}^F$ and by $\mathit{Good}^P(c_m, \mathsf{Some}(m, r_m)) : \mathit{predicate}^P$.

The type $\mathit{predicate}^F$ is a sum type, where each type constructor corresponds to exactly one predicate proven in a protocol. The arity of each case P_i^F matches the arity of the corresponding logical predicate P_i. Each type argument of a type constructor P_i^F is of type α *RevHid* to allow the distinction between hidden and revealed values. The values may be user identifiers ($\alpha := \mathit{uid}_{\mathit{pub}}$) or any other value ($\alpha := \mathit{bitstring}$).

As explained above, the type $\mathit{predicate}^P$ represents every value occurring in type $\mathit{predicate}^F$ with two values: one value of type *commitment* and one value of type *option*.

Types *formula* and *statement*:

The type *formula* captures the logical formula that can be proven by a zero-knowledge proof. The type *statement* captures the corresponding zero-knowledge statement. For instance, a logical formula *Alice* says $\mathit{Good}(m)$ is represented by the value $\mathsf{Says}(\mathsf{Revealed}(\mathit{vk}_{\mathit{Alice}}), \mathit{Good}^F(\mathsf{Revealed}(m))$: *formula* and by the value $\mathsf{Says_p}(c_{\mathit{vk}_{\mathit{Alice}}}, \mathsf{Some}(\mathit{vk}_{\mathit{Alice}}, r_{\mathit{vk}_{\mathit{Alice}}}), c_{\mathit{sig}}, \mathsf{Some}(\mathit{sig}, r_{\mathit{sig}}), \mathit{Good}^P(c_m, \mathsf{Some}(m, r_m)))$: *statement*.

Every logical formula provable by the API is represented by a case in *formula* and *statement*.

Says modality proofs Says and $\mathsf{Says_p}$:

$$\begin{aligned}
&\mathsf{Says\ of}\ (z : \mathit{uid}_{\mathit{pub}}\ \mathit{RevHid}) * \mathit{predicate}^F\\
&\mathsf{Says_p\ of}\ (c_z : \mathit{commitment}) * ((z : \mathit{uid}_{\mathit{pub}}) * (r_z : \mathit{random}))\ \mathit{option}\ *\\
&\qquad (c_{\mathit{sig}} : \mathit{commitment}) * ((\mathit{sig} : \mathit{signature}) * (r_{\mathit{sig}} : \mathit{random}))\ \mathit{option}\ *\\
&\qquad \mathit{predicate}^P
\end{aligned}$$

Says and $\mathsf{Says_p}$ canonically encode the formula *Alice* says P where the principal *Alice* is encoded using a principal identifier $z : \mathit{uid}_{\mathit{pub}}\ \mathit{RevHid}$ and P corresponds to the predicate encoded by the $\mathit{predicate}^F$ in Says and to $\mathit{predicate}^P$ in $\mathsf{Says_p}$. The statement $\mathsf{Says_p}$ additionally contains a commitment to the digital signature *sig* as well as to the corresponding opening information.

Pseudonym ownership proofs SSP and $\mathsf{SSP_p}$:

$$\begin{aligned}
&\mathsf{SSP}\ \mathsf{of}\ (z : \mathit{uid}_{\mathit{pub}}\ \mathit{RevHid}) * (s : \mathit{string}\ \mathit{RevHid}) *\\
&\qquad (\mathit{psd} : \mathit{pseudo}\ \mathit{RevHid})\\
&\mathsf{SSP_p}\ \mathsf{of}\ (c_z : \mathit{commitment}) * ((z : \mathit{uid}_{\mathit{pub}}) * (r_z : \mathit{random}))\ \mathit{option} *\\
&\qquad (c_s : \mathit{commitment}) * ((s : \mathit{string}) * (r_s : \mathit{random}))\ \mathit{option} *\\
&\qquad (c_{\mathit{psd}} : \mathit{commitment}) *\\
&\qquad\quad ((\mathit{psd} : \mathit{pseudo}) * (r_{\mathit{psd}} : \mathit{random}))\ \mathit{option} *\\
&\qquad (c_x : \mathit{commitment})
\end{aligned}$$

The encoding of service-specific pseudonyms contains the principal identifier z of the owner of the pseudonym, the service s, and the pseudonym psd itself. The proof $\mathsf{SSP_p}$ additionally contains a commitment c_x to the signing key used in the computation of the pseudonym (see Section 2.4). To protect the signing key, the statement contains no opening information for it.

Relational proofs REL and $\mathsf{REL_p}$:

$$\begin{aligned}
&\mathsf{REL}\ \mathsf{of}\ (x : \mathit{bitstring}\ \mathit{RevHid}) * (\mathit{op} : \mathit{string}) * (y : \mathit{bitstring}\ \mathit{RevHid})\\
&\mathsf{REL_p}\ \mathsf{of}\ (c_x : \mathit{commitment}) * ((x : \mathit{bitstring}) * (r_x : \mathit{random}))\ \mathit{option} *\\
&\qquad (\mathit{op} : \mathit{string}) *\\
&\qquad (c_y : \mathit{commitment}) * ((y : \mathit{bitstring}) * (r_y : \mathit{random}))\ \mathit{option}
\end{aligned}$$

The encoding of relational proofs contains the two operands x and y as well as the relation *op*. Since the proven Groth-Sahai equations allow for deducing the proven operation, the operation occurs in plain in the types *formula* and *statement*.

Equation proofs EQN and $\mathsf{EQN_p}$:

$$\begin{aligned}
&\mathsf{EQN}\ \mathsf{of}\ (x : \mathit{bitstring}\ \mathit{RevHid}) * (\mathit{op} : \mathit{string}) * (y : \mathit{bitstring}\ \mathit{RevHid}) *\\
&\qquad (z : \mathit{bitstring}\ \mathit{RevHid})\\
&\mathsf{EQN_p}\ \mathsf{of}\ (c_x : \mathit{commitment}) * ((x : \mathit{bitstring}\ \mathit{RevHid}) * (r_x : \mathit{random}))\ \mathit{option} *\\
&\qquad (\mathit{op} : \mathit{string}) *\\
&\qquad (c_y : \mathit{commitment})\ * ((y : \mathit{bitstring}\ \mathit{RevHid}) * (r_y : \mathit{random}))\ \mathit{option} *\\
&\qquad (c_z : \mathit{commitment}) * ((z : \mathit{bitstring}\ \mathit{RevHid}) * (r_z : \mathit{random}))\ \mathit{option}
\end{aligned}$$

The encoding of proofs of mathematical operations contains the two operands x and y, the operation *op*, as well as the result z. Since the proven Groth-Sahai equations allow for deducing the proven operation, the operation in *formula* and *statement* occurs in plain. Here and throughout the remainder of this thesis, plain means that the values are not wrapped in type *RevHid* in *formula* and the value occurs directly in *statement* without commitment and opening information.

List membership proofs LM and $\mathsf{LM_p}$:

$$\begin{aligned}
&\mathsf{LM}\text{ of } (x : \mathit{pseudo}\ \mathit{RevHid}) * (b : \mathit{bitstring}\ \mathit{RevHid}) *\\
&\qquad (\ell : (\mathit{pseudo} * \mathit{bitstring})\ \mathit{list})\\
&\mathsf{LM_p}\text{ of } (c_x : \mathit{commitment}) * ((x : \mathit{pseudo}) * (r_x : \mathit{random}))\ \mathit{option} *\\
&\qquad (c_b : \mathit{commitment}) * ((b : \mathit{bitstring}) * (r_b : \mathit{random}))\ \mathit{option} *\\
&\qquad (\ell : (\mathit{pseudo} * \mathit{bitstring})\ \mathit{list})
\end{aligned}$$

The list-membership branch of the types *formula* and *statement* consists of the pseudonym x, the attribute b, and the list ℓ. Since we disallow the list to be hidden, the list occurs in plain.

List non-membership proofs LNM and $\mathsf{LNM_p}$:

$$\begin{aligned}
&\mathsf{LNM}\text{ of } (x : \mathit{pseudo}\ \mathit{RevHid}) *\\
&\qquad (\ell : (\mathit{pseudo} * \mathit{bitstring})\ \mathit{list})\\
&\mathsf{LNM_p}\text{ of } (c_x : \mathit{commitment}) * ((x : \mathit{pseudo}) * (r_x : \mathit{random}))\ \mathit{option} *\\
&\qquad (\ell : (\mathit{pseudo} * \mathit{bitstring})\ \mathit{list})
\end{aligned}$$

The list-nonmembership branch of the types *formula* and *statement* consists of the pseudonym x and the list ℓ. Since we disallow the list to be hidden, the list occurs plainly in *formula* and in *statement*.

Identity escrow proofs $\mathsf{EscrowInfo}$ and $\mathsf{EscrowInfo_p}$:

$$\begin{aligned}
&\mathsf{EscrowInfo}\text{ of } (z : \mathit{uid}_{\mathit{pub}}) * (x : \mathit{uid}_{\mathit{pub}}\ \mathit{RevHid}) *\\
&\qquad (R : \mathit{bitstring}\ \mathit{RevHid}) * (s : \mathit{string}\ \mathit{RevHid}) * (\mathit{idr} : \mathit{bitstring}\ \mathit{RevHid})\\
&\mathsf{EscrowInfo_p}\text{ of } (z : \mathit{uid}_{\mathit{pub}}) *\\
&\qquad (c_x : \mathit{commitment}) * ((x : \mathit{uid}_{\mathit{pub}}) * (r_x : \mathit{random}))\ \mathit{option} *\\
&\qquad (c_R : \mathit{commitment}) * ((R : \mathit{bitstring}) * (r_R : \mathit{random}))\ \mathit{option} *\\
&\qquad (c_s : \mathit{commitment}) * ((s : \mathit{string}) * (r_s : \mathit{random}))\ \mathit{option} *\\
&\qquad (c_{\mathit{idr}} : \mathit{commitment}) * ((\mathit{idr} : \mathit{pseudo}) * (r_{\mathit{idr}} : \mathit{random}))\ \mathit{option} *\\
&\qquad (c_r : \mathit{commitment})
\end{aligned}$$

The identity escrow case of *formula* consists of the following components: the verification key of the trusted third party z, the verification key x of the user, the value R that was signed and issued by the TTP (see Section 2.4), the service s for which the escrow identifier is issued, and the escrow identifier *idr* itself. The corresponding case of *statement* additionally contains the commitment c_r on the value r used to compute the escrow identifier. We stipulate that r is never revealed and that z (the public identifier of the trusted party) is never hidden; these values occur plainly.

Conjunctive proofs And and $\mathsf{And_p}$:

$$\begin{aligned}
&\mathsf{And}\text{ of } (f_1 : \mathit{formula}) * (f_2 : \mathit{formula})\\
&\mathsf{And_p}\text{ of } (p_1 : \mathit{statement}) * (p_2 : \mathit{statement})
\end{aligned}$$

The conjunction case of types *formula* and *statement* contains the two subproofs.

$$
\begin{array}{l}
\mathsf{mkId} : \mathit{string} \to \mathit{uid} * \mathit{uid}_{pub} \\
\mathsf{mkSays} : x : \mathit{uid} \to f : \mathit{predicate}^{F} \to \mathit{proof} \\
\mathsf{mkSSP} : x : \mathit{uid} \to s : \mathit{string} \to \mathit{proof} \\
\mathsf{mkREL} : f : \mathit{formula} \to \mathit{proof} \\
\mathsf{mkEQN} : f : \mathit{formula} \to \mathit{proof} \\
\mathsf{mkLM} : x : \mathit{pseudo} \to b : \mathit{string} \to \ell : \mathit{list} \to \mathit{proof} \\
\mathsf{mkLNM} : x : \mathit{pseudo} \to \ell : \mathit{list} \to \mathit{proof} \\
\mathsf{mkIDRev} : \mathit{proof} \to s : \mathit{string} \to \mathit{proof} \\
\mathsf{mk}_{\wedge} : \mathit{proof} * \mathit{proof} \to \mathit{proof} \\
\mathsf{split}_{\wedge} : \mathit{proof} \to \mathit{proof} * \mathit{proof} \\
\mathsf{mk}_{\vee} : \mathit{proof} \to \mathit{formula} \to \mathit{proof} \\
\mathsf{extractForm} : p : \mathit{proof} \to \mathit{formula} \\
\mathsf{hide} : \mathit{proof} \to \mathit{formula} \to \mathit{proof} \\
\mathsf{rerand} : \mathit{proof} \to \mathit{formula} \to \mathit{proof} \\
\mathsf{verify}_{\mathcal{F}^{\vee}} : \mathit{proof} \to f : \mathit{formula} \to \{z : \mathit{bool} \mid \forall \widetilde{x}.\ f = \underline{\mathcal{F}^{\vee}} \wedge z = \mathsf{true} \implies \mathcal{F}^{\vee}\}
\end{array}
$$

Table A.4.: E_{API}: API RCF interface functions.

Disjunctive proofs Or and Or_{p}:

$$
\begin{array}{l}
\mathsf{Or}\ \mathsf{of}\ (f_1 : \mathit{formula}) * (f_2 : \mathit{formula}) \\
\mathsf{Or}_{\mathsf{p}}\ \mathsf{of}\ (p_1 : \mathit{statement}) * (p_2 : \mathit{statement})
\end{array}
$$

The disjunction case of types *formula* and *statement* contains the two sub-proofs.

Type *proof*:

$$\mathsf{ZK}\ \mathsf{of}\ (\mathit{zkv} : \mathit{zero\text{-}knowledge}) * (\mathit{stm} : \mathit{statement})$$

The type *proof* consists of a value $\mathit{zkv} : \mathit{zero\text{-}knowledge}$, modeling the cryptographic zero-knowledge proof and of the proven statement $\mathit{stm} : \mathit{statement}$. This closely corresponds to the Groth-Sahai zero-knowledge proof system.

A.1.3. Strong Types and Typed API Methods

In this section, we introduce the strong types used in the API. Intuitively, strong types with logical refinements are given to trustworthy components such as the zero-knowledge proof verification function. Notice that the types described in Appendix A.1.2 do not describe trustworthy components as they can be created by the attacker. For instance, zero-knowledge proofs can be created by anybody. Intuitively, zero-knowledge proofs can still be used for authorization purposes, if the proven statement contains a digital signature from an honest protocol participant, i.e. a trustworthy source.

From a type-checking point of view, all types described in Appendix A.1.2 are equivalent to *unit*. Whenever possible, we use meaningful names (e.g., *commitment* and *proof*) in order to keep the API implementation easily accessible.

$$
\begin{aligned}
\mathcal{T}^o_y &:= +_{k=1}^{n} P^S_k(x_1 : T^k_1 * \cdots * x_{\ell_{k-1}} : T^k_{\ell_{k-1}} * \{x_{\ell_k} : T^k_{\ell_k} \mid y \textsf{ says } P_k(x_1, \ldots, x_{\ell_k})\}) \\
\mathcal{T}_y &:= y : \mathit{bitstring} * \mathcal{T}^o_y \\
\mathcal{U}^o_{sk} &:= +_{k=1}^{n} P^S_k(x_1 : T^k_1 * \cdots * x_{\ell_{k-1}} : T^k_{\ell_{k-1}} * \\
&\qquad\quad \{x_{\ell_k} : T^k_{\ell_k} \mid \exists z, y.\ sk = (z, y) \wedge y \textsf{ says } P_k(x_1, \ldots, x_{\ell_k})\}) \\
&\quad \text{where } T^j_i \in \{\mathit{bitstring}, \alpha\} \\
\\
\mathit{verkey} &:= \mu\alpha.\ \mathit{signature} \to \mathcal{T}_y \\
\mathit{sigkey} &:= (\mu\alpha.\ \mathcal{T}_y \to \mathit{signature}) * \mathit{verkey} = (\mathcal{T}_y\{\mathit{verkey}/\alpha\} \to \mathit{signature}) * \mathit{verkey}
\end{aligned}
$$

Table A.5.: Definition of the signing key type *sigkey* and verification key type *verkey*.

Stronger types for the API methods. The key idea of the API is that zero-knowledge proofs can be used to transport logical formulas between principals. Since zero-knowledge proofs are only significant if we are convinced of their validity, the natural way to extract these formulas from a proof is to use the verification method. Therefore, we strengthen the type of the verification method to express that after a successful verification, the expected formula holds true in the current typing environment. The final type looks as follows:

$$\textsf{verify}_{\mathcal{F}^\vee} : \mathit{proof} \to f : \mathit{formula} \to \{z : \mathit{bool} \mid \forall \widetilde{x}.\ f = \underline{\mathcal{F}^\vee} \wedge z = \textsf{true} \implies \mathcal{F}^\vee\},$$

where $\underline{\mathcal{F}}$ corresponds to the ML encoding of the logical formula $\mathcal{F}$. This type ensures that if the returned value z is $\textsf{true}$ and the formula f passed as input is the ML encoding $\underline{\mathcal{F}^\vee}$ of the logical formula $\mathcal{F}^\vee$, then the formula $\mathcal{F}^\vee$ is entailed. The universal quantification $\forall\widetilde{x}$ binds all names in the ML encoding that would otherwise be unbound. Furthermore, the subscript ${}_{\mathcal{F}^\vee}$ of the verification function binds the formula used in the return type. In particular, every formula has its own verification function.

We change the type of the mkSays method to accept as input only predicates rather than formulas. We stress that this is not a conceptual restriction of the API: arbitrary formulas consist of predicates that are connected using Boolean conjunction and disjunction. This choice merely keeps the already lengthy implementations at a reasonable size. Table A.4 depicts the resulting typed API.

Signing and verification keys. Signing and verification keys are an important cornerstone of the API. Although they are well-concealed within the API, these cryptographic objects are the key ingredients (pun intended) from a type-checking point of view. In particular, they are crucial in the process of transporting logical formulas from one principal to another principal. In a nutshell, a principal *Alice* can sign a value *only* if the corresponding logical formula holds true in the environment of *Alice*. Correspondingly, if a principal *Bob* verifies such a signature, then that principal can deduce that the formula corresponding to the signed value holds true.

The ability to transport logical formulas is anchored in the types of the signing function and the signing key. We have already established the convention that the superscript ^{P} is used for the predicate type constructors in proof statements and the superscript ^{F} for type constructors in formulas. To distinguish the predicate type constructors used in digital signatures, we use the superscript ^{S}.

The full type definitions are shown in Table A.5. The (open) type $\mathcal{T}_y^o$ is the type for messages. It is a large disjoint sum type that contains one case for every possible predicate P_k that occurs in a protocol; the logical predicate P_k is represented by the type constructor P_k^S. Every predicate P_k and, consequently, every type constructor P_k^S consists of a specific number ℓ_k of arguments x_i, each of which is of a specific type T_i^k, where i ranges from 1 to ℓ_k. The last argument of P_k^S is additionally refined with the logical predicate P_k itself, namely, y says $P_k(x_1, \ldots, x_{\ell_k})$. This refinement establishes the connection between the logical predicate and its representation in the disjoint sum. In this definition of $\mathcal{T}_y^o$, y is intentionally left unbound. We add the identity of the signer and, at the same time, bind the free variable y in the type $\mathcal{T}_y$. The value y will be instantiated by a verification key; we will see below that *verkey* <: *unit*, i.e., a value v : *verkey* can be used as a value v : *unit*. The binding between the signing key and the corresponding verification key is enforced by the signature creation function.

We defined *verkey* as $\mu\alpha.\ signature \rightarrow \mathcal{T}_y$ and *sigkey* as $\mu\alpha.\ \mathcal{T}_y \rightarrow signature$. The value k : *sigkey* is a sealing function and *verkey* is the corresponding unsealing function. If a value is applied to the sealing function of a value k : *sigkey*, the resulting value of type *signature* is interpreted as signature. Applying the unsealing function, i.e., the verification key, to a signature returns the sealed value, i.e., the signed message, with the original type.

The type $\mathcal{T}_y$ encodes says-predicates. For signing, however, we need to connect a user's signing key with the principal identifier (verification key), which, in turn, needs to be connected to the says-predicate. This is taken care of by the type $\mathcal{U}_{sk}^o$: the (open) type $\mathcal{U}_{sk}^o$ corresponds to $\mathcal{T}_y^o$ for the purpose of signing. As in the definition of $\mathcal{T}_y^o$, the type $\mathcal{U}_{sk}^o$ is a large disjoint sum type that contains one case for every possible predicate P_k that can occur. The main addition is the refinement of the last element in the tuple that logically relates the signing key and the verification key. Given type $\mathcal{U}_{sk}^o$, a principal can derive the logical refinement required by the type $\mathcal{T}_y$. More precisely, given the logical predicates $\exists z, y.\ sk = (z, y) \wedge y$ says $P_k(x_1, \ldots, x_{\ell_k})$ and $sk = (\_, vk)$, one can derive vk says $P_k(x_1, \ldots, x_{\ell_k})$. The formula $sk = (\_, vk)$ originates from extracting the verification key from the signing key.

Creating fresh values and seals. In RCF, we use only the function mkUn to create fresh values of type *unit*; these values can be combined to form more complex types, for instance, using disjoint unions, tuples, or logical refinements. mkUn takes as input a value of type *unit* and outputs a fresh value of type *unit*.

A glance at Table A.1 reveals that the empty tuple () is the only non-functional value that occurs in RCF. Naturally, one may wonder how equality checks that occur in the API methods in form of if-statements, can ever fail. The intuitive reason is that RCF does not actually perform any equality checks but enters the then-branch of an if-statement

$\mathsf{mkUn}:$	$\mathit{unit} \ \to\ \mathit{unit}$
$\mathsf{mkSeal}\langle\alpha\rangle:$	$\mathit{unit} \ \to\ (\,(\alpha \to \mathit{unit}) * (\mathit{unit} \to \alpha)\,)$
$\mathsf{fail}\langle\alpha\rangle:$	$\mathit{unit} \ \to\ \alpha$
$\mathsf{List.member}^{(i,j)}\langle\alpha_1,\ldots,\alpha_i,\beta_{i+1},\ldots,\beta_j\rangle:$	
	$(y_1 : \alpha_1) \ \to\ \cdots \ \to\ (y_i : \alpha_i) \ \to$
	$(\ell : \alpha_1 * \cdots * \alpha_i * \beta_{i+1} * \cdots * \beta_j \ \mathit{list}) \ \to$
	$\{x : \mathit{bool} \mid x = \mathsf{true} \Leftrightarrow \exists y_{i+1},\ldots,y_j.\ (y_1,\ldots,y_j) \in \ell\}$
$\mathsf{List.get}^{(i,j)}\langle\alpha_1,\ldots,\alpha_i,\beta_{i+1},\ldots,\beta_j\rangle:$	
	$(y_1 : \alpha_1) \ \to\ \cdots \ \to\ (y_i : \alpha_i) \ \to$
	$(\ell : \alpha_1 * \cdots * \alpha_i * \beta_{i+1} * \cdots * \beta_j \ \mathit{list}) \ \to$
	$\{(y'_1,\ldots,y'_i,y_{i+1},\ldots,y_j) : \alpha_1 * \cdots * \alpha_i * \beta_{i+1} * \cdots * \beta_j \mid$
	$\quad (y_1,\ldots,y_j) \in \ell \wedge \bigwedge_{k=1}^{i} y'_k = y_k\}$
$\mathsf{func}^{r}_{op}:$	$(x : \mathit{bitstring}) \ \to\ (y : \mathit{bitstring}) \ \to\ \{z : \mathit{bool} \mid z = \mathsf{true} \Leftrightarrow x \ op \ y\}$
$\mathsf{func}^{e}_{op}:$	$(x : \mathit{bitstring}) \ \to\ (y : \mathit{bitstring}) \ \to\ \{z : \mathit{bitstring} \mid z = x \ op \ y\}$
$\mathsf{PKI}:$	$(x : \mathit{uid}_{pub}) \ \to\ \{y : \mathit{verkey} \mid x = y\}$

Table A.6.: Typed library functions used by the API methods.

under the premise that the equality check succeeded and it enters the else-branch under the premise that the check failed. Since RCF type *unit* captures, for instance, integers and strings, the equality checks in a type-checked program will be performed on meaningful values.

$\mathsf{mkUn}: \ \mathit{unit} \ \to\ \mathit{unit}$

We encode cryptographic primitives using a sealing-based encoding [148]. In RCF, seals are created using the polymorphic function $\mathsf{mkSeal}\langle\alpha\rangle$ that takes as input a value of type *unit* and returns a seal for type α. The argument to mkSeal could be a (textual) description of the seal. The first component of the returned pair is the sealing function, the second component is the unsealing function.

$\mathsf{mkSeal}\langle\alpha\rangle: \ \mathit{unit} \ \to\ (\,(\alpha \to \mathit{unit}) * (\mathit{unit} \to \alpha)\,)$

Implementing malleable zero-knowledge proofs using a sealing-based abstraction. Several possibilities for implementing zero-knowledge proofs with seals exist. The implementation in particular has to exclude the following attack: given a valid proof p for a disjunctive statement $\mathit{stm}_1 \vee \mathit{stm}_2$, it must not be possible to determine whether only the statement stm_1 is valid, only the statement stm_2 is valid, or stm_1 and stm_2 both are valid. More precisely, we need to prevent an attacker from abstractly using the malleability to separate a disjunction and verifying the left and the right branch individually, thus determining which branches are valid.

$\mathsf{getOperation}^r$:	$(op : string) \rightarrow ((x : bitstring) \rightarrow (y : bitstring) \rightarrow bool)$
$\mathsf{getOperation}^e$:	$(op : string) \rightarrow ((x : bitstring) \rightarrow (y : bitstring) \rightarrow bitstring)$
rand :	$unit \rightarrow random$
sign :	$(sk : sigkey) \rightarrow (m : \mathcal{U}^o_{sk}) \rightarrow signature$
check_{sig} :	$(y : verkey) \rightarrow (sig : signature) \rightarrow \mathcal{T}_y\{verkey/\alpha\}$
$\mathsf{storeSK}$:	$sigkey \rightarrow uid$
$\mathsf{restoreSK}$:	$uid \rightarrow sigkey$
$\mathsf{computeR}$:	$(x : bitstring) \rightarrow (r : bitstring) \rightarrow bitstring$
$\mathsf{computePsd}$:	
	$(sk : sigkey) \rightarrow (s : string) \rightarrow \{x : pseudo \mid \exists y, z.\ sk = (y, z) \wedge \mathsf{SSP}(z, s, x)\}$
$\mathsf{computeIDR}$:	$(vk_{EA} : bitstring) \rightarrow (vk : bitstring) \rightarrow (r : bitstring) \rightarrow$
	$(R : bitstring) \rightarrow (s : string) \rightarrow \{idr : pseudo \mid \mathsf{EscrowInfo}(vk_{EA}, vk, R, s, idr)\}$
commit :	$bitstring * random \rightarrow commitment$
$\mathsf{openCommit}$:	$commitment \rightarrow bitstring * random$
commit_{sk} :	$sigkey * random \rightarrow commitment$
$\mathsf{openCommit}_{sk}$:	$commitment \rightarrow sigkey * random$
$\mathsf{getSome}$:	$(x : (unit * unit)\ option) \rightarrow \{y : unit * unit \mid x = \mathsf{Some}\ \ y\}$
$\mathsf{getRevealed}$:	$(x : unit\ RevHid) \rightarrow \{y : unit \mid x = \mathsf{Revealed}\ \ y\}$
$\mathsf{commitZK}$:	$statement * random \rightarrow zero\text{-}knowledge$
openZK :	$zero\text{-}knowledge \rightarrow statement * random$
$\mathsf{stripStm}$:	$statement \rightarrow statement$
$\mathsf{checkZK}$:	$proof \rightarrow bool$
$\mathsf{fakestm}$:	$formula \rightarrow statement$
$\mathsf{createZK}^{\mathsf{e}}$:	$statement \rightarrow random \rightarrow proof * zero\text{-}knowledge * statement$
$\mathsf{createZK}$:	$statement \rightarrow random \rightarrow proof * zero\text{-}knowledge * statement$
rerand_{stm} :	$statement \rightarrow statement \rightarrow statement$
$\mathsf{checkEq}^1\langle\alpha\rangle$:	$\alpha\ RevHid \rightarrow commitment \rightarrow \alpha\ option \rightarrow$
	$(bitstring * commitment)\ list\ ref \rightarrow bool$
$\mathsf{checkEq}$:	$statement \rightarrow formula \rightarrow bool$
verify_{stm} :	$statement \rightarrow bool$
verify :	$proof \rightarrow formula \rightarrow bool$
hide_{stm} :	$statement \rightarrow formula \rightarrow statement$
$\mathsf{combineOr}$:	$proof \rightarrow formula \rightarrow random \rightarrow proof$
$\mathsf{commuteOr}$:	$proof \rightarrow statement \rightarrow proof$
$\mathsf{commuteAnd}$:	$proof \rightarrow statement \rightarrow proof$

Table A.7.: Typed auxiliary functions used by the API methods.

We prevent this kind of distinction by introducing the value *zkv* : *zero-knowledge*. Intuitively, this value *zkv* models a cryptographic zero-knowledge proof, which makes a disjunction inseparable.[1]

We implement *zkv* using a dedicated seal that is only available to the API methods. This seal takes as input the proven statement *stm* and a randomness r. The returned handle is used as *zkv*. Different randomness results in different *zkv* values, reflecting that many different zero-knowledge proofs for the same statement exist. Having access to the unsealing function, the verification function will use *zkv* to retrieve the sealed statement stm' and enforce that stm' and *stm* of the proof to be verified match. More precisely, the structure as well as the commitments of stm' and *stm* have to be equal. Since the attacker does not have access to the unsealing function, she cannot produce a *zkv* value that matches either branch. Consequently, trying to split a disjunction and constructing a proof to determine which branches are valid fails.

Notation. In the following implementation, we will often write code of the form

$$\text{let } (x, y) = M\ N; A,$$

i.e., we apply a split operation on an expression. Technically, this is not valid RCF (see Table A.8). We use this notation as shorthand for

$$\begin{array}{l}\text{let } z = M\ N;\\ \text{let } (x, y) = z; A,\end{array}$$

since it is more readable: the code is shorter and it contains one variable less.

We will use this notation only when the formula $\{(x, y) = z\}$ (e.g., as introduced by rule Exp Split) is not relevant for the remaining type-checking process and we will only use this notation for the auxiliary functions. For the verification code macros and the verification function, we will strictly adhere to the RCF syntax.

External library functions and internal auxiliary functions. The API uses several standard library functions and auxiliary functions. These provide commonly used functionality such as list operations and computing hash values. The former are provided by standard libraries of programming languages and we implement the latter. Table A.7 overviews the used functions with their type.

In the following implementation, we focus on the functional part and do not assign line numbers to error-handling code because this code always type-checks. For instance, fail (see below) has return value α that always matches the required type.

fail$\langle\alpha\rangle$: $unit \to \alpha$:

The function fail implements the standard exception mechanism of modern programming languages.

[1] In fact, if a cryptographic proof leaked which branch of a disjunction holds true, then it would not be zero-knowledge: this proof reveals more than just the bare validity of the proven statement.

$\mathsf{List.member}^{(i,j)}\langle\alpha_1,\dots,\alpha_i,\beta_{i+1},\dots,\beta_j\rangle : (y_1 : \alpha_1) \to \dots \to (y_i : \alpha_i)$
$\to (\ell : \langle\alpha_1,\dots,\alpha_i,\beta_{i+1},\dots,\beta_j\rangle \mathit{list})$
$\to \{x : \mathit{bool} \mid x = \mathsf{true} \Leftrightarrow \exists y_{i+1},\dots,y_j.\ (y_1,\dots,y_j) \in \ell\}$

and

$\mathsf{List.get}^{(i,j)}\langle\alpha_1,\dots,\alpha_i,\beta_{i+1},\dots,\beta_j\rangle : (y_1 : \alpha_1) \to \dots \to (y_i : \alpha_i)$
$\to (\ell : \langle\alpha_1,\dots,\alpha_i,\beta_{i+1},\dots,\beta_j\rangle \mathit{list})$
$\to \{(y'_1,\dots,y'_i,y_{i+1},\dots,y_j) : \alpha_1 * \dots * \alpha_i * \beta_{i+1} * \dots * \beta_j \mid$
$(y_1,\dots,y_j) \in \ell \bigwedge_{k=1}^{i} y'_k = y_k\}$:

The two functions implement the list membership functionality and the list element retrieval functionality for lists of tuples. In the implementation, there are lists that contain tuples of different arity. Rather than stating a strongly-typed function for every such occasion, we add parameters i and j. The parameter j denotes the arity of the tuples stored in the list. The parameter i enables us to ask for statements of the form "is there a tuple in the list where the first i elements are y_1 to y_i".

For instance, let ℓ be a list that contains triples. If we want to know whether there is a triple in ℓ that has as first two elements a and b, we call $\mathsf{List.member}^{(2,3)}\ a\ b\ \ell$. The return value is true if and only if there is a tripe $(a,b,c) \in \ell$ for some c. This fact is also reflected in the logical refinement of the return value.

The semantics of $\mathsf{List.get}$ is similar: it works on lists that contain tuples of arity j and it takes i elements as input. The return value is the first element in the list, that contains as the first i elements y_1 to y_i.

For instance, let ℓ be a list comprising the triples (a,b',c) and (a,b,c) and assume that all variables are different. Then, $\mathsf{List.get}^{(1,3)}\ a\ \ell$ returns (a,b',c) because this triple occurs in the list first, the call $\mathsf{List.get}^{(2,3)}\ a'\ b\ \ell$ throws an exception because there is no triple in the list with the first component equal to a', and the call $\mathsf{List.get}^{(2,3)}\ a\ b\ \ell$ returns (a,b,c). The list membership is also reflected in the logical refinement.

In the following, we will not consider the intermediate values y'_k and the corresponding equalities $y'_k = y_k$. Instead, we immediately use the refinement $(y_1,\dots y_j) \in \ell$.

$\mathsf{func}^r_{op} : (x : \mathit{bitstring}) \to (y : \mathit{bitstring}) \to \{z : \mathit{bool} \mid z = \mathsf{true} \Leftrightarrow x\ \mathit{op}\ y\}$

and

$\mathsf{func}^e_{op} : (x : \mathit{bitstring}) \to (y : \mathit{bitstring}) \to \{z : \mathit{bitstring} \mid z = x\ \mathit{op}\ y\}$:

The families of functions func^r_{op} and func^e_{op} correspond to the usual mathematical comparison and computation functions.

For instance, if $x \le y$, then $\mathsf{func}^r_{\le}\ x\ y$ returns the Boolean value $z = \mathsf{true}$, which is refined with the logical predicate $z = \mathsf{true} \Leftrightarrow x \le y$; if $z = x + y$, then $\mathsf{func}^e_{+}\ x\ y$ returns a value $z' = z$, which is refined with the logical predicate $z' = x + y$.

$\mathsf{PKI} : (x : \mathit{uid}_{\mathit{pub}}) \to \{y : \mathit{verkey} \mid x = y\}$:

The PKI function represents a public-key infrastructure. It takes as input a value $x : \mathit{uid}_{\mathit{pub}}$, and returns a value $y : \mathit{verkey}$ that is equal to x. From a type-checking point of view, the PKI allows us to upgrade the type of a value from *unit* to *verkey*, a concrete implementation ensures that a key is trustworthy by checking that the key is part of the PKI.

We assume that the PKI function is initialized and ready to use. Concretely establishing a PKI is a well-recognized bootstrapping step for which many solutions exists (e.g., centralized PKIs such as VeriSign [216] or decentralized solutions such as webs of trust [207, 25]) and the function PKI reflects the PKI into the API.

$\mathsf{getOperation}^r : (op : string) \to ((x : bitstring) \to (y : bitstring) \to bool)$
and

$\mathsf{getOperation}^e : (op : string) \to ((x : bitstring) \to (y : bitstring) \to bitstring)$:
These helper functions translate a function description into the corresponding function and they are used in the implementation of the unrefined verification method. The function $\mathsf{getOperation}^r$ takes as input $op \in \{``=", ``\neq", ``\leq", ``<", ``>", ``\geq"\}$ and $\mathsf{getOperation}^e$ takes as input $op \in \{``+", ``-", ``*"\}$.

For instance, the call $\mathsf{getOperation}^r$ $``="$ returns the usual equality check, i.e., $\mathsf{getOperation}^r$ $``="$ x x returns true and $\mathsf{getOperation}^e$ $``+"$ returns the usual add operation, i.e., $\mathsf{getOperation}^e$ $``+"$ 4 3 returns 7.

```
let $\mathsf{getOperation}^r$ $(op : string)$: $((x : bitstring) \to (y : bitstring) \to bool)$ =
  if $op$ = $``\leq"$ then
    $\mathsf{func}^r_{\leq}$
  else if $op$ = $``<"$ then
    $\mathsf{func}^r_{<}$
  else if $op$ = $``="$ then
    $\mathsf{func}^r_{=}$
  else if $op$ = $``>"$ then
    $\mathsf{func}^r_{>}$
  else if $op$ = $``\geq"$ then
    $\mathsf{func}^r_{\geq}$
  else
    $\mathsf{fail}\langle bitstring \to bitstring \to bool\rangle$ ()
```

```
let $\mathsf{getOperation}^e$ $(op : string)$: $((x : bitstring) \to (y : bitstring) \to bitstring)$ =

  if $op$ = $``+"$ then
    $\mathsf{func}^e_{+}$
  else if $op$ = $``-"$ then
    $\mathsf{func}^e_{-}$
  else if $op$ = $``\cdot"$ then
    $\mathsf{func}^e_{\cdot}$
  else
    $\mathsf{fail}\langle bitstring \to bitstring \to bool\rangle$ ()
```

rand : $unit \rightarrow random$:

Randomness is an essential ingredient in zero-knowledge proofs: the commitments used in a proof require randomness and even the proof itself contains randomness. In RCF, randomness is generated by calling the rand function that takes as argument of type *unit* and returns a fresh (random) value of type *unit*.

```
let rand ($x : unit$): $random$ =
    mkUn ()
```

sign : $(sk : sigkey) \rightarrow (m : \mathcal{U}^o_{sk}) \rightarrow signature$:

The signing function takes as input a signing key $sk : sigkey$ and a message $m : \mathcal{U}^o_{sk}$ and returns a signature of type *signature* for m.

Despite of what we stated above, the sign function takes as input a signing key directly instead of a public identifier of type *uid*. The reason is that the sign function is not part of the API but an internal function. The API methods that are exposed to programmers expect values of type *uid*.

```
let sign ($sk : sigkey$) ($m : \mathcal{U}^o_{sk}$): $signature$ =
    let ($x, y$) = $sk$;
    $x$ ($y, m$)
```

The type $\mathcal{U}^o_{sk}$ by itself not closed because the variable *sk* is free in $\mathcal{U}^o_{sk}$. Due to the dependent function that binds *sk* in the type $\mathcal{U}^o_{sk}$, the type of the signing function is closed. The user identifier y that is signed along with the message occurs in the logical refinement and determines the actor of the says modality. Intuitively, the logical refinement is of the form y says m.

check$_{sig}$: $(y : verkey) \rightarrow (sig : signature) \rightarrow \mathcal{T}_y\{verkey/\alpha\}$:

The verification function takes as input a verification key $y : verkey$ and a signature $sig : signature$ and returns a value of type $\mathcal{T}^o_y\{verkey/\alpha\}$, i.e., the type $\mathcal{T}^o_y$ where all occurrences of α are replaced by *verkey*.

```
let check$_{sig}$ ($y : verkey$) ($sig : signature$): $\mathcal{T}_y\{verkey/\alpha\}$ =
    let $x'$ = $y$ $sig$;
    let ($x, m$) = $x'$;
    if $x$ = $y$ then
      $m$
    else
      fail⟨$\mathcal{T}_y\{verkey/\alpha\}$⟩ ()
```

In particular, the implementation ensures that the verification key used to verify the signature is also the verification key that is signed along with the message.

storeSK : $\mathit{sigkey} \;\rightarrow\; \mathit{uid}$
and

restoreSK : $\mathit{uid} \;\rightarrow\; \mathit{sigkey}$:

The functions storeSK and restoreSK are internally used to store and restore a signing key to and from a handle, respectively. Every principal has her own local two functions. The function storeSK takes as input a signing key and returns a handle for that key, the function restoreSK takes as input a handle and returns the stored signing key.

For instance, if *sk* is a secret key, calling storeSK *sk* locally stores *sk* and returns a handle *hdl*. That handle points to the stored key and can be used within the API. In practice, *hdl* is realized, e.g., as a filename or a URI. Since *hdl* does not contain any secret information, it can be sent over the internet. If the user with the locally stored secret key calls restoreSK *hdl*, the stored key *sk* is returned.

We symbolically implement these functions as a seal. The fresh handle to the signing key will only resolve if applied by its owner to the corresponding local restoreSK function to retrieve the signing key. To every other principal, it is of no use.

```
let (storeSK, restoreSK) = mkSeal⟨sigkey⟩ ();
```

computeR : $(x : \mathit{bitstring}) \rightarrow (r : \mathit{bitstring}) \rightarrow \mathit{bitstring}$,

computePsd : $(\mathit{sk} : \mathit{sigkey}) \rightarrow (s : \mathit{string}) \rightarrow$
$\{x : \mathit{pseudo} \mid \exists y, z.\ \mathit{sk} = (y, z) \wedge \mathsf{SSP}(z, s, x)\}$,
and

computeIDR : $(\mathit{vk}_{EA} : \mathit{bitstring}) \rightarrow (\mathit{vk} : \mathit{bitstring}) \rightarrow (r : \mathit{bitstring}) \rightarrow$
$(R : \mathit{bitstring}) \rightarrow (s : \mathit{string}) \rightarrow \{\mathit{idr} : \mathit{pseudo} \mid \mathsf{EscrowInfo}(\mathit{vk}_{EA}, \mathit{vk}, R, s, \mathit{idr})\}$ The

functions computeR, computePsd, and computeIDR model the mathematical operations detailed in Section 2.4. Although computeIDR and computePsd perform the same mathematical operations, we use different RCF functions to distinguish between their different application scenarios and, in particular, between their different input and output types.

computeR takes as input two values r and *vk* of type *bitstring* and returns a value R of type *bitstring*.

```
let computeR =
  let (a, _) = mkSeal⟨bitstring ∗ bitstring⟩ ();
  fun (x : bitstring) → fun (y : bitstring) → a (x, y)
```

computePsd takes as input the signing key *sk* : *sigkey* and a service description s : *string*, and it returns the SSP *psd* : *pseudo* for which additionally the logical predicate $\exists y, z.\ \mathit{sk} = (y, z) \wedge \mathrm{SSP}(z, s, \mathit{psd})$ holds.

```
let computePsd =
  let (a, _) = mkSeal⟨verkey ∗ string⟩ ();
  fun (x : sigkey) → fun (s : string) →
    let (_, vk) = x;
    let psd = a (vk, s);
    let _ = assume(SSP(vk, s, psd));
    psd
```

computeIDR takes as input five values vk_{EA}, vk, r, and R of type *bitstring*, and s of type *string*; it returns a value *idr* of type *pseudo* that additionally carries the logical formula EscrowInfo(vk_{EA}, vk, R, s, idr). The inputs are necessary to bind the corresponding values occurring in the refinement; the value r occurs only implicitly in the formula since it is only used to mathematically compute the value *idr*.

```
let computeIDR =
  let (a, _) = mkSeal⟨bitstring ∗ bitstring ∗ bitstring ∗ bitstring ∗ string⟩ ();
  fun (vk_EA : bitstring) → fun (vk : bitstring) → fun (r : bitstring) →
     fun (R : bitstring) → fun (s : bitstring) →
          let idr = a (vk_TTP, vk, r, R, s);
          let _ = assume(EscrowInfo(vk_EA, vk, R, s, idr));
          idr
```

commit : *bitstring* ∗ *random* → *commitment*
and

openCommit : *commitment* → *bitstring* ∗ *random*:
The functions commit and openCommit are sealing and unsealing functions, respectively, for values of type *bitstring*. We use the seal directly to model the commitment function and the handle to model commitments. The unsealing function openCommit is used only internally in the API methods; it cannot be accessed from the outside and it is not exported by the API.

In practice, the commit function corresponds to the mathematical operation that computes commitments from a given bit string and randomness. The openCommit function in general does not exist or it is computationally infeasible to compute. Symbolically, it is necessary for the zero-knowledge verification: in the concrete proof verification, we use algebraic properties. Since these cannot be soundly modeled symbolically [214], we symbolically open the commitments and use the committed values for the verification [34].

```
let (commit, openCommit) = mkSeal⟨bitstring ∗ random⟩ ()
```

commit_{sk} : $\mathit{sigkey} * \mathit{random} \to \mathit{commitment}$
and

$\mathsf{openCommit}_{sk}$: $\mathit{commitment} \to \mathit{sigkey} * \mathit{random}$:

The functions commit_{sk} and $\mathsf{openCommit}_{sk}$ are sealing and unsealing functions, respectively, for values of type $\mathit{sigkey} * \mathit{random}$. We use the seal directly to model the dedicated commitment function for signing keys and the handle to model the commitments to signing keys; the *random* part corresponds to the randomness used in the creation of the commitment. The unsealing function $\mathsf{openCommit}_{sk}$ is not available outside of the API implementation (in fact, the unsealing function is not of kind public, so the type system prevents us from giving the function to the attacker).

Analogous to the commit and openCommit operation, the commit_{sk} and the $\mathsf{openCommit}_{sk}$ operation model commitments to secret keys rather than bit strings.

```
let (commit$_{sk}$, openCommit$_{sk}$) = mkSeal$\langle sigkey * random \rangle$ ()
```

getSome : $(x : (\mathit{unit} * \mathit{unit})\ \mathit{option}) \to \{y : \mathit{unit} * \mathit{unit} \mid x = \mathsf{Some}\ y\}$
and

getRevealed : $(x : \mathit{unit}\ \mathit{RevHid}) \to \{y : \mathit{unit} \mid x = \mathit{RevHid}\ y\}$:

These two functions retrieve and return the values stored in a given *option* type and in a given *RevHid* type, respectively. If no value was contained, i.e., $x = \mathsf{None}$ or $x = \mathsf{Hidden}\ i$ for some i, an exception is raised.

These functions are abbreviations so that we can call a function and do not need to write match-statements in our code.

```
let getSome ($x$ : ($unit * unit$) $option$) : {$y$ : $unit * unit$ | $x$ = Some $y$} =
  match $x$ with
    | Some($y_1, y_2$) ⟹
      ($y_1, y_2$)
    | _ ⟹
      fail$\langle unit * unit \rangle$ ()
```

```
let getRevealed ($x$ : $unit$ $RevHid$) : {$y$ : $unit$ | $x$ = Revealed $y$} =
  match $x$ with
    | Revealed $y$ ⟹
      $y$
    | _ ⟹
      fail$\langle unit \rangle$ ()
```

commitZK : *statement* $*$ *random* → *zero-knowledge*
and

openZK : *zero-knowledge* → *statement* $*$ *random*:
The commitZK and the openZK functions are sealing and unsealing functions for values of type *statement* $*$ *random*, respectively. They are used to create the *zkv* value, the first component of values of type *proof*. The *random* part corresponds to the randomness used in a zero-knowledge proof. Both functions are only used internally and are not exported.

In practice, the commitZK function corresponds to the concrete mathematical computation of a zero-knowledge proof.[2] Analogous to the openCommit and the openCommit$_{sk}$ function, we need the openZK function to enable the symbolic zero-knowledge verification.

```
let (commitZK, openZK) = mkSeal$\langle statement * random \rangle$ ()
```

stripStm : *statement* → *statement*:
The function stripStm takes as input a value of type *statement* and returns that value where all the opening information are set to None, i.e., all opening information from commitments are removed.

```
stripStm $(stm : statement)$: $statement$ =
  match $stm$ with
    | $\mathsf{Says}_\mathsf{p}(c_z, \_, c_{sig}, \_, P)$ $\Longrightarrow$
      let $P'$ = match $P$ with
        | $P_1^P(c_{arg_1}, \_, \dots, c_{arg_{n_1}}, \_)$ $\Longrightarrow$
          $P_1^P(c_{arg_1}, \mathsf{None}, \dots, c_{arg_{n_1}}, \mathsf{None})$
        | $P_2^P(c_{arg_1}, \_, \dots, c_{arg_{n_2}}, \_)$ $\Longrightarrow$
          $\vdots$
        | $P_m^P(c_{arg_1}, \_, \dots, c_{arg_{n_m}}, \_)$ $\Longrightarrow$
          $\vdots$
          $P_m^P(c_{arg_1}, \mathsf{None}, \dots, c_{arg_{n_m}}, \mathsf{None})$ ;
      $\mathsf{Says}_\mathsf{p}(c_z, \mathsf{None}, c_{sig}, \mathsf{None}, P')$
    | $\mathsf{SSP}_\mathsf{p}(c_z, \_, c_s, \_, c_{psd}, \_, c_x)$ $\Longrightarrow$
        $\vdots$
    | $\mathsf{LM}_\mathsf{p}(c_x, \_, c_b, \_, \ell)$ $\Longrightarrow$
        $\vdots$
    | $\mathsf{LNM}_\mathsf{p}(c_x, \_, \ell)$ $\Longrightarrow$
        $\vdots$
```

[2] In a Groth-Sahai implementation, the output to the commitZK function corresponds to the values $\vec{\pi}$ and $\vec{\theta}$, see Groth and Sahai [128], revised May 23, 2012, page 12

```
    | REL_p(c_x, _, op, c_y, _)  ==>
        ⋮
    | EscrowInfo_p(z, c_x, _, c_R, _, c_s, _, c_idr, _, c_r)  ==>
        ⋮
    | And_p(stm_1, stm_2)  ==>
      And_p(stripStm stm_1, stripStm stm_2)
    | Or_p(stm_1, stm_2)  ==>
      Or_p(stripStm stm_1, stripStm stm_2)
    | _  ==>
      fail⟨proof * unit * statement⟩ ()
```

Listing A.15: Implementation of stripStm.

checkZK : *proof* → *bool*:

The checkZK methods is used to prevent the following symbolic attack that generally does not exist for zero-knowledge proofs: let z be a valid zero-knowledge proof and let d be a commitment contained within that proof. Furthermore, let c be a commitment different from d. If the attacker could replace d with c inside of z and the proof would still verify, the attacker would learn that the d and c are commitments to the same value.

Without the checkZK method, this attack is possible symbolically. In a concrete proof, this is prevented by mathematical properties. For instance, in the Groth-Sahai scheme, the randomness used to compute the commitments is also contained in the proof. If these randomnesses do not match, the verification fails.

```
let checkZK (p : proof) =
  match p with
    | ZK(zkv, stm)  ==>
      let (stm', _) = openZK zkv;
      let s_1 = stripStm stm;
      let s_2 = stripStm stm';
      if s_1 = s_2 then
        true
      else
        false
    | _  ==>
      fail⟨bool⟩ ()
```

fakestm : *formula* → *statement*:

We need two statements for a disjunctive proof: a valid branch and a (possibly) invalid branch. These have to be indistinguishable. The fakestm method generates statements that can be used as invalid branches of a disjunction. More precisely, it takes as input a formula and creates a statement that is indistinguishable from a statement contained in a valid zero-knowledge proof. The values revealed in the formula will be revealed in the statement; the hidden values will be hidden in the statement. The respective commitments will contain randomly chosen values.

The fakestm function takes as input a formula f and produces a statement for that formula. Intuitively, fakestm pattern-matches the given formula, extracts all the contained information to create commitments at the proper places. All cryptographic values such as digital signatures that occur in the proof but not in the formula are chosen randomly and are inserted into the proof. Consequently, a proof using a statement that was created by fakestm will not verify but they can be used as the "false" branch in a disjunctive proof.

```
fakestm $(f : \mathit{formula})$ : $\mathit{statement}$ =
  match $f$ with
    | And$(f_1, f_2)$ $\Longrightarrow$
      And$_\mathsf{p}$(fakestm $f_1$, fakestm $f_2$)
    | Or$(f_1, f_2)$ $\Longrightarrow$
      Or$_\mathsf{p}$(fakestm $f_1$, fakestm $f_2$)
    | Says$(vk', p')$ $\Longrightarrow$
      match p' with $P(y'_1, \ldots, y'_n)$ $\Longrightarrow$
        let $vk$ = getRevealed $vk'$;
        let $y_1$ = getRevealed $y'_1$;
        ⋮
        let $y_n$ = getRevealed $y'_n$;
        let $sig$ = rand();
        let $r_{sig}$ = rand();
        let $c_{sig}$ = commit$(sig, r_{sig})$;
        let $r_{vk}$ = rand();
        let $c_{vk}$ = commit$(vk, r_{vk})$;
        let $r_1$ = rand();
        let $c_1$ = commit$(y_1, r_1)$;
        ⋮
        let $r_n$ = rand();
        let $c_n$ = commit$(y_n, r_n)$;
        Says$_\mathsf{p}$$(c_{sig}$, Some$(sig, r_{sig})$, $c_{vk}$, Some$(vk, r_{vk})$,
            $P(c_1$, Some$(y_1, r_1), \ldots, c_n$, Some$(y_n, r_n)))$
      | _ $\Longrightarrow$
        fail$\langle \mathit{statement} \rangle$ ()
    | SSP$(vk', s', psd')$ $\Longrightarrow$
      let $vk$ = getRevealed $vk'$;
      let $s$ = getRevealed $s'$;
      let $psd$ = getRevealed $psd'$;
      let $r_{vk}$ = rand();
```

```
        let $c_{vk}$ = commit($vk, r_{vk}$);
        let $r_s$ = rand();
        let $c_s$ = commit($s, r_s$);
        let $r_{psd}$ = rand();
        let $c_{psd}$ = commit($psd, r_{psd}$);
        let $x$ = rand();
        let $r_x$ = rand();
        let $c_x$ = commit($x, r_x$);
        SSP$_{\mathsf{p}}$($c_{vk}$, Some($vk, r_{vk}$), $c_s$, Some($s, r_s$), $c_{psd}$,Some($psd, r_{psd}$), $c_x$)
    | LM($x', b', \ell$) $\Longrightarrow$
        let $x$ = getRevealed $x'$;
        let $b$ = getRevealed $b'$;
        let $r_x$ = rand();
        let $c_x$ = commit($x, r_x$);
        let $r_b$ = rand();
        let $c_b$ = commit($b, r_b$);
        LM$_{\mathsf{p}}$($c_x$, Some($x, r_x$), $c_b$, Some($b, r_b$), $\ell$)
    | LNM($x', \ell$) $\Longrightarrow$
        let $x$ = getRevealed $x'$;
        let $r_x$ = rand();
        let $c_x$ = commit($x, r_x$);
        LNM$_{\mathsf{p}}$($c_x$, Some($x, r_x$), $\ell$)
    | REL($x', op, y'$) $\Longrightarrow$
        let $x$ = getRevealed $x'$;
        let $y$ = getRevealed $y'$;
        let $r_x$ = rand();
        let $c_x$ = commit($x, r_x$);
        let$r_y$ = rand();
        let $c_y$ = commit($y, r_y$);
        REL$_{\mathsf{p}}$($c_x$, Some($x, r_x$), $op$, $c_y$, Some($y, r_y$))
    | EscrowInfo($z', x', r', s', idr'$) $\Longrightarrow$
        let $z$ = getRevealed $z'$;
        let $x$ = getRevealed $x'$;
        let $r$ = getRevealed $r'$;
        let $s$ = getRevealed $s'$;
        let $idr$ = getRevealed $idr'$;
        let $r_x$ = rand();
        let $c_x$ = commit($x, r_x$);
        let $r_r$ = rand();
        let $c_r$ = commit($r, r_r$);
        let $R$ = computeR $x$ $r$;
        let $r_R$ = rand();
        let $c_R$ = commit($R, r_R$)
        let $r_s$ = rand();
        let $c_s$ = commit($s, r_s$);
        let $r_{idr}$ = rand();
        let $c_{idr}$ = commit($idr, r_{idr}$)
        EscrowInfo$_\mathsf{p}$($z, c_x$, Some($x, r_x$), $c_R$, Some($R, r_R$),
                    $c_s$, Some($s, r_s$), $c_{idr}$, Some($idr, r_{idr}$), $c_r$)
      | _ $\Longrightarrow$
        fail$\langle statement \rangle$ ()
```

Listing A.17: Implementation of fakestm.

createZK$^\mathsf{e}$: *statement* → *random* → *proof* ∗ *zero-knowledge* ∗ *statement*
and

createZK : *statement* → *random* → *proof* ∗ *zero-knowledge* ∗ *statement*:
The API methods are a convenient interface to create zero-knowledge methods.

For instance, the mkSays method takes as input a public identifier and a predicate; it outputs a zero-knowledge proof for a Says$_\mathsf{p}$-statement, i.e., for the validity of a signature verification. To create the zero-knowledge proof, however, only the public identifier (verification key), the signature, and the predicate are necessary. The createZK method enables the creation of zero-knowledge proofs based on the values contained in the proof: given a verification key *vk*, a signature *sig*, and a predicate *p*, createZK will create a zero-knowledge proof for the statement *vk* says *p*. The proof verifies if *sig* is a valid signature on *p* that verifies using key *vk*.

The createZK function takes as input a non-conjunctive, non-disjunctive statement *stm*, a randomness *r* and creates a zero-knowledge proof *p* from the opening information contained in *stm*; the randomness *r* is used to compute the corresponding zero-knowledge value *zkv*.

```
createZK$^\mathsf{e}$ ($stm$ : $statement$) ($r$ : $random$):  $proof * zero\text{-}knowledge * statement$ =
  match $stm$ with
    | Says$_\mathsf{p}$(_, $o_z$, _, $o_{sig}$, $P'$) $\Longrightarrow$
      let ($z, r_z$) = getSome $o_z$;
      let ($sig, r_{sig}$) = getSome $o_{sig}$;
      let $c_z$ = commit ($z, r_z$);
      let $c_{sig}$ = commit($sig, r_{sig}$);
      let $P$ = match $P'$ with
        | $P_1^P$(_, $o_1, \ldots$, _, $o_{n_1}$) $\Longrightarrow$
          let ($arg_1, r_{arg_1}$) = getSome $o_1$;
           ⋮
          let ($arg_{n_1}, r_{arg_{n_1}}$) = getSome $o_{n_1}$;
          let $c_{arg_1}$ = commit($arg_1, r_{arg_1}$);
           ⋮
```

```
                let $c_{arg_{n_1}}$ = $\mathsf{commit}(arg_{n_1}, r_{arg_{n_1}})$;
                $P_1^P(c_{arg_1}, \mathsf{Some}(arg_1, r_{arg_1}), \ldots, c_{arg_{n_1}}, \mathsf{Some}(arg_{n_1}, r_{arg_{n_1}}))$
            | $P_2^P(\_, o_1, \ldots, \_, o_{n_2})$  $\Longrightarrow$
                ⋮
            | $P_m^P(\_, o_1, \ldots, \_, o_{n_m})$  $\Longrightarrow$
                ⋮
                $P_m^P(c_{arg_1}, \mathsf{Some}(arg_1, r_{arg_1}), \ldots, c_{arg_{n_m}}, \mathsf{Some}(arg_{n_m}, r_{arg_{n_m}}))$;
            | _  $\Longrightarrow$
                $\mathsf{fail}\langle predicate^P \rangle$ ()
        let $stm'$ = $\mathsf{Says_p}(c_z, \mathsf{Some}(z, r_z), c_{sig}, \mathsf{Some}(sig, r_{sig}), P)$;
        let $zkv$ = $\mathsf{commitZK}(stm', r)$;
        let $p$ = $\mathsf{ZK}(zkv, stm')$;
        $(p, zkv, stm')$
    | $\mathsf{SSP_p}(\_, o_z, \_, o_s, \_, o_{psd}, c_x)$  $\Longrightarrow$
            ⋮
    | $\mathsf{LM_p}(\_, o_x, \_, o_b, \ell)$  $\Longrightarrow$
            ⋮
    | $\mathsf{LNM_p}(\_, o_x, \ell)$  $\Longrightarrow$
            ⋮
    | $\mathsf{REL_p}(\_, o_x, op, \_, o_y)$  $\Longrightarrow$
            ⋮
    | $\mathsf{EscrowInfo_p}(z, \_, o_x, \_, o_R, \_, o_s, \_, o_{idr}, c_r)$  $\Longrightarrow$
            ⋮
    | _  $\Longrightarrow$
        $\mathsf{fail}\langle proof * zero\text{-}knowledge * statement \rangle$ ()
```

Listing A.18: Implementation of $\mathsf{createZK^e}$.

```
createZK  $(stm : statement)$  $(r : random)$:  $proof * zero\text{-}knowledge * statement$ =
  match $stm$ with
    | $\mathsf{Says_p}$ _  $\Longrightarrow$
      $\mathsf{createZK^e}$ $stm$ $r$
    | $\mathsf{SSP_p}$ _  $\Longrightarrow$
      $\mathsf{createZK^e}$ $stm$ $r$
    | $\mathsf{LM_p}$ _  $\Longrightarrow$
      $\mathsf{createZK^e}$ $stm$ $r$
    | $\mathsf{LNM_p}$ _  $\Longrightarrow$
      $\mathsf{createZK^e}$ $stm$ $r$
    | $\mathsf{REL_p}$ _  $\Longrightarrow$
      $\mathsf{createZK^e}$ $stm$ $r$
    | $\mathsf{EscrowInfo}_{\mathsf{p}}$ _ $\Longrightarrow$
      $\mathsf{createZK}^{\mathsf{e}}$ $stm$ $r$
    | $\mathsf{And}_{\mathsf{p}}(stm_1, stm_2)$ $\Longrightarrow$
      let $(r_1, r_2)$ = $r$;
      let $(p_1, zkv'_1, stm'_1)$ = createZK $stm_1$ $r_1$;
      let $(p_2, zkv'_2, stm'_2)$ = createZK $stm_2$ $r_2$;
      let $stm'$ = $\mathsf{And}_{\mathsf{p}}(stm'_1, stm'_2)$;
      let $zkv$ = $\mathsf{commitZK}(stm', r)$;
      let $p$ = $\mathsf{ZK}(zkv, stm')$;
      $(p, zkv, stm')$
    | $\mathsf{Or}_{\mathsf{p}}(stm_1, stm_2)$ $\Longrightarrow$
      let $(r_1, r_2)$ = $r$;
      let $(p_1, zkv'_1, stm'_1)$ = createZK $stm_1$ $r_1$;
      let $(p_2, zkv'_2, stm'_2)$ = createZK $stm_2$ $r_2$;
      let $stm'$ = $\mathsf{Or}_{\mathsf{p}}(stm'_1, stm'_2)$;
      let $zkv$ = $\mathsf{commitZK}(stm', r)$;
      let $p$ = $\mathsf{ZK}(zkv, stm')$;
      $(p, zkv, stm')$
    | _ $\Longrightarrow$
      $\mathsf{fail}\langle proof * zero\text{-}knowledge * statement\rangle$ ()
```

Listing A.19: Implementation of createZK.

Internally, the $\mathsf{createZK}^{\mathsf{e}}$ function recomputes all commitments using the opening information contained in the input statement; the commitments for values without opening information (e.g., signing keys) are copied. The createZK function relies on the $\mathsf{createZK}^{\mathsf{e}}$ function for all non-conjunctive and non-disjunctive statements. The cases for the other two kinds of statements recursively call the createZK function, as expected.

rerand_{stm} : *statement* $\rightarrow$ *statement* $\rightarrow$ *statement*:

The rerand_{stm} function takes as input a statement stm to be re-randomized and a statement g that guides this process, i.e., g determines which values in stm are to be re-randomized. More precisely, if an opening information in g is different from None, then the randomness contained in the corresponding commitment in stm is re-randomized with freshly-chosen randomness; commitments for values that do not contain opening information (e.g., signing keys) are always re-randomized.

For instance, let $stm = \mathsf{Says}_{\mathsf{p}}(c_{vk}, \mathsf{Some}(vk, r_{vk}), c_{sig}, \mathsf{P}(c_m, \mathsf{Some}(m, r_m)))$ and let $g = \mathsf{Says}_{\mathsf{p}}(\_, \mathsf{Some}(\_, \_), c_{sig}, \mathsf{P}(\_, \mathsf{None}))$, calling rerand_{stm} stm g returns $\mathsf{Says}_{\mathsf{p}}(c'_{vk}, \mathsf{Some}(vk, r'_{vk}), c'_{sig}, \mathsf{P}(c_m, \mathsf{Some}(m, r_m)))$ where the primed values have changed due to the re-randomization process. For the process, only the structure of g but not the contained values are important.

Notice that using the Groth-Sahai implementation, it is possible to re-randomize values and to choose the randomness that will be added to the randomness contained in the corresponding commitment [42]. Since a symbolic model of such algebraic structures in inherently unsound [214], we do not consider this selective re-randomization. Furthermore, a concrete implementation uses the group operation to combine the freshly-chosen randomness with the already existing one.

```
$\mathsf{rerand}_{stm}$  $(stm : statement)$ $\rightarrow$ $(g : statement)$  :  $statement$ =
  match $stm$ with
    | $\mathsf{Says}_{\mathsf{p}}(c_z, o_z, c_{sig}, o_{sig}, P)$ $\Longrightarrow$
        ⋮
    | $\mathsf{SSP}_{\mathsf{p}}(c_z, o_z, c_s, o_s, c_{psd}, o_{psd}, c_x)$ $\Longrightarrow$
      match $g$ with $\mathsf{SSP}_{\mathsf{p}}(\_, o^g_z, \_, o^g_s, \_, o^g_{psd}, \_)$ $\Longrightarrow$
          let $(c'_z, o'_z)$ =
            if $o^g_z = \mathsf{None}$ then
              $(c_z, o_z)$
            else
              let $r = \mathsf{rand}$ ();
              let $c^n_z$ =
                let $(v, \_)$ = $\mathsf{openCommit}$ $c_z$;
                $\mathsf{commit}$ $(v, r)$;
              let $o^n_z$ =
                if $o_z = \mathsf{None}$ then
                  $\mathsf{None}$
                else
                  let $(v, r_v) = \mathsf{getSome}$ $o_z$;
                  if $c_z = \mathsf{commit}$ $(v, r_v)$ then
                     $\mathsf{Some}$ $(v, r)$
                  else
                     $o_z$;
              $(c^n_z, o^n_z)$;
          let $(c'_s, o'_s)$ =
            if $o^g_s = \mathsf{None}$ then
              $(c_s, o_s)$
            else
              let $r = \mathsf{rand}$ ();
              let $c^n_s$ =
                let $(v, \_)$ = $\mathsf{openCommit}$ $c_s$;
                $\mathsf{commit}$ $(v, r)$;
              let $o^n_s$ =
                if $o_s = \mathsf{None}$ then
                  $\mathsf{None}$
                else
                  let $(v, r_v) = \mathsf{getSome}$ $o_s$;
                  if $c_s = \mathsf{commit}$ $(v, r_v)$ then
```

```
                 Some $(v, r)$
              else
                 $o_s$;
          $(c_s^n, o_s^n)$;
      let $(c'_{psd}, o'_{psd})$ =
        if $o_{psd}^g$ = None then
          $(c_{psd}, o_{psd})$
        else
          let $r$ = rand ();
          let $c_{psd}^n$ =
            let $(v, \_)$ = openCommit $c_{psd}$;
            commit $(v, r)$;
          let $o_{psd}^n$ =
            if $o_{psd}$ = None then
              None
            else
              let $(v, r_v)$ = getSome $o_{psd}$;
              if $c_{psd}$ = commit $(v, r_v)$ then
                 Some $(v, r)$
              else
                 $o_{psd}$;
          $(c_{psd}^n, o_{psd}^n)$;
      let $c'_x$ =
        let $r$ = rand ();
        let $(v, \_)$ = openCommit $c_x$;
        commit $(v, r)$;
      $\mathsf{SSP_p}(c_z^n, o_z^n, c_s^n, o_s^n, c_{psd}^n, o_{psd}^n, c_x^n)$
    | _ $\Longrightarrow$
      fail$\langle proof \rangle$ ()
| $\mathsf{LM_p}(c_x, o_x, c_b, o_b, \ell)$ $\Longrightarrow$
    ⋮
| $\mathsf{LNM_p}(c_x, o_x, \ell)$ $\Longrightarrow$
    ⋮
| $\mathsf{REL_p}(c_x, o_x, op, c_y, o_y)$ $\Longrightarrow$
    ⋮
| $\mathsf{EscrowInfo_p}(z, c_x, o_x, c_R, o_R, c_s, o_s, c_{idr}, o_{idr}, c_r)$ $\Longrightarrow$
    ⋮
| $\mathsf{And_p}(stm_1, stm_2)$ $\Longrightarrow$
  match $g$ with $\mathsf{And_p}(g_1, g_2)$ $\Longrightarrow$
      $\mathsf{And_p}(\mathsf{rerand}_{stm}\ stm_1\ g_1, \mathsf{rerand}_{stm}\ stm_2\ g_2)$
    | _ $\Longrightarrow$
```

```
            fail⟨$statement$⟩ ()
71      | Or_p($stm_1, stm_2$) ⟹
72        match $g$ with Or_p($g_1, g_2$) ⟹
73            Or_p(rerand_{$stm$} $stm_1$ $g_1$, rerand_{$stm$} $stm_2$ $g_2$)
          | _ ⟹
            fail⟨$statement$⟩ ()
        | _ ⟹
          fail⟨$statement$⟩ ()
```

The rerand function is a large case split. It uses the re-randomization guide g : *formula* to determine whether to re-randomize a specific commitment or not. More precisely, if the opening information is different from None (e.g., line 6), then we choose new randomness and apply it to the respective commitment (e.g., line 12). If the original opening information was removed (e.g., line 14) or modified (e.g., line 18), the original opening information is used. Otherwise, the new randomness is inserted (e.g., line 19)

We stress that the check in line 18 is crucial. Otherwise, it is possible to extract the content of a commitment by supplying any value that is different from None.

checkEq^1: α *RevHid* $\rightarrow$ *bitstring* $\rightarrow$ (*bitstring* $*$ *commitment*) *list ref* $\rightarrow$ *bool*:

The function checkEq^1 is used to enforce that hidden values with the same index are equal and that revealed values in the proof and the formula match. It is heavily used in the checkEq method below.

The function checkEq^1 takes as input a value $v_1 : \alpha$ *RevHid* from a formula, a commitment v_2 : *commitment* from a proof statement, opening information v_3 : α *option* to v_2, and a list reference ℓ. checkEq^1 verifies that if v_1 is revealed, then it matches v_3 (in particular, v_3 is not None) and that if v_1 matches Hidden x for some x, then v_2 is equal to all values also hidden with the index x. The list reference ℓ is used to keep track of the index-commitment pairs. It is updated in the process accordingly.

```
   let checkEq^1 ($v_1 : \alpha$ $RevHid$) ($v_2 : commitment$) ($v_3 : \alpha$ $option$)
       ($\ell$ : ($bitstring * commitment$) $list$ $ref$) : $bool$ =
1    match $v_1$ with
2      | Revealed $x$ ⟹
3        match $v_3$ with
4          | Some($y, r_y$) ⟹
5            if $x$ = $y$ then
6              true
             else
               false
           | _ ⟹
               false
7      | Hidden $x$ ⟹
8        match $v_3$ with None ⟹
```

```
      if List.member$^{(1,2)}$ $x$ (!$\ell$) = true then
        let ($\_, v$) = List.get$^{(1,2)}$ $x$ (!$\ell$);
        if $v$ = $v_2$ then
          true
        else
          false
      else
        let _ = $\ell$ := Cons(($x, v_2$), !$\ell$);
        true
    | _ $\Longrightarrow$
      false
```

checkEq: $(\mathit{stm} : \mathit{statement}) \to (f : \mathit{formula}) \to \mathit{bool}$:

The checkEq function returns true only if the given statement matches the provided formula. It traverses the types *statement* and *formula*, enforcing that they structurally match. Additionally, checkEq applies checkEq^1 to establish the required equality between hidden values with the same index. In other words, checkEq enforces that the given formula matches the given statement.

For instance, let $\mathit{stm} = \mathsf{Says_p}(c_{vk}, \mathsf{Some}(vk, r_{vk}), c_{sig}, \mathsf{P}(c_m, \mathsf{None}, c_n, \mathsf{None}, c_\ell, \mathsf{Some}(\ell, r_\ell)))$ and $f = \mathsf{Revealed}\ vk\ \mathsf{says}\ \mathsf{P}(\mathsf{Hidden}\ 1, \mathsf{Hidden}\ 1, \mathsf{Revealed}\ \ell))$, i.e. the first and second argument to the predicate P are equal, the third argument is ℓ. The function checkEq enforces that the two hidden values are equal (i.e., the two commitments c_m and c_n contain the same value) and that the value ℓ contained in the statement is equal to the value contained in the formula.

The equality between lists and between operations (described by strings) is defined as expected; values without opening information (e.g., signing keys in the SSP predicate) are not checked since the real implementation establishes the equality via the user identifier (see Section 2.4).

```
let checkEq ($\mathit{stm} : \mathit{statement}$) ($f : \mathit{formula}$) : $\mathit{bool}$ =
  let $\ell$ = ref NIL;
  match $\mathit{stm}$ with
    | Says$_\mathsf{p}$($c_z, o_z, p$) $\Longrightarrow$
      match $p$ with
        | $P_1^P(c_1, o_1, \ldots, c_{n_1}, o_{n_1})$ $\Longrightarrow$
          match $f$ with
            | Says($z, P'$) $\Longrightarrow$
              match $P'$ with
                | $P_1^F(x_1, \ldots, x_{n_1})$ $\Longrightarrow$
                  if checkEq$^1$ $z$ $c_z$ $o_z$ $\ell$ then
                  if checkEq$^1$ $x_1$ $c_1$ $o_1$ $\ell$ then
```

```
                      ⋮
                    if checkEq$^1$ $x_{n_1-1}$ $c_{n_1-1}$ $o_{n_1-1}$ $\ell$ then
                      checkEq$^1$ $x_{n_1}$ $c_{n_1}$ $o_{n_1}$ $\ell$
                    else
                      false
                      ⋮
                    else
                      false
                  | _ $\Longrightarrow$
                    false
              | _ $\Longrightarrow$
                false
          | $P_2^P(c_1, o_1, \ldots, c_{n_2}, o_{n_2})$ $\Longrightarrow$
            ⋮
          | $P_m^P(c_1, o_1, \ldots, c_{n_m}, o_{n_m})$ $\Longrightarrow$
            ⋮
          | _ $\Longrightarrow$
            false
      | SSP$_\mathsf{p}(c_z, o_z, c_s, o_s, c_{psd}, o_{psd}, c_x)$ $\Longrightarrow$
        match $f$ with
          | SSP$(z, s, psd)$ $\Longrightarrow$
            if checkEq$^1$ $z$ $c_z$ $o_z$ $\ell$ then
              if checkEq$^1$ $s$ $c_s$ $o_s$ $\ell$ then
                checkEq$^1$ $psd$ $c_{psd}$ $o_{psd}$ $\ell$
              else
                false
            else
              false
          | _ $\Longrightarrow$
              false
      | REL$_\mathsf{p}(c_x, o_x, op, c_y, o_y)$ $\Longrightarrow$
        match $f$ with
          | REL$(x, op', y)$ $\Longrightarrow$
            if checkEq$^1$ $x$ $c_x$ $o_x$ $\ell$ then
              if checkEq$^1$ $y$ $c_y$ $o_y$ $\ell$ then
                $op$ = $op'$
              else
                false
            else
              false
      ⋮
```

```
    | $\mathsf{LM_p}(c_x, o_x, c_b, o_b, \ell)$ $\Longrightarrow$
      match $f$ with
        | $\mathsf{LM}(x, b, \ell')$ $\Longrightarrow$
          if $\mathsf{checkEq}^1$ $x$ $c_x$ $o_x$ $\ell$ then
            if $\mathsf{checkEq}^1$ $b$ $c_b$ $o_b$ $\ell$ then
              $\ell$ = $\ell'$
            else
              false
          else
            false
    ⋮
    | $\mathsf{Or_p}(stm_1, stm_2)$ $\Longrightarrow$
      match $f$ with
        | $\mathsf{Or}(f_1, f_2)$ $\Longrightarrow$
          if checkEq $stm_1$ $f_1$ then
            checkEq $stm_2$ $f_2$
          else
            false
        | _ $\Longrightarrow$
          false
    | _ $\Longrightarrow$
      false
```

verify_{stm}: *statement* → *bool*:

The verify_{stm} function takes as input a statement *stm* and verifies its validity.

For instance, if the given statement is a *vk* says *p*-statement, verify_{stm} opens the commitments and calls the signature verification on the committed signature with the predicate *p* as message and *vk* as verification key.

Internally, verify_{stm} matches *stm* and checks the validity of the respective case *f*.

```
$\mathsf{verify}_{stm}$ $(p : statement)$: $bool$ =
  match $p$ with
    | $\mathsf{And_p}(p_1, p_2)$ $\Longrightarrow$
      if $\mathsf{verify}_{stm}$ $p_1$ = true then
        $\mathsf{verify}_{stm}$ $p_2$
      else
        false
    | $\mathsf{Or_p}(p_1, p_2)$ $\Longrightarrow$
      if $\mathsf{verify}_{stm}$ $p_1$ = true then
        true
      else
        $\mathsf{verify}_{stm}$ $p_2$
    | $\mathsf{REL_p}(c_x, \_, op, c_y, \_)$ $\Longrightarrow$
```

```
      let $(x, r_x)$ = openCommit($c_x$);
      let $(y, r_y)$ = openCommit($c_y$);
      let $op''$ = getOperation$^r$ $op$;
      $op''$ $x$ $y$
    | EQN$_\mathsf{p}$($c_x, \_, op, c_y, \_, c_z, \_$) $\Longrightarrow$
      let $(x, r_x)$ = openCommit($c_x$);
      let $(y, r_y)$ = openCommit($c_y$);
      let $(z, r_z)$ = openCommit($c_z$);
      let $op''$ = getOperation$^e$ $op$;
      if $z = op''$ $x$ $y$ then
        true
      else
        false
    | Says$_\mathsf{p}$($c_{sig}, \_, c_z, \_, P(c_1, \_, \ldots, c_n, \_)$) $\Longrightarrow$
      let $(sig, r_{sig})$ = openCommit($c_{sig}$);
      let $(z, r_z)$ = openCommit($c_z$);
      let $(y_1, r_1)$ = openCommit($c_1$);
      $\vdots$
      let $(y_n, r_n)$ = openCommit($c_n$);
      let $(z'', q)$ = $z$ $sig$;
      if $z$ = $z''$ then
        true
      else
        false
    | EscrowInfo$_\mathsf{p}$($z, c_x, \_, c_R, \_, c_s, \_, c_{idr}, \_, c_r$) $\Longrightarrow$
      let $(s, r_s)$ = openCommit($c_s$);
      let $(idr, r_{idr})$ = openCommit($c_{idr}$);
      let $(x, r_x)$ = openCommit($c_x$);
      let $(r, r_r)$ = openCommit($c_r$);
      let $(R, r_R)$ = openCommit($c_R$);
      let $R'$ = computeR $x$ $r$;
      let $idr'$ = computeIDR $z$ $x$ $r$ $R$ $s$;
      if $R$ = $R'$ then
        if $idr$ = $idr'$ then
          true
        else
          false
    | SSP$_\mathsf{p}$($c_y, \_, c_s, \_, c_{psd}, \_, c_x$) $\Longrightarrow$
      let $(y, r_y)$ = openCommit($c_y$);
      let $(s, r_s)$ = openCommit($c_s$);
      let $(psd, r_{psd})$ = openCommit($c_{psd}$);
      let $(x, r_x)$ = openCommit$_{sk}$($c_x$);
      let $(\_, w)$ = $x$;
```

```
        let $psd'$ = computePsd $x$ $s$;
        if $psd'$ = $psd$ then
          if $w$ = $y$ then
            true
          else
            false
      | LM$_\mathsf{p}$($c_x$, _, $c_b$, _, $\ell$) $\Longrightarrow$
        let ($x$, $r_x$) = openCommit($c_x$);
        let ($b$, $r_b$) = openCommit($c_b$);
        List.member$^{(2,2)}$$\langle pseudo * bitstring \rangle$ $x$ $b$ $\ell$;
      | LNM$_\mathsf{p}$($c_x$, _, $\ell$) $\Longrightarrow$
        let ($x$, $r_x$) = openCommit($c_x$);
        let $b_\ell$ = List.member$^{(1,2)}$$\langle pseudo * bitstring \rangle$ $x$ $\ell$;
        if $b_\ell$ = true then
          false
        else
          true
      | _ $\Longrightarrow$ false
```

Listing A.23: Implementation of verify$_{stm}$.

verify : *proof* → *formula* → *bool*:

The verify API method takes as input a proof p : *proof*, a formula f : *formula*, and returns true if p is a valid proof for f. This method is generally accessible and, unlike the verify$_{\mathcal{F}^\vee}$ method, does not depend on a certain logical formula. However, it returns only a Boolean value without a logical refinement.

```
  verify ($p$ : $proof$) ($f$ : $formula$) : $bool$ =
1   match $p$ with ZK($zkv$, $stm$) $\Longrightarrow$
2       let $\ell$ = $ref$ [];
3       let $b_1$ = checkEq $stm$ $f$ $\ell$;
4       let $b_2$ = checkZK $p$;
5       if $b_1$ = true then
6         if $b_2$ = true then
7           verify$_{stm}$ $stm$
          else
            false
        else
          false
    | _ $\Longrightarrow$ false
```

Internally, verify first creates the list reference ℓ (line 2) that will be passed to verify$_{stm}$ and is used to keep track of the equality of hidden values with the same index. Next, we check that the statement contained inside p is a statement for the zero-knowledge proof (lines 1 to 5); intuitively, this check enforces that the commitments in both parts are equal (see checkEq above). Finally, $\mathsf{verify}_{\mathit{stm}}$ verifies that the statement *stm* in p is a valid zero-knowledge statement for formula f. Since we checked that stm and the zero-knowledge proof also match, this method verifies that p is a valid zero-knowledge proof for formula f.

$\mathsf{hide}_{\mathit{stm}}$: *statement* → *formula* → *statement*:

The $\mathsf{hide}_{\mathit{stm}}$ function takes as input a statement *stm* and a formula f, and it returns the statement that is derived from *stm* by hiding all values that are hidden in f.

For instance, let $\mathit{stm} = \mathsf{Says}_\mathsf{p}(c_{vk}, \mathsf{Some}(vk, r_{vk}), c_{sig}, \mathsf{P}(c_m, \mathsf{Some}(m, r_m)))$ and let $f = \mathsf{Hidden}\ 1\ \mathsf{says}\ \mathsf{Revealed}\ m$, then $\mathsf{hide}_{\mathit{stm}}\ \mathit{stm}\ f$ returns $\mathsf{Says}_\mathsf{p}(c_{vk}, \mathsf{None}, \mathsf{P}(c_m, \mathsf{Some}(m, r_m)))$.

Internally, $\mathsf{hide}_{\mathit{stm}}$ matches *stm* and f and if a value in f is hidden (via the Hidden constructor of the α *RevHid* type), then the corresponding opening information are replaced by None, otherwise the opening information are not modified.

```
$\mathsf{hide}_{stm}$ $(stm : statement)$ $\to$ $(f : formula)$ : $statement$ =
  match $stm$ with
    | $\mathsf{Says_p}(c_z, o_z, c_{sig}, o_{sig}, P)$ $\Longrightarrow$
        ⋮
    | $\mathsf{SSP_p}(c_z, o_z, c_s, o_s, c_{psd}, o_{psd}, c_x)$ $\Longrightarrow$
      match $f$ with $\mathsf{SSP}(z', s', psd')$ $\Longrightarrow$
          let $o'_z$ = match $z'$ with
            | Revealed _ $\Longrightarrow$ $o_z$
            | _ $\Longrightarrow$ None;
          let $o'_s$ = match $s'$ with
            | Revealed _ $\Longrightarrow$ $o_s$
            | _ $\Longrightarrow$ None;
          let $o'_{psd}$ = match $psd'$ with
            | Revealed _ $\Longrightarrow$ $o_{psd}$
            | _ $\Longrightarrow$ None;
          $\mathsf{SSP_p}(c_z, o'_z, c_s, o'_s, c_{psd}, o'_{psd}, c_x)$
        | _ $\Longrightarrow$
          fail$\langle proof \rangle$ ()
    | $\mathsf{LM_p}(c_x, o_x, c_b, o_b, \ell)$ $\Longrightarrow$
        ⋮
    | $\mathsf{LNM_p}(c_x, o_x, \ell)$ $\Longrightarrow$
        ⋮
    | $\mathsf{REL_p}(c_x, o_x, op, c_y, o_y)$ $\Longrightarrow$
        ⋮
    | $\mathsf{EscrowInfo_p}(z, c_x, o_x, c_R, o_R, c_s, o_s, c_{idr}, o_{idr}, c_r)$ $\Longrightarrow$
```

```
          ⋮
  | $\mathsf{And_p}(stm_1, stm_2) \Longrightarrow$
    match $f$ with $\mathsf{And}(f_1, f_2) \Longrightarrow$
        $\mathsf{And_p}(\mathsf{hide}_{stm}\ stm_1\ f_1, \mathsf{hide}_{stm}\ stm_2\ f_2)$
      | _ $\Longrightarrow$
        $\mathsf{fail}\langle statement\rangle\ ()$
  | $\mathsf{Or_p}(stm_1, stm_2) \Longrightarrow$
    match $f$ with $\mathsf{Or}(f_1, f_2) \Longrightarrow$
        $\mathsf{Or_p}(\mathsf{hide}_{stm}\ stm_1\ f_1, \mathsf{hide}_{stm}\ stm_2\ f_2)$
      | _ $\Longrightarrow$
        $\mathsf{fail}\langle statement\rangle\ ()$
  | _ $\Longrightarrow$
    $\mathsf{fail}\langle statement\rangle\ ()$
```

combineOr : *proof* → *formula* → *random* → *proof*:

The combineOr function is used to implement the $\mathsf{mk}_\vee$ API method. It takes as input a proof p : *proof*, a disjunctive formula f : *formula*, and randomness r : *random*. It returns a proof for the formula f that is valid only if p is a valid proof for either the left or the right branch of the disjunctive formula f.

```
combineOr $(p : proof)$ $(f : formula)$ $(r_{fake} : random)$ :
    $proof * zero\text{-}knowledge * statement$ =
 1  match $p$ with $\mathsf{ZK}(zkv, stm) \Longrightarrow$
 2      let $(t, r)$ = openZK $zkv$;
 3      match $f$ with $\mathsf{Or}(f_1, f_2) \Longrightarrow$
 4          if verify $p$ $f_1$ then
 5            let $stm_{fake}$ = fakestm $f_2$;
 6            let $stm'$ = $\mathsf{Or_p}(stm, stm_{fake})$;
 7            let $zkv'$ = commitZK$((t, stm_{fake}), (r, r_{fake}))$;
 8            let $p'$ = $\mathsf{ZK}(zkv', stm')$;
 9            $p'$
10          else
11            let $stm_{fake}$ = fakestm $f_1$;
12            let $stm'$ = $\mathsf{Or_p}(stm_{fake}, stm)$;
13            let $zkv'$ = commitZK$((stm_{fake}, t), (r_{fake}, r))$;
14            let $p'$ = $\mathsf{ZK}(zkv', stm')$;
15            $p'$
        | _ $\Longrightarrow$ $\mathsf{fail}\langle proof\rangle\ ()$
    | _ $\Longrightarrow$ $\mathsf{fail}\langle proof\rangle\ ()$
```

The function combineOr first matches the proof p to gain access to the zero-knowledge value *zkv* and the proven statement *stm*. The zero-knowledge value is opened (line 2) to obtain access to the randomness r used in the creation of p. Next, combineOr matches the formula f with a disjunction. The following if-statement decides at which position the valid branch (i.e., the one evidenced by p) is placed.

Inside the then- and the else-branch, first the statement $\mathit{stm}_{\mathit{fake}}$ for the invalid branch (i.e., the branch not evidenced by p) is created with the fakestm function. The statement stm' for the created disjunction consists of the $\mathsf{Or_p}$ constructor applied to the original statement stm of p and $\mathit{stm}_{\mathit{fake}}$. The zero-knowledge value zkv' for the disjunction consists of the disjunctive statement stm' and randomness. This randomness consists of the original randomness r of p and the input randomness r_{fake}. Finally, the disjunctive proof p' is constructed and returned. The order of the construction depends on whether p is a proof for f_1 or not.

Notice that if p is not a proof for either f_1 or f_2, then it is treated as a proof for f_2. Furthermore, in contrast to the cryptographic description in Section 2.4, we allow for the creation of disjunctive statements from proofs where not all opening information are available. Our symbolic model is general and also captures schemes that support the creation of disjunctive proofs without re-computing the whole proof.

commuteOr : *proof* → *statement* → *proof*
and

commuteAnd : *proof* → *statement* → *proof*:

The RCF model of zero-knowledge proofs does not allow for commuting the statement. For instance, a proof for a statement of the form $\mathsf{Or_p}(\mathit{stm}_1, \mathit{stm}_2)$ cannot be changed into $\mathsf{Or_p}(\mathit{stm}_2, \mathit{stm}_1)$; in a concrete implementation, however, such a commutative transformation might be a very simple operation. In fact, the cryptographic realization detailed in Section 2.4 allows for a trivial change of order. To reflect this concrete operation into the symbolic model, the functions commuteAnd and commuteOr are necessary. The two functions enable the respective transformations on the symbolic zero-knowledge proofs.

The functions commuteOr and commuteAnd both take as input a value p : *proof* and a statement stm : *statement*. The statement stm denotes where in p the order of a conjunction or disjunction will be swapped. The implementation consists only of match-statements without any insight and is therefore omitted.

Precision of the abstract zero-knowledge model. It is mandatory that the API types *proof* and *statement* are public. Certain kinds of computational proofs, however, require information whose abstract counterpart is neither public nor tainted. For instance, the pseudonym proofs computationally require the signing key of type *sigkey* and the type *sigkey* is neither public nor tainted. Intuitively, the reason is that a signing key is a tuple. The second component, the verification key is public (see Lemma A.11) and the first component, the sealing function, is tainted (using Kind Fun). The kinding rule for tuples requires that both components are either public or tainted. Consequently, type *sigkey* is neither public nor tainted. The RCF kinding mechanism propagates this property and deems all types that use such non-public, non-tainted types as also non-public and non-tainted. Thus, if the type *statement* contains type *sigkey*, then *statement* would also be neither public nor tainted and, as a result, type *proof* would not be public and the type system would prevent proofs from being be sent over a publicly readable network such as the Internet.

We address the problem by keeping the commitments to signing keys in the type *statement* but we do not keep the corresponding opening information. This relaxation is no threat to the soundness of our abstraction. In fact, quite the opposite is the case: an abstract attacker can create zero-knowledge proofs that the attacker could not create computationally (because the attacker lacks the knowledge of the signing key).

For instance, the implementation of createZK recomputes all commitments in a statement from the given opening information and returns a proof for the resulting value of type *statement*; the commitment to signing keys is used without requiring the key itself. Thus, an attacker can create a proof, for instance, an SSP proof, without supplying the opening information of some commitments.

At first glance, this seems to be a severe flaw in our model since we use SSPs to authenticate users. The issue, however, is immediately resolved because we only consider SSPs that are equipped with a valid says-statement. Since the says-statement necessarily requires the knowledge of the secret signing key (or the knowledge of a signature that, in turn, requires the signing key), the attacker cannot abuse the createZK function to authenticate users without their consent.[3]

In fact, since an abstract attacker can create abstract zero-knowledge proofs without knowing the opening information to the signing key, we technically lose the abstract of-knowledge property for these values. This is not a problem in our setting since the predicates we transfer do not rely on a principal knowing the witnesses but only on the validity of the proven formula.

Main API methods. The rest of the section is organized in two parts.

- first, we implement the main API methods except for the verification method. The implementation is often surprisingly short because the creation functions only perform checks regarding the well-formedness of the inputs;
- secondly, we develop important concepts for the implementation of the verify method and finally implement the verify method. From a type-checking and from a logical point of view, the verification method is the most important and most complex function because it allows principals to deduce logical formulas from data received from an untrusted public network.

We start with the mkId API method.

mkId : $(x : \mathit{string}) \rightarrow (\mathit{uid} * \mathit{uid}_{\mathit{pub}})$:

mkId takes a textual description $x : \mathit{string}$ as input and returns a freshly generated signing key handle/public key pair.

[3] Naturally, if a user already authenticated herself, the attacker can use that authentication information with the createZK method to create another authentication information. The newly derived information, however, would carry the same logical formula, i.e., the user has already consented to that authentication.

```
mkId $(x : string)$ : $uid * uid_{pub}$ =
  let $sk$ = mkSeal$\langle \mathcal{T}_y\{verkey/\alpha\}\rangle$ $x$;
  let $(\_, vk')$ = $sk$;
  let $hdl$ = storeSK $sk$;
  $(hdl$, fold$\langle \mathcal{T}_y\{verkey/\alpha\}$, $verkey\rangle$ $vk')$
```

The signing keys are the pairs of sealing and unsealing functions, and verification keys are the unsealing function. We apply the fold constructor to obtain the desired recursive type *verkey* for the verification key.

mkSays : $uid \rightarrow predicate^F \rightarrow proof$:

mkSays takes as input a handle to the signing key x' of the principal A executing the API method (recall that the API does not allow for direct access to secret key material) and a predicate y. For the sake of readability, we use a predicate rather than a formula, since the core insight lies in the transport of the logical formula and not the complexity thereof. mkSays outputs a proof that, when verified by a principal B, will allow B to logically entail A says F where f is the RCF encoding of predicate F. Since calling the mkSays method expresses the intention of the executing principal to state the provided formula, we internalized the necessary assumption into the code of the API method.

```
mkSays $(x' : uid)$ $(f : predicate^F)$: $proof$
  let $x$ = restoreSK $x'$;
  let $(w, z)$ = $x$;
  match $f$ with
    | $P_1^F$(Revealed $y_1, \dots,$ Revealed $y_{n_1}$) $\Longrightarrow$
      let $y'$ = $P_1^S(y_1, \dots, y_{n_1})$;
      let t = assume $z$ says $P_1(y_1, \dots, y_{n_1})$;
      let $sig$ = sign $x$ $(z, y')$;
      let $r_{sig}$ = rand();
      let $c_{sig}$ = commit$(sig, r_{sig})$;
      let $r_z$ = rand();
      let $c_z$ = commit$(z, r_z)$;
      let $r_1$ = rand();
      let $c_1$ = commit$(y_1, r_1)$;
      ⋮
      let $r_{n_1}$ = rand();
      let $c_{n_1}$ = commit$(y_{n_1}, r_{n_1})$;
      let $stm$ = Says$_{\mathsf{p}}(c_{sig}$, Some$(sig, r_{sig}), c_z$, Some$(z, r_z)$,
          $P_1^P(c_1$, Some$(y_1, r_1), \dots, c_{n_1}$, Some$(y_{n_1}, r_{n_1})))$;
      let $r_{zkv}$ = rand();
      let $zkv$ = commitZK $(stm, r_{zkv})$;
      ZK $(zkv, stm)$
```

```
20      | $P_2^F$(Revealed $y_1,\ldots,$Revealed $y_{n_2}$)  $\Longrightarrow$
          ⋮
21      | $P_m^F$(Revealed $y_1,\ldots,$Revealed $y_{n_m}$)  $\Longrightarrow$
22        let $y'$ = $P_m^S(y_1,\ldots,y_{n_m})$;
23        let t = assume $z$ says $P_m(y_1,\ldots,y_{n_m})$;
24        let $sig$ = $w$ $(z,y')$;
25        let $r_{sig}$ = rand();
26        let $c_{sig}$ = commit($sig,r_{sig}$);
27        let $r_z$ = rand();
28        let $c_z$ = commit($z,r_z$);
29        let $r_1$ = rand();
30        let $c_1$ = commit($y_1,r_1$);
          ⋮
31        let $r_{n_m}$ = rand();
32        let $c_{n_m}$ = commit($y_{n_m},r_{n_m}$);
33        let $stm$ = $\mathsf{Says}_\mathsf{p}$($c_{sig}$, Some($sig,r_{sig}$), $c_z$, Some($z,r_z$),
              $P_m^P$($c_1$, Some($y_1,r_1$), $\ldots$, $c_{n_m}$, Some($y_{n_m},r_{n_m}$)))
34        let $r_{zkv}$ = rand();
35        let $zkv$ = commitZK ($stm,r_{zkv}$);
36        ZK ($zkv,stm$)
        | _ $\Longrightarrow$ fail$\langle proof\rangle$ ()
```

We implement mkSays as a match statement over all possible predicates in the system. As a result, mkSays is general and does not require a formula annotation as is needed for the verification function (see below). mkSays first splits the signing key (line 2) to obtain the signing component used in creating the signature (line 3), matches predicate y with the type $predicate^F$ to access the arguments for the P_i^F (lines 4, 20, …, 21) respectively, and creates the signature. The matching of the supplied formula without any hidden components enforces that all values in the formula are revealed and, in particular, available for computation. This corresponds to the of-knowledge property of the zero-knowledge proofs.

The rest of the proof draws the randomness, computes the commitments, and builds the proof. The code for the different cases only differs in the matched pattern and on the number of calls to rand and commit (they depend on the arity of the matched pattern).

mkSSP : $uid \rightarrow bitstring \rightarrow proof$:

mkSSP takes as input the handle of the signing key x' of the principal running the code and the service description s. It extracts the signing key x using x' and uses computePsd, x, and s to compute the pseudonym psd.

The remainder of the function draws the randomness for the commitments, computes the commitments, and builds the proof. In accordance with the API description, the opening information for commitment c_x on the signing key are not included in the proof. This prevents accidental or intentional leaking of secret key material.

```
mkSSP $(x' : uid)$ $(s : bitstring)$ : $proof$ =
  let $x$ = restoreSK $x'$;
  let $(\_, y)$ = $x$;
  let $psd$ = computePsd $x$ $s$;
  let $r_y$ = rand();
  let $c_y$ = commit$(y, r_y)$;
  let $r_s$ = rand();
  let $c_s$ = commit$(s, r_s)$;
  let $r_{psd}$ = rand();
  let $c_{psd}$ = commit$(psd, r_{psd})$;
  let $r_x$ = rand();
  let $c_x$ = commit$_{sk}(x, r_x)$;
  let $stm$ = $\mathsf{SSP_p}(c_y, \mathsf{Some}(y, r_y), c_s, \mathsf{Some}(s, r_s), c_{psd}, \mathsf{Some}(psd, r_{psd}), c_x)$
  let $r_{zkv}$ = rand();
  let $zkv$ = commitZK$(stm, r_{zkv})$;
  ZK$(zkv, stm)$
```

mkREL : *formula* → *proof*:

mkREL takes as input the formula y that describes an (in)equality relation. The function draws randomness, builds the corresponding commitments, and creates the zero-knowledge proof. Since the operation can be deduced from the performed cryptographic operation, the proven operation occurs inside the abstract proof in plain.

```
mkREL $(y : formula)$: $proof$ =
  match $y$ with REL(Revealed $x$, $op$, Revealed $y$)
      let $r_x$ = rand();
      let $c_x$ = commit$(x, r_x)$;
      let $r_y$ = rand();
      let $c_y$ = commit$(y, r_y)$;
      let $stm$ = $\mathsf{REL_p}(c_x, \mathsf{Some}(x, r_x), op, c_y, \mathsf{Some}(y, r_y))$
      let $r_{zkv}$ = rand();
      let $zkv$ = commitZK$(stm, r_{zkv})$;
      ZK$(zkv, stm)$
    | _ $\Longrightarrow$
      fail$\langle proof \rangle$ ()
```

mkLM : *pseudo* → *string* → (*pseudo* ∗ *string*) *list* → *proof*:

mkLM takes as argument a pseudonym x, an attribute b, and a list ℓ. The function draws randomness, builds the corresponding commitments, and creates the zero-knowledge proof. We only create commitments and opening information for x and b since, by convention, lists are always revealed.

```
mkLM $(x : \mathit{pseudo})$ $(b : \mathit{string})$ $(\ell : (\mathit{pseudo} * \mathit{string})\ \mathit{list})$ : $\mathit{proof}$ =
  let $r_x$ = rand();
  let $c_x$ = commit$(x, r_x)$;
  let $r_b$ = rand();
  let $c_b$ = commit$(b, r_b)$;
  let $\mathit{stm}$ = $\mathsf{LM_p}(c_x, \mathsf{Some}(x, r_x), c_b, \mathsf{Some}(b, r_b), \ell)$
  let $r_{\mathit{zkv}}$ = rand();
  let $\mathit{zkv}$ = commitZK$(\mathit{stm}, r_{\mathit{zkv}})$;
  ZK$(\mathit{zkv}, \mathit{stm})$
```

mkLNM : $\mathit{pseudo} \rightarrow (\mathit{pseudo} * \mathit{string})\ \mathit{list} \rightarrow \mathit{proof}$:

mkLNM takes as argument a pseudonym x and a list ℓ. The function draws randomness, builds the corresponding commitments, and creates the zero-knowledge proof. Analogously to the list membership proof, we only create commitments and opening information for x since we use the convention that lists cannot be hidden.

```
mkLNM $(x : \mathit{pseudo})$ $(\ell : (\mathit{pseudo} * \mathit{string})\ \mathit{list})$ : $\mathit{proof}$ =
  let $r_x$ = rand();
  let $c_x$ = commit$(x, r_x)$;
  let $\mathit{stm}$ = $\mathsf{LNM_p}(c_x, \mathsf{Some}(x, r_x), \ell)$
  let $r_{\mathit{zkv}}$ = rand();
  let $\mathit{zkv}$ = commitZK$(\mathit{stm}, r_{\mathit{zkv}})$;
  ZK$(\mathit{zkv}, \mathit{stm})$
```

mkIDRev : $\mathit{proof} \rightarrow \mathit{bitstring} \rightarrow \mathit{proof}$:

mkIDRev takes as input a special proof p constructed by a trusted third party and a service description s. The TTP can, in case of a dispute or another cogent reason reveal the identity of the user. The proof p shows that the principal A running the code is registered with the trusted third party and contains the values used by the TTP to identify A (see Section 2.4). Since mkIDRev creates a proof from scratch and all values in p are used, p must contain all the opening information.

The escrow value R is computed using the computeR function, the escrow identifier idr is computed by applying the function computeIDR to z, x, r, R, and the service description s.

After the computation, the method draws the randomness to compute the commitments and creates the proof. As the value r is considered secret, we only include the commitment c_r on r in the proof.

```
mkIDRev $(p : proof)$ $(s : string)$: $proof$ =
  match $p$ with $\mathsf{Says_p}(c_{sig}, o_{sig}, c_z, o_z, f)$ $\Longrightarrow$
      match $f$ with $\mathsf{EscrowId}(c_x, o_x, c_r, o_r)$ $\Longrightarrow$
          let $(z, r_z)$ = getSome $o_z$;
          let $(x, r_x)$ = getSome $o_x$;
          let $(r, r_r)$ = getSome $o_r$;
          let $R$ = computeR $x$ $r$;
          let $idr$ = computeIDR $z$ $x$ $r$ $R$ $s$;
          let $r_R$ = rand();
          let $c_R$ = commit$(R, r_R)$;
          let $r_{idr}$ = rand();
          let $c_{idr}$ = commit$(idr, r_{idr})$;
          let $r_s$ = rand();
          let $c_s$ = commit$(s, r_s)$;
          let $stm$ = $\mathsf{EscrowInfo_p}(z, c_x, \mathsf{Some}(x, r_x), c_R, \mathsf{Some}(R, r_R),$
              $c_s, \mathsf{Some}(s, r_s), c_{idr}, \mathsf{Some}(idr, r_{idr}), c_r)$
          let $r_{zkv}$ = rand();
          let $zkv$ = commitZK$(stm, r_{zkv})$;
          ZK$(zkv, stm)$
        | _ $\Longrightarrow$ fail$\langle proof \rangle$ ()
    | _ $\Longrightarrow$ fail$\langle proof \rangle$ ()
```

$\mathsf{mk}_\wedge$: $(proof * proof) \rightarrow proof$:

$\mathsf{mk}_\wedge$ takes as input two proofs p_1 and p_2, and returns the proof for the logical conjunction $p_1 \wedge p_2$.

```
mk$_\wedge$ $(p : proof * proof)$ : $proof$ =
  let $(p_1, p_2)$ = $p$;
  match $p_1$ with ZK$(zkv_1, stm_1)$ $\Longrightarrow$
      match $p_2$ with ZK$(zkv_2, stm_2)$ $\Longrightarrow$
          let $(t_1, r_1)$ = openZK $zkv_1$;
          let $(t_2, r_2)$ = openZK $zkv_2$;
          let $zkv$ = commitZK$(\mathsf{And_p}(t_1, t_2), (r_1, r_2))$;
          ZK$(zkv, \mathsf{And_p}(stm_1, stm_2))$
        | _ $\Longrightarrow$ fail$\langle proof \rangle$ ()
    | _ $\Longrightarrow$ fail$\langle proof \rangle$ ()
```

$\mathsf{mk}_\wedge$ function matches the two proofs p_1 and p_2 to obtain the two corresponding zero-knowledge values zkv_1 and zkv_2 as well as the two corresponding statements stm_1 and stm_2. For the unsealed contents (t_1, r_1) and (t_2, r_2) of zkv_1 and zkv_2, respectively, it holds that t_1 equals stm_1 and t_2 equals stm_2 up to opening information that has been removed via the hide API method from stm_1 and stm_2. Consequently, the correct content of the zkv value consists of the constructed value $\mathsf{And_p}(t_1, t_2)$.

The randomness for the zero-knowledge proof is a subtle matter. Since the cryptographic realization does not require any special randomness (the two proofs are concatenated), we reflect this implementation by using the paired randomness of the two sub-proofs as randomness for the conjunction. Thus, exactly as in the cryptographic implementation, the proof for a logical conjunction only depends on its two sub-proofs.

split$_\wedge$: *proof* → (*proof* ∗ *proof*):

split$_\wedge$ takes as input a proof. If this proof is a conjunction, it returns the two conjuncted proofs by reversing the operations conducted by mk$_\wedge$.

```
   split_∧ (p : proof) : proof ∗ proof =
1    match p with ZK(zkv, stm) ⟹
2        match stm with And_p(stm_1, stm_2) ⟹
3            let (t, r) = openZK zkv;
4            let (r_1, r_2) = r;
5            match t with And_p(t_1, t_2) ⟹
6                let zkv_1 = commitZK(t_1, r_1);
7                let zkv_2 = commitZK(t_2, r_2);
8                (ZK(zkv_1, stm_1), ZK(zkv_2, stm_2))
              | _ ⟹ fail⟨proof ∗ proof⟩ ()
          | _ ⟹ fail⟨proof ∗ proof⟩ ()
      | _ ⟹ fail⟨proof ∗ proof⟩ ()
```

The implementation performs the following steps: it matches the given proof p against a conjunction proof (line 1), splits the statement to retrieve the two sub-statements of the individual sub-proofs (line 2), opens the *zkv* value (line 3), splits the randomness (line 4) and reassembles the zero-knowledge proofs for the two sub-proofs, accordingly (lines 5 through 8).

mk$_\vee$: *proof* → *formula* → *proof*:

The method mk$_\vee$ takes as input a proof p and a formula f. If f is a disjunctive formula and p is a valid proof for the left or the right branch of the disjunction, then mk$_\vee$ returns a disjunctive proof for f. Internally, mk$_\vee$ exploits the combineOr function to create the desired zero-knowledge proof.

```
mk_∨ (p : proof) (f : formula) : proof =
  let r = rand();
  combineOr p f r
```

hide : *proof* → *formula* → *proof*:

The hide method takes as input a proof p and formula f and outputs the proof where all the values specified by f are hidden. hide takes as input a proof p and a formula

f, and it returns the proof obtained by hiding all variables specified by f. Internally, hide relies on the hide_{stm} function.

```
hide $(p : proof)$ $(f : formula)$ : $proof$ =
  match $p$ with ZK$(zkv, stm)$ $\Longrightarrow$
      let $stm'$ = $\mathsf{hide}_{\mathsf{stm}}$ $stm$ $f$;
      ZK$(zkv, stm')$
    | _ $\Longrightarrow$
      fail$\langle proof\rangle$ ()
```

rerand : $proof \rightarrow statement \rightarrow proof$:

The function rerand implements the re-randomization of Groth-Sahai zero-knowledge proofs as first observed by Belenkiy et al. [42]. It takes as input a proof p, statement stm, and it returns the re-randomization of p. The re-randomization is guided by stm (see rerand_{stm}).

```
let rerand $(p : proof)$ $(stm : statement)$ : $proof$ =
  match $p$ with
    | ZK$(zkv, stm_2)$ $\Longrightarrow$
      let $r'$ = rand ();
      let $(stm_1, r)$ = openZK $zkv$;
      let $stm'_1$ = $\mathsf{rerand}_{stm}$ $stm_1$ $stm$;
      let $stm'_2$ = $\mathsf{rerand}_{stm}$ $stm_2$ $stm$;
      let $zkv'$ = commitZK $(stm'_1, r')$;
      ZK$(zkv', stm'_2)$
    | _ $\Longrightarrow$
      fail$\langle proof\rangle$ ()
```

We symbolically implement re-randomization by choosing a fresh randomness for the zero-knowledge proof. We do not consider arithmetic properties of randomness, i.e., we do not model protocols that rely on the fact that randomness can be canceled.

The verification function $\mathsf{verify}_{\underline{\mathcal{F}^\vee}}$**.** We begin to detail the code for the proof verification function. Before we start giving the code, we define all the necessary ingredients of the verification function.

For technical type-checking reasons, we cannot provide one implementation that covers all cases. Intuitively, one implementation would need to call itself recursively in case of a conjunction proof. All variables and logical formulas occurring within these recursive calls are lost, after the verification function returns. Therefore, we could at most make existentially quantified statements about the sub-proofs. In particular, the equalities between the values could never be established. For instance, consider the following formula from Example 2.5 of Section 2.3:

$$\begin{aligned}
&\exists w_{Pat}, w_{date}, w_{results}, w_{psd'}.\\
&\quad x_{Doc} \text{ says } \mathsf{Visit}(w_{Pat}, w_{date}, w_{results})\\
&\quad \wedge\ w_{Pat} \text{ says } \mathsf{Rating}(x_{opinion})\\
&\quad \wedge\ \mathsf{SSP}(w_{Pat}, x_{Internist}, x_{psd})\\
&\quad \wedge\ \mathsf{SSP}(w_{Pat}, x_{Dentist}, w_{psd'})\\
&\quad \wedge\ (w_{psd'}, \_) \notin x_{\ell}
\end{aligned} \tag{A.1}$$

The recursive verification function would verify each part of the proof individually. Ultimately, we would logically obtain the following formula:

$$\begin{aligned}
&\exists w_{Pat}, w_{date}, w_{results}.\ x_{Doc} \text{ says } \mathsf{Visit}(w_{Pat}, w_{date}, w_{results})\\
&\wedge\ \exists w_{Pat}.\ w_{Pat} \text{ says } \mathsf{Rating}(x_{opinion})\\
&\wedge\ \exists w_{Pat}.\ \mathsf{SSP}(w_{Pat}, x_{Internist}, x_{psd})\\
&\wedge\ \exists w_{Pat}, w_{psd'}.\ \mathsf{SSP}(w_{Pat}, x_{Dentist}, w_{psd'})\\
&\wedge\ \exists w_{psd'}.\ (w_{psd'}, \_) \notin x_{\ell}
\end{aligned} \tag{A.2}$$

At this point, however, we cannot argue about the equality of existentially quantified variables, e.g., why the variable w_{Pat} from the first line should hide the same value as the variable w_{Pat} from the second line. Intuitively, all the intermediate representations of the hidden variables are lost inside the function calls and we cannot retrieve the equality between these anymore. Thus, instead of giving one implementation for the verification function, we provide code macros that are assembled into the final implementation.

Technically, the code macros are contexts. Intuitively, a context is an expression with a unique hole and into this hole, another context or another expression can be inserted. We write $C[\bullet]$ to denote the context C and the hole is denoted by $\bullet$. Let C' be another context or an expression. $C[\ C'\ ]$ is the result of replacing the unique $\bullet$ in context C with the context C'. If C' is a context, the result is a context again, and if C' is an expression, the result will be an expression.

Clearly, the code macros depend on the proven formula. For instance, verifying a list membership proof requires different procedures than verifying a says-statement. The macros, however, do not only depend on the statement but also on the code assembled for the parts of the statement proven before. This dependence of the code macros on the previously assembled code is crucial and subtler than hinted in Equation A.1 and Equation A.2. We use logical maps to record and enforce equalities of variables within the code macros. The maps rely on the following definition that identifies variables with their canonical positional index.

Intuitively, the following definition extracts the offset $\vec{\Delta}$ of a formula, the positions of principal identifiers $\mathcal{I}_{vk}$, i.e., verification keys, and the variables Vars occurring in a formula ordered from "left to right". Intuitively, $\mathcal{I}_{vk}$ determines which elements of Vars are user identifiers. In the implementation, we ensure that we associate user identifiers with verification keys that have the strong type *verkey*. We use $\vec{\Delta}$ to properly adjust the indices of $\mathcal{I}_{vk}$ in the right branch of conjunctions and disjunctions by considering the respective left branch and the number of variables occurring therein.

Definition A.1 (Formula offset, verification key indices, and variables of conjunctive formulas)**.** *Let $eI(f : formula)$ be the function that extracts information from f as follows:*

$$eI\left(\mathsf{Says}\begin{pmatrix} x_0 : uid_{pub}\ RevHid, \\ P_k^F(x_1 : T_1^k, \\ \dots, \\ \{x_n : T_n^k \mid \mathcal{F}^e\}) \end{pmatrix}\right) ::= \begin{pmatrix} n+1, \\ \{0\} \cup \{i \mid T_i^k = uid_{pub}\ RevHid\}, \\ (x_0, \dots, x_n)) \end{pmatrix}$$

$$eI\left(\mathsf{SSP}\begin{pmatrix} vk : uid_{pub}\ RevHid, \\ s : bitstring\ RevHid, \\ psd : bitstring\ RevHid \end{pmatrix}\right) ::= (3, \emptyset, (vk, s, psd))$$

$$eI\left(\mathsf{REL}\begin{pmatrix} x : bitstring\ RevHid, \\ op : string, \\ y : bitstring\ RevHid \end{pmatrix}\right) ::= (3, \emptyset, (x, op, y))$$

$$eI\left(\mathsf{EQN}\begin{pmatrix} x : bitstring\ RevHid, \\ op : string, \\ y : bitstring\ RevHid, \\ z : bitstring\ RevHid \end{pmatrix}\right) ::= (4, \emptyset, (x, op, y, z))$$

$$eI\left(\mathsf{LM}\begin{pmatrix} x : pseudo\ RevHid, \\ b : bitstring\ RevHid, \\ \ell : list \end{pmatrix}\right) ::= (3, \emptyset, (x, b, \ell))$$

$$eI\left(\mathsf{LNM}\begin{pmatrix} x : pseudo\ RevHid, \\ \ell : list \end{pmatrix}\right) ::= (2, \emptyset, (x, \ell))$$

$$eI\left(\mathsf{EscrowInfo}\begin{pmatrix} vk_{EA} : uid_{pub}, \\ vk : uid_{pub}\ RevHid, \\ R : bitstring\ RevHid, \\ s : bitstring\ RevHid, \\ idr : bitstring\ RevHid \end{pmatrix}\right) ::= (5, \emptyset, (vk_{EA}, vk, R, s, idr))$$

$$eI(\mathsf{And}(f_1, f_2)) ::= (n + n', S \cup (n + S'), V@V'),$$

where $(n, S, V) = eI(f_1)$, $(n', S', V') = eI(f_2)$, $i + M := \{i + m \mid m \in M\}$, and $(x_0, \dots, x_n)@(y_0, \dots, y_m) := (x_0, \dots, x_n, y_0, \dots, y_m)$.

Notice that we explicitly differentiate between uid_{pub} and bitstring in the definition of $eI(f)$, i.e., we consider the ML definition of uid_{pub} and bitstring since both coincide with type unit in RCF*.*

Let $(n, S, V) = eI(f)$. We call $\vec{\Delta}(f) := n$ the formula offset of f*. We call $\mathcal{I}_{vk}(f) := S$* the verification key indices of f*, and we call $\mathsf{Vars}(f) := V$* the variables occurring in f*.*

The verification key indices report the positions in a formula that are filled with public identifiers. This notion is important because we use it to identify variables that have the strong type *verkey*. More precisely, we will later require that for each user identifier

$x : uid_{pub}$ at these positions, there is a variable y such that $x = y$ and $y : verkey$ w.r.t. the current typing environment. The formula offset indicates how many elements a formula contains and is important to keep the logical maps aligned. The variables of a formula allow us to name variables in a formula.

For instance, let

$$\mathcal{F}^\wedge := \exists y.\ \mathit{Prof}\ \mathsf{says}\ \mathsf{Reg}(y) \wedge y\ \mathsf{says}\ \mathsf{Eval}(sec, good) \tag{A.3}$$

with the encoding

$$\underline{\mathcal{F}^\wedge} := \mathsf{And}\left(\begin{array}{c} \overbrace{\mathsf{Says}(\underbrace{\mathsf{Revealed}\ x_{Prof}}_{0}, \mathsf{Reg}^F(\underbrace{\mathsf{Hidden}\ y}_{1}))}^{f_1}, \\ \underbrace{\mathsf{Says}(\overbrace{\mathsf{Hidden}\ y}^{2}, \mathsf{Eval}^F(\overbrace{\mathsf{Revealed}\ x_{sec}}^{3}, \overbrace{\mathsf{Revealed}\ x_{good}}^{4}))}_{f_2} \end{array} \right)$$

Then

- $\overrightarrow{\Delta}(f_1) = 2$, $\overrightarrow{\Delta}(f_2) = 3$, and $\overrightarrow{\Delta}(\underline{\mathcal{F}^\wedge}) = 5$
- $\mathcal{I}_{vk}(\underline{\mathcal{F}^\wedge}) = \{0, 1, 2\}$
- $\mathsf{Vars}(\underline{\mathcal{F}^\wedge}) = (\mathsf{Revealed}\ x_{Prof}, \mathsf{Hidden}\ y, \mathsf{Hidden}\ y, \mathsf{Revealed}\ x_{sec}, \mathsf{Revealed}\ x_{good})$

We deploy three maps: $\mathcal{E}_F$, ψ, and ϕ. Intuitively,

- $\mathcal{E}_F$ takes as input a positional index i of a variable and returns the smallest positional index $j \leq i$ such that the value at position j is equal to the value at position i (in the formula). For instance, the example in Equation A.3 yields the following map

$$\begin{array}{c} \mathcal{E}_{\mathcal{F}^\wedge}(0) = 0 \quad \mathcal{E}_{\mathcal{F}^\wedge}(1) = 1 \quad \mathcal{E}_{\mathcal{F}^\wedge}(2) = 1 \\ \mathcal{E}_{\mathcal{F}^\wedge}(3) = 3 \quad \mathcal{E}_{\mathcal{F}^\wedge}(4) = 4 \end{array}$$

- ψ partitions the variables occurring in the verification function $\mathsf{verify}_{\underline{\mathcal{F}^\vee}}$ w.r.t. equality, i.e., variables in one partition are pairwise equal. This is of paramount importance: the sealing mechanism will provide access to the witnesses (i.e., the hidden values) in the verification function. We use ψ to prove equality of these witnesses and these equalities allow us to logically represent equal witnesses by a single existentially quantified variable.
- ϕ tracks variables that can be typed with the strong type *verkey* under the current typing environment. The reason for the tracking is of a very technical nature and inherently necessary for the type-checking process. For instance, the signature verification function requires as argument a value of type *verkey*. If a value, however, is hidden in a formula, then the formula does not provide us with a value of that type

and the proof only contains values of type *bitstring*. In these situations, ϕ will point us to a variable $y : \mathit{verkey}$ that can be used as argument to the verification function (the code performs the necessary equality checks to justify using y as verification key). Notice that y may only be known within the verification by unsealing the witnesses, i.e., the value in the proven statement corresponding to y may be hidden.

We formally define the maps using Definition A.1.

Definition A.2 ($\mathcal{E}_{\mathcal{F}^\wedge}$, ψ, and ϕ)**.** *Let $f := \underline{\mathcal{F}^\wedge}$ be a formula without disjunctions and $(x_0, \ldots, x_n) = \mathsf{Vars}(\underline{\mathcal{F}^\wedge})$. We define the* index map function *$\mathcal{E}_{\mathcal{F}^\wedge}$ as follows:*

$$\mathcal{E}_{\mathcal{F}^\wedge} : \mathbb{N} \to \mathbb{N}, \quad \mathcal{E}_{\mathcal{F}^\wedge}(i) = \min_{0 \le j \le i} x_j \overset{\mathcal{F}}{=} x_i$$

where $x_i \overset{\mathcal{F}}{=} x_j$ denotes that the values or variables corresponding to x_i and x_j in the logical formula $\mathcal{F}^\wedge$ are equal, i.e., the variables x_i and x_j correspond to the same existentially quantified variable in $\mathcal{F}^\wedge$ or that the variables x_i and x_j correspond to the same value in $\mathcal{F}^\wedge$.

$$\psi : \mathbb{N} \to \wp(\mathit{Vars}) \qquad\qquad \phi : \mathbb{N} \to \mathit{Vars}.$$

where Vars denotes the set of all RCF *variables and $\wp(S)$ denotes the power set of the set S. Additionally, we define the usual update:*

$$\phi[x \mapsto y](z) = \begin{cases} y & \text{if } x = z \\ \phi(z) & \text{otherwise} \end{cases} \qquad\qquad \psi[x \mapsto y](z) = \begin{cases} y & \text{if } x = z \\ \psi(z) & \text{otherwise} \end{cases}$$

Initially (i.e., if no update is applied) $\forall x.\ \phi(x) = \bot$ and $\forall x.\ \psi(x) = \emptyset$.

The updates are contained in the proper locations in the code macros and are marked in the special line numbers.

The following proposition states that extending a formula $\mathcal{F}^\wedge$ with a logical conjunction does not change the index map restricted to the old formula $\mathcal{F}^\wedge$.

Proposition A.1. *Let $\mathcal{F}_1^\wedge$ and $\mathcal{F}_2^\wedge$ be a conjunctive formulas. Then, $\forall i < \overrightarrow{\Delta}(\mathcal{F}_1^\wedge)$. $\mathcal{E}_{\mathcal{F}_1^\wedge}(i) = \mathcal{E}_{\mathcal{F}_1^\wedge \wedge \mathcal{F}_2^\wedge}(i)$.*

Notation (Extending formulas with logical conjunctions)**.** In the following, we will discuss formulas that are extended by applying a logical conjunctions to it. To simplify our soundness proof, we stipulate that formulas are in a normal form. More precisely, we extended formulas only with elementary formulas, i.e., a formula $\mathcal{F}^\wedge$ is extended to $\mathcal{F}^{\wedge\prime} := \mathcal{F}^\wedge \wedge \mathcal{F}^e$ for some elementary formula $\mathcal{F}^e$. Since logical conjunction is commutative and associative, this restricts only the syntax but does not change the logical meaning.

Notation (Index map $\mathcal{E}$ subscript)**.** As stated by Proposition A.1, the index map $\mathcal{E}$ induced by a formula $\mathcal{F}^\wedge$ and by a formula $\mathcal{F}^\wedge \wedge \mathcal{F}^e$ is the same for all indices $i < \overrightarrow{\Delta}(\underline{\mathcal{F}^\wedge})$. Therefore, if the formula is clear from the context, we will drop it from the subscript for the sake of readability in the rest of the thesis.

Above, we stated that we use the logical map ψ to keep track of the variables that are equal to each other. We now define the code that will enforce these equalities so that we can use them while type-checking later on.

Definition A.3 (ψ-induced code)**.** *We define* $M_\psi := \{S \mid S = \psi(i) \text{ for some } i\}$,

$$\mathsf{context}^{=}(\{x_1, \dots, x_n\})[\bullet] := \left\{\begin{array}{l} \texttt{if } x_1 = x_2 \texttt{ then} \\ \quad \vdots \\ \texttt{if } x_1 = x_n \texttt{ then} \\ \quad \bullet \\ \texttt{else} \\ \quad \mathsf{false} \\ \quad \vdots \\ \texttt{else} \\ \quad \mathsf{false} \end{array}\right.$$

and $\mathsf{context}^{=}(\{S_1, \dots, S_n\})[\bullet]$ *as*

$$\mathsf{context}^{=}(\{S_1, \dots, S_n\})[\bullet] := \mathsf{context}^{=}(S_1)[\mathsf{context}^{=}(S_2)[\dots[\ \mathsf{context}^{=}(S_n)[\bullet]\ ]\dots]].$$

We let $\mathsf{context}^{=}(\psi) := \mathsf{context}^{=}(M_\psi)$.

The last piece missing before we can finally define the verification function is the formula translation $[\![\underline{\mathcal{F}}, \mathit{zkv}, \mathit{stm}, f, \omega]\!]$. This translation defines how we assemble the individual code macros into the final, formula-specific verification function. This translation canonically follows the structure of $\underline{\mathcal{F}}$.

Definition A.4 (Formula translation)**.** *We define the formula translation* $[\![\underline{\mathcal{F}}, \mathit{zkv}, \mathit{stm}, f, \omega]\!]$ *for a general formula* $\mathcal{F}$*, a zero-knowledge value* zkv : *zero-knowledge, a statement* stm : *statement, a formula* f : *formula, and positional index* ω*.*

$$\begin{array}{rcl}
\left[\!\!\left[\begin{array}{l}\mathsf{Says}(x_{vk}, P_k^F(x_1, \dots, x_n)), \\ \mathit{zkv}, \mathit{stm}, f, \omega\end{array}\right]\!\!\right] & ::= & \mathit{Says}\text{-Macro}\begin{pmatrix}\mathsf{Says}(x_{vk}, P_k^F(x_1, \dots, x_n)), \\ \mathit{stm}, f, \omega\end{pmatrix} \\
[\![\mathsf{SSP}(x_{vk}, x_s, x_{psd}), \mathit{zkv}, \mathit{stm}, f, \omega]\!] & ::= & \mathit{SSP}\text{-Macro}(\mathsf{SSP}(x_{vk}, x_s, x_{psd}), \mathit{stm}, f, \omega) \\
[\![\mathsf{REL}(x, \mathit{op}, y), \mathit{zkv}, \mathit{stm}, f, \omega]\!] & ::= & \mathit{REL}\text{-Macro}(\mathsf{REL}(x, \mathit{op}, y), \mathit{stm}, f, \omega) \\
[\![\mathsf{EQN}(x, \mathit{op}, y, z), \mathit{zkv}, \mathit{stm}, f, \omega]\!] & ::= & \mathit{EQN}\text{-Macro}(\mathsf{EQN}(x, \mathit{op}, y, z), \mathit{stm}, f, \omega) \\
[\![\mathsf{LM}(x, b, \ell), \mathit{zkv}, \mathit{stm}, f, \omega]\!] & ::= & \mathit{LM}\text{-Macro}(\mathsf{LM}(x, b, \ell), \mathit{stm}, f, \omega) \\
[\![\mathsf{LNM}(x, \ell), \mathit{zkv}, \mathit{stm}, f, \omega]\!] & ::= & \mathit{LNM}\text{-Macro}(\mathsf{LNM}(x, \ell), \mathit{stm}, f, \omega) \\
\left[\!\!\left[\begin{array}{l}\mathsf{EscrowInfo}\begin{pmatrix}x_{EA}, x_{vk}, \\ x_R, x_s, x_{idr}\end{pmatrix}, \\ \mathit{zkv}, \mathit{stm}, f, \omega\end{array}\right]\!\!\right] & ::= & \mathit{Escrow}\text{-Macro}\left(\begin{array}{l}\mathsf{EscrowInfo}\begin{pmatrix}x_{EA}, x_{vk}, \\ x_R, x_s, x_{idr}\end{pmatrix}, \\ \mathit{stm}, f, \omega\end{array}\right) \\
\left[\!\!\left[\mathsf{And}(\underline{\mathcal{F}_1^\wedge}, \underline{\mathcal{F}_2^\wedge}), \mathit{zkv}, \mathit{stm}, f, \omega\right]\!\!\right] & ::= & \mathit{And}\text{-Macro}\begin{pmatrix}\mathsf{And}(\underline{\mathcal{F}_1^\wedge}, \underline{\mathcal{F}_2^\wedge}), \mathit{stm}, f, \\ \omega, \omega + \Delta^{\rightarrow}(\underline{\mathcal{F}_1^\wedge})\end{pmatrix} \\
\left[\!\!\left[\mathsf{Or}(\underline{\mathcal{F}_1^\vee}, \underline{\mathcal{F}_2^\vee}), \mathit{zkv}, \mathit{stm}, f, \omega\right]\!\!\right] & ::= & \mathit{Or}\text{-Macro}(\mathsf{Or}(\underline{\mathcal{F}_1^\vee}, \underline{\mathcal{F}_2^\vee}), \mathit{zkv}, \mathit{stm}, f)
\end{array}$$

Verification code macros. We discuss the code macros that are the building blocks of the verification function. Since each verification function is specialized for a given formula, the code macros will be tailored for that formula. All code macros are contexts. The corresponding hole • is always placed at the point of the verification code that is reached if all checks turned out positive, i.e., if the verification up to this point succeeded.

The verify method is parameterized with the formula $\mathcal{F}^\vee$ to be verified. The code macros and the refinement of the return value are tailored for that formula. The method takes as input the proof p : *proof* and the formula y : *formula*. The refinement on the returned Boolean intuitively states that if the return value is true, then $\mathcal{F}^\vee$ holds.

```
verify$_{\underline{\mathcal{F}^\vee}}$ $(p : proof)$ $(f : formula)$ : $\{x : bool \mid \forall \tilde{z}.\ f = \underline{\mathcal{F}^\vee} \ \wedge\ x = \mathsf{true} \implies \mathcal{F}^\vee\}$ =
  match $p$ with ZK$(zkv, stm)$ ⇒
    let $c_1$ = checkEq $stm$ $f$;
    let $c_2$ = checkZK $p$;
    if $c_1$ = true then
      if $c_2$ = true then
        $[\![\underline{\mathcal{F}^\vee}, zkv, stm, f, 0]\!]$ [ context$^{=}(\psi)$[true] ]
      else
        false
    else
      false
```

Listing A.39: verify top-level structure

The code first obtains access to the proven statement *stm* to check whether the statement inside p matches the formula f. Then, it ensures, that the zero-knowledge value of p and the proven statement match. Only if both checks succeed, are the specific tests for for $\mathcal{F}^\vee$ executed. Finally, when the logical map ψ is set up by the macros, we run the code that checks for the equalities induced by ψ. We emphasize that the conditional code for ψ is created after the translation function has set up that map.

```
$Or$-Macro(Or$(\underline{\mathcal{F}_1^\vee}, \underline{\mathcal{F}_2^\vee}), zkv, stm, f)$ $\triangleq$
  match $stm$ with Or$_\mathsf{p}(stm_1, stm_2)$ $\implies$
      match $f$ with Or$(f_1, f_2)$ $\implies$
          let $tmp_{zkv}$ = openZK $zkv$;
          let $(stm', r)$ = $tmp_{zkv}$;
          match $stm'$ with Or$_\mathsf{p}(stm_1, stm_2)$ $\implies$
              let $zkv_1$ = commitZK $(stm'_1, r)$;
              let $zkv_2$ = commitZK $(stm'_2, r)$;
              let $res_1$ = verify$_{\underline{\mathcal{F}_1^\vee}}$ ZK$(zkv_1, stm_1)$ $f_1$;
              if $res_1$ = true then
                $res_1$
              else
```

```
                    $\mathsf{verify}_{\underline{\mathcal{F}_2^\vee}}$  $\mathsf{ZK}(zkv_2, r)$  $f_2$
               | _  $\Longrightarrow$  $\mathsf{false}$
          | _  $\Longrightarrow$  $\mathsf{false}$
     | _  $\Longrightarrow$  $\mathsf{false}$
```

Listing A.40: *Or*-Macro($\mathsf{Or}(\underline{\mathcal{F}_1^\vee}, \underline{\mathcal{F}_2^\vee}), zkv, stm, f$).

The macro for the logical disjunction first matches the given statement *stm* and the given formula f, and then destructs the zero-knowledge value. The reason is that it verifies the left branch of the disjunction and, should that verification return false, verifies the right branch. If the left branch returns true, then the result is returned, otherwise the result of the verification of the right branch is returned. Since the verification expects as input a value of type proof, we assemble the proofs $\mathsf{ZK}(zkv_1, r)$ and $\mathsf{ZK}(zkv_2, r)$ for both branches on the fly using the randomness contained in the original proof. Since these proof values are only used internally and never accessible outside of the verification function, using this randomness causes no problems.

Since the disjunction is valid if either of the branches is valid (and in particular, the disjunctive formula $\mathcal{F}^\vee$ is implied if either of its branches is valid), this check is sufficient to return the refinement $\mathcal{F}^\vee$. Also notice that this fact is also sufficient to extend a possible existential quantification across a disjunction, i.e., if $\exists \widetilde{x}.\ \mathcal{F}_1^\vee$ holds, then $\exists \widetilde{x}, \widetilde{x}'.\ \mathcal{F}_1^\vee \vee \mathcal{F}_2^\vee$ holds.

```
$And$-Macro($\mathsf{And}(\underline{\mathcal{F}_1^\wedge}, \underline{\mathcal{F}_2^\wedge}), stm, f, \omega_1, \omega_2$)  $\triangleq$
  match $stm$ with $\mathsf{And_p}(stm_1, stm_2)$  $\Longrightarrow$
      match $f$ with $\mathsf{And}(f_1, f_2)$  $\Longrightarrow$
          $[\![ \underline{\mathcal{F}_1^\wedge}, *, stm_1, f_1, \omega_1 ]\!]$  [  $[\![ \underline{\mathcal{F}_2^\wedge}, *, stm_2, f_2, \omega_2 ]\!]$  ]
        | _  $\Longrightarrow$  $\mathsf{false}$
    | _  $\Longrightarrow$  $\mathsf{false}$
```

Listing A.41: *And*-Macro($\mathsf{And}(\underline{\mathcal{F}_1^\wedge}, \underline{\mathcal{F}_2^\wedge}), stm, f, \omega_1, \omega_2$).

The macro for logical conjunctions first executes the code for the left branch and if this branch verifies successfully, then the code for the right branch is executed.

Formally, the translation function requires us to input a zero-knowledge value. Since this value is only necessary for disjunctions that, by the required disjunctive form, may not appear underneath conjunctions, we intentionally neglect the code for assembling some zero-knowledge values and input no value (denoted by $*$).

Notation. The following macros conditionally contain code that is highlighted in gray. This code is added depending on whether values in the corresponding formula are revealed or not. For instance, in the following says-macro, we add code for every individual arguments of the stated predicate that is revealed.

```
$Says$-Macro($\mathsf{Says}(x_{vk}, P_k^F(x_1, \ldots, x_n)), stm, f, \omega$)
1     match $stm$ with $\mathsf{Says_p}(c_{sig}, \_, c_z, \_, p')$ $\Longrightarrow$
2     match $p'$ with $P_k^P(c_{arg_1}, \_, \ldots, c_{arg_n}, \_)$ $\Longrightarrow$
3         match $f$ with $\mathsf{Says}(arg_0', f')$ $\Longrightarrow$
4         match $f'$ with $P_k^F(arg_1', \ldots, arg_n')$ $\Longrightarrow$
5             let $tmp_{sig}$ = $\mathsf{openCommit}$ $c_{sig}$;
6             let $(sig, r_{sig})$ = $tmp_{sig}$;
7             let $tmp_z$ = $\mathsf{openCommit}$ $c_z$;
8             let $(arg_0, r_{arg_0})$ = $tmp_z$;
              $\vdots$
9             let $tmp_{n-1}$ = $\mathsf{openCommit}$ $c_{arg_{n-1}}$;
10            let $(arg_{n-1}, r_{arg_{n-1}})$ = $tmp_{n-1}$;
11            let $tmp_n$ = $\mathsf{openCommit}$ $c_{arg_n}$;
12            let $(arg_n, r_{arg_n})$ = $tmp_n$;
   Add lines 13 and $U_0^\psi$
   for each $i > 0$ if $\exists x.\ arg_i' = \mathsf{Revealed}\ x$, i.e., $i$-th argument is revealed
13            let $arg_i^o$ = $\mathsf{getRevealed}$ $arg_i'$;
$U_0^\psi$    $\psi := \psi[\mathcal{E}(\omega + i) \mapsto \psi(\mathcal{E}(\omega + i)) \cup \{arg_i^o\}]$
   EndAdd
15            let $z''$ = match $arg_0'$ with
16              | $\mathsf{Revealed}$ $x$ $\Longrightarrow$
17                $\mathsf{PKI}$ $x$
$U_0^\phi$        $\phi := \phi[\mathcal{E}(\omega) \mapsto z'']$
19              | _ $\Longrightarrow$
20                $\phi(\mathcal{E}(\omega))$
21            if $z''$ = $arg_0$ then
22              let $m$ = $\mathsf{check}_{sig}$ $z''$ $sig$;
23              match $m$ with
24                | $P_k^S(y_1', \ldots, y_n')$ $\Longrightarrow$
25                  if $arg_1$ = $y_1'$ then
                    $\vdots$
26                  if $arg_n$ = $y_n'$ then
$U_1^\phi$             $\phi := \phi[\mathcal{E}(\omega + 1) \mapsto y_1']$
                       $\vdots$
$U_n^\phi$             $\phi := \phi[\mathcal{E}(\omega + n) \mapsto y_n']$
$U_1^\psi$             $\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{arg_0\}]$
$U_2^\psi$             $\psi := \psi[\mathcal{E}(\omega + 1) \mapsto \psi(\mathcal{E}(\omega + 1)) \cup \{arg_1\}]$
                       $\vdots$
```

```
$U^{\psi}_{n+1}$                        $\psi := \psi[\mathcal{E}(\omega+n) \mapsto \psi(\mathcal{E}(\omega+n)) \cup \{arg_n\}]$
32                                      •
                                      else
                                        false
                                        ⋮
                                      else
                                        false
                                   | _  ⟹  false
                            else
                             false
                         | _  ⟹  false
                         | _  ⟹  false
                    | _  ⟹  false
                    | _  ⟹  false
```

Listing A.42: *Says*-Macro($\mathsf{Says}(x_{vk}, P^F_k(x_1, \ldots, x_n)), stm, f, \omega$)

The says-macro matches f and *stm* with the expected pattern and starts to unseal all the commitments. The purpose of the code lines 13 and U^{ψ}_0 is to bind the revealed values (i.e., the values themselves, not the *RevHid* values) to the variables arg^o_i. These variables are then stored in the map ψ together with the content of the corresponding commitments arg_i. Consequently, the equality tests performed by the $\mathsf{context}^{=}(\psi)$ will enforce that the values stored in the commitments and corresponding revealed values in a formula are equal in case the return value is true.

We establish the authenticity of the user that originally created the zero-knowledge proof is determined in lines 15-20. If the user is revealed, we use the PKI to establish the type *verkey* for the corresponding value. Otherwise, we use the map $\mathcal{E}$ to retrieve the previously established verification key. Here, we see why the well-formedness of formulas (see Section 2.5.2.3 and Appendix A.2) is important: without well-formed formulas, e.g., if all user identifiers are hidden, we cannot obtain the corresponding strongly-typed verification keys since the map $\mathcal{E}$ is empty and there is no value to retrieve from the PKI.

If the committed user identifier matches the signing key derived in line 15 and the committed values match the values contained within the signature (lines 21-26), the verification successfully finished.

The remaining lines U^{ϕ}_1-U^{ψ}_{n+1} update the logical maps, ensuring that newly obtained verification keys (since these messages are taken from the signature, verification keys are given type *verkey*) are stored in ϕ (lines U^{ϕ}_1-U^{ϕ}_n) and that the necessary equality constraints are recorded in ψ (lines U^{ψ}_1-U^{ψ}_{n+1}).

```
$SSP$-Macro($\mathsf{SSP}(x_{vk}, x_s, x_{psd}), stm, f, \omega$) $\triangleq$
  match $stm$ with $\mathsf{SSP_p}(c_z, \_, c_s, \_, c_{psd}, \_, c_x)$ ⟹
      match $f$ with $\mathsf{SSP}(z', s', psd')$ ⟹
          let $tmp_x$ = $\mathsf{openCommit}_{sk}$ $c_x$;
```

```
4            let $(x, r_x)$ = $tmp_x$;
5            let $tmp_z$ = openCommit $c_z$;
6            let $(z, r_z)$ = $tmp_z$;
7            let $tmp_s$ = openCommit $c_s$;
8            let $(s, r_s)$ = $tmp_s$;
9            let $tmp_{psd}$ = openCommit $c_{psd}$;
10           let $(psd, r_{psd})$ = $tmp_{psd}$;
11           let $(x'', x')$ = $x$;
      Add lines 12 and $U_0^\psi$
      if $\exists y.\ z' = \mathsf{Revealed}\ y$, i.e., the verification key is revealed
12           let $z^o$ = getRevealed $z'$;
$U_0^\psi$   $\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{z^o\}]$
      End Add
      Add lines 14 and$U_1^\psi$
      if $\exists y.\ s' = \mathsf{Revealed}\ y$, i.e., the service is revealed
14           let $s^o$ = getRevealed $s'$;
$U_1^\psi$   $\psi := \psi[\mathcal{E}(\omega+1) \mapsto \psi(\mathcal{E}(\omega+1)) \cup \{s^o\}]$
      End Add
      Add lines 16 and $U_2^\psi$
      if $\exists y.\ psd' = \mathsf{Revealed}\ y$, i.e., the pseudonym is revealed
16           let $psd^o$ = getRevealed $psd'$;
$U_2^\psi$   $\psi := \psi[\mathcal{E}(\omega+2) \mapsto \psi(\mathcal{E}(\omega+2)) \cup \{psd^o\}]$
      End Add
$U_3^\psi$   $\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{z, x'\}]$
$U_4^\psi$   $\psi := \psi[\mathcal{E}(\omega+1) \mapsto \psi(\mathcal{E}(\omega+1)) \cup \{s\}]$
$U_5^\psi$   $\psi := \psi[\mathcal{E}(\omega+2) \mapsto \psi(\mathcal{E}(\omega+2)) \cup \{psd\}]$
21           let $psd''$ = computePsd $x$ $s$;
22           if $psd''$ = $psd$ then
23             •
             else
               false
         | _ ⟹ false
      | _ ⟹ false
```

Listing A.43: SSP-Macro($\mathsf{SSP}(x_{vk}, x_s, x_{psd}), stm, f, \omega$)

The first step is the usual opening of all the commitments. The notable difference from the other code is the splitting of the signing key x to obtain the corresponding verification key x' in line 11. We will use x' to tie the verification key used in computing the service-specific pseudonym to the user identifier contained therein using ψ (line U_3^ψ).

Similarly to lines 13 and U_0^ψ from the says-macro, the lines 12 and U_2^ψ enforce that the values revealed in the formula are bound to the respective variables z^o, s^o, and psd^o. These are put into the logical map ψ and are compared to the corresponding variables z, s, and psd in the code $\mathsf{context}^{=}(\psi)$ at the end of the verification; z, s, and psd are put into the logical map in lines U_3^ψ-U_5^ψ.

The whole purpose of the SSP proof is to show that a pseudonym is computed correctly. We compute the pseudonym corresponding to the given signing key and service in line 21 and compare to the given pseudonym line 22. This modus operandi of the symbolic implementation very closely matches the concrete cryptographic implementation.

```
$Rel$-Macro($\mathsf{REL}(x, op, y), stm, f, \omega$) $\triangleq$
1        match $stm$ with $\mathsf{REL_p}(c_x, \_, op, c_y, \_)$ $\Longrightarrow$
2            match $f$ with $\mathsf{REL}(x', op', y')$ $\Longrightarrow$
3                let $tmp_x$ = openCommit $c_x$;
4                let $(x, r_x)$ = $tmp_x$;
5                let $tmp_y$ = openCommit $c_y$;
6                let $(y, r_y)$ = $tmp_y$;
     Add lines 7 and $U_0^\psi$
     if $\exists z.\ x' = \mathsf{Revealed}\ z$, i.e., the left operand is revealed
7                let $x^o$ = getRevealed $x'$;
$U_0^\psi$       $\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x^o\}]$
     End Add
     Add lines 9 and $U_1^\psi$
     if $\exists z.\ y' = \mathsf{Revealed}\ z$, i.e., the right operand is revealed
9                let $y^o$ = getRevealed $y'$;
$U_1^\psi$       $\psi := \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{y^o\}]$
     End Add
$U_2^\psi$       $\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x\}]$
$U_3^\psi$       $\psi := \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{y\}]$
13               let $b$ = $\mathsf{func}^r_{op}$ $x$ $y$;
14               if $b$ = true then
15                 if $op$ = $op'$ then
16                   •
                   else
                     false
                 else
                   false
             | _ $\Longrightarrow$ false
```

```
  | _  $\Longrightarrow$  false
```

Listing A.44: *Rel*-Macro($\mathsf{REL}(x, op, y), stm, f, \omega$)

The code macro for relational proofs proceeds along the usual pattern. The commitments are opened and the values revealed in the formula are bound to variables and added the map ψ. We validate the proven relation in line 13 and the result in line 14. Additionally, we ensure that the checked relation matches the one requested in the formula in line 15.

```
      $Eqn$-Macro($\mathsf{EQN}(x, op, y, z), stm, f, \omega$) $\triangleq$
1       match $stm$ with $\mathsf{EQN_p}(c_x, \_, op, c_y, \_, c_z, \_)$ $\Longrightarrow$
2           match $f$ with $\mathsf{EQN}(x', op', y', z')$ $\Longrightarrow$
3               let $tmp_x$ = $\mathsf{openCommit}$ $c_x$;
4               let $(x, r_x)$ = $tmp_x$;
5               let $tmp_y$ = $\mathsf{openCommit}$ $c_y$;
6               let $(y, r_y)$ = $tmp_y$;
7               let $tmp_z$ = $\mathsf{openCommit}$ $c_z$;
8               let $(z, r_z)$ = $tmp_z$
      Add lines 9 and $U_0^\psi$
      if $\exists z.\, x' = \mathsf{Revealed}\ z$, i.e., the left operand is revealed
9               let $x^o$ = $\mathsf{getRevealed}$ $x'$;
$U_0^\psi$             $\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x^o\}]$
      End Add
      Add lines 11 and $U_1^\psi$
      if $\exists z.\, y' = \mathsf{Revealed}\ z$, i.e., the right operand is revealed
11              let $y^o$ = $\mathsf{getRevealed}$ $y'$;
$U_1^\psi$             $\psi := \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{y^o\}]$
      End Add
      Add lines 13 and $U_2^\psi$
      if $\exists z.\, y' = \mathsf{Revealed}\ z$, i.e., the right operand is revealed
13              let $z^o$ = $\mathsf{getRevealed}$ $z'$;
$U_2^\psi$             $\psi := \psi[\mathcal{E}(\omega + 3) \mapsto \psi(\mathcal{E}(\omega + 3)) \cup \{z^o\}]$
      End Add
$U_3^\psi$             $\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x\}]$
$U_4^\psi$             $\psi := \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{y\}]$
$U_5^\psi$             $\psi := \psi[\mathcal{E}(\omega + 3) \mapsto \psi(\mathcal{E}(\omega + 3)) \cup \{z\}]$
18              let $b$ = $\mathsf{func}^e_{op}$ $x$ $y$;
19              if $b$ = $z$ then
```

```
20              if $op$ = $op'$ then
21                $\bullet$
                else
                  false
              else
                false
          | _ $\Longrightarrow$ false
      | _ $\Longrightarrow$ false
```

Listing A.45: Eqn-Macro($\mathsf{EQN}(x, op, y, z), stm, f, \omega$)

As usual, the commitments are opened and the values revealed in the formula are bound to variables and added the map ψ. We validate the proven equation in line 18, we compare the obtained result in line 19 and we check the performed operation in line 20.

```
$LM$-Macro($\mathsf{LM}(x, b, \ell), stm, f, \omega$) $\triangleq$
1         match $stm$ with $\mathsf{LM_p}(c_x, \_, c_b, \_, \ell)$ $\Longrightarrow$
2             match $f$ with $\mathsf{LM}(x', b', \ell')$ $\Longrightarrow$
3                 let $tmp_x$ = openCommit $c_x$;
4                 let $(x, r_x)$ = $tmp_x$;
5                 let $tmp_b$ = openCommit $c_b$;
6                 let $(b, r_b)$ = $tmp_b$;
          Add lines 7 and $U_0^\psi$
          if $\exists y.\ x' = \mathsf{Revealed}\ y$, i.e., the pseudonym is revealed
7                 let $x^o$ = getRevealed $x'$;
$U_0^\psi$                $\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x^o\}]$
          End Add
          Add lines 9 and $U_1^\psi$
          if $\exists x.\ b' = \mathsf{Revealed}\ y$, i.e., the attribute is revealed
9                 let $b^o$ = getRevealed $b'$;
$U_1^\psi$                $\psi := \psi[\mathcal{E}(\omega+1) \mapsto \psi(\mathcal{E}(\omega+1)) \cup \{b^o\}]$
          End Add
$U_2^\psi$                $\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x\}]$
$U_3^\psi$                $\psi := \psi[\mathcal{E}(\omega+1) \mapsto \psi(\mathcal{E}(\omega+1)) \cup \{b\}]$
$U_4^\psi$                $\psi := \psi[\mathcal{E}(\omega+2) \mapsto \psi(\mathcal{E}(\omega+2)) \cup \{\ell, \ell'\}]$
14                let $r$ = List.member$^{(2,2)}\langle pseudo * bitstring\rangle$ $x$ $b$ $\ell$;
15                if $r$ = true then
16                  $\bullet$
                else
                  false
```

```
              | _  $\Longrightarrow$  false
          | _  $\Longrightarrow$  false
```

Listing A.46: LM-Macro($\mathsf{LM}(x, b, \ell), \mathit{stm}, f, \omega$)

The code macro for list membership proofs proceeds along the usual pattern. The commitments are opened and the values revealed in the formula are bound to variables and added the map ψ. We validate the list membership in line 14 and 15.

```
$\mathit{LNM}$-Macro($\mathsf{LNM}(x, \ell), \mathit{stm}, f, \omega$) $\triangleq$
1     match $\mathit{stm}$ with $\mathsf{LNM}_{\mathsf{p}}(c_x, \_, \ell)$  $\Longrightarrow$
2         match $f$ with $\mathsf{LNM}(x', \ell')$  $\Longrightarrow$
3             let $\mathit{tmp}_x$ = $\mathsf{openCommit}$ $c_x$;
4             let $(x, r_x)$ = $\mathit{tmp}_x$;
      Add lines 5 and $U_0^\psi$
      if $\exists y.\, x' = \mathsf{Revealed}\ y$, i.e., the pseudonym is revealed
5             let $x^o$ = $\mathsf{getRevealed}$ $x'$;
$U_0^\psi$            $\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x^o\}]$
      End Add
$U_1^\psi$            $\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x\}]$
$U_2^\psi$            $\psi := \psi[\mathcal{E}(\omega + 1) \mapsto \psi(\mathcal{E}(\omega + 1)) \cup \{\ell, \ell'\}]$
9             let $r$ = $\mathsf{List.member}^{(1,2)}\langle \mathit{pseudo} * \mathit{bitstring} \rangle$ $x$ $\ell$;
10            if $r$ = false then
11              $\bullet$
              else
                false
           | _  $\Longrightarrow$  false
       | _  $\Longrightarrow$  false
```

Listing A.47: LNM-Macro($\mathsf{LNM}(x, \ell), \mathit{stm}, f, \omega$)

As usual, the code macro for list non-membership proofs opens the commitments, and the values revealed in the formula are bound to variables and added the map ψ. We validate the list non-membership in line 9 and 10. Notice that for the non-membership, we need the result r test to be false in order to continue.

```
$\mathit{Escrow}$-Macro($\mathsf{EscrowInfo}(x_{EA}, x_{vk}, x_R, x_s, x_{idr}), \mathit{stm}, f, \omega$) $\triangleq$
1     match $\mathit{stm}$ with $\mathsf{EscrowInfo}_{\mathsf{p}}(z, c_x, \_, c_R, \_, c_s, \_, c_{idr}, \_, c_r)$  $\Longrightarrow$
2         match $f$ with $\mathsf{EscrowInfo}(z', x', R', s', \mathit{idr}')$  $\Longrightarrow$
3             let $\mathit{tmp}_x$ = $\mathsf{openCommit}$ $c_x$;
4             let $(x, r_x)$ = $\mathit{tmp}_x$;
```

```
5                    let $tmp_R$ = openCommit $c_R$;
6                    let $(R, r_R)$ = $tmp_R$;
7                    let $tmp_s$ = openCommit $c_s$;
8                    let $(s, r_s)$ = $tmp_s$;
9                    let $tmp_{idr}$ = openCommit $c_{idr}$;
10                   let $(idr, r_{idr})$ = $tmp_{idr}$;
11                   let $tmp_r$ = openCommit $c_r$;
12                   let $(r, r_r)$ = $tmp_r$;
$U_0^\psi$           $\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{z'\}]$
    Add lines 14 and $U_1^\psi$
    if $\exists y.\ x' = \mathsf{Revealed}\ y$, i.e., if the user identifier is revealed
14                   let $x^o$ = getRevealed $x'$;
$U_1^\psi$           $\psi := \psi[\mathcal{E}(\omega + 1) \mapsto \psi(\mathcal{E}(\omega + 1)) \cup \{x^o\}]$
    End Add
    Add lines 16 and $U_2^\psi$
    if $\exists y.\ R' = \mathsf{Revealed}\ y$, i.e., if the value $R$ is revealed
16                   let $R^o$ = getRevealed $R'$;
$U_2^\psi$           $\psi := \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{R^o\}]$
    End Add
    Add lines 18 and $U_3^\psi$
    if $\exists y.\ s' = \mathsf{Revealed}\ y$, i.e., if the service is revealed
18                   let $s^o$ = getRevealed $s'$;
$U_3^\psi$           $\psi := \psi[\mathcal{E}(\omega + 3) \mapsto \psi(\mathcal{E}(\omega + 3)) \cup \{s^o\}]$
    End Add
    Add lines 20 and $U_4^\psi$
    if $\exists y.\ idr' = \mathsf{Revealed}\ y$, i.e., the escrow identifier is revealed
20                   let $idr^o$ = getRevealed $idr'$;
$U_4^\psi$           $\psi := \psi[\mathcal{E}(\omega + 4) \mapsto \psi(\mathcal{E}(\omega + 4)) \cup \{idr^o\}]$
    End Add
$U_5^\psi$           $\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{z\}]$
$U_6^\psi$           $\psi := \psi[\mathcal{E}(\omega + 1) \mapsto \psi(\mathcal{E}(\omega + 1)) \cup \{x\}]$
$U_7^\psi$           $\psi := \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{R\}]$
$U_8^\psi$           $\psi := \psi[\mathcal{E}(\omega + 3) \mapsto \psi(\mathcal{E}(\omega + 3)) \cup \{s\}]$
$U_9^\psi$           $\psi := \psi[\mathcal{E}(\omega + 4) \mapsto \psi(\mathcal{E}(\omega + 4)) \cup \{idr\}]$
27                   let $idr''$ = computeIDR $z$ $x$ $r$ $R$ $s$;
28                   if $idr$ = $idr''$ then
```

```
            •
          else
            else
        | _  $\Longrightarrow$  false
    | _  $\Longrightarrow$  false
```

Listing A.48: *Escrow*-Macro($\mathsf{EscrowInfo}(x_{EA}, x_{vk}, x_R, x_s, x_{idr}), \mathit{stm}, f, \omega$)

Following the usual pattern, the commitments are opened and the values revealed in the formula are bound to variables and added to the map ψ. Since we stipulate that the identity z of the involved trusted third party is always revealed, there is no code added to extract the identity when it is revealed. Instead, we immediately add z into the map ψ in line U_0^ψ.

A.2. Well-Typedness of the RCF Implementation

We start with the definitions and lemmas that will ultimately pave our way to showing that our API methods are well-typed. We start formalizing our notion of well-formed formula. Intuitively, a formula is well-formed if we can check that it originates from a trustworthy principal of the system. Later in the type-checking proofs, the well-formedness will establish that all values that are used as verification keys can be given the verification key type *verkey*. Since type-checking depends on the types of values, well-formedness is a central notion and is of paramount importance in our proofs.

We call a key u *registered* if it is publicly verifiable that u belongs to a principal of the system. Typically, the owner of a key u registers her key in a public-key infrastructure (PKI) to establish a publicly-verifiable connection between her identity and her public key. This infrastructure can be hierarchically-ordered such as VeriSign [216] or it can be distributed such as webs of trust [207].

Definition A.5 (Trustworthiness of keys). *A key u is* trustworthy *in a monomial $\mathcal{M} = \bigwedge_{i=1}^{m} ap_i$ iff one of the following conditions holds:*

- *$u = vk$ is registered;*
- *there exists $ap_j = \mathsf{Says}(u_k, F)$ such that u is a variable occurring free in F and u_k is trustworthy in $\mathcal{M}^{<j} := \bigwedge_{i=1}^{j-1} ap_i$.*

If we apply this definition to our zero-knowledge proofs, the intuitive meaning is that a key is trustworthy if the key is not hidden and it originates from a known principal of the system, or if it is hidden but it is authenticated (via a says-predicate) that is issued by a trustworthy key.

Definition A.6 (Disjunctive form)**.** *We say a formula $\mathcal{F}$ is in* disjunctive form *if and only if $\mathcal{F} = \exists\widetilde{x}.\ \bigvee_{i=1}^{m} \mathcal{M}_i$, where $\mathcal{M}_i = \bigwedge_{j=1}^{n} ap_j$.*

Lemma A.1 states the well-known result that every logical formula F can be rewritten in disjunctive form. A disjunctive normal form can be obtained, for instance, by lexicographical order. We write $dnf(\mathcal{F})$ for the disjunctive normal form of $\mathcal{F}$.

Definition A.7 (Well-formedness of formulas)**.** *A monomial $\mathcal{M} = \bigwedge_{i=1}^{m} ap_i$ is* well-formed *if and only if for every $ap_i = \mathsf{Says}(u_k, F)$, and for every $ap_j = \mathsf{SSP}(u_\ell, s, psd)$, u_k and u_ℓ are trustworthy in $\mathcal{M}^{<i}$ and $\mathcal{M}^{<j}$, respectively.*

A formula S such that $dnf(S) = \exists\widetilde{x}.\ \bigvee_{i=1}^{m} \mathcal{M}_i$ is well-formed if each $\mathcal{M}_i$ is well-formed.

Lemma A.1 (Representation lemma)**.** *For every Boolean formula f, there is a formula g in disjunctive form such that $f \Leftrightarrow g$.*

Proof. The proof is by induction: whenever there is a conjunction on top of a disjunction, apply the distributivity law $(x \vee y) \wedge z \Leftrightarrow (x \wedge z) \vee (y \wedge z)$ or $z \wedge (x \vee y) \Leftrightarrow (z \wedge x) \vee (z \wedge y)$. Possible negations will be pushed down using De'Morgan's laws $\neg(x \wedge y) \Leftrightarrow \neg x \vee \neg y$ and $\neg(x \vee y) \Leftrightarrow \neg x \wedge \neg y$. □

In the following, we assume that all formulas are well-formed and in disjunctive normal form. Note that we do not have negations in our formulas.

Notation and auxiliary lemmas. In this paragraph, we review and briefly discuss important definitions, notations, and lemmas that play a role in the well-typedness proof below. In particular, we will also show all the typing rules since they play a central role in the proofs. The definitions, lemmas, and rules are borrowed from Bengtson et al. [49].

Table A.8 introduces the syntax of RCF. We assume collections of names, variables, and type variables. A name is an identifier, generated at run time, for a channel, while a variable is a placeholder for a value. The RCF syntax is to be read as a reference only; in previous sections, we relaxed this syntax and use the usual programming language syntax, i.e., we use arg_0 as variable names rather than writing x_{arg_0}. If Φ is a phrase of syntax, we write $\Phi\{M/x\}$ to denote the outcome of substituting a value M for each free occurrence of the variable x in Φ. We identify syntax up to the capture-avoiding renaming of bound names and variables. Table A.9 introduces the two RCF kinds and the $\bar{\cdot}$ notation. As mentioned above, intuitively, a public value can be sent to the attacker and a tainted value originates from the attacker. Table A.11 shows the deduction system that decides the well-formedness of typing environments E. Intuitively, this syntactic restriction ensures that, if E is read from left to right, there are no unbound variables in E and every name is at most bound once. Table A.12 and Table A.13 depict the kinding and subtyping rules. Since RCF is equipped with a classical subtyping relation as well as a subtyping based on kinds, both sets of rules often work closely together. Finally, Table A.14 and Table A.15 show all the RCF typing rules. The typing rules depend on all the rules introduced above.

a, b, c	names
x, y, z	variables
$h ::=$	value constructors
fold	constructor for recursive types
inl	left constructor for sum types
inr	right constructor for sum types
h_i	constructor for arbitrary sum types $\mathsf{datatype}\ U = h_1\ \mathsf{of}\ T_1 \mid \cdots \mid h_n\ \mathsf{of}\ T_n$
$M, N ::=$	values
x	variables
$()$	unit
$\text{fun } x \to A$	function (scope of x is A)
(M, N)	pairs
$h\ M$	construction
$A, B ::=$	expressions
M	value
$M\ N$	application
$M = N$	syntactic equality
$\mathsf{let}\ x = A;\ B$	let (scope of x is B)
$\mathsf{let}\ (x, y) = M;\ B$	pair split (scope of x, y is B)
$\mathsf{match}\ M\ \mathsf{with}\ h\ x \to A\ \mathsf{else}\ B$	constructor match (scope of x is A)
$(\nu a)A$	restriction (scope of a is A)
$A \Rsh B$	fork
$a!M$	transmission of M on channel a
$a?$	receive message off channel a
$\mathsf{assume}\ F$	assumption of the formula F
$\mathsf{assert}\ F$	assertion of formula F

Table A.8.: Syntax of RCF

$\nu ::= \mathsf{pub} \mid \mathsf{tnt}$	kind (public or tainted)

Let $\overline{\nu}$ be defined as $\overline{\mathsf{pub}} := \mathsf{tnt}$ and $\overline{\mathsf{tnt}} := \mathsf{pub}$.

Table A.9.: Syntax of kinds

$$\begin{array}{ll}
\mu ::= & \text{environment entry} \\
\quad \alpha & \quad \text{type variable} \\
\quad \alpha :: \nu & \quad \text{kinding for recursive type } \alpha \\
\quad \alpha <: \alpha' & \quad \text{subtyping for recursive types } \alpha \neq \alpha' \\
\quad a \updownarrow T & \quad \text{channel name} \\
\quad x : T & \quad \text{variable} \\
E ::= \mu_1, \ldots, \mu_\ell &
\end{array}$$

$$\begin{array}{l}
\mathit{dom}(\alpha) = \{\alpha\} \\
\mathit{dom}(\alpha :: \nu) = \{\alpha\} \\
\mathit{dom}(\alpha <: \alpha') = \{\alpha, \alpha'\} \\
\mathit{dom}(a \updownarrow T) = \{a\} \\
\mathit{dom}(x : T) = \{x\} \\
\mathit{dom}(E) = \mathit{dom}(\mu_1) \cup \cdots \cup \mathit{dom}(\mu_\ell)
\end{array}$$

$$\mathit{recvar}(E) = \{\alpha, \alpha' \mid (\alpha <: \alpha') \in E\} \cup \{\alpha \mid (\alpha :: \nu) \in E\}$$

Let $E = \mu_1, \ldots, \mu_\ell$. We write $\mu \in E$ to denote that $\mu = \mu_i$ for some $i \in \{1, \ldots, \ell\}$. We use the notation $T <:> T'$ to denote $T <: T'$ and $T' <: T$.

Table A.10.: Syntax of typing environments.

Empty Env

$$E \vdash \diamond$$

Env Entry

$$\frac{E \vdash \diamond \qquad \mathit{fnfv}(\mu) \subseteq \mathit{dom}(E) \qquad \mathit{dom}(\mu) \cap \mathit{dom}(E) = \emptyset}{E, \mu \vdash \diamond}$$

Type

$$\frac{E \vdash \diamond \qquad \mathit{fnfv}(T) \subseteq \mathit{dom}(E)}{E \vdash T}$$

Derive

$$\frac{E \vdash \diamond \qquad \mathit{fnfv}(F) \subseteq \mathit{dom}(E) \qquad \mathit{forms}(E) \vdash F}{E \vdash F}$$

$$\mathit{forms}(E) ::= \begin{cases} \{F\{y/x\}\} \cup \mathit{forms}(y : T) & \text{if } E = (y : \{x : T \mid F\}) \\ \mathit{forms}(E') \cup \mathit{forms}(\mu) & \text{if } E = E', \mu \\ \emptyset & \text{otherwise} \end{cases}$$

Table A.11.: Rules for well-formedness and deduction.

$$\textsc{Kind Var}\quad \frac{E \vdash \diamond \qquad (\alpha :: \nu) \in E}{E \vdash \alpha :: \nu} \qquad\qquad \textsc{Kind Unit}\quad \frac{E \vdash \diamond}{E \vdash \mathit{unit} :: \nu}$$

$$\textsc{Kind Fun}\quad \frac{E \vdash T :: \overline{\nu} \qquad E, x : T \vdash (\Pi x : T.\, U) :: \nu}{E \vdash (\Pi x : T.\, U) :: \nu} \qquad\qquad \textsc{Kind Pair}\quad \frac{E \vdash T :: \nu \qquad E, x : T \vdash U :: \nu}{E \vdash (\Sigma x : T.\, U) :: \nu}$$

$$\textsc{Kind Sum}\quad \frac{E \vdash T :: \nu \qquad E \vdash U :: \nu}{E \vdash (T + U) :: \nu} \qquad \textsc{Kind Rec}\quad \frac{E, \alpha :: \nu \vdash T :: \nu}{E \vdash (\mu\alpha.\, T) :: \nu} \qquad \textsc{Kind Refine Public}\quad \frac{E \vdash \{x : T \mid \mathcal{F}\} \qquad E \vdash T :: \mathsf{pub}}{E \vdash \{x : T \mid \mathcal{F}\} :: \mathsf{pub}}$$

$$\textsc{Kind Refine Tainted}\quad \frac{E \vdash T :: \mathsf{tnt} \qquad E, x : T \vdash C}{E \vdash \{x : T \mid \mathcal{F}\} :: \mathsf{tnt}} \qquad\qquad \textsc{Kind OK Tainted}\quad \frac{E \vdash \{F\} \qquad E \vdash F}{E \vdash \{F\} :: \mathsf{tnt}}$$

Table A.12.: Kinding rules: $E \vdash T :: \nu$

$$\textsc{Sub Refl}\quad \frac{E \vdash T \qquad \mathit{recvar}(E) \cap \mathit{fnfv}(T) = \emptyset}{E \vdash T <: T} \qquad\qquad \textsc{Sub Public Tainted}\quad \frac{E \vdash T :: \mathsf{pub} \qquad E \vdash U :: \mathsf{tnt}}{E \vdash T <: U}$$

$$\textsc{Sub Fun}\quad \frac{E \vdash T' <: T \qquad E \vdash U <: U'}{E \vdash (\Pi x : T.\, U) <: (\Pi x : T'.\, U')} \qquad\qquad \textsc{Sub Pair}\quad \frac{E \vdash T <: T' \qquad E, x : T \vdash U <: U'}{E \vdash (\Sigma x : T.\, U) <: (\Sigma x : T'.\, U')}$$

$$\textsc{Sub Sum}\quad \frac{E \vdash T <: T' \qquad E \vdash U <: U'}{E \vdash (T + U) <: (T' + U')} \qquad\qquad \textsc{Sub Var}\quad \frac{E \vdash \diamond \qquad E \vdash (\alpha <: \alpha') \in E}{E \vdash \alpha <: \alpha'}$$

$$\textsc{Sub Refine Left}\quad \frac{E \vdash \{x : T \mid \mathcal{F}\} \qquad E \vdash T <: T'}{E \vdash \{x : T \mid \mathcal{F}\} <: T'} \qquad\qquad \textsc{Sub Refine Right}\quad \frac{E \vdash T <: T' \qquad E, x : T \vdash \mathcal{F}}{E \vdash T <: \{x : T' \mid \mathcal{F}\}}$$

Table A.13.: Subtyping rules: $E \vdash T <: U$

$$\textsc{Val Var}\quad \frac{E \vdash \diamond \qquad (x : T) \in E}{E \vdash x : T} \qquad \textsc{Val Unit}\quad \frac{E \vdash \diamond}{E \vdash () : \mathit{unit}} \qquad \textsc{Val Fun}\quad \frac{E, x : T \vdash A : U}{E \vdash \text{fun } x \to A : (\Pi x : T.\, U)}$$

$$\textsc{Val Pair}\quad \frac{E \vdash M : T \qquad E \vdash N : U\{M/x\}}{E \vdash (M, N) : \Sigma x : T.\, U} \qquad \textsc{Val Refine}\quad \frac{E \vdash M : T \qquad E \vdash F\{M/x\}}{E \vdash M : \{x : T \mid F\}}$$

$$\textsc{Val Inl Inr Fold}\quad \frac{h : (T, U) \qquad E \vdash M : T \qquad E \vdash U}{E \vdash h(M) : U} \qquad \textsc{Val Ok}\quad \frac{E \vdash F}{E \vdash () : \{F\}}$$

$\mathsf{fold} : (T\{\mu\alpha.\, T/\alpha\}, \mu\alpha.\, T)$
$h_i : (T_i, U)$ for types of the form $\mathsf{datatype}\ U = h_1\ \mathsf{of}\ T_1 \mid \cdots \mid h_n\ \mathsf{of}\ T_n$

Table A.14.: Rules for values: $E \vdash M : T$

Definition A.8 (Free names and free variables, executable typing environment)**.** *We write fnfv*(Φ) *for the set of names and variables occurring free in a phrase of syntax* Φ*. We say a phrase is closed to mean it has no free variables (although it may have free names), i.e., fv*$(\Phi) = \emptyset$*.*

We call a typing environment E executable *if and only if recvar*$E = \emptyset$ *(see Table A.10 for the definition of recvar).*

The definition of *fv* (free variables) and *fn* (free names) is analogous to that of *fnfv* and helpful when describing properties of the syntax phrases. The notion of executable environment is a technical necessity and we introduce it as it is a precondition to several lemmas that we use. Intuitively, a typing environment is executable if it only contains entries of the form $(a \updownarrow T)$ and $(x : T)$ for some names a, variables x, and types T.

Logical formulas F that are tracked in typing environments E are a substantial part of our proofs. Technically, a formula F does not occur in E plainly but it is always attached to a refined variable $v : \{x : \mathit{unit} \mid F\}$ where x is some fresh variable that does not occur in F. We use the abbreviated notation $\{F\}$ of ok types to denote $\{\_ : \mathit{unit} \mid F\}$. In this type, the variable of the refinement type is unnamed (as it does not occur in F). Formally, we use fresh variables that occur nowhere else to instantiate the placeholders _. With a slight abuse of notation, we also write $\mu := \{F\}$ to occur in typing environments to denote $\mu := \_ : \{F\}$ where the placeholder is also instantiated with a fresh variable that does not occur in F and, in particular, that does not occur anywhere else in E. As mentioned above, we use the convention that $\mathcal{F}^{\vee}$ denotes formulas in disjunctive normal form without negation, $\mathcal{F}^{\wedge}$ denotes formulas that contain no disjunctions and no negations, and $\mathcal{F}^{e}$ denotes positive elementary formulas.

Exp Subsum

$$\frac{E \vdash A : T \qquad E \vdash T <: T'}{E \vdash A : T'}$$

Exp Appl

$$\frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\}}$$

Exp Split

$$\frac{\begin{array}{c} E \vdash M : (\Sigma x : T.\, U) \\ E, x : T, y : U, \_ : \{(x, y) = M\} \vdash A : V \qquad \{x, y\} \cap fv(V) = \emptyset \end{array}}{E \vdash \mathsf{let}\ (x, y) = M;\ A : V}$$

Exp Match

$$\frac{\begin{array}{c} E \vdash M : T \\ h : (H, T) \qquad E, x : H, \_ : \{h\ x = M\} \vdash A : U \qquad E, \_ : \{\forall x.\ h\ x \neq M\} \vdash B : U \end{array}}{E \vdash \mathsf{match}\ M\ \mathsf{with}\ h\ x \to A\ \mathsf{else}\ B : U}$$

Exp Eq

$$\frac{E \vdash M : T \qquad E \vdash N : U \qquad x \notin fv(M, N)}{E \vdash M = N : \{x : bool \mid (x = \mathsf{true} \wedge M = N) \vee (x = \mathsf{false} \wedge M \neq N)\}}$$

Exp Assume

$$\frac{E \vdash \diamond \qquad fnfv(\mathcal{F}) \subseteq dom(E)}{E \vdash \mathsf{assume}\ \mathcal{F} : \{\mathcal{F}\}}$$

Exp Assert

$$\frac{E \vdash F}{E \vdash \mathsf{assert}\ F : unit}$$

Exp Let

$$\frac{E \vdash A : T \qquad E, x : T \vdash B : U \qquad x \notin fv(U)}{E \vdash \mathsf{let}\ x = A;\ B : U}$$

We omit the rules for channels (Exp Res, Exp Send, and Exp Recv), and for parallel execution (Exp Fork), as they are not used in the proofs.

Table A.15.: Rules for expressions: $E \vdash A : T$

$E \vdash \diamond$	E is syntactically well-formed
$E \vdash T$	in E, T is syntactically well-formed
$E \vdash \mathcal{F}$	formula $\mathcal{F}$ is derivable from E
$E \vdash T :: \nu$	in E, type T has kind ν
$E \vdash T <: U$	in E, type T is a subtype of U
$E \vdash A : T$	in E, expression A has type T

Table A.16.: Judgments of the RCF type system.

Regarding the well-typedness of an expression, we state the following lemmas needed in the proofs below. Intuitively, Lemma A.2 states that removing entries from a well-formed environment "from the right" yields a well-formed environment; Corollary A.1 says that if an expression type-checks under a typing environment E, then E is well-formed.

Lemma A.2 (Shortening typing environments)**.** *Let* $E, \mu \vdash \diamond$*. Then* $E \vdash \diamond$*.*

Proof. Follows immediately as a consequence of rule Env Entry. □

Lemma A.3 (Derived judgments (Lemma 2 [49]))**.**

(1) If $E \vdash T$*, then* $E \vdash \diamond$ *and* $\mathit{fnfv}(T) \subseteq \mathit{dom}(E)$*.*

(2) If $E \vdash F$*, then* $E \vdash \diamond$ *and* $\mathit{fnfv}(F) \subseteq \mathit{dom}(E)$*.*

(3) If $E \vdash T :: v$*, then* $E \vdash T$*.*

(4) If $E \vdash T <: U$*, then* $E \vdash T$ *and* $E \vdash U$*.*

(5) If $E \vdash A : T$*, then* $E \vdash T$ *and* $\mathit{fnfv}(A) \subseteq \mathit{dom}(E)$

In the following proofs, we let let $\mathcal{J}$ range over judgments $\{\diamond, T, \mathcal{F}, T :: \nu, T <: U, A : T\}$.

Corollary A.1 (Type-checking implies well-formedness)**.** *Let* E *be a typing environment. If* $E \vdash \mathcal{J}$*, then* $E \vdash \diamond$*.*

Proof of Corollary A.1. The claim is a direct consequence of Lemma A.3.
□

In our proofs, we will often find ourselves in a situation where we know that certain facts can be proven by a typing environment E, E', but we need to prove these facts with an environment of the form E, μ, E', i.e., the environment E, E' extended with the entry in μ. Intuitively, we would like to drop μ and prove the fact with the environment E, E'. This intuition is formalized and proven by the following weakening lemma.

Lemma A.4 (Weakening, Lemma 6 [49])**.** *If* $E, E' \vdash \mathcal{J}$ *and* $E, \mu, E' \vdash \diamond$*, then* $E, \mu, E' \vdash \mathcal{J}$*, where* μ *corresponds to one single entry in the typing environment.*

Similar to weakening, it will be handy to be able to add formulas to our typing environment E that we can logically derive from E. This strengthening and formalized as follows:

Lemma A.5 (Anon Variable Strengthening, Lemma 4 [49])**.** *If* $E, \{C\}, E' \vdash \mathcal{J}$ *and* $\mathit{forms}(E, E') \vdash C$*, then* $E, E' \vdash \mathcal{J}$*, where* $\mathit{forms}(E'')$ *returns all the logical formulas (i.e., formulas in refinement types) contained in* E''*.*

Many of the upcoming type-checking proofs will be discharged to the following lemma. Intuitively, the lemma states that if an expression does not contain assertions and all exported variables and names are of type *unit*, then the expression type-checks.

Lemma A.6. *Opponent typability (Lemma 34 [49]) Let* $E \vdash \diamond$ *and* E *be executable. If* O *is an expression containing no* assert *such that* $(a \updownarrow \mathit{unit}) \in E$ *for all names* $a \in \mathit{fn}(O)$*, and* $(x : \mathit{unit}) \in E$ *for each variable* $x \in \mathit{fv}(O)$*, then* $E \vdash O : \mathit{unit}$*.*

In the code, we make heavy use of if-statements. These are syntactic sugar that we encode via match-statements into RCF as follows:

Definition A.9 (Encoding and type-checking of conditionals)**.** *We encode conditionals as follows:*

$$\begin{array}{lcl} \mathsf{if}\ M = N\ \mathsf{then} & & \\ \quad A & & \mathsf{let}\ x = (M = N)\ \mathsf{in} \\ \mathsf{else} & \rightsquigarrow & \mathsf{match}\ x\ \mathsf{with\ true} \rightarrow A\ \mathsf{else}\ B \\ \quad B & & \end{array}$$

where x *is a fresh variable that occurs nowhere else. Additionally, we add the following (derived) rule to the* RCF *type-system:*

$$\textsc{Exp If}\qquad \frac{E \vdash M : T \qquad E \vdash N : U \qquad E, \{M = N\} \vdash A : V \qquad E \vdash B : V}{E \vdash \mathsf{if}\ M = N\ \mathsf{then}\ A\ \mathsf{else}\ B : V}$$

The following lemma justifies that we can extend the RCF type system by the Exp If rule.

Lemma A.7. *Rule* Exp If *is derivable in F7.*

Proof. We show that the hypotheses of Exp If are strong enough to imply the premises deriving from type-checking the de-sugared code for the if statement. We start by type-checking the de-sugared version of the code under a typing environment E.

$$\begin{array}{ll} 1 & \mathsf{let}\ x = (M = N)\ \mathsf{in} \\ 2 & \overbrace{\mathsf{match}\ x\ \mathsf{with\ true} \rightarrow A\ \mathsf{else}\ B}^{C:=} \end{array}$$

In the following, we let the type $W := \{y : \mathit{bool} \mid x = \mathsf{true} \wedge M = N \vee x = \mathsf{false} \wedge M \neq N\}$.

Code (line 1):

$$\mathsf{let}\ x = (M = N)\ \mathsf{in}$$

Environment:

$$E$$

Rules:

$$\textsc{Exp Eq}\ \frac{E \vdash M : T \qquad E \vdash N : U \qquad x \notin \mathit{fv}(M,N)}{E \vdash (M = N) : W}$$

$$\textsc{Exp Let}\ \frac{E \vdash (M = N) : W \qquad E, x : W \vdash C : V \qquad x \notin \mathit{fv}(V)}{E \vdash \mathsf{let}\ x = (M = N)\ \mathsf{in}\ C : V}$$

Proof Obligations:

1. $E \vdash M : T$
2. $E \vdash N : U$
3. $x \notin \mathit{fv}(M,N)$
4. $E, x : W \vdash C : V$

Proving 4 requires the application of rule Exp Match.

Code (line 2):

$$\mathsf{match}\ x\ \mathsf{with\ true} \to A\ \mathsf{else}\ B$$

Environment:

$$E' := E, x : W$$

Rules:

Exp Match

$$\frac{E' \vdash x : \mathit{bool} \qquad \mathsf{inr} : (\mathit{unit}, \mathit{bool}) \qquad E', () : \mathit{unit}, \{\overbrace{\mathsf{inr}\ ()}^{=\mathsf{true}} = x\} \vdash A : V \qquad E', \{\forall y.\ \mathsf{inr}\ y \neq x\} \vdash B : V}{E' \vdash \mathsf{match}\ x\ \mathsf{with\ true} \to A\ \mathsf{else}\ B : V}$$

Proof Obligations:

4. $E, x : W \vdash x : \mathit{bool}$
5. $E, x : W, () : \mathit{unit}, \{\mathsf{inr}() = x\} \vdash A : V$
6. $E, x : W, \{\forall y.\ \mathsf{inr}\ y \neq x\} \vdash B : V$
7. $\mathsf{inr} : (\mathit{unit}, \mathit{bool})$

We now show that the hypotheses of Exp If imply the hypotheses of the derivation for the de-sugared if-statement, i.e., we show that (a)-(d) imply (1)-(6).

(a) $E \vdash M : T$

(b) $E \vdash N : U$

(c) $E, \{M = N\} \vdash A : V$

(d) $E \vdash B : V$

(1) $E \vdash M : T$

(2) $E \vdash N : U$

(3) $x \notin \mathit{fv}(M, N, V)$

(4) $E, x : W \vdash x : \mathit{bool}$

(5) $E, x : W, () : \mathit{unit}, \{\mathsf{inr}() = x\} \vdash A : V$

(6) $E, x : W, \{\forall y.\ \mathsf{inr}\ y \neq x\} \vdash B : V$

(1): $E \vdash M : T$

Immediately by (a)

(2): $E \vdash N : U$

Immediately by (b).

(3): $x \notin \mathit{fv}(M, N, V)$

Follows since x is fresh and occurs nowhere else.

(4): $E, x : W \vdash x : \mathit{bool}$

Follows since $W := \{y : \mathit{bool} \mid \overbrace{x = \mathsf{true} \wedge M = N \vee x = \mathsf{false} \wedge M \neq N}^{=:F}\}$ and $E, x : W \vdash W <: \mathit{bool}$ by Sub Refine Left and Sub Refl.

$$\textsc{Sub Refine Left}\ \dfrac{\textsc{Sub Refl}\ \dfrac{\textsc{Type}\ \dfrac{\begin{array}{c}E, x : W \vdash \diamond \\ \mathit{fnfv}(\mathit{bool}) \subseteq \mathit{dom}(E, x : W)\end{array}}{E, x : W \vdash \mathit{bool}} \quad \mathit{recvar}(E, x : W) \cap \mathit{fnfv}(\mathit{bool}) = \emptyset}{E, x : W \vdash \mathit{bool} <: \mathit{bool}} \quad \dfrac{\begin{array}{c}E, x : W \vdash \diamond \\ \mathit{fnfv}(\{x : \mathit{bool} \mid F\}) \subseteq \mathit{dom}(E)\end{array}}{E, x : W \vdash \{x : \mathit{bool} \mid F\}}\ \textsc{Type}}{E, x : W \vdash \{y : \mathit{bool} \mid F\} <: \mathit{bool}}$$

We are left with the obligations to show that

(i) $\mathit{fnfv}(\mathit{bool}) \subseteq \mathit{dom}(E, x : W)$ (ii) $\mathit{recvar}(E, x : W) \cap \mathit{fnfv}(\mathit{bool}) = \emptyset$
(iii) $E, x : W \vdash \diamond$ (iv) $\mathit{fnfv}(\{x : \mathit{bool} \mid F\}) \subseteq \mathit{dom}(E)$

Proof steps:	Proven statement:
Immediately, since *bool* contains neither free names nor free variables.	(i) and (ii)
The only thing left to consider is that x might cause a double-binding in the typing environment, causing the well-formedness check to fail (more precisely, the condition $\mathit{dom}(\mu) \cap \mathit{dom}(E) = \emptyset$ in rule Env Entry). This is excluded since x is fresh and occurs nowhere else.	(iii)
We first notice that the names and variables occurring free in F, i.e., x and $\mathit{fnfv}(M, N)$, are contained in $\mathit{dom}(E, x : W)$: obviously, $x \in \mathit{dom}(E, x : W)$, the relation $\mathit{fnfv}(M, N) \subseteq \mathit{dom}(E, x : W)$ follows from premises (a) and (b) applied to Lemma A.3 (5).	(iv)

(5): $E, x : W, () : \mathit{unit}, \{\mathsf{inr}() = x\} \vdash A : V$

This case is the most involved one in the proof.

Proof steps:	Proven statement:
This is proven as hypothesis (c) of rule Exp If.	$E, \{M = N\} \vdash A : V$
The shape of the current typing environment is almost the one from (c). We apply Lemma A.4 (Weakening) thrice to add the entries $\{\mathsf{inr}\ () = x\}$, $() : \mathit{unit}$, and $x : W$ in that order to our current typing environment. The order is important to maintain the well-formedness condition required by the weakening lemma. We stress that adding x does not break any well-formedness since x is fresh and does not occur anywhere else.	$\underbrace{E, x : W, () : \mathit{unit}, \{\mathsf{inr}() = x\}}_{E_1}, \{M = N\} \vdash A : V$
We note that $\mathit{forms}(E_1) \vdash M = N$, i.e., the formula contained in $x : W$ combined with the formula $\mathsf{inr}\ () = x$ (i.e., $x = \mathsf{true}$), yields the desired formula $M = N$. We apply Lemma A.5 (Strengthening) which allows us to drop $\{M = N\}$ from our typing environment.	$E_1 \vdash A : V$
This is the required hypothesis (5) of the desugared version of the if statement.	

(6): $E, x : W, \{\forall y.\ \mathsf{inr}\ y \neq x\} \vdash B : V$

Proof steps:	Proven statement:
Proven as premise (d).	$E \vdash B : V$
We apply Lemma A.4 (Weakening) twice on (d), adding the $x : W$ and $\{\forall y.\ \mathsf{inr}\ y \neq x\}$ in that order. Since x is fresh, and the free names and free variables in W are bound in E, the well-formedness of the extended environment is not affected.	$E, x : W, \{\forall y.\ \mathsf{inr}\ y \neq x\} \vdash B : V$
This is the required hypothesis (6).	

We have proven that the premises of rule Exp If imply the premises required to type-check the de-sugared version of an if-statement. This concludes our proof. □

Also, in our code we often have large cascades of if-statements. The following lemma eases the type-checking effort for these constructions.

Lemma A.8 (Type-checking cascaded if-statements). *Let C be a cascade of ℓ if statements that only test equality between two variables and the cascade ends with C', i.e., C is of the form*

$$C := \left\{ \begin{array}{l} \mathsf{if}\ M_1 = N_1\ \mathsf{then} \\ \quad \mathsf{if}\ M_2 = N_2\ \mathsf{then} \\ \qquad \ddots \\ \qquad\quad \mathsf{if}\ M_\ell = N_\ell\ \mathsf{then} \\ \qquad\qquad C' \\ \qquad\quad \mathsf{else} \\ \qquad\qquad \mathsf{fail}\langle V \rangle\ () \\ \qquad \iddots \\ \quad \mathsf{else} \\ \qquad \mathsf{fail}\langle V \rangle\ () \\ \mathsf{else} \\ \quad \mathsf{fail}\langle V \rangle\ () \end{array} \right.$$

If $E, \{M_1 = N_1\}, \ldots, \{M_\ell = N_\ell\} \vdash C' : V$ for some type V and for all i, $E \vdash M_i : T_i$ and $E \vdash N_i : U_i$ for some T_i and U_i, then $E \vdash C : V$.

Proof. The proof is by induction on the number of cascaded if statements. □

RCF only allows pairs rather than n-ary tuple. We use the following abbreviation to encode arbitrary tuples in RCF.

Definition A.10 (General tuples)**.** *We use the convention that pairs are right-associative and we write*

$$x_1 : T_1 * \cdots * x_{n-1} : T_{n-1} * T_n$$

to denote $\sum x_1 : T_1.\ \sum x_2 : T_2.\ \ldots \sum x_{n-1} : T_{n-1}.\ T_n$*, where the scope of* x_i *is* $T_{i+1}, \ldots, T_n$*, and we write*

$$T_1 * \cdots * T_n$$

if the pair is not dependent.

Definition A.11 (General functions)**.** *We use the convention that functions are right-associative and we write*

$$x_1 : T_1 \to \cdots \to x_{n-1} : T_{n-1} \to T_n$$

to denote $\Pi x_1 : T_1.\ \Pi x_2 : T_2.\ \ldots \Pi x_{n-1} : T_{n-1}.\ T_n$*, where the scope of* x_i *is* $T_{i+1}, \ldots, T_n$*, and we write*

$$T_1 \to \cdots \to T_n$$

if the function is not dependent.

The RCF calculus allows for splitting of pairs, however, not for splitting tuples of arbitrary length. We introduce syntactic sugar for such a construct.

Definition A.12 (Splitting tuples)**.** *We use the following syntactic sugar:*

$$\mathsf{let}\ (x_1, \ldots, x_n) = x\ \mathsf{in}\ A$$

as syntactic sugar for

$$\begin{array}{l} \mathsf{let}\ (x_1, x^1) = x\ \mathsf{in} \\ \mathsf{let}\ (x_2, x^2) = x^1\ \mathsf{in} \\ \quad\vdots \\ \mathsf{let}\ (x_{n-1}, x_n) = x^{n-1}\ \mathsf{in}\ A \end{array}$$

Additionally, we add the following (derived) rule to the type system:

$$\textsc{Exp Split}^n \quad \frac{\begin{array}{c} E, x_1 : T_1, \ldots, x_n : T_n, \{(x_1, \ldots, x_n) = M\} \vdash A : V \\ E \vdash M : (x_1 : T_1 * \cdots * x_{n-1} : T_{n-1} * T_n) \qquad \{x_1, \ldots, x_n\} \cap fv(V) = \emptyset \end{array}}{E \vdash \mathsf{let}\ (x_1, \ldots, x_n) = M\ \mathsf{in}\ A : V}$$

Lemma A.9. Exp Split^n *is derivable in F7.*

Proof. We show that the hypotheses of Exp Split^n imply the hypotheses of the de-sugared code. We start by type-checking the de-sugared version of the code under the typing environment E.

$$
\begin{array}{l}
\mathsf{let}\ (x_1, x^1) = M;\\
\mathsf{let}\ (x_2, x^2) = x^1;\\
\quad\vdots\\
\mathsf{let}\ (x_{n-1}, x_n) = x^{n-2};\ A
\end{array}
$$

where $x^1, \dots, x^{n-2}$ are fresh variables that occur nowhere else.

We denote the hypothesis of rule Exp Split^n as follows:

(a) $E \vdash M : U_1$

(b) $E, x_1 : T_1, \dots, x_n : T_n, \{(x_1, \dots, x_n) = M\} \vdash A : V$

(c) $\{x_1, \dots, x_n\} \cap \mathit{fv}(V) = \emptyset$

where we let $U_i := x_i : T_i * \cdots * x_{n-1} : T_{n-1} * T_n$. We type-check the de-sugared code as illustrated below

$$
\textsc{Exp Split}\ \dfrac{
\textsc{Exp Split}\ \dfrac{
\textsc{Exp Split}\ \dfrac{
\dfrac{
\textsc{Exp Split}\ \dfrac{
\begin{array}{c}
\begin{array}{r} E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = M\}, \dots, \\ x^{n-2} : U_{n-1}, x_{n-1} : T_{n-1}, x_n : T_n, \{(x_{n-1}, x_n) = x^{n-2}\} \end{array} \vdash A : V \\
\begin{array}{r} E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = M\}, \dots, \\ x^{n-2} : U_{n-1}, x_{n-1} : T_{n-1}, x_n : T_n, \{(x_{n-1}, x_n) = x^{n-2}\} \end{array} \vdash x^{n-2} : (x_{n-1} : T_{n-1} * T_n) \\
\{x_{n-1}, x_n\} \cap \mathit{fv}(V) = \emptyset
\end{array}
}{\vdots}
}{
\begin{array}{r} E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = M\}, \\ x_2 : T_2, x^2 : U_2, \{(x_2, x^2) = x^1\} \end{array} \vdash \begin{array}{l} \mathsf{let}\ (x_3, x^3) = x^2; \dots \\ \mathsf{let}\ (x_{n-1}, x_n) = x^{n-2};\ A : V \end{array}
}
\quad \{x_2, x^2\} \cap \mathit{fv}(V) = \emptyset \quad E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = M\} \vdash x^1 : (x_2 : T_2, U_3)
}{
E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = M\} \vdash \begin{array}{l} \mathsf{let}\ (x_2, x^2) = x^1;\ \dots \\ \mathsf{let}\ (x_{n-1}, x_n) = x^{n-2};\ A : V \end{array}
}
\quad E \vdash x : (x_1 : T_1 * U_2) \quad \{x_1, x^1\} \cap \mathit{fv}(V) = \emptyset
}{
E \vdash \mathsf{let}\ (x_1, x^1) = M;\ \mathsf{let}\ (x_2, x^2) = x^1;\ \dots\ \mathsf{let}\ (x_{n-1}, x_n) = x^{n-2};\ A : V
}
$$

This result is easily obtained by induction on the arity of the tuple.

(1) $E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = M\}, \dots, x^{n-2} : U_{n-1}, x_{n-1} : T_{n-1}, x_n : T_n, \{(x_{n-1}, x_n) = x^{n-2}\} \vdash A : V$

(2) $E \vdash x : (x_1 : T_1 * U_2)$

(3) $E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = M\} \vdash x^1 : (x_2 : T_2 * U_3)$

(4) $\forall 3 < i < n.\ E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = M\}, \dots, x^{i-2} : U_{i-1}, \{(x_{i-2}, x^{i-2}) = x^{i-3}\} \vdash x^{i-2} : (x_{i-1} : T_{i-1} * U_i)$

(5) $E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = M\}, \dots, x^{n-2} : U_{n-1}, \{(x_{n-2}, x^{n-2}) = x^{n-3}\} \vdash x^{n-2} : (x_{n-1} : T_{n-1} * T_n)$

(6) $\forall 1 \leq i \leq n-2.\ \{x_i, x^i\} \cap fv(V) = \emptyset$

(7) $\{x_{n-1}, x_n\} \cap fv(V) = \emptyset$

We prove that the hypotheses (a)-(c) of rule Exp Split^n are strong enough to entail the obligations collected above.

(1): $\begin{array}{l} E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = M\}, \ldots, \\ \qquad x^{n-2} : U_{n-1}, x_{n-1} : T_{n-1}, x_n : T_n, \{(x_{n-1}, x_n) = x^{n-2}\} \vdash A : V \end{array}$

This one is the most involved one in the proof.

Proof steps:	Proven statement:
This is proven as hypothesis (b) of Exp Split^n.	$E, x_1 : T_1, \ldots, x_n : T_n, \{(x_1, \ldots, x_n) = M\} \vdash A : V$
We apply Lemma A.4 (Weakening) $2n-3$ times ($n-1$ times for entries of the form $\{(x_i, x^i) = x^{i-1}\}$ and $n-2$ times for variables of the form x^i) to introduce $x^1 : U_2, \ldots, x^{n-2} : U_{n-1}$ and the entries $\{(x_1, x^1) = M\}, \ldots, \{(x_{n-1}, x_n) = x^{n-2}\}$.	$\underbrace{\begin{array}{r} E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = M\}, \ldots, \\ x_{n-2} : T_{n-2}, x^{n-2} : U_{n-1}, \\ \{(x_{n-2}, x^{n-2}) = x^{n-3}\}, \\ x_{n-1} : T_{n-1}, x_n : T_n, \\ \{(x_{n-1}, x_n) = x^{n-2}\} \\ \{(x_1, \ldots, x_n) = M\} \end{array}}_{=:E_1} \vdash A : V$
We can see that $forms(E_1) \vdash (x_1, \ldots, x_n) = M$ (viewing the n-tuple as nested pairs). This is the case because we can use the substitution property of the logic and derive $(x_1, \ldots, x_n) = M$ from $\{(x_1, x^1) = M\}, \{(x_2, x^2) = x^1\}, \ldots, \{(x_{n-1}, x_n) = x^{n-2}\}$. Hence, by using strengthening (Lemma A.5), we can drop the last entry in the environment.	$\begin{array}{r} E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = M\}, \ldots, \\ x_{n-2} : T_{n-2}, x^{n-2} : U_{n-1}, \\ \{(x_{n-2}, x^{n-2}) = x^{n-3}\}, \\ x_{n-1} : T_{n-1}, x_n : T_n, \\ \{(x_{n-1}, x_n) = x^{n-2}\} \end{array} \vdash A : V$
This is the required premise (1) of the de-sugared version.	

(2): $E \vdash M : x_1 : T_1 * U_2$

Follows immediately from (a) ($E \vdash M : U_1$) since U_i is defined as $U_i := x_i : T_i * \cdots * x_{n-1} : T_{n-1} * T_n$.

(3): $E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = M\} \vdash x^1 : (x_2 : T_2 * U_3)$

(4): $\forall 3 < i < n.\ E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = M\}, \dots, x^{i-2} : U_{i-1}, \{(x_{i-2}, x^{i-2}) = x^{i-3}\}$
$\vdash x^{i-2} : (x_{i-1} : T_{i-1} * U_i)$

(5): $E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = M\}, \dots, x^{n-2} : U_{n-1}, \{(x_{n-2}, x^{n-2}) = x^{n-3}\}$
$\vdash x^{n-2} : (x_{n-1} : T_{n-1} * T_n)$

For these cases, we will use rule Val Var. This rule has two premises, namely that the current typing environment is well-formed and that there is an entry in the environment of the current variable with the appropriate type.

Since hypothesis (b) holds, we can apply Corollary A.1 to derive that the extended typing environment E used in (b) is well-formed. All the typing environments used in (4) and the environment used in (3) can be derived from E by dropping entries "from the right". In particular, all such environments are well-formed by Lemma A.2.

All the environments contain the respective variables since $x^i : U_{i+1}$ and $U_j := x_j : T_j * \cdots * x_{n-1} : T_{n-1} : T_n$. We can apply Val Var and obtain the desired result.

(6): $\forall 1 \leq i \leq n-2.\ \{x_i, x^i\} \cap fv(V) = \emptyset$

(7): $\{x_{n-1}, x_n\} \cap fv(V) = \emptyset \cap fv(V) = \emptyset$

Follows from (c) ($\{x_1, \dots, x_n\} \cap fv(V) = \emptyset$) and the fact that $x^1, \dots, x^{n-2}$ are fresh and occur nowhere else.

□

The RCF calculus does not allow to match type-constructors and, at the same time split the arguments if it is a tuple. We use the following syntactic sugar.

Definition A.13 (Matching of constructor with tuples as arguments). *We use the following syntactic sugar:*

$$\begin{array}{l}
\mathsf{match}\ M\ \mathsf{with} \\
\quad \mid h_1\ (x^1_1, \dots, x^1_{m_1})\ \rightarrow A_1 \\
\quad \mid h_2\ (x^2_1, \dots, x^2_{m_2})\ \rightarrow A_2 \\
\quad\quad \vdots \\
\quad \mid h_n\ (x^n_1, \dots, x^n_{m_n})\ \rightarrow A_n \\
\quad \mid \_ \rightarrow A_{\mathsf{fail}}
\end{array}$$

as abbreviation for

$$
\begin{array}{l}
\mathsf{match}\ M\ \mathsf{with}\ h_1\ x^1 \rightarrow \\
\quad \mathsf{let}\ (x^1_1,\ldots,x^1_{m_1}) = x^1\ \mathsf{in}\ A_1 \\
\mathsf{else\ match}\ M\ \mathsf{with}\ h_2\ x^2 \rightarrow \\
\quad \mathsf{let}\ (x^2_1,\ldots,x^2_{m_2}) = x^2\ \mathsf{in}\ A_2 \\
\ \vdots \\
\mathsf{else\ match}\ M\ \mathsf{with}\ h_n\ x^n \rightarrow \\
\quad \mathsf{let}\ (x^n_1,\ldots,x^n_{m_n}) = x^n\ \mathsf{in}\ A_n \\
\mathsf{else}\ A_{\mathsf{fail}};
\end{array}
$$

where the variables x^i are fresh variables that occur nowhere else.

Additionally, we add the following (derived) rule to the type system:

Exp Match-Split$^n_{(m_1,\ldots,m_n)}$

$$
\frac{\begin{array}{c}
E \vdash M : T \\
\forall 0 < i \leq n.\ h_i : (H_i, T) \qquad \forall 0 < i \leq n.\ H_i = x^i_1 : T^i_1 * \cdots * x^i_{m_i-1} : T^i_{m_i-1} * T^i_{m_i} \\
\forall 0 < i \leq n.\ E, x^i_1 : T^i_1, \ldots, x^i_{m_i} : T^i_{m_i}, \{M = h_i(x^i_1,\ldots,x^i_{m_i})\} \vdash A_i : U \qquad E \vdash A_{\mathsf{fail}} : U \\
\{x^i_j \mid 1 \leq i \leq n, 1 \leq j \leq m_i\} \cap fv(U) = \emptyset
\end{array}}{E \vdash \mathsf{match}\ M\ \mathsf{with}\ \mid h_1\ (x^1_1,\ldots,x^1_{m_1})\ \rightarrow A_1 \ldots \mid h_n\ (x^n_1,\ldots,x^n_{m_n})\ \rightarrow A_n \mid \_ \rightarrow A_{\mathsf{fail}} : U}
$$

For the sake of readbility, we omit the parameters n and $(m_1,\ldots,m_n)$ if they are obvious from the context.

Lemma A.10. Exp Match-Split$^n_{(m_1,\ldots,m_n)}$ *is derivable in F7.*

Proof. We show that the hypothesis of the individual cases are strong enough to imply the premises required to type-check the de-sugared version of the match construct. In our proof, we will first show that the individual match statements and tuple splits of the de-sugared version can be type-checked. Then we will give an inductive argument that shows that the logical formulas we accumulate in the else branches can be reconstructed using weakening (Lemma A.4).

We denote the hypothesis of Exp Match-Split$^n_{(m_1,\ldots,m_n)}$ as follows:

(a) $E \vdash M : T$

(b) $\forall 0 < i \leq n.\ h_i : (H_i, T)$

(c) $\forall 0 < i \leq n.\ H_i = x^i_1 : T^i_1, \ldots, x^i_{m_i-1} : T^i_{m_i-1} * T^i_{m_i}$

(d) $\forall 0 < i \leq n.\ E, x^i_1 : T^i_1, \ldots, x^i_{m_i} : T^i_{m_i}, \{M = h_i(x^i_1,\ldots,x^i_{m_i})\} \vdash A_i : T_i$

(e) $\{x^i_j \mid 1 \leq i \leq n, 1 \leq j \leq m_i\} \cap fv(U) = \emptyset$

We type-check the i-th the de-sugared code to gather the obligations which we need to prove. We do not consider the else case here; we will reason about it in our inductive argument.

$$\dfrac{\begin{array}{c} E, x^i : H_i, \{h\ x^i = M\} \vdash x^i : (x_1^i : T_1^i * \cdots * T_{m_i}^i) \\ \left.\begin{array}{r} E, x^i : H_i, \{h\ x^i = M\}, x_1^i : T_1^i, \ldots, x_{m_i}^i : T_{m_i}^i, \\ \{(x_1^i, \ldots, x_{m_i}^i) = x^i\} \end{array}\right\vdash A : T_i \\ \{x_1^i, \ldots, x_{m^i}^i\} \cap \mathit{fv}(T_i) = \emptyset \end{array}}{E, x^i : H_i, \{h\ x^i = M\} \vdash \mathsf{let}\ (x_1^i, \ldots, x_{m_i}^i) = x^i\ \mathsf{in}\ A_i : T_i}\ \textsc{Exp Split}^{m_i}$$

$$\textsc{Exp Match}\ \dfrac{E \vdash M : T \qquad h_i : (H_i, T)}{E \vdash \mathsf{match}\ M\ \mathsf{with}\ h_i\ x^i \to \mathsf{let}\ (x_1^i, \ldots, x_{m_i}^i) = x^i\ \mathsf{in}\ A_i}$$

(1) $E \vdash M : T$

(2) $\forall i.\ h_i : (H_i, T)$

(3) $\forall i.\ E, x^i : H_i, \{h\ x^i = M\} \vdash x^i : (x_1^i : T_1^i * \cdots * T_{m_i}^i)$

(4) $\forall i.\ E, x^i : H_i, \{h\ x^i = M\}, x_1^i : T_1^i, \ldots, x_{m_i}^i : T_{m_i}^i, \{(x_1^i, \ldots, x_{m_i}^i) = x^i\} \vdash A : T_i$

(5) $\forall i.\ \{x_1^i, \ldots, x_{m^i}^i\} \cap \mathit{fv}(T_i) = \emptyset$

We prove that the hypotheses (a)-(e) of rule $\textsc{Exp Match-Split}^n_{(m_1,\ldots,m_n)}$ are strong enough to entail the obligations collected above.

(1): $E \vdash M : T$

Immediate by (a) ($E \vdash M : T$).

(2): $\forall i.\ h_i : (H_i, T)$

Immediate by (c) ($\forall 1 \leq i \leq n.\ h_i : (H_i, T)$).

(3): $\forall i.\ E, x^i : H_i, \{h\ x^i = M\} \vdash x^i : (x_1^i : T_1^i * \cdots * T_{m_i}^i)$

Since H_i is exactly defined as $H_i = x_1^i : T_1^i, \ldots, x_{m_i-1}^i : T_{m_i-1}^i * T_{m_i}^i$ by (c) ($\forall 0 < i \leq n.\ H_i = x_1^i : T_1^i, \ldots, x_{m_i-1}^i : T_{m_i-1}^i * T_{m_i}^i$), where all newly-introduced variables are fresh, the typing environment is well-formed. Furthermore, the typing environment contains all $x^i : H_i$. The statement follows by applying rule Val Var.

(4): $\forall i.\ E, x^i : H_i, \{h\ x^i = M\}, x_1^i : T_1^i, \ldots, x_{m_i}^i : T_{m_i}^i, \{(x_1^i, \ldots, x_{m_i}^i) = x^i\} \vdash A : T_i$

This is the most involved proof step.

Proof steps:	Proven statement:
Proven as hypothesis (d) of rule EXP MATCH-SPLIT$^n_{(m_1,\ldots,m_n)}$.	$E, x_1^i : T_1^i, \ldots, x_{m_i}^i : T_{m_i}^i, \{M = h_i(x_1^i, \ldots, x_{m_i}^i)\} \vdash A_i : U$
We apply weakening thrice to add the two entries $x^i : H_i$, $\{M = h_i\ x^i\}$, and $\{(x_1^i, \ldots, x_{m_i}^i) = x^i\}$ to the typing environment.	$E, x^i : H_i, \{M = h_i\ x^i\}, x_1^i : T_1^i, \ldots, x_{m_i}^i : T_{m_i}^i, \{(x_1^i, \ldots, x_{m_i}^i) = x^i\}, \{M = h_i(x_1^i, \ldots, x_{m_i}^i)\} \vdash A_i : T_i$
The formulas in the environment, in particular $\{M = h_i\ x^i\}$ and $\{(x_1^i, \ldots, x_{m_i}^i) = x^i\}$ logically entail the formula $M = h_i(x_1^i, \ldots, x_{m_i}^i)$. We use strengthening (Lemma A.5) to drop the last environment entry $\{M = h_i(x_1^i, \ldots, x_{m_i}^i)\}$.	$E, x^i : H_i, \{M = h_i\ x^i\}, x_1^i : T_1^i, \ldots, x_{m_i}^i : T_{m_i}^i, \{(x_1^i, \ldots, x_{m_i}^i) = x^i\} \vdash A_i : T_i$
This is the required hypothesis (4).	

(5): $\forall i.\ \{x_1^i, \ldots, x_{m^i}^i\} \cap fv(T_i) = \emptyset$

Immediately, since all variables x_j^i are fresh and occur nowhere else.

In the above proof, we have lazily neglected to type-check the else branch of the match statements. We now argue why the else branches will also type-check.

Using a simple inductive argument, we see that the i-th else branch will type-check with the typing environment $E, \{\forall x.\ h_1\ x \neq M\}, \ldots, \{\forall x.\ h_i\ x \neq M\}$. Above, we have proven that every branch of the n branches will type-check under the typing environment E. Therefore, we can apply weakening i times and add the entries $\{\forall x.\ h_1\ x \neq M\}, \ldots, \{\forall x.\ h_i\ x \neq M\}$ to accommodate the typing environment of the i-th else branch. □

Lemma A.11 (Type *verkey* is public)**.** *Let $E \vdash \diamond$. Then $E \vdash verkey :: \mathsf{pub}$.*

Proof. First, we review the definition of type *verkey*:

$$\mathcal{T}_y^o := +_{k=1}^n P_k^S(x_1 : T_1^k * \cdots * x_{\ell_{k-1}} : T_{\ell_{k-1}}^k * \{x_{\ell_k} : T_{\ell_k}^k \mid y \ \mathsf{says}\ P_k(x_1, \ldots, x_{\ell_k})\})$$
$$\text{where } T_i^j \in \{bitstring, \alpha\}$$
$$\mathcal{T}_y := y : bitstring * \mathcal{T}_y^o$$
$$verkey := \mu\alpha.\ signature \to \mathcal{T}_y$$

Notice that the variables x_j are fresh and occur nowhere else.

1. While proving that $E \vdash \mathit{verkey} :: \mathit{pub}$, we naturally apply

$$\dfrac{E, \alpha :: \nu \vdash T :: \nu}{E \vdash (\mu\alpha.\, T) :: \nu}\ \textsc{Kind Rec}.$$

   This rule assumes that $\alpha :: \mathsf{pub}$ (where α corresponds to *verkey*) and proceeds to the inner type.

2. We are left to show that $E, \alpha :: \mathsf{pub} \vdash (\mathit{signature} \to \mathcal{T}_y) :: \mathsf{pub}$. We apply

$$\dfrac{E \vdash T :: \overline{\nu} \qquad E, x : T \vdash (\Pi x : T.\, U) :: \nu}{E \vdash (\Pi x : T.\, U) :: \nu}\ \textsc{Kind Fun}.$$

   T corresponds to *bitstring* and U corresponds to $\mathcal{T}_y$. Since $\mathit{bitstring} := \mathit{unit}$, we get that $\mathit{bitstring} :: \mathsf{tnt}$ by rule

$$\dfrac{E \vdash \diamond}{E \vdash \mathit{unit} :: \nu}\ \textsc{Kind Unit}$$

   immediately.

3. $\mathcal{T}_y$ is a sum type. We apply

$$\dfrac{E \vdash T :: \nu \qquad E \vdash U :: \nu}{E \vdash (T + U) :: \nu}\ \textsc{Kind Sum}.$$

   Without loss of generality, we only consider the k-th case; all other cases are analogous.

4. To show that the individual sub-cases of $\mathcal{T}_y$ are of kind pub, we apply

$$\dfrac{E \vdash T :: \nu \qquad E, x : T \vdash U :: \nu}{E \vdash (\Sigma x : T.\, U) :: \nu}\ \textsc{Kind Pair}$$

   recursively (the general tuple $(M_1, \ldots, M_n)$ is defined as $(M_1, (\cdots (M_{n-1}, M_n)) \cdots)$, see Definition A.11). We denote the extended typing environment after the i-th step as $E_i := E_{i-1}, x_i : T_i^k$ where $E_0 := E$. Since the variables x_i occur only in this type, $E_i \vdash \diamond$ for all i.

5. The individual tuple elements are either of type *bitstring* or of type α. The type α is in the environment as kind pub (see step 1)). As such, rule Kind Var yields that $\alpha :: \mathsf{pub}$. The other type is defined to be *unit* and is also pub.

$$\dfrac{E \vdash \diamond \qquad (\alpha :: \nu) \in E}{E \vdash \alpha :: \nu}\ \textsc{Kind Var} \qquad\qquad \dfrac{E \vdash \diamond}{E \vdash \mathit{unit} :: \nu}\ \textsc{Kind Unit}$$

6. The very last element in the tuple is refined. Rule

$$\frac{E \vdash \{x : T \mid \mathcal{F}\} \qquad E \vdash T :: \mathsf{pub}}{E \vdash \{x : T \mid \mathcal{F}\} :: \mathsf{pub}} \;\textsc{Kind Refine Public}$$

forces us to prove statements. We already argued in that step 5) that $T^k_{\ell_k} :: \mathsf{pub}$.

7. For showing that $E_{\ell_{k-1}} \vdash \overbrace{\{x_{\ell_k} : T^k_{\ell_k} \mid y \;\mathsf{says}\; P_k(x_1, \ldots, x_k)\}}^{=:U}$, we apply rule

$$\frac{E \vdash \diamond \qquad \mathit{fnfv}(T) \subseteq \mathit{dom}(E)}{E \vdash T} \;\textsc{Type}.$$

The type is closed since all the values occurring in the refinement are bound above. More precisely, all variables x_i are contained in $E_{\ell_{k-1}}$ by the definition of E_i; variable x_{ℓ_k} is bound due to the refinement type. Consequently, $\mathit{fnfv}(T) \subseteq \mathit{dom}(E_{\ell_{i-1}})$. As argued in step 4), $E_{\ell_{k-1}} \vdash \diamond$.

This concludes the proof.

□

Corollary A.2. *Let $E \vdash \diamond$ and let x be such that $E \vdash x : \mathit{verkey}$. Then $E \vdash x : \mathit{unit}$.*

Proof. Follows immediately from Lemma A.11 and rules Sub Public Tainted, Kind Unit, and Exp Subsum.

$$\frac{E \vdash T :: \mathsf{pub} \qquad E \vdash U :: \mathsf{tnt}}{E \vdash T <: U} \;\textsc{Sub Public Tainted} \qquad \frac{E \vdash \diamond}{E \vdash \mathit{unit} :: \nu} \;\textsc{Kind Unit} \qquad \frac{E \vdash A : T \qquad E \vdash T <: T'}{E \vdash A : T'} \;\textsc{Exp Subsum}$$

□

Proposition A.2 (Kinding *option* and *RevHid*)**.** *Let $E \vdash \diamond$. Then $E \vdash T\ \mathit{option} :: \nu$ and $E \vdash T\ \mathit{RevHid} :: \nu$ if and only if $E \vdash T :: \nu$.*

Proof. We recall that $\alpha\ \mathit{option} := \mathsf{Some\ of}\ \alpha \mid \mathsf{None}$ and $\alpha\ \mathit{RevHid} := \mathsf{Revealed\ of}\ \alpha \mid \mathsf{Hidden\ of}\ \mathit{bitstring}$. The claim then follows directly from rules

$$\frac{E \vdash T :: \nu \qquad E \vdash U :: \nu}{E \vdash (T + U) :: \nu} \;\textsc{Kind Sum} \qquad \frac{E \vdash \diamond}{E \vdash \mathit{unit} :: \nu} \;\textsc{Kind Unit},$$

since T always corresponds to α and $U = \mathit{unit}$ in all cases (recall that $\mathsf{None} := \mathsf{None\ of}\ \mathit{unit}$).

□

Lemma A.12 ($\mathit{unit} <: \{\mathit{predicate}^F, \mathit{predicate}^P, \mathit{formula}, \mathit{statement}, \mathit{proof}\} <: \mathit{unit}$)**.** *Let $E \vdash \diamond$. Then $E \vdash T <: \mathit{unit}$ and $E \vdash \mathit{unit} <: T$ for $T \in \{\mathit{predicate}^F, \mathit{predicate}^P, \mathit{formula}, \mathit{statement}, \mathit{proof}\}$.*

Proof. The types are defined in Table A.2. We first show that the types $\mathit{predicate}^F$, $\mathit{predicate}^P$, *formula*, *statement*, and *proof* are public and tainted. We notice that all types that occur within the respective types are of kind ν by Proposition A.2 and Kind Unit. The intermediate claim follows by applying

$$\frac{E \vdash T :: \nu \qquad E \vdash U :: \nu}{E \vdash (T + U) :: \nu}\ \textsc{Kind Sum} \qquad \frac{E \vdash T :: \nu \qquad E, x : T \vdash U :: \nu}{E \vdash (\Sigma x : T.\, U) :: \nu}\ \textsc{Kind Pair} \qquad \frac{E \vdash \diamond}{E \vdash \mathit{unit} :: \nu}\ \textsc{Kind Unit}$$

similarly to the proof of Lemma A.11 and of Proposition A.2. The claim itself follows by rule Sub Public Tainted

$$\frac{E \vdash T :: \mathsf{pub} \qquad E \vdash U :: \mathsf{tnt}}{E \vdash T <: U}\ \textsc{Sub Public Tainted}$$

and Kind Unit. □

The last proposition captures that the type *unit* and any type that, as base types, only contains *unit* are subtypes of one another.

Proposition A.3. *Let $E \vdash \diamond$ and let T and U be types such that $E \vdash T <:> \mathit{unit}$ and $E \vdash U <:> \mathit{unit}$. Then $E \vdash T \to U <:> \mathit{unit}$, $E \vdash T * U <:> \mathit{unit}$, $E \vdash T + U <:> \mathit{unit}$, $E \vdash \mu\alpha.\, T <:> \mathit{unit}$.*

Corollary A.3. *Let $E \vdash \diamond$. Then for all types T constructed of only unit, we have that $E \vdash T <:> \mathit{unit}$.*

The above lemmas and propositions are the building block to showing that the auxiliary functions are well-typed.

A.2.1. Type-Checking Auxiliary Functions

This subsection is devoted to showing that all the auxiliary functions depicted in Table A.7 are well-typed. We will assume that the library functions listed in Table A.6 are implemented and contained in the typing environment E. More precisely, we assume that for all functions $f : T$ in that table, $E \vdash f : T$. Furthermore, we will use $\mathcal{R}$ to denote the respective remainder of the function body that is to be type-checked.

Convention. Here and through the remainder of this work, we implicitly use Val Fun to type-check functions.

$$\textsc{Val Fun}\quad \frac{E, x : T \vdash A : U}{E \vdash \text{fun } x \to A : (\Pi x : T.\ U)}$$

More precisely, when type-checking a function $f : (x_1 : T_1) \to \cdots \to (x_n : T_n)$ under typing environment E, we implicitly apply Val Fun n times and type-check the body of f under the typing environment $E, x_1 : T_1, \ldots, x_n : T_n$. Additionally, we will silently fold and unfold iso-recursive values and we will not argue why the current typing environment is well-formed, since all variables occurring in the functions are fresh (we will, however, highlight why variables occurring free in a type are bound). Furthermore, we intentionally name variables that occur free in a type the same way as they occur in the typing environment. Although this formally prevents us from type-checking the code (mostly because of conditions of the form $x \notin fv(T)$), it increases the readability and can easily be fixed by using consistent α-renaming. Finally, we will omit the else-branches and the | _ $\implies$ catch-all cases since they always contain either a fail (which always has the right type) or, in case of the verification function, false as return value, which makes the logical refinement vacuously true.

Since we have many auxiliary functions and most of the proofs are straightforward or follow immediately from the opponent typability lemma (Lemma A.6), we keep the proofs high-level, only highlighting the crucial points. We use the order given by Table A.7.

getOperation^r : $(op : string) \to ((x : bitstring) \to (y : bitstring) \to bool)$,

getOperation^e : $(op : string) \to ((x : bitstring) \to (y : bitstring) \to bitstring)$, and

rand : $unit \ \to \ random$:

The types of these functions are equivalent to *unit* and they only contains free variables of that type (e.g., $\leq$ can be seen as a variable of type *unit*, the type of mkUn is equivalent to *unit*). Consequently, the function type-checks by Lemma A.6.

$\mathsf{sign} : (\mathit{sk} : \mathit{sigkey}) \;\to\; (m : \mathcal{U}^o_{\mathit{sk}}) \;\to\; \mathit{signature}$:

First, we quickly recall the signing and verification key types:

$$\begin{aligned}
\mathcal{T}^o_y &:= +^n_{k=1}\, P^S_k(x_1 : T^k_1 * \cdots * x_{\ell_{k-1}} : T^k_{\ell_{k-1}} * \{x_{\ell_k} : T^k_{\ell_k} \mid y \;\mathsf{says}\; P_k(x_1, \ldots, x_{\ell_k})\}) \\
\mathcal{T}_y &:= y : \mathit{bitstring} * \mathcal{T}^o_y \\
\mathcal{U}^o_{\mathit{sk}} &:= +^n_{k=1}\, P^S_k(x_1 : T^k_1 * \cdots * x_{\ell_{k-1}} : T^k_{\ell_{k-1}} * \\
&\qquad \{x_{\ell_k} : T^k_{\ell_k} \mid \exists z, y.\ \mathit{sk} = (z, y) \wedge y \;\mathsf{says}\; P_k(x_1, \ldots, x_{\ell_k})\}) \\
&\quad \text{where } T^j_i \in \{\mathit{bitstring}, \alpha\}
\end{aligned}$$

$$\begin{aligned}
\mathit{verkey} &:= \mu\alpha.\ \mathit{signature} \to \mathcal{T}_y \\
\mathit{sigkey} &:= (\mu\alpha.\ \mathcal{T}_y \to \mathit{signature}) * \mathit{verkey}
\end{aligned}$$

In particular, the only difference between $\mathcal{U}^o_{\mathit{sk}}$ and $\mathcal{T}^o_y$ is the logical refinement $\exists z, y.\ \mathit{sk} = (z, y)$. We use this fact to derive that a message $m : \mathcal{U}^o_{\mathit{sk}}$ is also a message $m : \mathcal{T}^o_y$. Formally, we apply Sub Refine.

$$\textsc{Sub Refine}\quad \frac{E \vdash T <: T' \qquad E, x : \{x : T \mid \mathcal{F}\} \vdash \mathcal{F}'}{E \vdash \{x : T \mid \mathcal{F}\} <: \{x : T' \mid \mathcal{F}'\}}$$

In the following, let $E' := E, \mathit{sk} : \mathit{sigkey}, m : \mathcal{U}^o_{\mathit{sk}}$.

Code (line 1):

$$\mathsf{let}\ (x, y) = \mathit{sk};$$

Environment:

$$E'$$

Rules:

$$\textsc{Exp Split}\quad \frac{\begin{array}{c} E \vdash M : (\Sigma x : T.\ U) \\ E, x : T, y : U, \_ : \{(x, y) = M\} \vdash A : V \qquad \{x, y\} \cap \mathit{fv}(V) = \emptyset \end{array}}{E \vdash \mathsf{let}\ (x, y) = M;\ A : V}$$

Proof Obligations:

1. $E' \vdash \mathit{sk} : \mathit{sigkey}$
2. $\{x, y\} \cap \mathit{fv}(V) = \emptyset$
3. $\overbrace{E', x : (\mu\alpha.\ \mathcal{T}_y \to \mathit{signature}), y : \mathit{verkey}, \{(x, y) = \mathit{sk}\}}^{=:E'_1} \vdash \mathcal{R} : \mathit{signature}$

The first and the second obligation are immediate by Type and the fact that *signature* does not contain free names, respectively.

Code (line 2):

$$x\ (y, m)$$

Environment:

$$E'_1$$

Rules:

$$\frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\}}\ \textsc{Exp Appl}$$

Proof Obligations:

1. $E'_1 \vdash x : (\mu\alpha.\, \mathcal{T}_y \to \mathit{signature})$
2. $E'_1 \vdash (y, m) : \mathcal{T}_y$

The first obligation is trivially fulfilled since $x : (\mu\alpha.\, \mathcal{T}_y \to \mathit{signature}) \in E'_1$ by

$$\frac{E \vdash \diamond \qquad (x : T) \in E}{E \vdash x : T}\ \textsc{Val Var}.$$

For the second obligation, we use Corollary A.2 to derive that $E'_1 \vdash y : \mathit{bitstring}$. Rule Val Pair together with the above observation that $m : \mathcal{T}^o_y$ yields that $(y, m) : \mathcal{T}_y$.

The resulting type of the application is *bitstring*, which concludes the proof.

$\mathsf{check}_{sig} : (y : \mathit{verkey}) \ \to\ (\mathit{sig} : \mathit{signature}) \ \to\ \mathcal{T}_y\{\mathit{verkey}/\alpha\}$:
We let $E' := E, y : \mathit{verkey}, \mathit{sig} : \mathit{signature}$.

Code (line 1):

$$\text{let } x' = y\ \mathit{sig}$$

Environment:

$$E'$$

Rules:

$$\frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\}}\ \textsc{Exp Appl}$$

$$\frac{E \vdash A : T \qquad E, x : T \vdash B : U \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U}\ \textsc{Exp Let}$$

Proof Obligations:

1. $E' \vdash y : \mathit{verkey}$
2. $E' \vdash \mathit{sig} : \mathit{bitstring}$
3. ($E' \vdash (y\ \mathit{sig}) : \mathcal{T}_y$, which is proven by the two obligations above as Exp Appl)
4. $x' \notin \mathit{fv}(\mathcal{T}_y)$
5. $E', x' : \mathcal{T}_y \vdash \mathcal{R} : \mathcal{T}_y$

The first two proof obligations are trivially fulfilled. Since $\mathit{fv}(\mathcal{T}_y) = \{y\}$, the requirement $x' \notin \mathit{fv}(\mathcal{T}_y)$ follows.

Code (line 2):

$$\text{let } (x, m) = x';$$

Environment:

$$E'_1 := E', x' : \mathcal{T}_y$$

Rules:

Exp Split

$$\frac{E \vdash M : (\Sigma x : T.\ U) \qquad E, x : T, y : U, \_ : \{(x, y) = M\} \vdash A : V \qquad \{x, y\} \cap \mathit{fv}(V) = \emptyset}{E \vdash \mathsf{let}\ (x, y) = M;\ A : V}$$

Proof Obligations:

1. $E'_1 \vdash x' : \mathcal{T}_y$
2. $\{x, m\} \cap \{\mathcal{T}_y\} = \emptyset$
3. $E'_1, x : \mathit{bitstring}, m : \mathcal{T}^o_y, \{(x, m) = x'\} \vdash \mathcal{R} : \mathcal{T}_y$

The first obligation is immediate, the second follows from inspection. At this point, we stress that the free variable y in the type of $m : \mathcal{T}^o_y$ is replaced by the variable x, i.e., if we returned m as is, we could not meet the required return type: in the return type $\mathcal{T}^o_y$, the argument variable y takes the place of the free variable y.

Code (line 3):

$$\text{if } x = y \text{ then}$$

Environment:

$$E'_2 := E'_1, x : \mathit{bitstring}, m : \mathcal{T}^o_y, \{(x, m) = x'\}$$

Rules:

Exp If

$$\frac{E \vdash M : T \qquad E \vdash N : U \qquad E, \{M = N\} \vdash A : V \qquad E \vdash B : V}{E \vdash \mathsf{if}\ M = N\ \mathsf{then}\ A\ \mathsf{else}\ B : V}$$

Proof Obligations:

1. $E_2' \vdash x : \mathit{bitstring}$
2. $E_2' \vdash y : \mathit{verkey}$
3. $E_2', \{x = y\} \vdash m : \mathcal{T}_y$

The first two obligations are immediate. For the final requirement, we recall that the free variable y within the refinement of type $m : \mathcal{T}_y^o$ is replaced by x but the argument variable y takes the place of the free variable y in the refinement of the return type $\mathcal{T}_y^o$. We use the equality $x = y$ to substitute x with y in the refinement of type $m : \mathcal{T}_y^o$ which now equals the return type. This concludes the proof.

$\mathsf{storeSK} : \mathit{sigkey} \;\rightarrow\; \mathit{uid}$ and

$\mathsf{restoreSK} : \mathit{uid} \;\rightarrow\; \mathit{sigkey}$:
Immediate by using Exp Appl to the library function mkSeal.

$\mathsf{computeR} : (x : \mathit{bitstring}) \rightarrow (r : \mathit{bitstring}) \rightarrow \mathit{bitstring}$,

$\mathsf{computePsd} : (\mathit{sk} : \mathit{sigkey}) \rightarrow (s : \mathit{string}) \rightarrow \{x : \mathit{pseudo} \mid \exists y, z.\ \mathit{sk} = (y, z) \wedge \mathsf{SSP}(z, s, x)\}$, and

$\mathsf{computeIDR} : \begin{array}{l}(\mathit{vk}_{EA} : \mathit{bitstring}) \rightarrow (\mathit{vk} : \mathit{bitstring}) \rightarrow (r : \mathit{bitstring}) \rightarrow (R : \mathit{bitstring}) \\ \quad \rightarrow (s : \mathit{string})\end{array}$:
The type-checking is straightforward, since the matching logical formula is assumed within the respective function.

$\mathsf{commit} : \mathit{bitstring} * \mathit{random} \rightarrow \mathit{commitment}$ and

$\mathsf{openCommit} : \mathit{commitment} \rightarrow \mathit{bitstring} * \mathit{random}$:
Immediate by using Exp Appl to the library function mkSeal.

$\mathsf{commit}_{\mathit{sk}} : \mathit{sigkey} * \mathit{random} \rightarrow \mathit{commitment}$ and

$\mathsf{openCommit}_{\mathit{sk}} : \mathit{commitment} \rightarrow \mathit{sigkey} * \mathit{random}$:
Immediate by using Exp Appl to the library function mkSeal.

$\mathsf{getSome} : (x : (\mathit{unit} * \mathit{unit})\ \mathit{option}) \rightarrow \{y : \mathit{unit} * \mathit{unit} \mid x = \mathsf{Some}\ y\}$ and

$\mathsf{getRevealed} : (x : \mathit{unit}\ \mathit{RevHid}) \rightarrow \{y : \mathit{unit} \mid x = \mathit{RevHid}\ y\}$:
The type-checking is straightforward. The respective logical refinement originates from the rule Exp Match.

Exp Match

$$\frac{\begin{array}{c} E \vdash M : T \\ h : (H, T) \qquad E, x : H, \_ : \{h\ x = M\} \vdash A : U \qquad E, \_ : \{\forall x.\ h\ x \neq M\} \vdash B : U \end{array}}{E \vdash \mathsf{match}\ M\ \mathsf{with}\ h\ x \rightarrow A\ \mathsf{else}\ B : U}$$

$\mathsf{commitZK} : \mathit{statement} * \mathit{random} \rightarrow \mathit{commitment}$,

$\mathsf{openZK} : \mathit{commitment} \rightarrow \mathit{statement} * \mathit{random}$,

$\mathsf{stripStm} : \mathit{statement} \to \mathit{statement}$,

$\mathsf{checkZK} : \mathit{proof} \to \mathit{bool}$,

$\mathsf{fakestm} : \mathit{formula} \to \mathit{statement}$,

$\mathsf{createZK}^{\mathsf{e}} : \mathit{statement} \to \mathit{random} \to \mathit{proof} * \mathit{zero\text{-}knowledge} * \mathit{statement}$, and

$\mathsf{createZK} : \mathit{statement} \to \mathit{random} \to \mathit{proof} * \mathit{zero\text{-}knowledge} * \mathit{statement}$:
Type-check by opponent typability (Lemma A.6).

$\mathsf{rerand}_{\mathit{stm}} : \mathit{statement} \to \mathit{statement} \to \mathit{statement}$:
Large parts of the code type-check using opponent typability (Lemma A.6). However, the branches for SSPs and escrow identifiers contain the free variables $\mathsf{commit}_{\mathit{sk}}$ and $\mathsf{openCommit}_{\mathit{sk}}$ that are not of type *unit*. The type-checking is nonetheless straightforward since the logical refinements occur solely as part of the type *sigkey* and are not returned or used in any way.

$\mathsf{checkEq}^1 : \alpha\ \mathit{RevHid} \to \mathit{bitstring} \to (\mathit{bitstring} * \mathit{commitment})\ \mathit{list}\ \mathit{ref} \to \mathit{bool}$:
The implementation of $\mathsf{checkEq}^1$ uses the refined list library functions. As a result, the type of these functions is not equivalent to *unit*. The type-checking, however, is straightforward since the logical refinements are not internally used and they are not returned as a refinement to the return value.

$\mathsf{checkEq} : \mathit{statement} \to \mathit{formula} \to \mathit{bool}$:
Type-checks by opponent typability (Lemma A.6).

$\mathsf{verify}_{\mathit{stm}} : \mathit{statement} \to \mathit{bool}$:
The function uses refined library functions. The type-checking, however, is straightforward since the logical refinements are not used.

$\mathsf{verify} : \mathit{proof} \to \mathit{formula} \to \mathit{bool}$,

$\mathsf{hide}_{\mathit{stm}} : \mathit{statement} \to \mathit{formula} \to \mathit{statement}$,

$\mathsf{combineOr} : \mathit{proof} \to \mathit{formula} \to \mathit{random} \to \mathit{proof}$,

$\mathsf{commuteOr} : \mathit{proof} \to \mathit{statement} \to \mathit{proof}$, and

$\mathsf{commuteAnd} : \mathit{proof} \to \mathit{statement} \to \mathit{proof}$:
Type-check by opponent typability (Lemma A.6).

A.2.2. Type-Checking Main API Methods

We proceed to showing the well-typedness of the main API methods. From a type-checking point of view, all of the proof creation methods are mostly trivially well-typed. This is not surprising since they mimic the corresponding cryptographic implementation, which almost only creates commitments; the complicated cryptographic operations take place during the zero-knowledge verification. As we will see, this is reflected into the well-typedness proof since the verification is the most complex method to type-check.

For the following proofs, we have two requirements on the typing environment E: the library and the auxiliary functions need to be contained in E with their proper types and E must be well-formed. This is the most basic typing environment that we can use to type-check the main API methods.

Definition A.14 (Basic typing environment)**.** *We call a typing environment E* basic *if and only if the following three conditions hold:*

- $E \vdash \diamond$;
- *For all auxiliary functions $f : T$ (see Table A.7), $E \vdash f : T$;*
- *For all library functions $f : T$ (see Table A.6), $E \vdash f : T$.*

Lemma A.13 (mkSays well-typed)**.** *Let E be a basic. Then, $E \vdash \mathsf{mkSays} : (x' : uid) \to (f : predicate^F) \to proof$.*

Proof sketch.

```
mkSays $(x' : uid)$ $(f : predicate^F)$: $proof$
1   let $x$ = restoreSK $x'$;
2   let $(w, z)$ = $x$;
3   match $f$ with
4     | $P_1^F$(Revealed $y_1, \ldots,$ Revealed $y_{n_1}$) $\Longrightarrow$
5       let $y'$ = $P_1^S(y_1, \ldots, y_{n_1})$;
6       let t = assume $z$ says $P_1(y_1, \ldots, y_{n_1})$;
7       let $sig$ = sign $x$ $(z, y')$;
        ⋮
```

Although the types of the values used internally are complicated, the proof is straightforward. The key insights are that the signing key x corresponding to the handle x' is extracted in line (1). The split in line (2) yields the logical formula $\{x = (w, z)\}$; this formula together with the predicate assumed in line (6) allow for deriving that the value $y' : \mathcal{U}^o_{sk}\{z/y\}$; the second component z of the pair (w, z) replaces the free variable y in the type $\mathcal{U}^o_{sk}$, yielding a closed type. The resulting logical refinement is only needed for type-checking the sign method in line (7). Since the rest of the proof only deals with values of type *unit*, the remainder of the type-checking procedure poses no problems. □

Lemma A.14 (mkSSP well-typed)**.** *Let E be a basic. Then, $E \vdash \mathsf{mkSSP} : (x : uid) \to (s : string) \to proof$.*

Proof sketch.

```
mkSSP (x' : uid) (s : bitstring) : proof =
1   let x = restoreSK x';
2   let (_, y) = x;
3   let psd = computePsd x s;
      ⋮
```

The only difficulty that may arise in this proof is the occurrence of logical refinements, e.g., by the signing key x in line (1). The refinements, however, are not used and the type-checking process is straightforward. □

Lemma A.15 (mkIDRev well-typed)**.** *Let E be a basic. Then, $E \vdash \mathsf{mkSSP} : (x : uid) \to (s : string) \to proof$.*

Proof sketch.

```
mkIDRev (p : proof) (s : string): proof =
1   match p with Says_p(c_sig, o_sig, c_z, o_z, f)  ⟹
2       match f with EscrowId(c_x, o_x, c_r, o_r)  ⟹
3           let (z, r_z) = getSome o_z;
4           let (x, r_x) = getSome o_x;
5           let (r, r_r) = getSome o_r;
6           let R = computeR x r;
7           let idr = computeIDR z x r R s;
              ⋮
```

The only difficulty that may arise in this proof is the occurrence of logical refinements, caused by the use of computeIDR in line (7). The refinement, however, is not used and the type-checking process is straightforward. □

Lemma A.16. *Let E be basic. Then, the following holds:*

- $E \vdash \mathsf{mkREL} : (y : formula) \to proof$
- $E \vdash \mathsf{mkLM} : (x : pseudo) \to (s : string) \to (\ell : (pseudo * string)\ list) \to proof$
- $E \vdash \mathsf{mkLNM} : (x : pseudo) \to (\ell : (pseudo * string)\ list) \to proof$
- $E \vdash \mathsf{mk}_\wedge : (p : proof * proof) \to proof$
- $E \vdash \mathsf{split}_\wedge : (p : proof) \to (proof * proof)$

- $E \vdash \mathsf{mk}_{\vee} : (p : \mathit{proof}) \to (f : \mathit{formula}) \to \mathit{proof}$
- $E \vdash \mathsf{hide} : (p : \mathit{proof}) \to (f : \mathit{formula}) \to \mathit{proof}$
- $E \vdash \mathsf{rerand} : (p : \mathit{proof}) \to (\mathit{stm} : \mathit{statement}) \to \mathit{proof}$

Proof. Follows from opponent typability (Lemma A.6). □

A.2.2.1. Type-Checking $\mathsf{verify}_{\mathcal{F}^{\vee}}$

Finally, we type-check the $\mathsf{verify}_{\underline{\mathcal{F}^{\vee}}}$ function and complete the type-checking of the RCF implementation. Since the verification function is assembled of different parts, we will type-check each part individually and prove that every part respects a strong invariant. Finally, we will show that this invariant is strong enough to deduce the logical refinement of the return type of the verification.

Type-checking contexts. The verification macros are contexts. Formally, we cannot type-check a context because there is no rule that can handle the hole •. We introduce the rule

$$\frac{\textsc{Exp Context}^{\mathcal{F}}}{E \vdash \bullet : \{x : \mathit{bool} \mid \forall \tilde{z}.\ y = \underline{\mathcal{F}}\ \wedge\ x = \mathsf{true} \implies \mathcal{F}\}}$$

that allows us to formally type-check contexts. The annotated logical formula is part of the refinement given to the hole. This is necessary for the invariance proof below. Naturally, the final verification code is not a context and, therefore, does not use this rule.

Typing environment after type-checking. During the invariance proof, we will need to keep track of the typing environment resulting from type-checking the code contexts. In particular, after we finished type-checking one verification macro, we need to reason about the proceeding verification macro. For this reasoning, we inherently need the typing environment extended with the binding that we just established when checking the current macro. Since the RCF typing rules do not yield access to the typing environment after the type-checking process is finished, we extend the typing rules. We stress that this modification merely offers a way for us to formalize the notion of "typing environment after type-checking" and does not affect the type system in any other way. In particular, it does not invalidate any soundness results of the original type system.

Additional bindings are only introduced in expressions that contain a continuation process. For instance, the expression

$$\mathsf{let}\ (x, r_x) = \mathsf{getSome}\ o_x\ \mathsf{in}\ \mathcal{R}$$

adds the binding $x : T, r_x : \mathit{random}$ for some type T to the continuation process $\mathcal{R}$. As a

$$\textsc{Ext Exp Subsum}\quad \frac{E \vdash A : T \leadsto E' \qquad E \vdash T <: T'}{E \vdash A : T' \leadsto E}$$

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \leadsto E}$$

$$\textsc{Ext Exp Match}\quad \frac{\begin{array}{c} E \vdash M : T \qquad h : (H, T) \\ E, x : H, \_ : \{h\ x = M\} \vdash A : U \leadsto E' \qquad E, \_ : \{\forall x.\ h\ x \neq M\} \vdash B : U \leadsto E'' \end{array}}{E \vdash \mathsf{match}\ M\ \mathsf{with}\ h\ x \to A\ \mathsf{else}\ B : U \leadsto E'}$$

$$\textsc{Ext Exp Eq}\quad \frac{E \vdash M : T \qquad E \vdash N : U \qquad x \notin \mathit{fv}(M, N)}{E \vdash M = N : \{x : \mathit{bool} \mid (x = \mathsf{true} \wedge M = N) \vee (x = \mathsf{false} \wedge M \neq N)\} \leadsto E}$$

$$\textsc{Ext Exp Assume}\quad \frac{E \vdash \diamond \qquad \mathit{fnfv}(\mathcal{F}) \subseteq \mathit{dom}(E)}{E \vdash \mathsf{assume}\ \mathcal{F} : \{\mathcal{F}\} \leadsto E}$$

$$\textsc{Ext Exp Assert}\quad \frac{E \vdash F}{E \vdash \mathsf{assert}\ F : \mathit{unit} \leadsto E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \leadsto E'' \qquad E, x : T \vdash B : U \leadsto E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \leadsto E'}$$

$$\textsc{Ext Exp If}\quad \frac{E \vdash M : T \qquad E \vdash N : U \qquad E, \{M = N\} \vdash A : V \leadsto E' \qquad E \vdash B : V \leadsto E''}{E \vdash \mathsf{if}\ M = N\ \mathsf{then}\ A\ \mathsf{else}\ B : V \leadsto E'}$$

$$\textsc{Ext Exp Split}^{n}\quad \frac{\begin{array}{c} E, x_1 : T_1, \ldots, x_n : T_n, \{(x_1, \ldots, x_n) = M\} \vdash A : V \leadsto E' \\ E \vdash M : (x_1 : T_1 * \cdots * x_{n-1} : T_{n-1} * T_n) \qquad \{x_1, \ldots, x_n\} \cap \mathit{fv}(V) = \emptyset \end{array}}{E \vdash \mathsf{let}\ (x_1, \ldots, x_n) = M\ \mathsf{in}\ A : V \leadsto E'}$$

$$\textsc{Ext Exp Match-Split}^{n}_{(m_1,\ldots,m_n)}\quad \frac{\begin{array}{c} E \vdash M : T \\ \forall 0 < i \leq n.\ h_i : (H_i, T) \qquad \forall 0 < i \leq n.\ H_i = x^i_1 : T^i_1 * \cdots * x^i_{m_i - 1} : T^i_{m_i - 1} * T^i_{m_i} \\ \forall 0 < i \leq n.\ E, x^i_1 : T^i_1, \ldots, x^i_{m_i} : T^i_{m_i}, \{M = h_i(x^i_1, \ldots, x^i_{m_i})\} \vdash A_i : U \leadsto E'_i \\ E \vdash A_{\mathsf{fail}} : U \leadsto E''' \\ \{x^i_j \mid 1 \leq i \leq n, 1 \leq j \leq m_i\} \cap \mathit{fv}(U) = \emptyset \end{array}}{\begin{array}{c} E \vdash \mathsf{match}\ M\ \mathsf{with} \\ \mid h_1\ (x^1_1, \ldots, x^1_{m_1})\ \to A_1 \ldots \mid h_n\ (x^n_1, \ldots, x^n_{m_n})\ \to A_n \mid \_ \to A_{\mathsf{fail}} : U \leadsto E'_1 \end{array}}$$

$$\textsc{Ext Exp Context}^{\mathcal{F}}\quad \frac{}{E \vdash \bullet : \{x : \mathit{bool} \mid \forall \tilde{z}.\ y = \underline{\mathcal{F}}\ \wedge\ x = \mathsf{true} \implies \mathcal{F}\} \leadsto E}$$

We omit the rules for channels (Exp Res, Exp Send, and Exp Recv), and for parallel execution (Exp Fork), as they are not used in the proofs.

Table A.17.: Excerpt of the extended rules for expressions: $E \vdash A : T \leadsto E'$

consequence, we extend the rule

Exp Split

$$\frac{E \vdash M : (\Sigma x : T.\ U) \qquad E, x : T, y : U, \_ : \{(x,y) = M\} \vdash A : V \qquad \{x,y\} \cap \mathit{fv}(V) = \emptyset}{E \vdash \mathsf{let}\ (x,y) = M;\ A : V}$$

to

Ext Exp Split

$$\frac{\begin{array}{c} E \vdash M : (\Sigma x : T.\ U) \\ E, x : T, y : U, \_ : \{(x,y) = M\} \vdash A : V \rightsquigarrow E' \qquad \{x,y\} \cap \mathit{fv}(V) = \emptyset \end{array}}{E \vdash \mathsf{let}\ (x,y) = M;\ A : V \rightsquigarrow E'}$$

Most rules do not require the type-checking of a continuation process. In fact, only a few of the typing rules for expressions have that requirement. We show all the extended rules for expressions in Table A.17. For instance, rule

Ext Exp Subsum

$$\frac{E \vdash A : T \rightsquigarrow E' \qquad E \vdash T <: T'}{E \vdash A : T' \rightsquigarrow E}$$

that ignores the typing environment E' and returns the initial environment E. For all other rules, i.e., the value typing, kinding, and subtyping, the extension is not necessary: they use the typing environment to derive facts but they do not extend the environment.

In some expressions, it is not clear which environment to return in the extended rules. For instance, rule

Ext Exp Match-Split$^{n}_{(m_1,\ldots,m_n)}$

$$\frac{\begin{array}{c} E \vdash M : T \\ \forall 0 < i \leq n.\ h_i : (H_i, T) \qquad \forall 0 < i \leq n.\ H_i = x^i_1 : T^i_1 * \cdots * x^i_{m_i - 1} : T^i_{m_i - 1} * T^i_{m_i} \\ \forall 0 < i \leq n.\ E, x^i_1 : T^i_1, \ldots, x^i_{m_i} : T^i_{m_i}, \{M = h_i(x^i_1, \ldots, x^i_{m_i})\} \vdash A_i : U \rightsquigarrow E'_i \\ E \vdash A_{\mathsf{fail}} : U \rightsquigarrow E''' \\ \{x^i_j \mid 1 \leq i \leq n, 1 \leq j \leq m_i\} \cap \mathit{fv}(U) = \emptyset \end{array}}{\begin{array}{c} E \vdash \mathsf{match}\ M\ \mathsf{with} \\ \mid h_1\ (x^1_1, \ldots, x^1_{m_1})\ \to A_1 \ldots \mid h_n\ (x^n_1, \ldots, x^n_{m_n})\ \to A_n \mid \_ \to A_{\mathsf{fail}} : U \rightsquigarrow E'_1 \end{array}}$$

takes the continuation of the first match. We resolve this ambiguity by restricting the expressions which we consider. Intuitively, we need processes where the continuation is uniquely determined, i.e., where there is no significant branching. We call these expressions linear (to denote that there is no significant branch).

Definition A.15 (Linear context)**.** *We call a context* linear *if and only if C is of either of the following forms:*

- $C = \bullet$*;*
- $C = \mathsf{match}\ M\ \mathsf{with}\ \mid h(x) \implies C' \mid \_ \implies \mathsf{false}$ *where C' is a linear context;*

- $C = \mathsf{if}\ x = y\ \mathsf{then}\ C'\ \mathsf{else}\ \mathsf{false}$ *where* C' *is a linear context;*
- $C = \mathsf{let}\ x = A; C'$ *where* C' *is a linear context;*
- $C = \mathsf{let}\ (x_1, \ldots, x_n) = A; C'$ *where* C' *is a linear context.*

Lemma A.17 (Linear context environment extension)**.** *Let* C *be a linear context and let* E' *be a typing environment such that* $E' \vdash C : \{x : \mathit{bool} \mid x = \mathsf{true} \implies F\} \rightsquigarrow E''$ *for some formula* F*, and let*

$$\dfrac{\textsc{Ext Exp Context}^{\mathcal{F}}}{E \vdash \bullet : \{x : \mathit{bool} \mid \forall \tilde{z}.\ y = \underline{\mathcal{F}}\ \wedge\ x = \mathsf{true} \implies \mathcal{F}\} \rightsquigarrow E}$$

be the only application of this rule. Then $E = E''$*.*

Proof. Let C, E', and E'' be as in the lemma. We show that the hole lies always in the branch whose typing environment is returned by the extended judgments.

The proof is by induction on the structure of linear processes. The base case is trivial since the only possible rule to apply is Ext Exp Context$^{\mathcal{F}}$. For the induction, there are only three cases to consider. A glimpse at the extended rules in Table A.17 shows that the environment passed down is always in the position where the linear context C' resides (see Definition A.15). Applying the induction hypothesis yields the desired result. □

Lemma A.18 (Transitivity linear process)**.** *Let* C_1 *and* C_2 *be two linear contexts. Then* $C_1[C_2]$ *is a linear process.*

Proof. The proof is by induction on the length of the linear process C_2. □

Lemma A.19 (Linear verification code)**.** *Let* $\mathcal{F}$ *be a formula and let* $C = [\![\mathcal{F}^{\wedge}, \mathit{zkv}, \mathit{stm}, f, \omega]\!]$ *for some* zkv*,* stm*,* f*, and* ω*. Then* C *is a linear context.*

Proof. The claim follows by inspection of the code macros and Lemma A.18. □

Here and throughout the rest of the thesis, we let $T^{\mathcal{F}}$ denote $T^{\mathcal{F}} := \{x : \mathit{bool} \mid \forall \tilde{z}.\ f = \underline{\mathcal{F}}\ \wedge\ x = \mathsf{true} \implies \mathcal{F}\}$. Notice that f is free in this type. It will be closed by the overall type of the verification function

$$\mathsf{verify}_{\mathcal{F}^{\wedge}} : (p : \mathit{proof}) \to (f : \mathit{formula}) \to T^{\mathcal{F}^{\wedge}}.$$

Definition A.16 (ψ-induced equalities)**.** *Let* $M_\psi := \{S_1, \ldots, S_m\}$ *be defined as in Definition A.3. We define*

$$(\{x_1, \ldots, x_n\})^{=} := \{x_1 = x_2\}, \ldots, \{x_1 = x_n\}, \{x_2 = x_3\}, \ldots, \{x_{n-1} = x_n\}$$

and

$$(\psi)^{=} := (M_\psi)^{=} := (S_1)^{=}, \ldots, (S_m)^{=}.$$

The following lemma establishes the anticipated relation between the code induced by ψ and the equalities induced by ψ.

Lemma A.20 ($\mathsf{context}^{=}(\psi)$ and $(\psi)^{=}$)**.** *Let ψ be a logical mapping as described above, $C := \mathsf{context}^{=}(\psi)$, and E be a typing environment. Then*

(1) C is a linear context

(2) if $E, (\psi)^{=} \vdash A : T$ for some expression A and some type T, then $E \vdash C[A] : T$.

Proof. The part 1 follows by inspecting the code generated for $\mathsf{context}^{=}(\psi)$ as defined in Definition A.3. Part 2 follows from Lemma A.8. □

Well-typedness of $\mathsf{verify}_{\underline{\mathcal{F}^{\vee}}}$. Finally, we have all the machinery to prove that our verification method is well-typed. We now state the intuition of appropriateness of typing environments, the central notion in the proof. Then we give a quick road map how the proofs will proceed and we will finally prove the well-typedness of the verification function.

The central notion of our well-typedness proof is the appropriateness $\mathsf{appropriate}(E, E', \psi, \phi, \mathcal{F}^{\wedge}, \mathit{stm}, f)$ of typing environments and the logical maps with respect to a formula $\mathcal{F}^{\wedge}$. We establish this notion as an invariance throughout all of our following proofs. Intuitively, if $\mathsf{appropriate}(E, E', \psi, \phi, \mathcal{F}^{\wedge}, \mathit{stm}, f)$ holds, then E type-checks the verification code for formula $\mathcal{F}^{\wedge}$. The typing environment E' is the resulting typing environment that is already strong enough to proof the formula $\mathcal{F}^{\wedge}$. The maps ψ and ϕ are needed to establish the invariance if $\mathcal{F}^{\wedge}$ is extended.

The proofs will proceed as follows.

(i) First, we show that if, for a given formula $\mathcal{F}^{\wedge}$, typing environments, and mappings, appropriateness holds, then we also establish appropriateness for the extended formula $\mathcal{F}^{\wedge} \wedge \mathcal{F}^{e}$. More precisely, we show that appropriateness is an invariant of the verification code.

(ii) Second, we show that starting from a basic typing environment and the verification code for a formula, we can establish the appropriateness after we finished type-checking the code.

Formally, this is a proof by induction. The base case corresponds to the empty formula and the induction step corresponds to the proof from step (i).

(iii) Third, we show that the verification function is well-typed.

Formally, we conclude that the code for formulas $\mathcal{F}^{\wedge}$ containing no disjunctions is well-typed and, leveraging this result, show that the code macro for disjunctions is also well-typed.

In our proof, we will also need to reason about the logical maps ψ and ϕ and relate them to the code that was type-checked.

Definition A.17 (Mapping extension)**.** *Let* $\mathcal{F}^\wedge$ *be a formula and* ψ *and* ϕ *be empty mappings. We call* ϕ' *and* ψ' extended by $\mathcal{F}^\wedge$ *if* ϕ' *is derived from* ϕ *and* ψ' *is derived from* ψ *only from the modification described in the lines* U_i^ϕ *and* U_j^ψ *in* $[\![\underline{\mathcal{F}^\wedge}, zkv, stm, f, \omega]\!]$ *for some* zkv*,* stm*,* f*, and* ω*. We write* $\mathsf{ext}(\mathcal{F}^\wedge, \psi', \phi')$*.*

We proceed to the central definition of this section:

Definition A.18 (Appropriate)**.** *Let* E *and* E' *be typing environments,* p *and* f *be variables,* ψ *and* ϕ *be the mappings as described above, and* $\mathcal{F}^\wedge$ *be a formula. Furthermore, let the context* $C = [\![\underline{\mathcal{F}^\wedge}, zkv, stm, f, 0]\!]$ *as in the definition, and let* $(x_0, \ldots, x_n) := \mathsf{Vars}(\underline{\mathcal{F}^\wedge})$*. Then, the predicate*

$$\mathsf{appropriate}(E, E', \psi, \phi, \mathcal{F}^\wedge, stm, f).$$

holds if and only if, all of the following conditions are fulfilled:

(1) $E \vdash C : \{x : bool \mid \forall \tilde{z}.\ f = \underline{\mathcal{F}^\wedge}\ \wedge\ x = \mathsf{true} \implies \mathcal{F}^\wedge\} \rightsquigarrow E'$*;*

(2) E *is basic and* $E' \vdash \diamond$*;*

(3) $\mathsf{ext}(\mathcal{F}^\wedge, \psi, \phi)$*;*

(4) $\forall i \in \mathcal{I}_{vk}(\underline{\mathcal{F}^\wedge}).\ E' \vdash \phi(i) : verkey$*;*

(5) $\forall i.\ \exists y.\ (x_i = \mathsf{Revealed}\ y \wedge y \in \psi(\mathcal{E}(i)))\ \vee\ (\exists z.\ x_i = \mathsf{Hidden}\ z \wedge y \in \psi(\mathcal{E}(i)))$*;*

(6) $E \vdash stm : statement$ *and* $E \vdash f : formula$*;*

(7) $E' \vdash \{f = \underline{\mathcal{F}^\wedge}\}$*;*

(8) $E', (\psi)^= \vdash \mathcal{F}^\wedge$*.*

Intuitively, the individual parts of the definition of $\mathsf{appropriate}(E, E', \psi, \phi, \mathcal{F}^\wedge, stm, f)$ express the following:

(1) $E \vdash C : \{x : bool \mid \forall \tilde{z}.\ f = \underline{\mathcal{F}^\wedge}\ \wedge\ x = \mathsf{true} \implies \mathcal{F}^\wedge\} \rightsquigarrow E'$:
E is the typing environment initially used to start type-checking the verification context. E' is the resulting typing environment when arriving at the hole $\bullet$ of the current context.

(2) E is basic and $E' \vdash \diamond$:
This requirement is mostly a sanity check since, when E is basic and E' is well-formed, then E' is also basic.

(3) $\mathsf{ext}(\mathcal{F}^\wedge, \psi, \phi)$:
This requirement only enforces that the maps contain exactly the modifications that are described in the verification macros.

(4) $\forall i \in \mathcal{I}_{vk}(\underline{\mathcal{F}^\wedge}).\ E' \vdash \phi(i) : verkey$:
The key ingredient to deriving logical formulas during the verification process are verification keys. This requirement states that if a formula contains a public user identifier x at position i, i.e., a handle to a verification key, then there is a variable $y = \phi(i)$ that corresponds to this handle and $E' \vdash y : verkey$.

(5) $\forall i.\ \exists y.\ (x_i = \mathsf{Revealed}\ y \wedge y \in \psi(\mathcal{E}(i)))\ \vee\ (\exists z.\ x_i = \mathsf{Hidden}\ z \wedge y \in \psi(\mathcal{E}(i)))$:
This requirement states that all values, no matter whether they are revealed or hidden, occur in the equality map ψ. In the revealed case, the variable itself occurs in ψ. Otherwise, there is an index inside of the Hidden constructor. However, there is a value also for hidden values in ψ (in the RCF implementation, that value y corresponds to the sealed value inside the commitment corresponding to the hidden value).

(6) $E \vdash \mathit{stm} : \mathit{statement}$ and $E \vdash f : \mathit{formula}$:
Since *stm* and f are an essential part of the verification macros, they have to occur in the typing environment. Ultimately, the formula f is an input to the verification function and the statement is extracted from the proof $p : \mathit{proof}$ that is also an input to the verification function.

(7) $E' \vdash \{f = \underline{\mathcal{F}^\wedge}\}$:
This requirement ensures that we maintain a close binding between the formula f that will be an input into the verification function and the formula which we are proving.

(8) $E', (\psi)^= \vdash \mathcal{F}^\wedge$:
This point states the expected condition, that the formulas contained in the typing environment after type-checking the verification context and the equalities induced by the map ψ are sufficient to logically deduce the formula $\mathcal{F}^\wedge$.

Proposition A.4 (Appropriate for true)**.** *Let E be basic such that $\{\mathit{stm} : \mathit{statement}, f : \mathit{formula}\} \subseteq \mathit{dom}(E)$ and let ψ and ϕ be empty maps, i.e., $\forall x.\ \psi(x) = \bot = \phi(x)$. Then* $\mathsf{appropriate}(E, E, \psi, \phi, \mathsf{true}, \mathit{stm}, f)$.

We now prove that type-checking the verification macros maintains an invariance that is close to appropriateness.

Definition A.19 (Almost appropriate)**.** *Let E, E', and E'' be typing environments, $\mathit{stm}, \mathit{stm}'$ and f, f' be variables, ψ, ψ' and ϕ, ϕ' be the mappings, and $\mathcal{F}^\wedge$ and $\mathcal{F}^e$ be formulas. Furthermore, let the context $C = [\![\underline{\mathcal{F}^\wedge}, \mathit{zkv}, \mathit{stm}, f, 0]\!]$ and $C' = [\![\underline{\mathcal{F}^e}, \mathit{zkv}', \mathit{stm}', f', \vec{\Delta}\ \underline{\mathcal{F}^\wedge}]\!]$, let $(x_0, \ldots, x_{n-1}) = \mathsf{Vars}(\underline{\mathcal{F}^\wedge \wedge \mathcal{F}^e})$ for some zero-knowledge values zkv and zkv′, and let* $\mathsf{appropriate}(E, E', \psi, \phi, \mathcal{F}^\wedge, \mathit{stm}, f)$ *hold. Then, the predicate*

$$\mathsf{appropriate}'(E, E', E'', \psi', \phi', \mathcal{F}^\wedge, \mathcal{F}^e, \mathit{stm}', f').$$

holds if and only if, all of the following conditions are fulfilled:

(1) $E' \vdash C' : \{x : \mathit{bool} \mid \forall \tilde{z}.\ y = \underline{\mathcal{F}^\wedge \wedge \mathcal{F}^e}\ \wedge\ x = \mathsf{true} \implies \mathcal{F}^\wedge \wedge \mathcal{F}^e\} \rightsquigarrow E''$;

(2) E *is basic and* $E'' \vdash \diamond$;

(3) $\mathsf{ext}(\mathcal{F}^\wedge \wedge \mathcal{F}^e, \psi', \phi')$;

(4) $\forall i \in \mathcal{I}_{\mathit{vk}}(\underline{\mathcal{F}^\wedge \wedge \mathcal{F}^e}).\ E'' \vdash \phi'(i) : \mathit{verkey}$;

(5) $\forall i.\ \exists y.\ (x_i = \mathsf{Revealed}\ y \wedge y \in \psi(\mathcal{E}(i)))\ \vee\ (\exists z.\ x_i = \mathsf{Hidden}\ z \wedge y \in \psi(\mathcal{E}(i)))$;

(6) $E' \vdash \mathit{stm}' : \mathit{proof}$ *and* $E' \vdash f' : \mathit{formula}$*;*

(7) $E'' \vdash \{f' = \underline{\mathcal{F}^e}\}$*;*

(8) $E'', (\psi')^{=} \vdash \mathcal{F}^{\wedge} \wedge \mathcal{F}^e$*.*

Intuitively, almost appropriate matches appropriate up to the requirement, that the resulting typing environment shows that the formula f equals the complete formula $\underline{\mathcal{F}^{\wedge} \wedge \mathcal{F}^e}$. This missing link allows for considering all elementary cases by themselves and, in the main theorem, use this consideration in an inductive proof. There, the code for the logical conjunction paired with the almost appropriateness ensures that the corresponding formula matches the logical conjunction. The following proposition captures this intuition.

Proposition A.5 (Linking appropriateness and almost-appropriateness)**.** *Let* E, E', E'' *be typing environments,* ψ, ψ', ϕ, ϕ' *be mappings,* $\mathcal{F}^{\wedge}, \mathcal{F}^e$ *be formulas, and* $\mathit{stm}, \mathit{stm}', f, f'$ *such that* $\mathsf{appropriate}(E, E', \psi, \phi, \mathcal{F}^{\wedge}, \mathit{stm}, f)$ *and* $\mathsf{appropriate}'(E, E', E'', \psi', \phi', \mathcal{F}^{\wedge}, \mathcal{F}^e, \mathit{stm}', y')$*.*

If $E'' \vdash f'' : \mathit{formula}$ *and* $E'' \vdash \mathit{stm}'' : \mathit{statement}$ *for some variables* f'' *and* stm''*, and* $E'', (\psi')^{=} \vdash \{f'' = \underline{\mathcal{F}^{\wedge} \wedge \mathcal{F}^e}\}$*, then* $\mathsf{appropriate}(E, E'', \psi', \phi', \mathcal{F}^{\wedge} \wedge \mathcal{F}^e, \mathit{stm}'', f'')$*.*

Lemma A.21 (Almost appropriate invariance)**.** *Let* E*,* E'*,* ψ*,* ϕ*,* $\mathcal{F}^{\wedge}$*,* stm*,* f *be such that* $\mathsf{appropriate}(E, E', \psi, \phi, \mathcal{F}^{\wedge}, \mathit{stm}, f)$ *and let* stm' *and* f' *be such that* $E' \vdash \mathit{stm}' : \mathit{statement}$ *and* $E' \vdash f' : \mathit{formula}$*. Then for every elementary formula* $\mathcal{F}^e$*, there are a typing environment* E'' *and mappings* ψ'*,* ϕ' *such that* $\mathsf{appropriate}'(E, E', E'', \psi', \phi', \mathcal{F}^{\wedge}, \mathcal{F}^e, \mathit{stm}', f')$*.*

Proof. Let E, E', ψ, ϕ, $\mathcal{F}^{\wedge}$, stm, f be such that $\mathsf{appropriate}(E, E', \psi, \phi, \mathcal{F}^{\wedge}, \mathit{stm}, f)$ and let stm' and f' be such that $E' \vdash \mathit{stm}' : \mathit{statement}$ and $E' \vdash f' : \mathit{formula}$ be as in the lemma.

We assume $\mathsf{appropriate}(E, E', \psi, \phi, \mathcal{F}^{\wedge\prime}, \mathit{stm}', f')$ and we show that type-checking an additional elementary formula $\mathcal{F}^e$ yields an almost appropriate state.

In the following proof, we stress that we never violate the well-formedness of any typing environment due to the freshness of all names and implicit α-renaming. Consequently, all typing environments we use are basic since they all originate from E.

We will omit type-checking the non-matching branches: they immediately return false. The return type $\mathsf{false} : \{x : \mathit{bool} \mid \forall \tilde{z}.\ y = \underline{\mathcal{F}^{\wedge}}\ \wedge\ x = \mathsf{true} \implies \mathcal{F}^{\wedge}\}$ vacuously holds true since x turns the complete premise of the implication into false. We will also skip the initial matching statements. They hold by inspecting the type definitions in Table A.2 and Table A.3.

We will implicitly use rule

$$\frac{\begin{array}{l}\textsc{Val Var}\\ E \vdash \diamond \qquad (x : T) \in E\end{array}}{E \vdash x : T}$$

by arguing that a variable along with a certain type is entered in the current typing environment. Finally, we use Lemma A.17 (linear context environment extension) and Lemma A.19 (linear verification code) to derive that the environment E'' used to type-check the hole • in the respective verification context is also the environment returned by the extended rules.

Case $\mathcal{F}^e \triangleq A$ says $P_k(m_1, \ldots, m_n)$**:**

Code (line 5):

$$\text{let } tmp_{sig} = \mathsf{openCommit}\ c_{sig};$$

Environment:

$$\underbrace{\begin{array}{l} E', c_{sig} : commitment, c_z : commitment, c_{arg_1} : commitment, \ldots, c_{arg_n} : commitment, \\ \quad \{stm = \mathsf{Says}_\mathsf{p}(c_{sig}, \_, c_z, \_, P_k^P(c_{arg_1}, \_, \ldots, c_{arg_n}, \_))\} \\ arg'_0 : uid_{pub}\ RevHid, arg'_1 : unit, \ldots, arg'_n : unit, \\ \quad \{f = \mathsf{Says}(arg'_0, P_k^F(arg'_1, \ldots, arg'_n))\} \end{array}}_{=:E'_1}$$

Rules:

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\ U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin fv(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_1 \vdash \mathsf{openCommit} : commitment \to bitstring * random$
2. $E'_1 \vdash c_{sig} : commitment$
3. $tmp_{sig} \notin fv(T^{\mathcal{F}^\wedge})$
4. $E'_1, tmp_{sig} : bitstring * random \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

Since E'_1 is basic, the first obligation follows. The second obligation follows because $c_{sig} : commitment \in E'_1$. The third obligation holds since the variable does not occur in the return type.

Code (line 6):

$$\text{let } (sig, r_{sig}) = tmp_{sig};$$

Environment:

$$\underbrace{E'_1, tmp_{sig} : bitstring * random}_{=:E'_2}$$

Rules:

$$\textsc{Ext Exp Split}\quad \frac{\begin{array}{c} E \vdash M : (\Sigma x : T.\ U) \\ E, x : T, y : U, \_ : \{(x, y) = M\} \vdash A : V \rightsquigarrow E' \qquad \{x, y\} \cap fv(V) = \emptyset \end{array}}{E \vdash \mathsf{let}\ (x, y) = M;\ A : V \rightsquigarrow E'}$$

Proof Obligations:

1. $E_2' \vdash \mathit{tmp}_{\mathit{sig}} : \mathit{bitstring} * \mathit{random}$
2. $\{\mathit{sig}, r_{\mathit{sig}}\} \cap \mathit{fv}(T^{\mathcal{F}^\wedge}) = \emptyset$
3. $\begin{array}{l} E_2', \mathit{tmp}_{\mathit{sig}} : \mathit{bitstring} * \mathit{random}, \mathit{sig} : \mathit{signature}, \\ \quad r_{\mathit{sig}} : \mathit{random}, \{(\mathit{sig}, r_{\mathit{sig}}) = \mathit{tmp}_{\mathit{sig}}\} \end{array} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

Since $\mathit{tmp}_{\mathit{sig}} : \mathit{bitstring} * \mathit{random} \in E_2'$, the first obligation follows. The second obligation follows because $\mathit{tmp}_{\mathit{sig}}$ is fresh and does not occur anywhere else. For the next type-checking steps, we recall that $\mathit{bitstring} = \mathit{signature} = \mathit{unit}$, i.e., we can safely add sig with type $\mathit{signature}$ into the typing environment.

We repeat this step until we arrive at line 12 (the arguments for the obligations are analogous to the two type-checking steps above.

Code (lines 7-12):

$$\begin{array}{l} \text{let } \mathit{tmp}_z = \mathsf{openCommit}\ c_z; \\ \text{let } (\mathit{arg}_0, r_{\mathit{arg}_0}) = \mathit{tmp}_z; \\ \vdots \\ \text{let } \mathit{tmp}_{n-1} = \mathsf{openCommit}\ c_{\mathit{arg}_{n-1}}; \\ \text{let } (\mathit{arg}_{n-1}, r_{\mathit{arg}_{n-1}}) = \mathit{tmp}_{n-1}; \\ \text{let } \mathit{tmp}_n = \mathsf{openCommit}\ c_{\mathit{arg}_n}; \\ \text{let } (\mathit{arg}_n, r_{\mathit{arg}_n}) = \mathit{tmp}_n; \end{array}$$

In the proceeding proof, we implicitly use Lemma A.4 (weakening) to skip the logical formulas obtained by the splitting process since we do not need them.

We type-check under the assumption that the i-th argument is revealed.

Code (line 13):

$$\text{let } \mathit{arg}_i^o = \mathsf{getRevealed}\ \mathit{arg}_i';$$

Environment:

$$\underbrace{\begin{array}{l} E_2', \mathit{tmp}_{\mathit{sig}} : \mathit{bitstring} * \mathit{random}, \mathit{sig} : \mathit{signature}, r_{\mathit{sig}} : \mathit{random}, \\ \mathit{tmp}_z : \mathit{bitstring} * \mathit{random}, \mathit{arg}_0 : \mathit{bitstring}, r_{\mathit{arg}_0} : \mathit{random}, \ldots \\ \mathit{tmp}_n : \mathit{bitstring} * \mathit{random}, \mathit{arg}_n : \mathit{bitstring}, r_{\mathit{arg}_n} : \mathit{random} \end{array}}_{=:E_3'}$$

Rules:

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_3 \vdash \mathsf{getRevealed} : (x : \mathit{unit\ RevHid}) \ \rightarrow \ \{y : \mathit{unit} \mid x = \mathsf{Revealed}\ y\}$
2. $E'_3 \vdash \mathit{arg}'_i : \mathit{unit\ RevHid}$
3. $\mathit{arg}^o_i \notin \mathit{fv}(T^{\mathcal{F}^\wedge})$
4. $E'_3, \mathit{arg}^o_i : \{y : \mathit{unit} \mid \mathit{arg}'_i = \mathsf{Revealed}\ y\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The obligations hold analogously to the previous step.

Code (line U^ψ_0):

$$\psi := \psi[\mathcal{E}(\omega + i) \mapsto \psi(\mathcal{E}(\omega + i)) \cup \{\mathit{arg}^o_i\}]$$

We extend the map ψ to include the revealed arguments of the formula. Together with the map extension in lines U^ψ_1-U^ψ_{n+1}, we ensure that the formula matches the proof and that the desired formula can be proven.

Code (lines 15-20):

```
let $z''$ = match $\mathit{arg}'_0$ with
   | Revealed $x$ $\Longrightarrow$
      PKI $x$
      $\phi := \phi[\mathcal{E}(\omega) \mapsto z'']$
   | _ $\Longrightarrow$
      $\phi(\mathcal{E}(\omega))$;
```

Environment:

$$\underbrace{E'_3, \mathit{arg}^o_i : \{y : \mathit{unit} \mid \mathit{arg}'_i = \mathsf{Revealed}\ y\}}_{=:E'_4}$$

Rules:

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

$$\textsc{Ext Exp Match}\quad \frac{\begin{array}{c} E \vdash M : T \qquad h : (H, T) \\ E, x : H, \_ : \{h\ x = M\} \vdash A : U \rightsquigarrow E' \qquad E, \_ : \{\forall x.\ h\ x \neq M\} \vdash B : U \rightsquigarrow E'' \end{array}}{E \vdash \mathsf{match}\ M\ \mathsf{with}\ h\ x \rightarrow A\ \mathsf{else}\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_4 \vdash \mathit{arg}'_0 : \mathit{uid}_{\mathit{pub}}\ \mathit{RevHid}$
2. $E'_4 \vdash \mathsf{Revealed} : (\mathit{uid}_{\mathit{pub}}, \mathit{uid}_{\mathit{pub}}\ \mathit{RevHid})$
3. $E'_4, x : \mathit{unit}, \{\mathit{arg}'_0 = \mathsf{Revealed}\ x\} \vdash \mathsf{PKI} : (x : \mathit{unit}) \ \rightarrow\ \{y : \mathit{verkey} \mid x = y\}$
4. $E'_4, x : \mathit{unit}, \{\mathit{arg}'_0 = \mathsf{Revealed}\ x\} \vdash x : \mathit{unit}$
5. $E'_4, x : \mathit{unit}, \{\mathit{arg}'_0 = \mathsf{Revealed}\ x\} \vdash \mathsf{PKI}\ x : \mathit{verkey}$
6. $z'' \notin \mathit{fv}(T^{\mathcal{F}^\wedge}) = \emptyset$
7. $E'_4, \{\forall y.\ \mathit{arg}'_0 \neq \mathsf{Revealed}\ y\} \vdash \phi(\mathcal{E}(\omega)) : \mathit{verkey}$
8. if the verification key is revealed: $E'_4, z'' : \{y : \mathit{verkey} \mid \mathit{arg}'_0 = \mathsf{Revealed}\ y\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$
9. otherwise: $E'_4, z'' : \mathit{verkey} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first four premises hold due to E'_4 being basic and because the variables have been added to the typing environments in previous steps. The fifth premise follows immediately by Exp Appl.

Since z'' does not occur in $T^{\mathcal{F}^\wedge}$, the sixth obligation follows.

For the remaining obligations, we notice that due to the assumed appropriateness of the formula $\mathcal{F}^\wedge$ in combination with the well-formedness of the logical formula: If $\mathcal{E}(\omega) = \omega$, i.e., there is no occurrence of a variable y at a position $i < \omega$ that is equal to arg'_0 in the formula $\mathcal{F}^\wedge$, then the variable occurs here for the first time. Since it is a verification key, well-formedness requires that the variable is revealed: were we type-checking the "hiding" branch, it would contradict the well-formedness of the proven formula. If there is such a position $i < \omega$, then the appropriateness of formula $\mathcal{F}^\wedge$ yields that $E' \vdash \phi(\mathcal{E}(i)) : \mathit{verkey}$. Since $\mathcal{E}(i) = \mathcal{E}(\omega)$ by definition of the map $\mathcal{E}$ and by extending E' without breaking the well-formedness, we derive that $E'_4, \{\forall y.\ \mathit{arg}'_0 \neq \mathsf{Revealed}\ y\} \vdash \phi(\mathcal{E}(\omega)) : \mathit{verkey}$. In either case, $E'_4 \vdash \phi(\mathcal{E}(\omega)) : \mathit{verkey}$, i.e., z'' will have type verkey in the resulting typing environment. If the verification key is revealed, the matching yields the formula $\{\mathit{arg}'_0 = \mathsf{Revealed}\ x\}$ and the application of the auxiliary function PKI yields a variable of type $\{y : \mathit{verkey} \mid x = y\}$. Combining them gives the type $\{y : \mathit{verkey} \mid \mathit{arg}'_0 = \mathsf{Revealed}\ y\}$.

Code (line 21):

$$\text{if } z'' = \mathit{arg}_0 \text{ then}$$

Environment:

$$\underbrace{E'_4, z'' : \mathit{verkey}}_{=:E'_5}$$

Rules:

Ext Exp If

$$\frac{E \vdash M : T \qquad E \vdash N : U \qquad E, \{M = N\} \vdash A : V \rightsquigarrow E' \qquad E \vdash B : V \rightsquigarrow E''}{E \vdash \mathsf{if}\ M = N\ \mathsf{then}\ A\ \mathsf{else}\ B : V \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_5 \vdash z'' : \mathit{verkey}$
2. $E'_5 \vdash \mathit{arg}_0 : \mathit{unit}$
3. $E'_5, \{z'' = \mathit{arg}_0\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first two obligations follow since the respective variables have been added to the typing environment in previous steps.

Code (line 22):

$$\text{let } m = \mathsf{check}_{sig}\ z''\ sig;$$

Environment:

$$\underbrace{E'_5, \{z'' = arg_0\}}_{=:E'_6}$$

Rules:

$$\frac{E \vdash M : (\Pi x : T.\ U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}\ \textsc{Ext Exp Appl}$$

$$\frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin fv(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}\ \textsc{Ext Exp Let}$$

Proof Obligations:

1. $E'_6 \vdash \mathsf{check}_{sig} : (y : verkey) \ \rightarrow\ (sig : signature) \ \rightarrow\ \mathcal{T}_y\{verkey/\alpha\}$
2. $E'_6 \vdash z'' : verkey$
3. $E'_6 \vdash sig : signature$
4. $E'_6 \vdash m : \mathcal{T}_y\{z''/y\}$
5. $m \notin fv(T^{\mathcal{F}^\wedge})$
6. $E'_6, m : \mathcal{T}_y\{z''/y\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first three obligations hold since E'_6 is basic and because the respective variables have been added to the typing environment with the proper types. The fourth obligation is a direct consequence of Exp Appl. The fifth obligation holds since m does not occur in $T^{\mathcal{F}^\wedge}$.

Code (lines 23-24):

$$\begin{array}{l}\text{match } m \text{ with}\\ \quad \mid P^S_k(y'_1, \dots, y'_n) \implies\end{array}$$

Environment:

$$\underbrace{E'_6, m : \mathcal{T}_y\{z''/y\}}_{=:E'_7}$$

Rules:

$$\begin{array}{l}\textsc{Ext Exp Match-Split}^{n}_{(m_1,\ldots,m_n)}\\ \dfrac{\begin{array}{c} E \vdash M : T \qquad \forall 0 < i \leq n.\ h_i : (H_i, T) \qquad \forall 0 < i \leq n.\ H_i = x^i_1 : T^i_1 * \cdots * x^i_{m_i-1} : T^i_{m_i-1} * T^i_{m_i} \\ \forall 0 < i \leq n.\ E, x^i_1 : T^i_1, \ldots, x^i_{m_i} : T^i_{m_i}, \{M = h_i(x^i_1, \ldots, x^i_{m_i})\} \vdash A_i : U \rightsquigarrow E'_i \\ E \vdash A_{\mathsf{fail}} : U \rightsquigarrow E'' \\ \{x^i_j \mid 1 \leq i \leq n, 1 \leq j \leq m_i\} \cap fv(U) = \emptyset \end{array}}{E \vdash \mathsf{match}\ M\ \mathsf{with} \mid h_1\ (x^1_1, \ldots, x^1_{m_1})\ \to A_1 \ldots \mid h_n\ (x^n_1, \ldots, x^n_{m_n})\ \to A_n \mid \_ \to A_{\mathsf{fail}} : U \rightsquigarrow E'_1}\end{array}$$

Proof Obligations:

1. $E'_7 \vdash m : \mathcal{T}_y\{z''/y\}$
2. $E'_7 \vdash P^S_k : (x^k_1 : T^k_1 * \cdots * x^k_{n-1} : T^k_{n-1} * \{x^k_n : T^k_n \mid z''\ \mathsf{says}\ P_k(x^k_1, \ldots, x^k_n)\},\ \mathcal{T}_y\{z''/y\})$
3. $\begin{array}{r} E'_7, y'_1 : T^k_1, \ldots, y'_n : \{x^k_n : T^k_n \mid z''\ \mathsf{says}\ P_k(y'_1, \ldots, y'_{n-1}, x^k_n)\}, \\ \{m = P^S(y'_1, \ldots, y'_n)\} \end{array} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first hypothesis follows since m has been added to the typing environment in the previous step. The second hypothesis follows by inspecting the definition of type *verkey* (see Table A.5).

Code (lines 25-26):

$$\begin{array}{l} \text{if } arg_1 = y'_1 \text{ then} \\ \vdots \\ \text{if } arg_n = y'_n \text{ then} \end{array}$$

Environment:

$$\underbrace{E'_7, y'_1 : T^k_1, \ldots, y'_n : \{x^k_n : T^k_n \mid z''\ \mathsf{says}\ P_k(y'_1, \ldots, y'_{n-1}, x^k_n)\}, \{m = P^S(y'_1, \ldots, y'_n)\}}_{=:E'_8}$$

Rules:

$$\begin{array}{l}\textsc{Ext Exp If}\\ \dfrac{E \vdash M : T \qquad E \vdash N : U \qquad E, \{M = N\} \vdash A : V \rightsquigarrow E' \qquad E \vdash B : V \rightsquigarrow E''}{E \vdash \mathsf{if}\ M = N\ \mathsf{then}\ A\ \mathsf{else}\ B : V \rightsquigarrow E'}\end{array}$$

Proof Obligations:

1. $\forall i.\ E'_8 \vdash arg_i : unit$
2. $\forall i.\ E'_8 \vdash y_i : T^k_i$
3. $E'_8, \{arg_1 = y'_1\}, \ldots \{arg_n = y'_n\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first two obligations hold because all the variables have been entered into the typing environment in previous steps.

Code (lines U_1^{ϕ}-U_{n+1}^{ψ}):

$$\begin{aligned}
&\phi := \phi[\mathcal{E}(\omega+1) \mapsto y_1'] \\
&\vdots \\
&\phi := \phi[\mathcal{E}(\omega+n) \mapsto y_n'] \\
&\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{arg_0\}] \\
&\psi := \psi[\mathcal{E}(\omega+1) \mapsto \psi(\mathcal{E}(\omega+1)) \cup \{arg_1\}] \\
&\vdots \\
&\psi := \psi[\mathcal{E}(\omega+n) \mapsto \psi(\mathcal{E}(\omega+n)) \cup \{arg_n\}]
\end{aligned}$$

At this point, we greatly extend the maps ψ and ϕ. We add all the newly-derived variables y_i' into the map ϕ. Since only the y_i' are potentially typed *verkey*, we ensure that $E_8' \vdash \phi(\mathcal{E}(\omega+i)) : \mathit{verkey}$.

Furthermore, we add all the variables arg_i to the correct position in the map ψ. This ensures that if the content of two commitments are equal, then we can also reflect this back into the proven logical formula, even if the values are hidden. Notice that this step also connects the proof to the proven formula: the equalities between the y_i' and the arg_i are proven in lines 25-26; the equality between z'' and arg_0 is proven in line 21.

Finally, we arrived at $\bullet$.

Code (line 28):

$$\bullet$$

Environment:

$$\underbrace{E_8', \{arg_1 = y_1'\}, \ldots, \{arg_n = y_n'\}}_{=:E_9'}$$

Rules:

$$\frac{\textsc{Ext Exp Context}^{\mathcal{F}}}{E \vdash \bullet : \{x : \mathit{bool} \mid \forall \tilde{z}.\ y = \underline{\mathcal{F}} \ \wedge\ x = \mathsf{true} \implies \mathcal{F}\} \rightsquigarrow E}$$

Proof Obligations:

none

We let $\phi' := \phi$ and $\psi' := \psi$ and argue why the current typing environment and maps are almost appropriate:

(1) $E' \vdash C' : \{x : \mathit{bool} \mid \forall \tilde{z}.\ f = \underline{\mathcal{F}^{\wedge} \wedge \mathcal{F}^{e}} \ \wedge\ x = \mathsf{true} \implies \mathcal{F}^{\wedge} \wedge \mathcal{F}^{e}\} \rightsquigarrow E''$:
We conducted the type-checking and we have derived that $E'' := E_9'$. We stress that this type only holds due to the typing rule Exp Context$^{\mathcal{F}}$. The typing environment cannot prove that $f = \underline{\mathcal{F}^{\wedge} \wedge \mathcal{F}^{e}}$. The final code, however, will be able to prove this type.

(2) E is basic and $E'' \vdash \diamond$:
We never double-bind values and all the free names and variables of a type are closed inside of the typing environment.

(3) $\mathsf{ext}(\mathcal{F}^\wedge \wedge \mathcal{F}^e, \psi', \phi')$:
From $\mathsf{appropriate}(E, E', \mathcal{F}^\wedge, \psi, \phi, \mathit{stm}, f)$, we immediately obtain that $\mathsf{ext}(\mathcal{F}^\wedge, \psi, \phi)$. The only modifications applied to ψ and ϕ are those described in the verification context. Since the verification context for logical conjunction and logical disjunction do not contain changes to ψ and ϕ, we conclude that $\mathsf{ext}(\mathcal{F}^\wedge \wedge \mathcal{F}^e, \psi', \phi')$.

(4) $\forall i \in \mathcal{I}_{vk}(\underline{\mathcal{F}^\wedge \wedge \mathcal{F}^e}).\ E'' \vdash \phi'(i) : \mathit{verkey}$:
For the variables in the formula $\mathcal{F}^\wedge$, this holds by the assumed appropriateness of E'. For the other values, we argued that all variables that might be given type *verkey* are entered into the map ϕ. More precisely, the variables are entered in lines U_1^ϕ-U_n^ϕ.

(5) $\forall i.\ \exists y.\ (x_i = \mathsf{Revealed}\ y \wedge y \in \psi(\mathcal{E}(i)))\ \vee\ (\exists z.\ x_i = \mathsf{Hidden}\ z \wedge y \in \psi(\mathcal{E}(i)))$:
If the i-th argument arg_i' to the says predicate is revealed, we have executed lines 9 and U_0^ψ for that variable. Consequently, we have added arg_i^o in to ψ' where $\mathit{arg}_i' = \mathsf{Revealed}\ \mathit{arg}_i^o$.

In either case, we add the variable arg_i to ψ' in line U_{i+1}^ψ. In particular, this variable will ensure that all values hidden by the same existential quantifier are represented by the same committed value (the equality between them is ensured by the map $\mathcal{E}$).

(6) $E' \vdash \mathit{stm}' : \mathit{statement}$ and $E' \vdash f' : \mathit{formula}$:
This is an assumption.

(7) $E'' \vdash \{f' = \underline{\mathcal{F}^e}\}$ and

(8) $E'', (\psi')^= \vdash \mathcal{F}^\wedge \wedge \mathcal{F}^e$:
The assumed appropriateness yields that $E', (\psi)^= \vdash \mathcal{F}^\wedge$. Since E'' is an extension of E' and ψ' is an extension of ψ, we obtain that $E'', (\psi')^= \vdash \mathcal{F}^\wedge$.

For the formula $\mathcal{F}^e$ with the encoding $\underline{\mathcal{F}^e} = \mathsf{Says}(\mathit{arg}_0', P_k^F(\mathit{arg}_1', \ldots, \mathit{arg}_n'))$, we consider the following formulas in E'' and the entries of ψ':

(a) lines 1-4: $\{f' = \mathsf{Says}(\mathit{arg}_0', P_k^F(\mathit{arg}_1', \ldots, \mathit{arg}_n'))\}$

(b) line 13: For all revealed entries: $\{\mathit{arg}_i' = \mathsf{Revealed}\ \mathit{arg}_i^o\}$

(c) line U_0^ψ $\psi[\mathcal{E}(\omega + i) \mapsto \psi(\mathcal{E}(\omega + i)) \cup \{\mathit{arg}_i^o\}]$

(d) lines 16-20: If arg_0' is revealed, then we additionally get $\{\mathit{arg}_0' = \mathsf{Revealed}\ z''\}$

(e) line 21: $\{z'' = \mathit{arg}_0\}$

(f) lines 22-24: $\{z''\ \mathsf{says}\ P_k(y_1', \ldots, y_n')\}$

(g) lines 25-26: $\{\mathit{arg}_1 = y_1'\}, \ldots \{\mathit{arg}_n = y_n'\}$

(h) lines U_1^ψ-U_{n+1}^ψ:

$$\begin{aligned}&\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{arg_0\}]\\ &\psi := \psi[\mathcal{E}(\omega+1) \mapsto \psi(\mathcal{E}(\omega+1)) \cup \{arg_1\}]\\ &\vdots\\ &\psi := \psi[\mathcal{E}(\omega+n) \mapsto \psi(\mathcal{E}(\omega+n)) \cup \{arg_n\}]\end{aligned}$$

First, we note that by (e), (f), and (g), we derive

$$arg_0 \text{ says } P_k(arg_1, \ldots, arg_n).$$

More precisely, we derive the logical formula where all values are derived from the commitments attached to the proof. If the i-th value is revealed in $\mathcal{F}^\wedge$, we obtain $arg'_i = \mathsf{Revealed}\ arg^o_i$ by (b) and by (d) we get $arg'_0 = \mathsf{Revealed}\ z''$. We obtain the expected equalities $arg^o_i = arg_i$ by (c) and (h), and $arg_0 = z''$ by (e); for all hidden values, we get the necessary equalities by (h). In particular, this substitution shows that $E'' \vdash \{f' = \underline{\mathcal{F}^e}\}$. Furthermore, it shows that $E'' \vdash \mathcal{F}^e$ since substituting all the arg^o_i for the revealed values and consistently replacing the committed values arg_j with existential quantified values proves $\mathcal{F}^e$. The equalities induced by ψ enable us to consistently existentially quantify across $\mathcal{F}^\wedge$ and $\mathcal{F}^e$, yielding $E'' \vdash \mathcal{F}^\wedge \wedge \mathcal{F}^e$.

Case $\mathcal{F}^e \triangleq \mathsf{SSP}(A, s, psd)$:

Code (line 3):

$$\text{let } tmp_x = \mathsf{openCommit}_{sk}\ c_x;$$

Environment:

$$\underbrace{\begin{aligned}&E', c_z : commitment, c_s : commitment, c_{psd} : commitment, c_x : commitment,\\ &\quad\{stm = \mathsf{SSP}_\mathsf{p}(c_z, \_, c_s, \_, c_{psd}, c_x)\}\\ &z' : uid_{pub}\ RevHid, s' : string\ RevHid, psd' : pseudo\ RevHid,\\ &\quad\{f = \mathsf{SSP}(z', s', psd')\}\end{aligned}}_{=:E'_1}$$

Rules:

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin fv(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_1 \vdash \mathsf{openCommit}_{sk} : commitment \to sigkey * random$
2. $E'_1 \vdash c_x : commitment$
3. $tmp_x \notin fv(T^{\mathcal{F}^\wedge})$
4. $E'_1, tmp_x : sigkey * random \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

Since E'_1 is basic, the first obligation follows. The second obligation follows since $c_x : \mathit{commitment} \in E'_1$. The third obligation holds since tmp_x does not occur in the return type.

Code (line 4):

$$\text{let } (x, r_x) = \mathit{tmp}_x;$$

Environment:

$$\underbrace{E'_1, \mathit{tmp}_x : \mathit{sigkey} * \mathit{random}}_{=:E'_2}$$

Rules:

$$\textsc{Ext Exp Split}\quad \frac{E \vdash M : (\Sigma x : T.\, U) \qquad E, x : T, y : U, \_ : \{(x, y) = M\} \vdash A : V \rightsquigarrow E' \qquad \{x, y\} \cap \mathit{fv}(V) = \emptyset}{E \vdash \mathsf{let}\ (x, y) = M;\ A : V \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_2 \vdash \mathit{tmp}_x : \mathit{sigkey} * \mathit{random}$
2. $\{x, r_x\} \cap \mathit{fv}(T^{\mathcal{F}^\wedge}) = \emptyset$
3. $E'_2, x : \mathit{sigkey}, r_x : \mathit{random}, \{(x, r_x) = \mathit{tmp}_x\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since $c_x : \mathit{commitment} \in E'_2$. The second obligation holds since the variables do not occur in the return type.

Code (line 5):

$$\text{let } \mathit{tmp}_z = \mathsf{openCommit}\ c_z;$$

Environment:

$$\underbrace{E'_2, x : \mathit{sigkey}, r_x : \mathit{random}, \{(x, r_x) = \mathit{tmp}_x\}}_{=:E'_3}$$

Rules:

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E_3' \vdash \mathsf{openCommit} : \mathit{commitment} \rightarrow \mathit{bitstring} * \mathit{random}$
2. $E_3' \vdash c_z : \mathit{commitment}$
3. $\mathit{tmp}_z \notin \mathit{fv}(T^{\mathcal{F}^\wedge})$
4. $E_3', \mathit{tmp}_z : \mathit{bitstring} * \mathit{random} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

Since E_3' is basic, the first obligation follows. The second obligation follows since $c_z : \mathit{commitment} \in E_3'$. The third obligation holds since the variable does not occur in the return type.

Code (line 6):

$$\text{let } (z, r_z) = \mathit{tmp}_z;$$

Environment:

$$\underbrace{E_3', \mathit{tmp}_z : \mathit{bitstring} * \mathit{random}}_{=:E_4'}$$

Rules:

Ext Exp Split

$$\frac{E \vdash M : (\Sigma x : T.\, U) \qquad E, x : T, y : U, \_ : \{(x, y) = M\} \vdash A : V \rightsquigarrow E' \qquad \{x, y\} \cap \mathit{fv}(V) = \emptyset}{E \vdash \mathsf{let}\ (x, y) = M;\ A : V \rightsquigarrow E'}$$

Proof Obligations:

1. $E_4' \vdash \mathit{tmp}_z : \mathit{bitstring} * \mathit{random}$
2. $\{z, r_z\} \cap \mathit{fv}(T^{\mathcal{F}^\wedge}) = \emptyset$
3. $E_4', z : \mathit{bitstring}, r_z : \mathit{random}, \{(z, r_z) = \mathit{tmp}_z\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since $\mathit{tmp}_z : \mathit{bitstring} * \mathit{random} \in E_4'$. The second obligation holds since the variables do not occur in the return type.

Code (line 7):

$$\text{let } \mathit{tmp}_s = \mathsf{openCommit}\ c_s;$$

Code (line 8):

$$\text{let } (s, r_s) = \mathit{tmp}_s;$$

Code (line 9):

$$\text{let } \mathit{tmp}_{psd} = \mathsf{openCommit}\ c_{psd};$$

Code (line 10):

$$\text{let } (\mathit{psd}, r_{psd}) = \mathit{tmp}_{psd};$$

Lines 7 through 10 are analogous to lines 5 and 6.

Code (line 11):

$$\text{let } (x'', x') = x;$$

Environment:

$$\underbrace{\begin{array}{l} E'_4, \mathit{tmp}_s : \mathit{bitstring} * \mathit{random}, s : \mathit{bitstring}, r_s : \mathit{random}, \{(s, r_s) = \mathit{tmp}_s\}, \\ \mathit{tmp}_{\mathit{psd}} : \mathit{bitstring} * \mathit{random}, \mathit{psd} : \mathit{bitstring}, r_{\mathit{psd}} : \mathit{random}, \{(\mathit{psd}, r_{\mathit{psd}}) = \mathit{tmp}_{\mathit{psd}}\} \end{array}}_{=:E'_5}$$

Rules:

Ext Exp Split

$$\frac{\begin{array}{c} E \vdash M : (\Sigma x : T.\, U) \\ E, x : T, y : U, \_ : \{(x, y) = M\} \vdash A : V \rightsquigarrow E' \qquad \{x, y\} \cap \mathit{fv}(V) = \emptyset \end{array}}{E \vdash \mathsf{let}\ (x, y) = M;\ A : V \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_5 \vdash x : \mathit{sigkey}$
2. $\{x'', x'\} \cap \mathit{fv}(T^{\mathcal{F}^\wedge}) = \emptyset$
3. $E'_5, x'' : (\mu\alpha.\, \mathcal{T}_y \rightarrow \mathit{signature}), x' : \mathit{verkey}, \{x = (x'', x')\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since $x : \mathit{sigkey} \in E'_5$ and because $\mathit{sigkey} := (\mu\alpha.\, \mathcal{T}_y \rightarrow \mathit{signature}) * \mathit{verkey}$, i.e., the type *sigkey* can be split. The second obligation holds because neither x' nor x'' occur in $T^{\mathcal{F}^\wedge}$.

We will type-check the code as if all values were revealed.

Code (line 12):

$$\text{let } z^o = \mathsf{getRevealed}\ z';$$

Environment:

$$\underbrace{E'_5, x'' : (\mu\alpha.\, \mathcal{T}_y \rightarrow \mathit{signature}), x' : \mathit{verkey}, \{(x'', x') = x\}}_{=:E'_6}$$

Rules:

Ext Exp Appl

$$\frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

Ext Exp Let

$$\frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_6 \vdash \mathsf{getRevealed} : (x : \mathit{unit}\ \mathit{RevHid}) \ \rightarrow\ \{y : \mathit{unit} \mid x = \mathsf{Revealed}\ y\}$
2. $E'_6 \vdash z' : \mathit{uid}_{\mathit{pub}}\ \mathit{RevHid}$
3. $z^o \notin \mathit{fv}(T^{\mathcal{F}^\wedge})$
4. $E'_6, z^o : \{y : \mathit{unit} \mid z' = \mathsf{Revealed}\ y\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since E'_6 is basic. The second obligation holds since $z' : \mathit{uid}_{\mathit{pub}}\ \mathit{RevHid} \in E'_6$. The third obligation follows since z^o does not occur in $T^{\mathcal{F}^\wedge}$.

Code (line U_0^ψ):

$$\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{z^o\}]$$

We extend the map ψ to include z^o.

Code (line 14):

$$\text{let } s^o = \mathsf{getRevealed}\ s';$$

Environment:

$$\underbrace{E'_6, z^o : \{y : \mathit{unit} \mid z' = \mathsf{Revealed}\ y\}}_{=:E'_7}$$

Rules:

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\ U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_7 \vdash \mathsf{getRevealed} : (x : \mathit{unit}\ \mathit{RevHid}) \ \rightarrow\ \{y : \mathit{unit} \mid x = \mathsf{Revealed}\ y\}$
2. $E'_7 \vdash s' : \mathit{bitstring}\ \mathit{RevHid}$
3. $s^o \notin \mathit{fv}(T^{\mathcal{F}^\wedge})$
4. $E'_7, s^o : \{y : \mathit{unit} \mid s' = \mathsf{Revealed}\ y\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since E'_7 is basic. The second obligation holds since $s' : \mathit{bitstring}\ \mathit{RevHid} \in E'_7$. The third obligation follows since s^o does not occur in $T^{\mathcal{F}^\wedge}$.

Code (line U_1^ψ):

$$\psi := \psi[\mathcal{E}(\omega + 1) \mapsto \psi(\mathcal{E}(\omega + 1)) \cup \{s^o\}]$$

We extend the map ψ to include s^o.

Code (line 16):

$$\text{let } psd^o = \mathsf{getRevealed}\ psd';$$

Environment:

$$\underbrace{E_7', s^o : \{y : unit \mid s' = \mathsf{Revealed}\ y\}}_{=:E_8'}$$

Rules:

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin fv(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E_8' \vdash \mathsf{getRevealed} : (x : unit\ RevHid)\ \rightarrow\ \{y : unit \mid x = \mathsf{Revealed}\ y\}$
2. $E_8' \vdash psd' : bitstring\ RevHid$
3. $psd^o \notin fv(T^{\mathcal{F}^\wedge})$
4. $E_8', psd^o : \{y : unit \mid psd' = \mathsf{Revealed}\ y\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since E_8' is basic. The second obligation holds since $psd' : bitstring\ RevHid \in E_8'$. The third obligation follows since psd^o does not occur in $T^{\mathcal{F}^\wedge}$.

Code (line U_2^ψ):

$$\psi := \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{psd^o\}]$$

We extend the map ψ to include psd^o.

Code (lines U_3^ψ-U_5^ψ):

$$\begin{aligned}
&\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{z, x'\}]\\
&\psi := \psi[\mathcal{E}(\omega + 1) \mapsto \psi(\mathcal{E}(\omega + 1)) \cup \{s\}]\\
&\psi := \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{psd\}]
\end{aligned}$$

We extend the map ψ to prove the equality of z and x', and to further include psd and s.

Code (line 21):

$$\text{let } psd'' = \mathsf{computePsd}\ x\ s;$$

Environment:

$$\underbrace{E_8', psd^o : \{y : unit \mid psd' = \mathsf{Revealed}\ y\}}_{=:E_9'}$$

Rules:

$$\dfrac{\textsc{Ext Exp Appl} \\ E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \leadsto E}$$

$$\dfrac{\textsc{Ext Exp Let} \\ E \vdash A : T \leadsto E'' \qquad E, x : T \vdash B : U \leadsto E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \leadsto E'}$$

Proof Obligations:

1. $E'_9 \vdash \mathsf{computePsd} : \begin{array}{l}(sk : \mathit{sigkey}) \ \to \ (s : \mathit{string}) \ \to \\ \{x : \mathit{pseudo} \mid \exists y, z.\ sk = (y, z) \wedge \mathsf{SSP}(z, s, x)\}\end{array}$
2. $E'_9 \vdash x : \mathit{sigkey}$
3. $E'_9 \vdash s : \mathit{string}$
4. $\mathit{psd}'' \notin \mathit{fv}(T^{\mathcal{F}^\wedge})$
5. $E'_9, \mathit{psd}'' : \{z : \mathit{pseudo} \mid \exists y', y.\ x = (y', y) \wedge \mathsf{SSP}(y, s, z)\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since E'_9 is basic. The second and third obligations hold since $\{x : \mathit{sigkey}, s : \mathit{string}\} \subset E'_9$. The fourth obligation follows since psd'' does not occur in $T^{\mathcal{F}^\wedge}$.

Code (line 22):

$$\text{if } \mathit{psd}'' = \mathit{psd} \text{ then}$$

Environment:

$$\underbrace{E'_9, \mathit{psd}'' : \{z : \mathit{pseudo} \mid \exists y', y.\ x = (y', y) \wedge \mathsf{SSP}(y, s, z)\}}_{=:E'_{10}}$$

Rules:

$$\dfrac{\textsc{Ext Exp If} \\ E \vdash M : T \qquad E \vdash N : U \qquad E, \{M = N\} \vdash A : V \leadsto E' \qquad E \vdash B : V \leadsto E''}{E \vdash \mathsf{if}\ M = N\ \mathsf{then}\ A\ \mathsf{else}\ B : V \leadsto E'}$$

Proof Obligations:

1. $E'_{10} \vdash \mathit{psd}'' : \{z : \mathit{pseudo} \mid \exists y', y.\ x = (y', y) \wedge \mathsf{SSP}(y, s, z)\}$
2. $E'_{10} \vdash \mathit{psd} : \mathit{pseudo}$
3. $E'_{10}, \{\mathit{psd}'' = \mathit{psd}\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since psd'' was added with this type in the step above. The second obligation holds since $\mathit{psd} : \mathit{pseudo} \in E'_{10}$.

Code (line 23):

$$\bullet$$

Environment:

$$\underbrace{E'_{10}, \{psd'' = psd\}}_{=:E'_{11}}$$

Rules:

$$\frac{\textsc{Ext Exp Context}^{\mathcal{F}}}{E \vdash \bullet : \{x : bool \mid \forall \tilde{z}.\ y = \underline{\mathcal{F}}\ \wedge\ x = \mathsf{true} \implies \mathcal{F}\} \rightsquigarrow E}$$

Proof Obligations:

none

We let $\phi' := \phi$ and $\psi' := \psi$ and argue why the current typing environment and maps are almost appropriate:

(1) $E' \vdash C' : \{x : bool \mid \forall \tilde{z}.\ y = \underline{\mathcal{F}^{\wedge} \wedge \mathcal{F}^{e}}\ \wedge\ x = \mathsf{true} \implies \mathcal{F}^{\wedge} \wedge \mathcal{F}^{e}\} \rightsquigarrow E''$:
We conducted the type-checking and we have derived that $E'' := E'_{11}$. We stress that this type only holds due to the typing rule $\textsc{Exp Context}^{\mathcal{F}}$. The typing environment cannot prove that $y = \underline{\mathcal{F}^{\wedge} \wedge \mathcal{F}^{e}}$. The final code, however, will be able to prove this type.

(2) E is basic and $E'' \vdash \diamond$:
We never double-bind values and all the free names and variables of a type are closed inside of the typing environment.

(3) $\mathsf{ext}(\mathcal{F}^{\wedge} \wedge \mathcal{F}^{e}, \psi', \phi')$:
From $\mathsf{appropriate}(E, E', \mathcal{F}^{\wedge}, \psi, \phi, stm, f)$, we immediately obtain that $\mathsf{ext}(\mathcal{F}^{\wedge}, \psi, \phi)$. The only modifications applied to ψ and ϕ are those described in the verification context. Since the verification context for logical conjunction and logical disjunction do not contain changes to ψ and ϕ, we conclude that $\mathsf{ext}(\mathcal{F}^{\wedge} \wedge \mathcal{F}^{e}, \psi', \phi')$.

(4) $\forall i \in \mathcal{I}_{vk}(\underline{\mathcal{F}^{\wedge} \wedge \mathcal{F}^{e}}).\ E'' \vdash \phi'(i) : verkey$:
For the variables in the formula $\mathcal{F}^{\wedge}$, this holds by the assumed appropriateness of E'. Since $\mathcal{I}_{vk}(\mathcal{F}^{e}) = \emptyset$, the obligation follows.

(5) $\forall i.\ \exists y.\ (x_i = \mathsf{Revealed}\ y \wedge y \in \psi(\mathcal{E}(i)))\ \vee\ (\exists z.\ x_i = \mathsf{Hidden}\ z \wedge y \in \psi(\mathcal{E}(i)))$:
If any of the user identifier z, the service description s, or the pseudonym psd are revealed, we will have executed lines 12 and U_0^{ψ}, lines 14 and U_1^{ψ}, and lines 16 and U_2^{ψ}, respectively. Consequently, we have derived that $\nu' = \mathsf{Revealed}\ \nu^{o}$ for $\nu \in \{z, s, psd\}$.

In either case, we add the variables z, s, and psd to ψ' in lines $U_3^{\psi} - U_5^{\psi}$. In particular, these variables will ensure that all values hidden by the same existential quantifier are represented by the same committed value (the equality between them is ensured by $\mathcal{E}$ that maps the indices of these variables to the same index).

(6) $E' \vdash stm' : statement$ and $E' \vdash f' : formula$:
This is an assumption.

(7) $E'' \vdash \{f' = \underline{\mathcal{F}^e}\}$ and

(8) $E'', (\psi')^{=} \vdash \mathcal{F}^\wedge \wedge \mathcal{F}^e$:
The assumed appropriateness yields that $E', (\psi)^{=} \vdash \mathcal{F}^\wedge$. Since E'' is an extension of E' and ψ' is an extension of ψ, we obtain that $E'', (\psi')^{=} \vdash \mathcal{F}^\wedge$.

For the formula $\mathcal{F}^e$ with the encoding $\underline{\mathcal{F}^e} = \mathsf{SSP}(z', s', psd')$, we consider the following formulas in E'' and the entries of ψ':

(a) lines 1-2: $\{f = \mathsf{SSP}(z', s', psd')\}$

(b) line 11: $\{(x'', x') = x\}$

(c) line 12-U_2^ψ: $\{z' = \mathsf{Revealed}\ z^o\}$, $\{s' = \mathsf{Revealed}\ s^o\}$, $\{psd' = \mathsf{Revealed}\ psd^o\}$ if the corresponding values are revealed.

(d) lines U_3^ψ-U_5^ψ:
$\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{z, x'\}]$
$\psi := \psi[\mathcal{E}(\omega + 1) \mapsto \psi(\mathcal{E}(\omega + 1)) \cup \{s\}]$
$\psi := \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{psd\}]$

(e) lines 21: $\{\exists y, z.\ x = (y, z) \wedge \mathsf{SSP}(z, s, psd'')\}$

(f) line 22: $\{psd'' = psd\}$

First, we note that by (e), (b), and (f), we derive

$$\mathsf{SSP}(z, s, psd).$$

More precisely, we derive the logical formula where all values are derived from the commitments attached to the proof. For the revealed arguments, we obtain the expected equalities by the respective entries into ψ' by (c); for all hidden values, we get the necessary equalities by (d). In particular, this substitution shows that $E'' \vdash \{f' = \underline{\mathcal{F}^e}\}$. Furthermore, it shows that $E'' \vdash \mathcal{F}^e$ since substituting all the "^{o}" variables for the revealed values and consistently replacing the committed values with existential quantified values proves $\mathcal{F}^e$. The equalities induced by ψ' enable us to consistently existentially quantify across $\mathcal{F}^\wedge$ and $\mathcal{F}^e$, yielding $E'' \vdash \mathcal{F}^\wedge \wedge \mathcal{F}^e$.

Case $\mathcal{F}^e \triangleq x\ op\ y$:

Code (line 3):

$$\text{let } tmp_x = \mathsf{openCommit}\ c_x;$$

Environment:

$$\underbrace{\begin{array}{l} E', c_x : commitment, op : string, c_y : commitment, \\ \quad \{stm = \mathsf{REL_p}(c_x, \_, op, \_, c_y)\} \\ x' : bitstring\ RevHid, op : string, y : bitstring\ RevHid, \\ \quad \{f = \mathsf{REL}(x', op, y')\} \end{array}}_{=:E_1'}$$

Rules:

$$\frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}\ \textsc{Ext Exp Appl}$$

$$\frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}\ \textsc{Ext Exp Let}$$

Proof Obligations:

1. $E'_1 \vdash \mathsf{openCommit} : \mathit{commitment} \to \mathit{bitstring} * \mathit{random}$
2. $E'_1 \vdash c_x : \mathit{commitment}$
3. $\mathit{tmp}_x \notin \mathit{fv}(T^{\mathcal{F}^\wedge})$
4. $E'_1, \mathit{tmp}_x : \mathit{bitstring} * \mathit{random} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

Since E'_1 is basic, the first obligation follows. The second obligation follows since $c_x : \mathit{commitment} \in E'_1$. The third obligation holds as the variable is fresh and does not occur in the return type.

Code (line 4):

$$\text{let } (x, r_x) = \mathit{tmp}_x;$$

Environment:

$$\underbrace{E'_1, \mathit{tmp}_x : \mathit{bitstring} * \mathit{random}}_{=:E'_2}$$

Rules:

$$\frac{\begin{array}{c} E \vdash M : (\Sigma x : T.\, U) \\ E, x : T, y : U, \_ : \{(x, y) = M\} \vdash A : V \rightsquigarrow E' \qquad \{x, y\} \cap \mathit{fv}(V) = \emptyset \end{array}}{E \vdash \mathsf{let}\ (x, y) = M;\ A : V \rightsquigarrow E'}\ \textsc{Ext Exp Split}$$

Proof Obligations:

1. $E'_2 \vdash \mathit{tmp}_x : \mathit{bitstring} * \mathit{random}$
2. $\{x, r_x\} \cap \mathit{fv}(T^{\mathcal{F}^\wedge}) = \emptyset$
3. $E'_2, x : \mathit{bitstring}, r_x : \mathit{random}, \{(x, r_x) = \mathit{tmp}_x\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since $\mathit{tmp}_x : \mathit{bitstring} * \mathit{random} \in E'_2$. The second obligation holds as the variables do not occur in the return type.

Code (line 5):

$$\text{let } \mathit{tmp}_y = \mathsf{openCommit}\ c_y;$$

Code (line 6):

$$\text{let } (y, r_y) = \mathit{tmp}_y;$$

These steps are analogous to those for lines 3 and 4.

We will type-check the code as if all values were revealed.

Code (line 7):

$$\text{let } x^o = \mathsf{getRevealed}\ x';$$

Environment:

$$\underbrace{\begin{array}{l} E'_2, x : \mathit{bitstring}, r_x : \mathit{random}, \{(x, r_x) = \mathit{tmp}_x\}, \\ \mathit{tmp}_y : \mathit{bitstring} * \mathit{random}, y : \mathit{bitstring}, r_y : \mathit{random}, \{(y, r_y) = \mathit{tmp}_y\} \end{array}}_{=:E'_3}$$

Rules:

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_3 \vdash \mathsf{getRevealed} : (x : \mathit{unit\ RevHid}) \;\to\; \{y : \mathit{unit} \mid x = \mathsf{Revealed}\ y\}$
2. $E'_3 \vdash x' : \mathit{bitstring\ RevHid}$
3. $x^o \notin \mathit{fv}(T^{\mathcal{F}^\wedge})$
4. $E'_3, x^o : \{y : \mathit{unit} \mid x' = \mathsf{Revealed}\ y\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since E'_3 is basic. The second obligation holds since $x' : \mathit{bitstring\ RevHid} \in E'_3$. The third obligation follows since x^o does not occur in $T^{\mathcal{F}^\wedge}$.

Code (line U_0^ψ):

$$\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x^o\}]$$

We extend the map ψ to include x^o.

Code (line 9):

$$\text{let } y^o = \mathsf{getRevealed}\ y';$$

Environment:

$$\underbrace{E_3', x^o : \{y : \mathit{unit} \mid x' = \mathsf{Revealed}\ y\}}_{=:E_4'}$$

Rules:

$$\frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}\ \textsc{Ext Exp Appl}$$

$$\frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}\ \textsc{Ext Exp Let}$$

Proof Obligations:

1. $E_4' \vdash \mathsf{getRevealed} : (x : \mathit{unit\ RevHid}) \ \rightarrow\ \{y : \mathit{unit} \mid x = \mathsf{Revealed}\ y\}$
2. $E_4' \vdash y' : \mathit{bitstring\ RevHid}$
3. $y^o \notin \mathit{fv}(T^{\mathcal{F}^\wedge})$
4. $E_4', y^o : \{y : \mathit{unit} \mid y' = \mathsf{Revealed}\ y\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since E_4' is basic. The second obligation holds since $y' : \mathit{bitstring\ RevHid} \in E_4'$. The third obligation follows since y^o does not occur in $T^{\mathcal{F}^\wedge}$.

Code (line U_1^ψ):

$$\psi := \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{y^o\}]$$

We extend the map ψ to include y^o.

Code (line U_2^ψ-U_3^ψ):

$$\begin{aligned}\psi &:= \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x\}]\\ \psi &:= \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{y\}]\end{aligned}$$

We extend the map ψ to include the committed values x and y.

Code (line 13):

$$\text{let } b = \mathsf{func}_{op}^{r}\ x\ y;$$

Environment:

$$\underbrace{E_4', y^o : \{y : \mathit{unit} \mid y' = \mathsf{Revealed}\ y\}}_{=:E_5'}$$

Rules:

$$\frac{\begin{array}{l}\textsc{Ext Exp Appl}\\ E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T\end{array}}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\frac{\begin{array}{l}\textsc{Ext Exp Let}\\ E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)\end{array}}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E_5' \vdash \mathsf{func}^r_{op} : (x : \mathit{bitstring}) \ \rightarrow\ (y : \mathit{bitstring}) \ \rightarrow\ \{z : \mathit{bool} \mid z = \mathsf{true} \Leftrightarrow x\ op\ y\}$
2. $E_5' \vdash x : \mathit{bitstring}$
3. $E_5' \vdash y : \mathit{bitstring}$
4. $b \notin \mathit{fv}(T^{\mathcal{F}^\wedge})$
5. $E_5', b : \{z : \mathit{bool} \mid z = \mathsf{true} \Leftrightarrow x\ op\ y\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since E_5' is basic. The second and third obligations hold since $\{x : \mathit{bitstring}, y : \mathit{bitstring}\} \subset E_5'$ (recall that $\mathit{bitstring} = \mathit{unit}$). The fourth obligation follows since b does not occur in $T^{\mathcal{F}^\wedge}$.

Code (line 14):

$$\text{if } b = \mathsf{true} \text{ then}$$

Environment:

$$\underbrace{E_5', b : \{z : \mathit{bool} \mid z = \mathsf{true} \Leftrightarrow x\ op\ y\}}_{=:E_6'}$$

Rules:

$$\frac{\begin{array}{l}\textsc{Ext Exp If}\\ E \vdash M : T \qquad E \vdash N : U \qquad E, \{M = N\} \vdash A : V \rightsquigarrow E' \qquad E \vdash B : V \rightsquigarrow E''\end{array}}{E \vdash \mathsf{if}\ M = N\ \mathsf{then}\ A\ \mathsf{else}\ B : V \rightsquigarrow E'}$$

Proof Obligations:

1. $E_6' \vdash b : \{z : \mathit{bool} \mid z = \mathsf{true} \Leftrightarrow x\ op\ y\}$
2. $E_6' \vdash \mathsf{true} : \mathit{bool}$
3. $E_6', \{b = \mathsf{true}\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation holds since b is contained in E_6' with the proper type. The second obligation holds since $\mathsf{true} \triangleq \mathsf{inr}()$ is defined to be of type *bool*.

Code (line 15):

$$\text{if } op = op' \text{ then}$$

Environment:

$$\underbrace{E'_6, \{b = \mathsf{true}\}}_{=:E'_7}$$

Rules:

$$\textsc{Ext Exp If}\quad \frac{E \vdash M : T \qquad E \vdash N : U \qquad E, \{M = N\} \vdash A : V \rightsquigarrow E' \qquad E \vdash B : V \rightsquigarrow E''}{E \vdash \mathsf{if}\ M = N\ \mathsf{then}\ A\ \mathsf{else}\ B : V \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_7 \vdash \mathit{op} : \mathit{string}$
2. $E'_7 \vdash \mathit{op}' : \mathit{string}$
3. $E'_7, \{\mathit{op} = \mathit{op}'\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first and second obligation hold by inspecting the *statement* and the *formula* data type.

Code (line 16):

$$\bullet$$

Environment:

$$\underbrace{E'_7, \{\mathit{op} = \mathit{op}'\}}_{=:E'_8}$$

Rules:

$$\textsc{Ext Exp Context}^{\mathcal{F}}\quad \frac{}{E \vdash \bullet : \{x : \mathit{bool} \mid \forall \tilde{z}.\ y = \underline{\mathcal{F}}\ \wedge\ x = \mathsf{true} \implies \mathcal{F}\} \rightsquigarrow E}$$

Proof Obligations:

none

We let $\phi' := \phi$ and $\psi' := \psi$ and argue why the current typing environment and maps are almost appropriate:

(1) $E' \vdash C' : \{x : \mathit{bool} \mid \forall \tilde{z}.\ y = \underline{\mathcal{F}^\wedge \wedge \mathcal{F}^e}\ \wedge\ x = \mathsf{true} \implies \mathcal{F}^\wedge \wedge \mathcal{F}^e\} \rightsquigarrow E''$:
We conducted the type-checking and we have derived that $E'' := E'_8$. We stress that this type only holds due to the typing rule Exp Context$^{\mathcal{F}}$. The typing environment cannot prove that $y = \underline{\mathcal{F}^\wedge \wedge \mathcal{F}^e}$. The final code, however, will be able to prove this type.

(2) E is basic and $E'' \vdash \diamond$:
We never double-bind values and all the free names and variables of a type are closed inside of the typing environment.

(3) $\mathsf{ext}(\mathcal{F}^\wedge \wedge \mathcal{F}^e, \psi', \phi')$:
From $\mathsf{appropriate}(E, E', \mathcal{F}^\wedge, \psi, \phi, \mathit{stm}, f)$, we immediately obtain that $\mathsf{ext}(\mathcal{F}^\wedge, \psi, \phi)$. The only modifications applied to ψ and ϕ are those described in the verification context. Since the verification context for logical conjunction and logical disjunction do not contain changes to ψ and ϕ, we conclude that $\mathsf{ext}(\mathcal{F}^\wedge \wedge \mathcal{F}^e, \psi', \phi')$.

(4) $\forall i \in \mathcal{I}_{vk}(\underline{\mathcal{F}^\wedge \wedge \mathcal{F}^e}).\ E'' \vdash \phi'(i) : \mathit{verkey}$:
For the variables in the formula $\mathcal{F}^\wedge$, this holds by the assumed appropriateness of E'. Since $\mathcal{I}_{vk}(\mathcal{F}^e) = \emptyset$, the obligation follows.

(5) $\forall i.\ \exists y.\ (x_i = \mathsf{Revealed}\ y \wedge y \in \psi(\mathcal{E}(i)))\ \vee\ (\exists z.\ x_i = \mathsf{Hidden}\ z \wedge y \in \psi(\mathcal{E}(i)))$:
If any of the committed values x and y are revealed, we have executed lines 7 and U_0^ψ, and lines 9 and U_1^ψ, respectively. Consequently, we have derived that $x' = \mathsf{Revealed}\ x^o$ and $y' = \mathsf{Revealed}\ y^o$ and put x^o and y^o into ψ.

In either case, we add the variables x and y to ψ' in lines $U_2^\psi - U_3^\psi$. In particular, these variables will ensure that all values hidden by the same existential quantifier are represented by the same committed value (the equality between them is ensured by the map $\mathcal{E}$).

(6) $E' \vdash \mathit{stm}' : \mathit{statement}$ and $E' \vdash f' : \mathit{formula}$:
This is an assumption.

(7) $E'' \vdash \{f' = \underline{\mathcal{F}^e}\}$ and

(8) $E'', (\psi')^= \vdash \mathcal{F}^\wedge \wedge \mathcal{F}^e$:
The assumed appropriateness yield that $E', (\psi)^= \vdash \mathcal{F}^\wedge$. Since E'' is an extension of E' and ψ' is an extension of ψ, we obtain that $E'', (\psi')^= \vdash \mathcal{F}^\wedge$.

For the formula $\mathcal{F}^e$ with the encoding $\underline{\mathcal{F}^e} = \mathsf{REL}(x', \mathit{op}, y')$, we consider the following formulas in E'' and the entries of ψ':

(a) lines 1-2: $\{f = \mathsf{REL}(x', \mathit{op}, y')\}$

(b) lines 7 and U_0^ψ:
$\{x' = \mathsf{Revealed}\ x^o\}$
$\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x^o\}]$
if the corresponding value is revealed.

(c) lines 9 and U_1^ψ:
$\{y' = \mathsf{Revealed}\ y^o\}$
$\psi := \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{y^o\}]$
if the corresponding value is revealed.

(d) lines U_2^ψ-U_3^ψ:
$\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x\}]$
$\psi := \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{y\}]$

(e) line 13: $\{b = \mathsf{true} \Leftrightarrow x\ \mathit{op}\ y\}$

(f) line 14: $\{b = \mathsf{true}\}$

First, we note that from (e) and (f), we derive

$$x \; op \; y.$$

More precisely, we derive the logical formula where all values are derived from the commitments attached to the proof. For the revealed arguments, we obtain the expected equalities by the respective entries into ψ' by (b) and (c); for all hidden values, we get the necessary equalities by (d). In particular, this substitution shows that $E'' \vdash \{f' = \underline{\mathcal{F}^e}\}$. Furthermore, it shows that $E'' \vdash \mathcal{F}^e$ since substituting all the "^{o}" variables for the revealed values and consistently replacing the committed values with existential quantified values proves $\mathcal{F}^e$. The equalities induced by ψ' enable us to consistently existentially quantify across $\mathcal{F}^\wedge$ and $\mathcal{F}^e$, yielding $E'' \vdash \mathcal{F}^\wedge \wedge \mathcal{F}^e$.

Case $\mathcal{F}^e \triangleq x \; op \; y = z$: The macro for this case is very close to the macro for relations. The only addition is argument z for which the corresponding commitments needs to be opened, possibly extracted if z is revealed in the formula, and added to the map ψ. Apart from that, the case is analogous to that of the relational proofs.

Case $\mathcal{F}^e \triangleq (x, b) \in \ell$:

Code (line 3):

$$\text{let } tmp_x = \mathsf{openCommit}\ c_x;$$

Environment:

$$\underbrace{\begin{array}{l} E', c_x : commitment, c_b : commitment, \ell : (pseudo * bitstring)\ list \\ \quad \{stm = \mathsf{REL_p}(c_x, \_, c_b, \_, \ell)\} \\ x' : bitstring\ RevHid, b' : bitstring\ RevHid, \ell' : (pseudo * bitstring)\ list \\ \quad \{f = \mathsf{REL}(x', b', \ell')\} \end{array}}_{=:E'_1}$$

Rules:

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin fv(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_1 \vdash \mathsf{openCommit} : commitment \to bitstring * random$
2. $E'_1 \vdash c_x : commitment$
3. $tmp_x \notin fv(T^{\mathcal{F}^\wedge})$
4. $E'_1, tmp_x : bitstring * random \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

Since E_1' is basic, the first obligation follows. The second obligation follows since $c_x : \mathit{commitment} \in E_1'$. The third obligation holds as tmp_x does not occur in $T^{\mathcal{F}^\wedge}$.

Code (line 4):

$$\text{let } (x, r_x) = \mathit{tmp}_x;$$

Environment:

$$\underbrace{E_1', \mathit{tmp}_x : \mathit{bitstring} * \mathit{random}}_{=:E_2'}$$

Rules:

Ext Exp Split

$$\frac{E \vdash M : (\Sigma x : T.\ U) \qquad E, x : T, y : U, \_ : \{(x, y) = M\} \vdash A : V \rightsquigarrow E' \qquad \{x, y\} \cap \mathit{fv}(V) = \emptyset}{E \vdash \mathsf{let}\ (x, y) = M;\ A : V \rightsquigarrow E'}$$

Proof Obligations:

1. $E_2' \vdash \mathit{tmp}_x : \mathit{bitstring} * \mathit{random}$
2. $\{x, r_x\} \cap \mathit{fv}(T^{\mathcal{F}^\wedge}) = \emptyset$
3. $E_2', x : \mathit{bitstring}, r_x : \mathit{random}, \{(x, r_x) = \mathit{tmp}_x\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since $\mathit{tmp}_x : \mathit{bitstring} * \mathit{random} \in E_2'$. The second obligation holds since the variables do not occur in the return type.

Code (line 5):

$$\text{let } \mathit{tmp}_b = \mathsf{openCommit}\ c_b;$$

Code (line 6):

$$\text{let } (b, r_b) = \mathit{tmp}_b;$$

This steps are analogous to lines 3 and 4.

We will type-check the code as if all values were revealed.

Code (line 7):

$$\text{let } x^o = \mathsf{getRevealed}\ x';$$

Environment:

$$\underbrace{\begin{array}{l} E_2', x : \mathit{bitstring}, r_x : \mathit{random}, \{(x, r_x) = \mathit{tmp}_x\} \\ \mathit{tmp}_b : \mathit{bitstring} * \mathit{random}, b : \mathit{bitstring}, r_b : \mathit{random}, \{(b, r_b) = \mathit{tmp}_b\} \end{array}}_{=:E_3'}$$

Rules:

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E_3' \vdash \mathsf{getRevealed} : (x : \mathit{unit\ RevHid}) \ \rightarrow\ \{y : \mathit{unit} \mid x = \mathsf{Revealed}\ y\}$
2. $E_3' \vdash x' : \mathit{bitstring\ RevHid}$
3. $x^o \notin \mathit{fv}(T^{\mathcal{F}^\wedge})$
4. $E_3', x^o : \{y : \mathit{unit} \mid x' = \mathsf{Revealed}\ y\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since E_3' is basic. The second obligation holds since $x' : \mathit{bitstring\ RevHid} \in E_3'$. The third obligation follows since x^o does not occur in $T^{\mathcal{F}^\wedge}$.

Code (line U_0^ψ):

$$\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x^o\}]$$

We extend the map ψ to include x^o.

Code (line 9):

$$\text{let } b^o = \mathsf{getRevealed}\ b';$$

Environment:

$$\underbrace{E_3', x^o : \{y : \mathit{unit} \mid x' = \mathsf{Revealed}\ y\}}_{=:E_4'}$$

Rules:

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E_4' \vdash \mathsf{getRevealed} : (x : \mathit{unit\ RevHid}) \ \rightarrow\ \{y : \mathit{unit} \mid x = \mathsf{Revealed}\ y\}$
2. $E_4' \vdash b' : \mathit{bitstring\ RevHid}$
3. $b^o \notin \mathit{fv}(T^{\mathcal{F}^\wedge})$
4. $E_4', b^o : \{y : \mathit{unit} \mid b' = \mathsf{Revealed}\ y\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since E'_4 is basic. The second obligation holds since $b' : \mathit{bitstring}\ \mathit{RevHid} \in E'_4$. The third obligation follows since b^o does not occur in $T^{\mathcal{F}^\wedge}$.

Code (line U^ψ_1):

$$\psi := \psi[\mathcal{E}(\omega+1) \mapsto \psi(\mathcal{E}(\omega+1)) \cup \{b^o\}]$$

We extend the map ψ to include b^o.

Code (line U^ψ_2-U^ψ_3):

$$\begin{array}{l} \psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x\}] \\ \psi := \psi[\mathcal{E}(\omega+1) \mapsto \psi(\mathcal{E}(\omega+1)) \cup \{b\}] \\ \psi := \psi[\mathcal{E}(\omega+2) \mapsto \psi(\mathcal{E}(\omega+2)) \cup \{\ell, \ell'\}] \end{array}$$

We extend the map ψ to include the committed values x and b. Additionally, we use ψ to enforce that the list ℓ contained in the proof and the list ℓ' provided with the formula coincide.

Code (line 14):

$$\text{let } r = \mathsf{List.member}^{(2,2)}\langle \mathit{pseudo} * \mathit{bitstring}\rangle\ x\ b\ \ell;$$

Environment:

$$\underbrace{E'_4, b^o : \{y : \mathit{unit} \mid b' = \mathsf{Revealed}\ y\}}_{=:E'_5}$$

Rules:

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\ U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_5 \vdash \mathsf{List.member}^{(2,2)}\langle \mathit{pseudo} * \mathit{bitstring}\rangle : (x : \mathit{pseudo}) \to (b : \mathit{bitstring}) \to$ $(\ell : (\mathit{pseudo} * \mathit{bitstring})\ \mathit{list}) \to \{z : \mathit{bool} \mid z = \mathsf{true} \Leftrightarrow (x, b) \in \ell\}$
2. $E'_5 \vdash x : \mathit{pseudo}$
3. $E'_5 \vdash b : \mathit{bitstring}$
4. $r \notin \mathit{fv}(T^{\mathcal{F}^\wedge})$
5. $E'_5, r : \{z : \mathit{bool} \mid z = \mathsf{true} \Leftrightarrow (x, b) \in \ell\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since E'_5 is basic. The second and third obligations hold since $\{x : \mathit{pseudo}, b : \mathit{bitstring}\} \subset E'_5$. The fourth obligation follows since r does not occur in $T^{\mathcal{F}^\wedge}$.

Code (line 15):

$$\text{if } r = \mathsf{true} \text{ then}$$

Environment:

$$\underbrace{E'_5, r : \{z : \mathit{bool} \mid z = \mathsf{true} \Leftrightarrow (x, b) \in \ell\}}_{=:E'_6}$$

Rules:

Ext Exp If

$$\frac{E \vdash M : T \qquad E \vdash N : U \qquad E, \{M = N\} \vdash A : V \rightsquigarrow E' \qquad E \vdash B : V \rightsquigarrow E''}{E \vdash \mathsf{if}\ M = N\ \mathsf{then}\ A\ \mathsf{else}\ B : V \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_6 \vdash r : \{z : \mathit{bool} \mid z = \mathsf{true} \Leftrightarrow (x, b) \in \ell\}$
2. $E'_6 \vdash \mathsf{true} : \mathit{bool}$
3. $E'_6, \{r = \mathsf{true}\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation holds since r is in E'_6 with the proper type. The second obligation holds since $\mathsf{true} \triangleq \mathsf{inr}()$ is defined to be of type *bool*.

Code (line 16):

$$\bullet$$

Environment:

$$\underbrace{E'_6, \{r = \mathsf{true}\}}_{=:E'_7}$$

Rules:

Ext Exp Context$^{\mathcal{F}}$

$$\frac{}{E \vdash \bullet : \{x : \mathit{bool} \mid \forall \tilde{z}.\ y = \underline{\mathcal{F}}\ \wedge\ x = \mathsf{true} \implies \mathcal{F}\} \rightsquigarrow E}$$

Proof Obligations:

none

We let $\phi' := \phi$ and $\psi' := \psi$ and argue why the current typing environment and maps are almost appropriate:

(1) $E' \vdash C' : \{x : \mathit{bool} \mid \forall \tilde{z}.\ y = \underline{\mathcal{F}^\wedge \wedge \mathcal{F}^e}\ \wedge\ x = \mathsf{true} \implies \mathcal{F}^\wedge \wedge \mathcal{F}^e\} \rightsquigarrow E''$:
We conducted the type-checking and we have derived that $E'' := E'_7$. We stress that this type only holds due to the typing rule Exp Context$^{\mathcal{F}}$. The typing environment cannot prove that $y = \underline{\mathcal{F}^\wedge \wedge \mathcal{F}^e}$. The final code, however, will be able to prove this type.

(2) E is basic and $E'' \vdash \diamond$:
We never double-bind values and all the free names and variables of a type are closed inside of the typing environment.

(3) $\mathsf{ext}(\mathcal{F}^\wedge \wedge \mathcal{F}^e, \psi', \phi')$:
From $\mathsf{appropriate}(E, E', \mathcal{F}^\wedge, \psi, \phi, \mathit{stm}, f)$, we immediately obtain that $\mathsf{ext}(\mathcal{F}^\wedge, \psi, \phi)$. The only modifications applied to ψ and ϕ are those described in the verification context. Since the verification context for logical conjunction and logical disjunction do not contain changes to ψ and ϕ, we conclude that $\mathsf{ext}(\mathcal{F}^\wedge \wedge \mathcal{F}^e, \psi', \phi')$.

(4) $\forall i \in \mathcal{I}_{vk}(\underline{\mathcal{F}^\wedge \wedge \mathcal{F}^e}).\ E'' \vdash \phi'(i) : \mathit{verkey}$:
For the variables in the formula $\mathcal{F}^\wedge$, this holds by the assumed appropriateness of E'. Since $\mathcal{I}_{vk}(\mathcal{F}^e) = \emptyset$, the obligation follows.

(5) $\forall i.\ \exists y.\ (x_i = \mathsf{Revealed}\ y \wedge y \in \psi(\mathcal{E}(i)))\ \vee\ (\exists z.\ x_i = \mathsf{Hidden}\ z \wedge y \in \psi(\mathcal{E}(i)))$:
If any of the committed variables x and b are revealed, we have executed lines 7 and U_0^ψ, and lines 9 and U_1^ψ, respectively. Consequently, we have derived that $x' = \mathsf{Revealed}\ x^o$ and $y' = \mathsf{Revealed}\ y^o$ and added x^o and y^o to ψ.

In either case, we add the variables x and b to ψ' in lines U_2^ψ-U_3^ψ. In particular, these variables will ensure that all values hidden by the same existential quantifier are represented by the same committed value (the equality between them is ensured by the map $\mathcal{E}$).

(6) $E' \vdash \mathit{stm}' : \mathit{statement}$ and $E' \vdash f' : \mathit{formula}$:
This is an assumption.

(7) $E'' \vdash \{f' = \underline{\mathcal{F}^e}\}$ and

(8) $E'', (\psi')^= \vdash \mathcal{F}^\wedge \wedge \mathcal{F}^e$:
The assumed appropriateness yield that $E', (\psi)^= \vdash \mathcal{F}^\wedge$. Since E'' is an extension of E' and ψ' is an extension of ψ, we obtain that $E'', (\psi')^= \vdash \mathcal{F}^\wedge$.

For the formula $\mathcal{F}^e$ with the encoding $\underline{\mathcal{F}^e} = \mathsf{LM}(x', b', \ell')$, we consider the following formulas in E'' and the entries of ψ':

(a) lines 1-2: $\{f = \mathsf{LM}(x', b', \ell')\}$

(b) line 7 and U_0^ψ:
$\{x' = \mathsf{Revealed}\ x^o\}$
$\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x^o\}]$ if the corresponding value is revealed.

(c) line 9 and U_1^ψ:
$\{b' = \mathsf{Revealed}\ b^o\}$
$\psi := \psi[\mathcal{E}(\omega + 1) \mapsto \psi(\mathcal{E}(\omega + 1)) \cup \{b^o\}]$ if the corresponding value is revealed.

(d) lines U_2^ψ-U_4^ψ:
$\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x\}]$
$\psi := \psi[\mathcal{E}(\omega + 1) \mapsto \psi(\mathcal{E}(\omega + 1)) \cup \{b\}]$
$\psi := \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{\ell, \ell'\}]$

(e) line 13: $\{r = \mathsf{true} \Leftrightarrow (x, b) \in \ell\}$

(f) line 14: $\{r = \mathsf{true}\}$

First, we note that by (e) and (f), we derive

$$(x, b) \in \ell.$$

More precisely, we derive the logical formula where all values are derived from the commitments attached to the proof. For the revealed arguments, we obtain the expected equalities by the respective entries into ψ' by (b) and (c); for all hidden values, we get the necessary equalities by (d). In particular, this substitution shows that $E'' \vdash \{f' = \underline{\mathcal{F}^e}\}$. Furthermore, it shows that $E'' \vdash \mathcal{F}^e$ since substituting all the "^{o}" variables for the revealed values and consistently replacing the committed values with existential quantified values proves $\mathcal{F}^e$. The equalities induced by ψ' enable us to consistently existentially quantify across $\mathcal{F}^\wedge$ and $\mathcal{F}^e$, yielding $E'' \vdash \mathcal{F}^\wedge \wedge \mathcal{F}^e$.

Case $\mathcal{F}^e \triangleq \forall b.\ (x, b) \notin \ell$:

Code (line 3):

$$\text{let } tmp_x = \mathsf{openCommit}\ c_x;$$

Environment:

$$\underbrace{\begin{array}{l} E', c_x : commitment, c_b : commitment, \ell : (pseudo * bitstring)\ list \\ \quad \{stm = \mathsf{REL_p}(c_x, \_, c_b, \_, \ell)\} \\ x' : bitstring\ RevHid, b' : bitstring\ RevHid, \ell' : (pseudo * bitstring)\ list \\ \quad \{f = \mathsf{REL}(x', b', \ell')\} \end{array}}_{=:E'_1}$$

Rules:

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\ U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin fv(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_1 \vdash \mathsf{openCommit} : commitment \to bitstring * random$
2. $E'_1 \vdash c_x : commitment$
3. $tmp_x \notin fv(T^{\mathcal{F}^\wedge})$
4. $E'_1, tmp_x : bitstring * random \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

Since E'_1 is basic, the first obligation follows. The second obligation follows since $c_x : commitment \in E'_1$. The third obligation holds since the variable does not occur in the return type.

Code (line 4):

$$\text{let } (x, r_x) = \mathit{tmp}_x;$$

Environment:

$$\underbrace{E'_1, \mathit{tmp}_x : \mathit{bitstring} * \mathit{random}}_{=:E'_2}$$

Rules:

$$\frac{\textsc{Ext Exp Split} \qquad E \vdash M : (\Sigma x : T.\ U) \qquad E, x : T, y : U, \_ : \{(x, y) = M\} \vdash A : V \rightsquigarrow E' \qquad \{x, y\} \cap \mathit{fv}(V) = \emptyset}{E \vdash \mathsf{let}\ (x, y) = M;\ A : V \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_2 \vdash \mathit{tmp}_x : \mathit{bitstring} * \mathit{random}$
2. $\{x, r_x\} \cap \mathit{fv}(T^{\mathcal{F}^\wedge}) = \emptyset$
3. $E'_2, x : \mathit{bitstring}, r_x : \mathit{random}, \{(x, r_x) = \mathit{tmp}_x\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since $\mathit{tmp}_x : \mathit{bitstring} * \mathit{random} \in E'_2$. The second obligation holds since the variables do not occur in the return type.

We will type-check the code as if the value x' was revealed.

Code (line 5):

$$\text{let } x^o = \mathsf{getRevealed}\ x';$$

Environment:

$$\underbrace{E'_2, x : \mathit{bitstring}, r_x : \mathit{random}, \{(x, r_x) = \mathit{tmp}_x\}}_{=:E'_3}$$

Rules:

$$\frac{\textsc{Ext Exp Appl} \qquad E \vdash M : (\Pi x : T.\ U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\frac{\textsc{Ext Exp Let} \qquad E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_3 \vdash \mathsf{getRevealed} : (x : \mathit{unit}\ \mathit{RevHid}) \ \rightarrow\ \{y : \mathit{unit} \mid x = \mathsf{Revealed}\ y\}$
2. $E'_3 \vdash x' : \mathit{bitstring}\ \mathit{RevHid}$
3. $x^o \notin \mathit{fv}(T^{\mathcal{F}^\wedge})$
4. $E'_3, x^o : \{y : \mathit{unit} \mid x' = \mathsf{Revealed}\ y\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since E_3' is basic. The second obligation holds since x' : *bitstring RevHid* $\in E_3'$. The third obligation follows since x^o does not occur in $T^{\mathcal{F}^\wedge}$.

Code (line U_0^ψ):

$$\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x^o\}]$$

We extend the map ψ to include x^o.

Code (line U_1^ψ-U_2^ψ):

$$\begin{array}{l} \psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x\}] \\ \psi := \psi[\mathcal{E}(\omega+1) \mapsto \psi(\mathcal{E}(\omega+1)) \cup \{\ell, \ell'\}] \end{array}$$

We extend the map ψ to include the committed value x. Additionally, we use ψ to enforce that the ℓ list contained in the proof and the list ℓ' provided with the formula coincide.

Code (line 9):

$$\text{let } r = \mathsf{List.member}^{(1,2)}\langle \mathit{pseudo} * \mathit{bitstring}\rangle\ x\ \ell;$$

Environment:

$$\underbrace{E_3', x^o : \{y : \mathit{unit} \mid x' = \mathsf{Revealed}\ y\}}_{=:E_4'}$$

Rules:

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E_4' \vdash \mathsf{List.member}^{(1,2)}\langle \mathit{pseudo} * \mathit{bitstring}\rangle : (x : \mathit{pseudo}) \to (b : \mathit{bitstring}) \to$ $(\ell : (\mathit{pseudo} * \mathit{bitstring})\ \mathit{list}) \to \{z : \mathit{bool} \mid z = \mathsf{true} \Leftrightarrow \exists b.\ (x, b) \in \ell\}$
2. $E_4' \vdash x : \mathit{pseudo}$
3. $r \notin \mathit{fv}(T^{\mathcal{F}^\wedge})$
4. $E_4', r : \{z : \mathit{bool} \mid z = \mathsf{true} \Leftrightarrow \exists b.\ (x, b) \in \ell\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since E_4' is basic. The second obligations hold since $x : \mathit{pseudo} \in E_4'$. The third obligation follows since r does not occur in $T^{\mathcal{F}^\wedge}$.

Code (line 10):

$$\text{if } r = \mathsf{false} \text{ then}$$

Environment:

$$\underbrace{E'_4, r : \{z : \mathit{bool} \mid z = \mathsf{true} \Leftrightarrow \exists b.\ (x, b) \in \ell\}}_{=:E'_5}$$

Rules:

$$\textsc{Ext Exp If}\quad \frac{E \vdash M : T \qquad E \vdash N : U \qquad E, \{M = N\} \vdash A : V \rightsquigarrow E' \qquad E \vdash B : V \rightsquigarrow E''}{E \vdash \mathsf{if}\ M = N\ \mathsf{then}\ A\ \mathsf{else}\ B : V \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_5 \vdash r : \{z : \mathit{bool} \mid z = \mathsf{true} \Leftrightarrow \exists b.\ (x, b) \in \ell\}$
2. $E'_5 \vdash \mathsf{false} : \mathit{bool}$
3. $E'_5, \{r = \mathsf{false}\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation holds since r is in E'_5 with the proper type. The second obligation holds since $\mathsf{false} \triangleq \mathsf{inl}()$ is defined to be of type *bool*.

Code (line 11):

$$\bullet$$

Environment:

$$\underbrace{E'_5, \{r = \mathsf{false}\}}_{=:E'_6}$$

Rules:

$$\textsc{Ext Exp Context}^{\mathcal{F}}\quad \frac{}{E \vdash \bullet : \{x : \mathit{bool} \mid \forall \tilde{z}.\ y = \underline{\mathcal{F}}\ \wedge\ x = \mathsf{true} \implies \mathcal{F}\} \rightsquigarrow E}$$

Proof Obligations:

none

We let $\phi' := \phi$ and $\psi' := \psi$ and argue why the current typing environment and maps are almost appropriate:

(1) $E' \vdash C' : \{x : \mathit{bool} \mid \forall \tilde{z}.\ y = \underline{\mathcal{F}^\wedge \wedge \mathcal{F}^e}\ \wedge\ x = \mathsf{true} \implies \mathcal{F}^\wedge \wedge \mathcal{F}^e\} \rightsquigarrow E''$:
We conducted the type-checking and we have derived that $E'' := E'_6$. We stress that this type only holds due to the typing rule Exp Context$^{\mathcal{F}}$. The typing environment cannot prove that $y = \underline{\mathcal{F}^\wedge \wedge \mathcal{F}^e}$. The final code, however, will be able to prove this type.

(2) E is basic and $E'' \vdash \diamond$:
We never double-bind values and all the free names and variables of a type are closed inside of the typing environment.

(3) $\mathsf{ext}(\mathcal{F}^\wedge \wedge \mathcal{F}^e, \psi', \phi')$:
From $\mathsf{appropriate}(E, E', \mathcal{F}^\wedge, \psi, \phi, \mathit{stm}, f)$, we immediately obtain that $\mathsf{ext}(\mathcal{F}^\wedge, \psi, \phi)$. The only modifications applied to ψ and ϕ are those described in the verification context. Since the verification context for logical conjunction and logical disjunction do not contain changes to ψ and ϕ, we conclude that $\mathsf{ext}(\mathcal{F}^\wedge \wedge \mathcal{F}^e, \psi', \phi')$.

(4) $\forall i \in \mathcal{I}_{vk}(\underline{\mathcal{F}^\wedge \wedge \mathcal{F}^e}).\ E'' \vdash \phi'(i) : \mathit{verkey}$:
For the variables in the formula $\mathcal{F}^\wedge$, this holds by the assumed appropriateness of E'. Since $\mathcal{I}_{vk}(\mathcal{F}^e) = \emptyset$, the obligation follows.

(5) $\forall i.\ \exists y.\ (x_i = \mathsf{Revealed}\ y \wedge y \in \psi(\mathcal{E}(i)))\ \vee\ (\exists z.\ x_i = \mathsf{Hidden}\ z \wedge y \in \psi(\mathcal{E}(i)))$:
If the committed variable x is revealed, we have executed lines 6 and U_0^ψ. Consequently, we have derived that $x' = \mathsf{Revealed}\ x^o$.

In either case, we add the variable x to ψ' in lines U_0^ψ. In particular, this variable will ensure that all values hidden by the same existential quantifier are represented by the same committed value (the equality between them is ensured by the map $\mathcal{E}$).

(6) $E' \vdash \mathit{stm}' : \mathit{statement}$ and $E' \vdash f' : \mathit{formula}$:
This is an assumption.

(7) $E'' \vdash \{f' = \underline{\mathcal{F}^e}\}$ and

(8) $E'', (\psi')^= \vdash \mathcal{F}^\wedge \wedge \mathcal{F}^e$:
The assumed appropriateness yield that $E', (\psi)^= \vdash \mathcal{F}^\wedge$. Since E'' is an extension of E' and ψ' is an extension of ψ, we obtain that $E'', (\psi')^= \vdash \mathcal{F}^\wedge$.

For the formula $\mathcal{F}^e$ with the encoding $\underline{\mathcal{F}^e} = \mathsf{LNM}(x', \ell')$, we consider the following formulas in E'' and the entries of ψ':

(a) lines 1-2: $\{f = \mathsf{LNM}(x', \ell')\}$

(b) line 5 and U_0^ψ:
$$\begin{array}{ll} \{x' = \mathsf{Revealed}\ x^o\} & \\ \psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x^o\}] & \text{if the corresponding value is revealed.} \end{array}$$

(c) lines U_1^ψ-U_2^ψ:
$$\begin{array}{l} \psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x\}] \\ \psi := \psi[\mathcal{E}(\omega + 1) \mapsto \psi(\mathcal{E}(\omega + 1)) \cup \{\ell, \ell'\}] \end{array}$$

(d) line 12: $\{r = \mathsf{true} \Leftrightarrow \exists b.\ (x, b) \in \ell\}$

(e) line 13: $\{r = \mathsf{false}\}$

First, we note that by (e) and (f), we derive

$$\begin{array}{rl} & r = \mathsf{true} \Leftrightarrow \exists b.\ (x, b) \in \ell(x, b) \in \ell \\ \Longleftrightarrow & r \neq \mathsf{true} \Leftrightarrow \neg \exists b.\ (x, b) \in \ell \\ \Longleftrightarrow & r = \mathsf{false} \Leftrightarrow \forall b.\ (x, b) \notin \ell \\ \overset{r=\mathsf{false}}{\Longrightarrow} & \forall b.\ (x, b) \notin \ell \end{array}$$

More precisely, we derive the logical formula where all values are derived from the commitments attached to the proof. For the revealed argument, we obtain the expected equality by (b); for all hidden values, we get the necessary equalities by (c). In particular, this substitution shows that $E'' \vdash \{f' = \underline{\mathcal{F}^e}\}$. Furthermore, it shows that $E'' \vdash \mathcal{F}^e$ since substituting x^o variable for the revealed values and consistently replacing the committed values with existential quantified values proves $\mathcal{F}^e$. The equalities induced by ψ' enable us to consistently existentially quantify across $\mathcal{F}^\wedge$ and $\mathcal{F}^e$, yielding $E'' \vdash \mathcal{F}^\wedge \wedge \mathcal{F}^e$.

Case $\mathcal{F}^e \triangleq \mathsf{EscrowInfo}(EA, A, R, s, idr)$:

Code (line 3):

$$\text{let } tmp_x = \mathsf{openCommit}\ c_x;$$

Environment:

$$\underbrace{\begin{array}{l} E', z : uid_{pub}, c_x : commitment, c_R : commitment, c_s : commitment, c_{idr}, c_r : commitment \\ \quad \{stm = \mathsf{EscrowInfo_p}(z, c_x, \_, c_R, \_, c_s, \_, c_{idr}, \_, c_r)\} \\ z' : uid_{pub}, x' : uid_{pub}\ RevHid, R' : bitstring\ RevHid, s' : string\ RevHid, \\ \quad idr' : pseudo\ RevHid, \{f = \mathsf{EscrowInfo}(z', x', R', s', idr')\} \end{array}}_{=:E'_1}$$

Rules:

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\ U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin fv(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_1 \vdash \mathsf{openCommit} : commitment \to bitstring * random$
2. $E'_1 \vdash c_x : commitment$
3. $tmp_x \notin fv(T^{\mathcal{F}^\wedge})$
4. $E'_1, tmp_x : bitstring * random \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

Since E'_1 is basic, the first obligation follows. The second obligation follows since $c_x : commitment \in E'_1$. The third obligation holds since the variable does not occur in the return type.

Code (line 4):

$$\text{let } (x, r_x) = \mathit{tmp}_x;$$

Environment:

$$\underbrace{E'_1, \mathit{tmp}_x : \mathit{bitstring} * \mathit{random}}_{=:E'_2}$$

Rules:

$$\textsc{Ext Exp Split}\quad \frac{\begin{array}{c} E \vdash M : (\Sigma x : T.\, U) \\ E, x : T, y : U, \_ : \{(x, y) = M\} \vdash A : V \rightsquigarrow E' \qquad \{x, y\} \cap \mathit{fv}(V) = \emptyset \end{array}}{E \vdash \mathsf{let}\ (x, y) = M;\ A : V \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_2 \vdash \mathit{tmp}_x : \mathit{bitstring} * \mathit{random}$
2. $\{x, r_x\} \cap \mathit{fv}(T^{\mathcal{F}^\wedge}) = \emptyset$
3. $E'_2, x : \mathit{bitstring}, r_x : \mathit{random}, \{(x, r_x) = \mathit{tmp}_x\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The second obligation follows since $\mathit{tmp}_x : \mathit{bitstring} * \mathit{random} \in E'_2$. The second obligation holds because the variables do not occur in the return type.

Code (line 5):

$$\text{let } \mathit{tmp}_R = \mathsf{openCommit}\ c_R;$$

Code (line 6):

$$\text{let } (R, r_R) = \mathit{tmp}_R;$$

Code (line 7):

$$\text{let } \mathit{tmp}_s = \mathsf{openCommit}\ c_s;$$

Code (line 8):

$$\text{let } (s, r_s) = \mathit{tmp}_s;$$

Code (line 9):

$$\text{let } \mathit{tmp}_{idr} = \mathsf{openCommit}\ c_{idr};$$

Code (line 10):

$$\text{let } (\mathit{idr}, r_{idr}) = \mathit{tmp}_{idr};$$

Code (line 11):

$$\text{let } \mathit{tmp}_r = \mathsf{openCommit}\ c_r;$$

Code (line 12):

$$\text{let } (r, r_r) = \mathit{tmp}_r;$$

The proofs for lines 5 through 12 are analogous to that of lines 3 and 4.

Code (line U_0^ψ):

$$\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{z'\}]$$

We extend the map ψ to include z'. Should the public identifier of the trusted third party occur several times in a proof, ψ enforces equality among them, i.e., the TTP will be represented by the same identifier across the proof.

We will type-check the code as if all values were revealed.

Code (line 14):

$$\text{let } x^o = \mathsf{getRevealed}\ x';$$

Environment:

$$\underbrace{\begin{array}{l} E_2', x : \mathit{bitstring}, r_x : \mathit{random}, \{(x, r_x) = \mathit{tmp}_x\} \\ \mathit{tmp}_R : \mathit{bitstring} * \mathit{random}, R : \mathit{bitstring}, r_R : \mathit{random}, \{(R, r_R) = \mathit{tmp}_R\} \\ \mathit{tmp}_s : \mathit{bitstring} * \mathit{random}, s : \mathit{bitstring}, r_s : \mathit{random}, \{(s, r_s) = \mathit{tmp}_s\} \\ \mathit{tmp}_{\mathit{idr}} : \mathit{bitstring} * \mathit{random}, \mathit{idr} : \mathit{bitstring}, r_{\mathit{idr}} : \mathit{random}, \{(\mathit{idr}, r_{\mathit{idr}}) = \mathit{tmp}_{\mathit{idr}}\} \\ \mathit{tmp}_r : \mathit{bitstring} * \mathit{random}, r : \mathit{bitstring}, r_r : \mathit{random}, \{(r, r_r) = \mathit{tmp}_r\} \end{array}}_{=:E_3'}$$

Rules:

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\ U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E_3' \vdash \mathsf{getRevealed} : (x : \mathit{unit\ RevHid}) \rightarrow \{y : \mathit{unit} \mid x = \mathsf{Revealed}\ y\}$
2. $E_3' \vdash x' : \mathit{uid}_{\mathit{pub}}\ \mathit{RevHid}$
3. $x^o \notin \mathit{fv}(T^{\mathcal{F}^\wedge})$
4. $E_3', x^o : \{y : \mathit{unit} \mid x' = \mathsf{Revealed}\ y\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since E_3' is basic. The second obligation holds since $x' : \mathit{uid}_{\mathit{pub}}\ \mathit{RevHid} \in E_3'$. The third obligation follows since x^o does not occur in $T^{\mathcal{F}^\wedge}$.

Code (line U_1^{ψ}):

$$\psi := \psi[\mathcal{E}(\omega+1) \mapsto \psi(\mathcal{E}(\omega+1)) \cup \{x^o\}]$$

We extend the map ψ to include x^o.

Code (line 16):

$$\text{let } R^o = \textsf{getRevealed } R';$$

Environment:

$$\underbrace{E_3', x^o : \{y : \mathit{unit} \mid x' = \textsf{Revealed } y\}}_{=:E_4'}$$

Rules:

$$\frac{\textsc{Ext Exp Appl} \quad E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\frac{\textsc{Ext Exp Let} \quad E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \textsf{let } x = A;\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E_4' \vdash \textsf{getRevealed} : (x : \mathit{unit\ RevHid}) \ \rightarrow\ \{y : \mathit{unit} \mid x = \textsf{Revealed } y\}$
2. $E_4' \vdash R' : \mathit{bitstring\ RevHid}$
3. $XXXR^o \notin \mathit{fv}(T^{\mathcal{F}^\wedge})$
4. $E_4', R^o : \{y : \mathit{unit} \mid R' = \textsf{Revealed } y\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since E_4' is basic. The second obligation holds since $R' : \mathit{bitstring\ RevHid} \in E_4'$. The third obligation follows since R^o does not occur in $T^{\mathcal{F}^\wedge}$.

Code (line U_2^{ψ}):

$$\psi := \psi[\mathcal{E}(\omega+2) \mapsto \psi(\mathcal{E}(\omega+2)) \cup \{R^o\}]$$

We extend the map ψ to include R^o.

Code (line 18):

$$\text{let } s^o = \textsf{getRevealed } s';$$

Environment:

$$\underbrace{E_4', s^o : \{y : \mathit{unit} \mid s' = \textsf{Revealed } y\}}_{=:E_5'}$$

Rules:

$$\dfrac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}\ \textsc{Ext Exp Appl}$$

$$\dfrac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}\ \textsc{Ext Exp Let}$$

Proof Obligations:

1. $E'_5 \vdash \mathsf{getRevealed} : (x : \mathit{unit\ RevHid}) \ \rightarrow\ \{y : \mathit{unit} \mid x = \mathsf{Revealed}\ y\}$
2. $E'_5 \vdash s' : \mathit{bitstring\ RevHid}$
3. $s^o \notin \mathit{fv}(T^{\mathcal{F}^\wedge})$
4. $E'_5, R^o : \{y : \mathit{unit} \mid s' = \mathsf{Revealed}\ y\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since E'_5 is basic. The second obligation holds since $s' : \mathit{bitstring\ RevHid} \in E'_5$. The third obligation follows since s^o does not occur in $T^{\mathcal{F}^\wedge}$.

Code (line U_3^ψ):

$$\psi := \psi[\mathcal{E}(\omega + 3) \mapsto \psi(\mathcal{E}(\omega + 3)) \cup \{s^o\}]$$

We extend the map ψ to include s^o.

Code (line 20):

$$\text{let } \mathit{idr}^o = \mathsf{getRevealed}\ \mathit{idr}';$$

Environment:

$$\underbrace{E'_5, \mathit{idr}^o : \{y : \mathit{unit} \mid \mathit{idr}' = \mathsf{Revealed}\ y\}}_{=:E'_6}$$

Rules:

$$\dfrac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}\ \textsc{Ext Exp Appl}$$

$$\dfrac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}\ \textsc{Ext Exp Let}$$

Proof Obligations:

1. $E'_6 \vdash \mathsf{getRevealed} : (x : \mathit{unit\ RevHid}) \ \rightarrow\ \{y : \mathit{unit} \mid x = \mathsf{Revealed}\ y\}$
2. $E'_6 \vdash \mathit{idr}' : \mathit{bitstring\ RevHid}$
3. $\mathit{idr}^o \notin \mathit{fv}(T^{\mathcal{F}^\wedge})$
4. $E'_6, \mathit{idr}^o : \{y : \mathit{unit} \mid \mathit{idr}' = \mathsf{Revealed}\ y\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since E'_6 is basic. The second obligation holds since $idr' : bitstring\ RevHid \in E'_6$. The third obligation follows since idr^o does not occur in $T^{\mathcal{F}^\wedge}$.

Code (line U_4^ψ):

$$\psi := \psi[\mathcal{E}(\omega + 4) \mapsto \psi(\mathcal{E}(\omega + 4)) \cup \{idr^o\}]$$

We extend the map ψ to include idr^o.

Code (lines U_5^ψ-U_9^ψ):

$$\begin{aligned}
&\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{z\}]\\
&\psi := \psi[\mathcal{E}(\omega + 1) \mapsto \psi(\mathcal{E}(\omega + 1)) \cup \{x\}]\\
&\psi := \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{R\}]\\
&\psi := \psi[\mathcal{E}(\omega + 3) \mapsto \psi(\mathcal{E}(\omega + 3)) \cup \{s\}]\\
&\psi := \psi[\mathcal{E}(\omega + 4) \mapsto \psi(\mathcal{E}(\omega + 4)) \cup \{idr\}]
\end{aligned}$$

We extend the map ψ to prove the equality of z and z', and to further include R, s, and idr.

Code (line 27):

$$\text{let } idr'' = \mathsf{computeIDR}\ z\ x\ r\ R\ s;$$

Environment:

$$\underbrace{E'_6, idr^o : \{y : unit \mid idr' = \mathsf{Revealed}\ y\}}_{=:E'_7}$$

Rules:

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin fv(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

Proof Obligations:

1. $E'_7 \vdash \mathsf{computeIDR} : (vk_{EA} : bitstring) \to (vk : bitstring) \to (r : bitstring) \to (R : bitstring) \to (s : string) \to \{idr : pseudo \mid \mathsf{EscrowInfo}(vk_{EA}, vk, R, s, idr)\}$
2. $E'_7 \vdash z : bitstring$
3. $E'_7 \vdash x : bitstring$
4. $E'_7 \vdash r : bitstring$
5. $E'_7 \vdash R : bitstring$
6. $E'_7 \vdash s : string$
7. $idr'' \notin fv(T^{\mathcal{F}^\wedge})$
8. $E'_7, idr'' : \{y : pseudo \mid \mathsf{EscrowInfo}(z, x, R, s, y)\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since E'_7 is basic. The second through the sixth obligations hold since the variables are contained in E'_7 with the proper types. The seventh obligation follows since idr'' does not occur in $T^{\mathcal{F}^\wedge}$.

Code (line 28):

$$\text{if } idr = idr'' \text{ then}$$

Environment:

$$\underbrace{E'_7, idr'' : \{z : pseudo \mid \mathsf{EscrowInfo}(z, x, R, s, y)\}}_{=:E'_8}$$

Rules:

$$\frac{E \vdash M : T \qquad E \vdash N : U \qquad E, \{M = N\} \vdash A : V \rightsquigarrow E' \qquad E \vdash B : V \rightsquigarrow E''}{E \vdash \mathsf{if}\ M = N\ \mathsf{then}\ A\ \mathsf{else}\ B : V \rightsquigarrow E'}\ \textsc{Ext Exp If}$$

Proof Obligations:

1. $E'_8 \vdash idr'' : \{z : pseudo \mid \mathsf{EscrowInfo}(z, x, R, s, y)\}$
2. $E'_8 \vdash idr : bitstring$
3. $E'_8, \{idr = idr''\} \vdash \mathcal{R} : T^{\mathcal{F}^\wedge}$

The first obligation follows since idr'' was added with this type in the step above. The second obligation holds since $idr : bitstring \in E'_8$.

Code (line 29):

$$\bullet$$

Environment:

$$\underbrace{E'_8, \{idr = idr''\}}_{=:E'_9}$$

Rules:

$$\frac{}{E \vdash \bullet : \{x : bool \mid \forall \tilde{z}.\ y = \underline{\mathcal{F}}\ \wedge\ x = \mathsf{true} \implies \mathcal{F}\} \rightsquigarrow E}\ \textsc{Ext Exp Context}^{\mathcal{F}}$$

Proof Obligations:

none

We let $\phi' := \phi$ and $\psi' := \psi$ and argue why the current typing environment and maps are almost appropriate:

(1) $E' \vdash C' : \{x : bool \mid \forall \tilde{z}.\ y = \underline{\mathcal{F}^\wedge \wedge \mathcal{F}^e}\ \wedge\ x = \mathsf{true} \implies \mathcal{F}^\wedge \wedge \mathcal{F}^e\} \rightsquigarrow E''$:
We conducted the type-checking and we have derived that $E'' := E'_9$. We stress that this type only holds due to the typing rule Exp Context$^{\mathcal{F}}$. The typing environment cannot prove that $y = \underline{\mathcal{F}^\wedge \wedge \mathcal{F}^e}$. The final code, however, will be able to prove this type.

(2) E is basic and $E'' \vdash \diamond$:
We never double-bind values and all the free names and variables of a type are closed inside of the typing environment.

(3) $\mathsf{ext}(\mathcal{F}^{\wedge} \wedge \mathcal{F}^{e}, \psi', \phi')$:
From $\mathsf{appropriate}(E, E', \mathcal{F}^{\wedge}, \psi, \phi, \mathit{stm}, f)$, we immediately obtain that $\mathsf{ext}(\mathcal{F}^{\wedge}, \psi, \phi)$. The only modifications applied to ψ and ϕ are those described in the verification context. Since the verification context for logical conjunction and logical disjunction do not contain changes to ψ and ϕ, we conclude that $\mathsf{ext}(\mathcal{F}^{\wedge} \wedge \mathcal{F}^{e}, \psi', \phi')$.

(4) $\forall i \in \mathcal{I}_{\mathit{vk}}(\underline{\mathcal{F}^{\wedge} \wedge \mathcal{F}^{e}})$. $E'' \vdash \phi'(i) : \mathit{verkey}$:
For the variables in the formula $\mathcal{F}^{\wedge}$, this holds by the assumed appropriateness of E'. Since $\mathcal{I}_{\mathit{vk}}(\mathcal{F}^{e}) = \emptyset$, the obligation follows.

(5) $\forall i.\ \exists y.\ (x_i = \mathsf{Revealed}\ y \wedge y \in \psi(\mathcal{E}(i)))\ \vee\ (\exists z.\ x_i = \mathsf{Hidden}\ z \wedge y \in \psi(\mathcal{E}(i)))$:
If any of the user identifier x, the value R, the service description s, and the escrow identifier idr are revealed, we will have executed lines 14 and U_1^{ψ}, lines 16 and U_2^{ψ}, lines 18 and U_3^{ψ}, and lines 20 and U_4^{ψ}, respectively. Consequently, we have derived that $w' = \mathsf{Revealed}\ w^o$ for $w \in \{x, R, s, \mathit{idr}\}$.

In either case, we add the variables z, x, R, s, and idr to ψ' in lines U_5^{ψ}-U_9^{ψ}. In particular, these variables will ensure that all values hidden by the same existential quantifier are represented by the same committed value (the equality between them is ensured by $\mathcal{E}$ that maps the indices of these variables to the same index).

(6) $E' \vdash \mathit{stm}' : \mathit{statement}$ and $E' \vdash f' : \mathit{formula}$:
This is an assumption.

(7) $E'' \vdash \{f' = \underline{\mathcal{F}^{e}}\}$ and

(8) $E'', (\psi')^{=} \vdash \mathcal{F}^{\wedge} \wedge \mathcal{F}^{e}$:
The assumed appropriateness yields that $E', (\psi)^{=} \vdash \mathcal{F}^{\wedge}$. Since E'' is an extension of E' and ψ' is an extension of ψ, we obtain that $E'', (\psi')^{=} \vdash \mathcal{F}^{\wedge}$.

For the formula $\mathcal{F}^{e}$ with the encoding $\underline{\mathcal{F}^{e}} = \mathsf{EscrowInfo}(z', x', R', s', \mathit{idr}')$, we consider the following formulas in E'' and the entries of ψ':

(a) lines 1-2: $\{f = \mathsf{EscrowInfo}(z', x', R', s', \mathit{idr}'\}$

(b) line U_0^{ψ}: $\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{z'\}]$

(c) line 14-U_4^{ψ}: $\{x' = \mathsf{Revealed}\ x^o\}$, $\{R' = \mathsf{Revealed}\ R^o\}$, $\{s' = \mathsf{Revealed}\ s^o\}$, $\{\mathit{idr}' = \mathsf{Revealed}\ \mathit{idr}^o\}$ along with the corresponding entries in ψ', if the corresponding values are revealed.

(d) lines U_5^{ψ}-U_9^{ψ}:

$$\begin{aligned}
&\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{z\}]\\
&\psi := \psi[\mathcal{E}(\omega+1) \mapsto \psi(\mathcal{E}(\omega+1)) \cup \{x\}]\\
&\psi := \psi[\mathcal{E}(\omega+2) \mapsto \psi(\mathcal{E}(\omega+2)) \cup \{R\}]\\
&\psi := \psi[\mathcal{E}(\omega+3) \mapsto \psi(\mathcal{E}(\omega+3)) \cup \{s\}]\\
&\psi := \psi[\mathcal{E}(\omega+4) \mapsto \psi(\mathcal{E}(\omega+4)) \cup \{idr\}]
\end{aligned}$$

(e) line 27: $\{\mathsf{EscrowInfo}(z, x, R, s, \mathit{idr}'')\}$

(f) line 28: $\{\mathit{idr} = \mathit{idr}''\}$

First, we note that by (e) and (f), we derive

$$\mathsf{EscrowInfo}(z, x, R, s, \mathit{idr}).$$

More precisely, we derive the logical formula where all values are derived from the commitments attached to the proof. For the revealed arguments, we obtain the expected equalities by the respective entries into ψ' by (c); for all hidden values, we get the necessary equalities by (d). In particular, this substitution shows that $E'' \vdash \{f' = \underline{\mathcal{F}^e}\}$. Furthermore, it shows that $E'' \vdash \mathcal{F}^e$ since substituting all the "^{o}" variables for the revealed values and consistently replacing the committed values with existential quantified values proves $\mathcal{F}^e$. The equalities induced by ψ' enable us to consistently existentially quantify across $\mathcal{F}^\wedge$ and $\mathcal{F}^e$, yielding $E'' \vdash \mathcal{F}^\wedge \wedge \mathcal{F}^e$.

This case concludes the proof. □

Lemma A.22 (Appropriate invariance)**.** *Let E be basic such that $E \vdash \mathit{stm} : \mathit{statement}$ and $E \vdash f : \mathit{formula}$. Then, for every formula $\mathcal{F}^\wedge$, there are a typing environment E' and mappings ψ', ϕ' such that* $\mathsf{appropriate}(E, E', \psi', \phi', \mathcal{F}^\wedge, \mathit{stm}, f)$.

Proof. Let E, stm and f be as in the lemma. Furthermore, let ψ and ϕ be empty mappings. The proof proceeds by induction on the structure of $\mathcal{F}^\wedge$. We consider the base case to be the "empty" formula true.

Case $\mathcal{F}^\wedge = \mathcal{F}^e$:
In this case $\mathcal{F}^\wedge = \mathcal{F}^e = \mathsf{true}$. We obtain $\mathsf{appropriate}(E, E, \mathsf{true}, \psi, \phi, \mathit{stm}, f)$ by Proposition A.4 (appropriate for true).

From Lemma A.21 (almost-appropriate invariance), we obtain that there are a typing environment E'' and mappings ψ', ϕ' such that $\mathsf{appropriate}'(E, E, E'', \psi', \phi', \mathsf{true}, \mathcal{F}^e, \mathit{stm}, f)$. Notice that we used stm and f as guaranteed by the theorem. Since $\mathcal{F}^\wedge = \mathcal{F}^e$, we can immediately apply Proposition A.5 (linking appropriateness and almost-appropriateness) and derive that $\mathsf{appropriate}(E, E'', \psi', \phi', \mathcal{F}^e, \mathit{stm}, f)$.

Case $\mathcal{F}^\wedge = \mathcal{F}^{\wedge\prime} \wedge \mathcal{F}^e$:
In this case, we type-check the verification code for the logical conjunction applied to $\mathcal{F}^{\wedge\prime}$ and $\mathcal{F}^e$.

```
$\mathit{And}$-Macro($\underline{\mathcal{F}^{\wedge\prime} \wedge \mathcal{F}^e}, p, f, 0, \overrightarrow{\Delta}(\underline{\mathcal{F}^{\wedge\prime}})$) $\triangleq$
  match $\mathit{stm}$ with $\mathsf{And}_\mathsf{p}(\mathit{stm}_1, \mathit{stm}_2)$ $\Longrightarrow$
```

```
2        match $f$ with And$(f_1, f_2)$ $\implies$
3              $[\![\underline{\mathcal{F}^{\wedge'}}, *, \mathit{stm}_1, f_1, \omega_1]\!]$ [ $[\![\underline{\mathcal{F}^{e}}, *, \mathit{stm}_2, f_2, \omega_2]\!]$ ]
```

We type-check this expression until we arrive at the code for $\mathcal{F}^{\wedge'}$ and $\mathcal{F}^e$. Since we want to prove that the resulting typing environment and maps are appropriate, we start with the given maps and the typing environment E.

Code (line 1):

$$\text{match } \mathit{stm} \text{ with } \mathsf{And}_{\mathsf{p}}(\mathit{stm}_1, \mathit{stm}_2) \implies$$

Environment:

$$E$$

Rules:

$$\frac{\begin{array}{c} E \vdash M : T \qquad \forall 0 < i \leq n.\ h_i : (H_i, T) \qquad \forall 0 < i \leq n.\ H_i = x_1^i : T_1^i * \cdots * x_{m_i - 1}^i : T_{m_i-1}^i * T_{m_i}^i \\ \forall 0 < i \leq n.\ E, x_1^i : T_1^i, \ldots, x_{m_i}^i : T_{m_i}^i, \{M = h_i(x_1^i, \ldots, x_{m_i}^i)\} \vdash A_i : U \leadsto E_i' \\ E \vdash A_{\mathsf{fail}} : U \leadsto E'' \\ \{x_j^i \mid 1 \leq i \leq n, 1 \leq j \leq m_i\} \cap \mathit{fv}(U) = \emptyset \end{array}}{E \vdash \mathsf{match}\ M\ \mathsf{with} \mid h_1\ (x_1^1, \ldots, x_{m_1}^1)\ \to A_1 \ldots \mid h_n\ (x_1^n, \ldots, x_{m_n}^n)\ \to A_n \mid \_ \to A_{\mathsf{fail}} : U \leadsto E_1'}\ \textsc{Ext Exp Match-Split}^n_{(m_1, \ldots, m_n)}$$

Proof Obligations:

$$E \vdash \mathit{stm} : \mathit{statement}$$
$$\mathsf{And}_{\mathsf{p}} : (\mathit{statement} * \mathit{statement}, \mathit{statement})$$
$$\{\mathit{stm}_1, \mathit{stm}_2\} \cap \mathit{fv}(T^{\mathcal{F}^{\wedge'} \wedge \mathcal{F}^e}) = \emptyset$$
$$E, \mathit{stm}_1 : \mathit{statement}, \mathit{stm}_2 : \mathit{statement}, \{\mathit{stm} = \mathsf{And}_{\mathsf{p}}(\mathit{stm}_1, \mathit{stm}_2)\} \vdash \mathcal{R} : T^{\mathcal{F}^{\wedge'} \wedge \mathcal{F}^e}$$

The first obligation is an assumption, the second obligation holds by definition of the type constructor $\mathsf{And}_{\mathsf{p}}$, and the third assumption follows since stm_1 and stm_2 do not occur in $T^{\mathcal{F}^{\wedge'} \wedge \mathcal{F}^e}$.

Code (line 2):

$$\text{match } f \text{ with } \mathsf{And}(f_1, f_2) \implies$$

Environment:

$$\underbrace{E, \mathit{stm}_1 : \mathit{statement}, \mathit{stm}_2 : \mathit{statement}, \{\mathit{stm} = \mathsf{And}_{\mathsf{p}}(\mathit{stm}_1, \mathit{stm}_2)\}}_{=: E_1}$$

Rules:

$$\frac{\begin{array}{c} E \vdash M : T \qquad \forall 0 < i \leq n.\ h_i : (H_i, T) \qquad \forall 0 < i \leq n.\ H_i = x_1^i : T_1^i * \cdots * x_{m_i - 1}^i : T_{m_i-1}^i * T_{m_i}^i \\ \forall 0 < i \leq n.\ E, x_1^i : T_1^i, \ldots, x_{m_i}^i : T_{m_i}^i, \{M = h_i(x_1^i, \ldots, x_{m_i}^i)\} \vdash A_i : U \leadsto E_i' \\ E \vdash A_{\mathsf{fail}} : U \leadsto E'' \\ \{x_j^i \mid 1 \leq i \leq n, 1 \leq j \leq m_i\} \cap \mathit{fv}(U) = \emptyset \end{array}}{E \vdash \mathsf{match}\ M\ \mathsf{with} \mid h_1\ (x_1^1, \ldots, x_{m_1}^1)\ \to A_1 \ldots \mid h_n\ (x_1^n, \ldots, x_{m_n}^n)\ \to A_n \mid \_ \to A_{\mathsf{fail}} : U \leadsto E_1'}\ \textsc{Ext Exp Match-Split}^n_{(m_1, \ldots, m_n)}$$

Proof Obligations:

$$\begin{array}{l} E_1 \vdash f : \mathit{formula} \\ \mathsf{And} : (\mathit{formula} * \mathit{formula}, \mathit{formula}) \\ \{f_1, f_2\} \cap \mathit{fv}(T^{\mathcal{F}^{\wedge\prime} \wedge \mathcal{F}^e}) = \emptyset \end{array}$$

$$\overbrace{E_1, f_1 : \mathit{formula}, f_2 : \mathit{formula}, \{f = \mathsf{And}_\mathsf{p}(f_1, f_2)\}}^{=:E_2} \vdash \mathcal{R}$$

The first three obligation are analogous to the arguments for line 1.

At this point, we notice that E_1 is basic and that

$$\begin{array}{l} E_2 \vdash \mathit{stm} : \mathit{statement} \\ E_2 \vdash \mathit{stm}_1 : \mathit{statement} \\ E_2 \vdash \mathit{stm}_2 : \mathit{statement} \\ E_2 \vdash f : \mathit{formula} \\ E_2 \vdash f_1 : \mathit{formula} \\ E_2 \vdash f_2 : \mathit{formula} \end{array}$$

We apply the induction hypothesis for the code of $\mathcal{F}^{\wedge\prime}$ and derive $\mathsf{appropriate}(E_1, E', \psi', \phi', \mathcal{F}^{\wedge\prime}, \mathit{stm}_1, f_1)$ for some typing environment E' and some mappings ψ' and ϕ'. We now have all the necessary premises to apply Lemma A.21 (almost-appropriate invariance). We derive that $\mathsf{appropriate}'(E_1, E', E'', \psi'', \phi'', \mathcal{F}^{\wedge\prime}, \mathcal{F}^{e\prime}, \mathit{stm}_2, f_2)$ for some typing environment E'' and mappings ψ'' and ϕ''.

At this point, the appropriateness of $\mathcal{F}^{\wedge\prime}$ yields that $E_1, (\phi')^{=} \vdash \{f_1 = \underline{\mathcal{F}^{\wedge\prime}}\}$; the almost-appropriateness of $\mathcal{F}^e$ yields $E'' \vdash \{f_2 = \underline{\mathcal{F}^e}\}$. In particular, $E_1 \vdash \{f = \mathsf{And}_\mathsf{p}(f_1, f_2)\}$. It follows that

$$\begin{aligned} f &= \mathsf{And}(f_1, f_2) \\ &= \mathsf{And}(\underline{\mathcal{F}^{\wedge\prime}}, \underline{\mathcal{F}^e}) \\ &= \underline{\mathcal{F}^{\wedge\prime} \wedge \mathcal{F}^e} \end{aligned}$$

We will now argue that $\mathsf{appropriate}(E, E'', \psi'', \phi'', \mathcal{F}^{\wedge}, \mathit{stm}, f)$:

(1) $E \vdash C : \{x : \mathit{bool} \mid \forall \tilde{z}.\ y = \underline{\mathcal{F}^{\wedge}} \ \wedge\ x = \mathsf{true} \implies \mathcal{F}^{\wedge}\} \rightsquigarrow E'$:
We have proven that the type-checking succeeds. In particular, the type-checking of the code for $\mathcal{F}^e$ proves this type.

(2) E is basic and $E'' \vdash \diamond$:
Follows since appropriateness and almost-appropriateness preserves the well-formedness.

(3) $\mathsf{ext}(\mathcal{F}^{\wedge}, \psi, \phi)$:
Follows by the appropriateness and the almost-appropriateness.

(4) $\forall i \in \mathcal{I}_{vk}(\underline{\mathcal{F}^{\wedge}}).\ E' \vdash \phi(i) : \mathit{verkey}$:
Follows immediately by the almost-appropriateness.

(5) $\forall i.\ \exists y.\ (x_i = \mathsf{Revealed}\ y \wedge y \in \psi(\mathcal{E}(i)))\ \vee\ (\exists z.\ x_i = \mathsf{Hidden}\ z \wedge y \in \psi(\mathcal{E}(i)))$:
Follows immediately by the almost-appropriateness.

(6) $E \vdash \mathit{stm} : \mathit{statement}$ and $E \vdash f : \mathit{formula}$:
This is an assumption.

(7) $E'' \vdash \{f = \underline{\mathcal{F}^\wedge}\}$:
The appropriateness and the almost-appropriateness establish that $E'' \vdash \{f_1 = \underline{\mathcal{F}^{\wedge\prime}}\}$ and $E'' \vdash \{f_2 = \underline{\mathcal{F}^e}\}$. Using the observation above, we immediately conclude that $E'' \vdash \{f = \underline{\mathcal{F}^\wedge}\}$.

(8) $E', (\psi)^= \vdash \mathcal{F}^\wedge$:
This follows immediately by the almost-appropriateness.

This concludes the proof. □

The final steps of the well-typedness proof are as follows: First, we use the appropriate invariance to type-check all verification functions for conjunctive formulas. Inductively, we will then type-check all verification functions for formulas with disjunctions. The crucial point will be that all formulas are in disjunctive normal form, i.e., the disjunctions are "stacked on top" of the conjunctions.

Lemma A.23 (Well-typedness of $\mathsf{verify}_{\mathcal{F}^\wedge}$)**.** *Let $\mathcal{F}^\wedge$ be a formula that only contains elementary formulas connected with logical conjunctions and let E be basic. Then* $E \vdash \mathsf{verify}_{\mathcal{F}^\wedge} : (p : \mathit{proof}) \to (f : \mathit{formula}) \to T^{\mathcal{F}^\wedge}$.

Proof.

```
verify_{$\underline{\mathcal{F}^\wedge}$} $(p : \mathit{proof})$ $(y : \mathit{formula})$ : $\{x : \mathit{bool} \mid \forall \tilde{z}.\ y = \underline{\mathcal{F}^\wedge}\ \wedge\ x = \mathsf{true} \implies \mathcal{F}^\wedge\}$ =
1   match $p$ with ZK$(\mathit{zkv}, \mathit{stm})$ ⇒
2     let $c_1$ = checkEq $\mathit{stm}$ $f$;
3     let $c_2$ = checkZK $p$;
4     if $c_1$ = true then
5       if $c_2$ = true then
6       $[\![\underline{\mathcal{F}^\wedge}, \mathit{zkv}, \mathit{stm}, f, 0]\!]$ [ context$^=(\psi)$[true] ]
```

Listing A.52: Excerpt of the implementation of verify.

Since we are familiar with how to type-check match statements, let statements and if statements, we immediately skip to code of the verification macros.

Code (line 5):

$$[\![\underline{\mathcal{F}^\wedge}, \mathit{zkv}, \mathit{stm}, f, 0]\!]\ [\ \mathsf{context}^=(\psi)[\mathsf{true}]\ ]$$

Environment:

$$\underbrace{\begin{array}{l} E, p : \mathit{proof}, f : \mathit{formula}, \mathit{zkv} : \mathit{zero\text{-}knowledge}, \mathit{stm} : \mathit{statement}, \{p = \mathsf{ZK}(\mathit{zkv}, \mathsf{stm})\} \\ c_1 : \mathit{bool}, c_2 : \mathit{bool}, \{c_1 = \mathsf{true}\}, \{c_2 = \mathsf{true}\} \end{array}}_{=:E_1}$$

At this point, we let ψ and ϕ be empty mappings. Notice that E_1 satisfies the premise of Lemma A.22 (Appropriate invariance), i.e., E_1 is basic and $E_1 \vdash \mathit{stm} : \mathit{statement}$ and $E_1 \vdash f : \mathit{formula}$. We conclude that there is a typing environment E'' and mappings ψ' and ϕ' such that $\mathsf{appropriate}(E_1, E'', \psi', \phi', \mathcal{F}^\wedge, \mathit{stm}, f)$. In particular, we obtain that $E'', (\psi')^= \vdash \mathcal{F}^\wedge$ and $E'', (\psi')^= \vdash \{f = \underline{\mathcal{F}^\wedge}\}$. Combining these two facts, we derive that $E'', \mathsf{context}^=(\psi') \vdash \{(f = \underline{\mathcal{F}^\wedge}) \wedge \mathcal{F}^\wedge\}$.

The final step in the verification is to type-check $E'' \vdash \mathsf{context}^=(\psi')[\mathsf{true}] : T^{\mathcal{F}^\wedge}$. We apply Lemma A.20 ($\mathsf{context}^=(\psi)$ and $(\psi)^=$) and derive that if $E'', \mathsf{context}^=(\psi') \vdash \mathsf{true} : T^{\mathcal{F}^\wedge}$, then $E'' \vdash \mathsf{context}^=(\psi')[\mathsf{true}] : T^{\mathcal{F}^\wedge}$. As observed above, $E'', (\psi')^= \vdash \mathcal{F}^\wedge$. We apply

$$\textsc{Val Refine}\quad \frac{E \vdash M : T \qquad E \vdash F\{M/x\}}{E \vdash M : \{x : T \mid F\}}$$

to conclude that $E'', (\psi')^= \vdash \mathsf{true} : \{x : \mathit{bool} \mid f = \underline{\mathcal{F}^\wedge} \wedge \mathcal{F}^\wedge\}$. We logically derive that $\mathsf{true} : \{x : \mathit{bool} \mid f = \underline{\mathcal{F}^\wedge} \wedge x = \mathsf{true} \implies \mathcal{F}^\wedge\}$ since $x = \mathsf{true}$ holds. Any of the other branches will result in returning false, which makes the refinement hold vacuously. This concludes the proof. □

Finally, we prove that the (general) verification function is well-typed.

Theorem A.1 (Well-typedness of $\mathsf{verify}_{\mathcal{F}^\vee}$)**.** *Let E be basic. Then $E \vdash \mathsf{verify}_{\mathcal{F}^\vee} : (p : \mathit{proof}) \to (f : \mathit{formula}) \to T^{\mathcal{F}^\vee}$.*

Proof. The proof proceeds by induction on the structure of $\mathcal{F}^\vee$. The base case is $\mathcal{F}^\vee = \mathcal{F}^\wedge$, i.e., the formula does not contain any disjunction.

Case $\mathcal{F}^\vee = \mathcal{F}^\wedge$:
The claim follows directly from Lemma A.23 (Well-typedness of $\mathsf{verify}_{\mathcal{F}^\wedge}$).

In the following, we assume that E is basic and $\mathsf{verify}_{\mathcal{F}^{\vee\prime}}$ is well-typed relative to E for all formulas $\mathcal{F}^{\vee\prime}$ that are structurally smaller w.r.t. disjunctions than $\mathcal{F}^\vee$.

Case $\mathcal{F}^\vee = \mathcal{F}^\vee_1 \vee \mathcal{F}^\vee_2$:
In this case, we have to type-check

```
verify$_{\underline{\mathcal{F}^\vee}}$ $(p : \mathit{proof})$ $(f : \mathit{formula})$ : $\{x : \mathit{bool} \mid \forall \tilde{z}.\ f = \underline{\mathcal{F}^\vee} \wedge x = \mathsf{true} \implies \mathcal{F}^\vee\}$ =
  match $p$ with ZK$(\mathit{zkv}, \mathit{stm})$ $\Rightarrow$
    let $c_1$ = checkEq $\mathit{stm}$ $f$;
    let $c_2$ = checkZK $p$;
    if $c_1$ = true then
      if $c_2$ = true then
        match $\mathit{stm}$ with Or$_\mathsf{p}(\mathit{stm}_1, \mathit{stm}_2)$ $\Longrightarrow$
        match $f$ with Or$(f_1, f_2)$ $\Longrightarrow$
            let $\mathit{tmp}_{\mathit{zkv}}$ = openZK $\mathit{zkv}$;
            let $(\mathit{stm}', r)$ = $\mathit{tmp}_{\mathit{zkv}}$;
            match $\mathit{stm}'$ with Or$_\mathsf{p}(\mathit{stm}_1, \mathit{stm}_2)$ $\Longrightarrow$
```

```
11                    let $zkv_1$ = commitZK $(stm'_1, r)$;
12                    let $zkv_2$ = commitZK $(stm'_2, r)$;
13                    let $res_1$ = verify$_{\underline{\mathcal{F}_1^{\vee}}}$ ZK$(zkv_1, stm_1)$ $f_1$;
14                    if $res_1$ = true then
15                      $res_1$
16                    else
17                      verify$_{\underline{\mathcal{F}_2^{\vee}}}$ ZK$(zkv_2, r)$ $f_2$
                  | _ $\Longrightarrow$ false
              | _ $\Longrightarrow$ false
              | _ $\Longrightarrow$ false
18      else
19        false
20    else
21      false
```

We skip in initial application of

$$\dfrac{E, x : T \vdash A : U}{E \vdash \text{fun } x \to A : (\Pi x : T.\, U)}\ \textsc{Val Fun}$$

and the type-checking of the two let-statements with the checks as in Lemma A.23 (Well-typedness of verify$_{\mathcal{F}^{\wedge}}$) and proceed with line 9 and the extended typing environment

$$\underbrace{\begin{array}{l} E, p : \mathit{proof}, f : \mathit{formula}, \mathit{zkv} : \mathit{zero\text{-}knowledge}, \mathit{stm} : \mathit{statement}, \{p = \mathsf{ZK}(\mathit{zkv}, \mathit{stm})\} \\ c_1 : \mathit{bool}, c_2 : \mathit{bool}, \{c_1 = \mathsf{true}\}, \{c_2 = \mathsf{true}\} \\ \mathit{stm}_1 : \mathit{statement}, \mathit{stm}_2 : \mathit{statement}, \{\mathit{stm} = \mathsf{Or}_{\mathsf{p}}(\mathit{stm}_1, \mathit{stm}_2)\} \\ f_1 : \mathit{formula}, f_2 : \mathit{formula}, \{f = \mathsf{Or}(f_1, f_2)\} \end{array}}_{=:E_1}$$

Code (lines 8):

$$\text{let } \mathit{tmp}_{\mathit{zkv}} = \mathsf{openZK}\ \mathit{zkv};$$

Environment:

$$E_1$$

Rules:

$$\dfrac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}\ \textsc{Ext Exp Appl}$$

$$\dfrac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}\ \textsc{Ext Exp Let}$$

Proof Obligations:

1. $E_1 \vdash \mathsf{openZK} : \mathit{zero\text{-}knowledge} \to \mathit{statement} * \mathit{random}$
2. $E_1 \vdash \mathit{zkv} : \mathit{zero\text{-}knowledge}$
3. $\mathit{tmp}_{\mathit{zkv}} \notin \mathit{fv}(T^{\mathcal{F}^{\vee}})$
4. $E_1, \mathit{tmp}_{\mathit{zkv}} : \mathit{statement} * \mathit{random} \vdash \mathcal{R} : T^{\mathcal{F}^{\vee}}$

Requirement 1 follows since E_1 is basic. Obligation 2 holds as *zkv* is contained in E_1 with the proper type. The third obligation holds because the value $\mathit{tmp}_{\mathit{zkv}}$ is not contained in the return type.

Code (lines 9):

$$\text{let } (\mathit{stm}', r) = \mathit{tmp}_{\mathit{zkv}};$$

Environment:

$$\underbrace{E_1, \mathit{tmp}_{\mathit{zkv}} : \mathit{statement} * \mathit{random}}_{=:E_2}$$

Rules:

$$\dfrac{\begin{array}{c} E \vdash M : (\Sigma x : T.\, U) \\ E, x : T, y : U, \_ : \{(x,y) = M\} \vdash A : V \rightsquigarrow E' \qquad \{x,y\} \cap \mathit{fv}(V) = \emptyset \end{array}}{E \vdash \mathsf{let}\ (x,y) = M;\ A : V \rightsquigarrow E'}\ \textsc{Ext Exp Split}$$

Proof Obligations:

1. $E_2 \vdash \mathit{tmp}_{\mathit{zkv}} : \mathit{statement} * \mathit{random}$
2. $\{\mathit{stm}', r\} \cap \mathit{fv}(T^{\mathcal{F}^{\vee}}) = \emptyset$
3. $E_2, \mathit{stm}' : \mathit{statement}, r : \mathit{random}, \{(\mathit{stm}', r) = \mathit{tmp}_{\mathit{zkv}}\} \vdash \mathcal{R} : T^{\mathcal{F}^{\vee}}$

Requirements 1 follows since we added the variable with the proper type in the step above. As the newly introduced variables do not occur in the type $T^{\mathcal{F}^{\vee}}$, obligation 2 follows.

Code (lines 10):

$$\text{match } \mathit{stm}' \text{ with } \mathsf{Or_p}(\mathit{stm}_1, \mathit{stm}_2) \implies$$

Environment:

$$\underbrace{E_2, \mathit{stm}' : \mathit{statement}, r : \mathit{random}, \{(\mathit{stm}', r) = \mathit{tmp}_{\mathit{zkv}}\}}_{=:E_3}$$

Rules:

$$\dfrac{\begin{array}{c} E \vdash M : T \qquad \forall 0 < i \le n.\ h_i : (H_i, T) \qquad \forall 0 < i \le n.\ H_i = x^i_1 : T^i_1 * \cdots * x^i_{m_i-1} : T^i_{m_i-1} * T^i_{m_i} \\ \forall 0 < i \le n.\ E, x^i_1 : T^i_1, \ldots, x^i_{m_i} : T^i_{m_i}, \{M = h_i(x^i_1, \ldots, x^i_{m_i})\} \vdash A_i : U \rightsquigarrow E'_i \\ E \vdash A_{\mathsf{fail}} : U \rightsquigarrow E'' \\ \{x^i_j \mid 1 \le i \le n, 1 \le j \le m_i\} \cap \mathit{fv}(U) = \emptyset \end{array}}{E \vdash \mathsf{match}\ M\ \mathsf{with} \mid h_1\ (x^1_1, \ldots, x^1_{m_1})\ \to A_1 \ldots \mid h_n\ (x^n_1, \ldots, x^n_{m_n})\ \to A_n \mid \_ \to A_{\mathsf{fail}} : U \rightsquigarrow E'_1}\ \textsc{Ext Exp Match-Split}^n_{(m_1,\ldots,m_n)}$$

Proof Obligations:

1. $E_3' \vdash \mathit{stm}' : \mathit{statement}$
2. $\{\mathit{stm}_1, \mathit{stm}_2\} \cap \mathit{fv}(T^{\mathcal{F}^\vee}) = \emptyset$
3. $\mathsf{Or}_\mathsf{p} : (\mathit{statement} * \mathit{statement}, \mathit{statement})$
4. $E_3', \mathit{stm}_1 : \mathit{statement}, \mathit{stm}_2 : \mathit{statement}, \{\mathit{stm}' = \mathsf{Or}_\mathsf{p}(\mathit{stm}_1, \mathit{stm}_2)\} \vdash \mathcal{R} : T^{\mathcal{F}^\vee}$

Requirement 1 follows since we added the variable with the proper type in the step above. As the newly introduced variables do not occur in the type $T^{\mathcal{F}^\vee}$, requirement 2 follows. The third requirement holds by inspection of the type definitions in Table A.2 and Table A.3.

Code (lines 11):

$$\text{let } \mathit{zkv}_1 = \mathsf{commitZK}\ (\mathit{stm}_1, r);$$

Environment:

$$\underbrace{E_3', \mathit{stm}_1 : \mathit{statement}, \mathit{stm}_2 : \mathit{statement}, \{\mathit{stm}' = \mathsf{Or}_\mathsf{p}(\mathit{stm}_1, \mathit{stm}_2)\}}_{=:E_4}$$

Rules:

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

$$\textsc{Val Pair}\quad \frac{E \vdash M : T \qquad E \vdash N : U\{M/x\}}{E \vdash (M, N) : \Sigma x : T.\, U}$$

Proof Obligations:

1. $E_4 \vdash \mathsf{commitZK} : (\mathit{statement} * \mathit{random}) \to \mathit{zero\text{-}knowledge}$
2. $E_4 \vdash \mathit{stm}_1 : \mathit{statement}$
3. $E_4 \vdash r : \mathit{random}$
4. $\mathit{zkv}_1 \notin \mathit{fv}(T^{\mathcal{F}^\vee})$
5. $E_4, \mathit{zkv}_1 : \mathit{zero\text{-}knowledge} \vdash \mathcal{R} : T^{\mathcal{F}^\vee}$

The first obligation follows because E_4 is basic. The second and third obligations hold as we added the variables with the proper type in the steps above. The forth obligation follows as the variable does not occur in the return type.

Code (lines 12):

$$\text{let } \mathit{zkv}_2 = \mathsf{commitZK}\ (\mathit{stm}_2, r);$$

Environment:

$$\underbrace{E_4', zkv_1 : \mathit{zero\text{-}knowledge}}_{=:E_5}$$

Rules:

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

$$\textsc{Val Pair}\quad \frac{E \vdash M : T \qquad E \vdash N : U\{M/x\}}{E \vdash (M, N) : \Sigma x : T.\, U}$$

Proof Obligations:

1. $E_5 \vdash \mathsf{commitZK} : (\mathit{statement} * \mathit{random}) \to \mathit{zero\text{-}knowledge}$
2. $E_5 \vdash \mathit{stm}_2 : \mathit{statement}$
3. $E_5 \vdash r : \mathit{random}$
4. $\mathit{zkv}_2 \notin \mathit{fv}(T^{\mathcal{F}^\vee})$
5. $E_5, \mathit{zkv}_2 : \mathit{zero\text{-}knowledge} \vdash \mathcal{R} : T^{\mathcal{F}^\vee}$

The reasoning is analogous to the step above.

Code (lines 13):

$$\text{let } \mathit{res}_1 = \mathsf{verify}_{\underline{\mathcal{F}_1^\vee}}\ \mathsf{ZK}(\mathit{zkv}_1, \mathit{stm}_1)\ f_1;$$

Environment:

$$\underbrace{E_5, \mathit{zkv}_2 : \mathit{zero\text{-}knowledge}}_{=:E_6}$$

Rules:

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E}$$

$$\textsc{Ext Exp Let}\quad \frac{E \vdash A : T \rightsquigarrow E'' \qquad E, x : T \vdash B : U \rightsquigarrow E' \qquad x \notin \mathit{fv}(U)}{E \vdash \mathsf{let}\ x = A;\ B : U \rightsquigarrow E'}$$

$$\textsc{Val Inl Inr Fold}\quad \frac{h : (T, U) \qquad E \vdash M : T \qquad E \vdash U}{E \vdash h(M) : U} \qquad\qquad \textsc{Val Pair}\quad \frac{E \vdash M : T \qquad E \vdash N : U\{M/x\}}{E \vdash (M, N) : \Sigma x : T.\, U}$$

Proof Obligations:

1. $E_6 \vdash \mathsf{verify}_{\underline{\mathcal{F}_1^\vee}} : (p : \mathit{proof}) \to (f : \mathit{formula}) \to T^{\mathcal{F}_1^\vee}$
2. $\mathsf{ZK} : (\mathit{zero\text{-}knowledge} * \mathit{statement}, \mathit{proof})$
3. $E_6 \vdash \mathit{zkv}_1 : \mathit{zero\text{-}knowledge}$
4. $E_6 \vdash \mathit{stm}_1 : \mathit{zero\text{-}knowledge}$
5. $E_6 \vdash f_1 : \mathit{formula}$
6. $\mathit{res}_1 \notin \mathit{fv}(T^{\mathcal{F}^\vee})$
7. $E_6, \mathit{res}_1 : T^{\mathcal{F}_1^\vee} \vdash \mathcal{R} : T^{\mathcal{F}^\vee}$

The first claim follows from the induction hypothesis. The second obligation follows from inspecting the type definition in Table A.2 and Table A.3. The third, fourth, and fifth requirement follow since the variables with the corresponding types are contained in E_6. The sixth requirement follows since res_1 does not occur in $T^{\mathcal{F}^\vee}$.

Code (lines 14-17):

```
if $\mathit{res}_1 = \mathsf{true}$ then
  $\mathit{res}_1$
else
  $\mathsf{verify}_{\underline{\mathcal{F}_2^\vee}}\ \mathsf{ZK}(\mathit{stm}_2, r)\ f_2$
```

Environment:

$$\underbrace{E_6, \mathit{res}_1 : T^{\mathcal{F}_1^\vee}}_{=:E_7}$$

Rules:

$$\textsc{Ext Exp If}\quad \frac{E \vdash M : T \qquad E \vdash N : U \qquad E, \{M = N\} \vdash A : V \rightsquigarrow E' \qquad E \vdash B : V \rightsquigarrow E''}{E \vdash \mathsf{if}\ M = N\ \mathsf{then}\ A\ \mathsf{else}\ B : V \rightsquigarrow E'}$$

$$\textsc{Ext Exp Appl}\quad \frac{E \vdash M : (\Pi x : T.\, U) \qquad E \vdash N : T}{E \vdash M\ N : U\{N/x\} \rightsquigarrow E} \qquad \textsc{Val Inl Inr Fold}\quad \frac{h : (T, U) \qquad E \vdash M : T \qquad E \vdash U}{E \vdash h(M) : U}$$

Proof Obligations:

1. $E_7 \vdash \mathit{res}_1 : T^{\mathcal{F}_1^\vee}$
2. $E_7 \vdash \mathit{stm}_2 : \mathit{statement}$
3. $E_7 \vdash r : \mathit{random}$
4. $E_7 \vdash f_2 : \mathit{formula}$
5. $E_7 \vdash \mathsf{true} : \mathit{bool}$
6. $\mathsf{ZK} : (\mathit{statement} * \mathit{random}, \mathit{proof})$
7. $E_7, \{\mathit{res}_1 = \mathsf{true}\} \vdash \mathit{res}_1 : T^{\mathcal{F}^\vee}$
8. $E_7 \vdash \mathsf{verify}_{\underline{\mathcal{F}_2^\vee}}\ \mathsf{ZK}(\mathit{stm}_2, r)\ f_2 : T^{\mathcal{F}^\vee}$

The first through the fourth requirements follow since the corresponding variables are contained in E_7 with the proper types. Requirement 5 and 6 follow by definition of *bool* and by inspection of the type definitions in Table A.2 and Table A.3.

For the eighth obligation, we notice that by induction hypothesis, the return type is $T^{\mathcal{F}_2^\vee}$. To summarize, the then-branch has the returns type $T^{\mathcal{F}_1^\vee}$ and the else-branch returns type $T^{\mathcal{F}_2^\vee}$ (the reasoning here is analogous to that of the line 13).

The two branches only prove their respective branch of the disjunction. However, if one branch of a disjunction holds true, then the whole disjunction is valid. Therefore, from type $T^{\mathcal{F}_i^\vee}$, $i \in \{1,2\}$, we derive $T^{\mathcal{F}^\vee}$ if we additionally can argue that the condition on the proven formula holds. This reasoning is taken care of in line 7 that shows that

$$\begin{aligned} f &= \mathsf{Or}(f_1, f_2) \\ &= \mathsf{Or}(\underline{\mathcal{F}_1^\vee}, \underline{\mathcal{F}_2^\vee}) \\ &= \underline{\mathcal{F}^\vee}. \end{aligned}$$

Should both res_1 and res_2 be false, then the verification returns value false and the type $T^{\mathcal{F}^\vee}$ vacuously holds. This concludes the proof. □

A.3. Well-Typedness of the ML Implementation

We have proven that the RCF implementation is well-typed. This implementation contains standard ML types such as *bitstring* and *bool* but it also contains refined types that do not exist in an ML context. Although we kept the number of functions that rely on refinement types to a minimum, we have to type-check every program that uses the refined API with the RCF type system.

The reason is that for programs containing non-ML types (i.e., types that are not equivalent to *unit*), the program deals with security-sensitive data and we cannot “skip” the type-checking process. The key idea therefore is to hide away the refined types. In a nutshell, we proceed as follows: we expose to programmers an augmented API that only exports functions that exclusively contain ML types, i.e., types that in RCF are captured by the type *unit*. Since the all types occurring in an ML program correspond to the RCF type *unit* and the augmented API also does not introduce non-*unit* types, we do not need to type-check the program with the RCF type system. Technically, we will apply the opponent typability lemma (Lemma A.6), which intuitively states that programs that only use ML types will always type-check. In particular, we do not have to conduct the type-checking.

To create an API with only ML types, we have to adapt the API. In particular, we have to enforce that the expected logical formulas hold since the ML programs also cannot use assertions. The core idea is to encapsulate the refinement types and the assertions inside of wrapper functions that, in turn, only export ML types. More precisely, we implement the method $\mathsf{verify}'_{\mathcal{F}}$. This method internally uses the refined method $\mathsf{verify}_{\mathcal{F}}$ and executes the expected assertion after a successful verification to enforce that formula $\mathcal{F}$ is logically entailed while it only exports ML types.

$\mathsf{mkId} : \mathit{string} \to \mathit{uid} * \mathit{uid}_{\mathit{pub}}$

$\mathsf{mkSays} : x : \mathit{uid} \to f : \mathit{predicate}^F \to \mathit{proof}$

$\mathsf{mkSSP} : x : \mathit{uid} \to s : \mathit{string} \to \mathit{proof}$

$\mathsf{mkREL} : f : \mathit{formula} \to \mathit{proof}$

$\mathsf{mkLM} : x : \mathit{pseudo} \to b : \mathit{string} \to \ell : \mathit{list} \to \mathit{proof}$

$\mathsf{mkLNM} : x : \mathit{pseudo} \to \ell : \mathit{list} \to \mathit{proof}$

$\mathsf{mkIDRev} : \mathit{proof} \to s : \mathit{string} \to \mathit{proof}$

$\mathsf{mk}_\wedge : \mathit{proof} * \mathit{proof} \to \mathit{proof}$

$\mathsf{split}_\wedge : \mathit{proof} \to \mathit{proof} * \mathit{proof}$

$\mathsf{mk}_\vee : \mathit{proof} \to \mathit{formula} \to \mathit{proof}$

$\mathsf{extractForm} : p : \mathit{proof} \to \mathit{formula}$

$\mathsf{hide} : \mathit{proof} \to \mathit{formula} \to \mathit{proof}$

$\mathsf{rerand} : \mathit{proof} \to \mathit{formula} \to \mathit{proof}$

$\mathsf{verify}'_{\mathcal{F}} : \mathit{proof} \to f : \mathit{formula} \to \mathit{bool}$

Table A.18.: ML API methods.

```
$\mathsf{verify}'_{\mathcal{F}}$ $(p : \mathit{proof})$ $(f : \mathit{formula})$ $: \mathit{bool}$ =
  let $r$ = $\mathsf{verify}_{\mathcal{F}}$ $p$ $f$;
  if $r = \mathsf{true}$ then
    assert $\mathcal{F}$;
    $r$
  else
    $r$
```

The type-checking of this function is straightforward since the logical formula $\mathcal{F}$ is guaranteed to hold by the return type of $\mathsf{verify}_{\mathcal{F}}$ and the formula $\{r = \mathsf{true}\}$ obtained by the match-statement.

Table A.18 show the augmented ML API. It contains all the methods that are also contained in the RCF API except for the refined verification method that has been substituted by the unrefined method $\mathsf{verify}'_{\mathcal{F}}$. While this interface still depends on the proven formulas (the $\mathsf{verify}'_{\mathcal{F}}$ method is parameterized by the proven formula), it does not contain refinement types.

Regarding assumptions and assertions, we have proven that if a zero-knowledge proof verification succeeds and returns true, then the passed formula (in encoded form) holds. Thus, there is no need for a programmer to explicitly state the assertion since it will always succeed by construction (the needed assumptions have already been internalized in the RCF API).

Using this API protects secret cryptographic material against accidental leakage. In particular, the programmer cannot access the signing key directly but only has access to a handle. Leaking this handle does not reveal the secret key. Of course, using techniques such as reflection, it is still possible to obtain access to the signing key directly, i.e., we only protect against accidental leakage. This, however, is not surprising since the signing key is stored locally on the user's computer and can always be extracted.

We let P_{API} be the code that defines all the API methods (including the necessary library functions). We prove the following theorem.

Theorem A.2. *Let A be an expression such that the set $S := fv(A)$ of free variables consists only of the methods listed in Table A.18. Then $P_{\mathsf{API}}; A$ is robustly safe, i.e., $\emptyset \vdash P_{\mathsf{API}}; A : unit$.*

Proof of Theorem A.2. Follows directly from Lemma A.6 (opponent typability). □

Theorem A.2 guarantees that any ML program linked against the ML API as shown in Table A.18 is well-typed, thus yielding a security by construction guarantee. In particular, the encoding of the verification method ensures that if a call to the method returns `true`, then the expected logical formula holds true (guaranteed by the internalized assertion).

B. ProVerif Code

We give the ProVerif models used to prove anonymity of the case studies from Chapter 3 as well as ObliviAd from Chapter 4.

B.1. Anonymous Webs of Trust

We present the ProVerif script for proving anonymity of the anonymous webs of trust case study. As detailed in Section 3.3.4, the model uses two random values per proof part to distinguish between a re-randomization initiated by a protocol participant and one initiated by the attacker.

```
type ZKVerSig.
type ZKProofVerSig.
type VerKey.
type CommitVerKey.
type SigKey.
type Signature.
type CommitSignature.
type Message.
type CommitMessage.
type Identity.
type RND.

free c : channel.

free Alice : Identity.
free Bob : Identity.

const EpsRnd : RND.

(* proof computation *)

fun zkVerSig(ZKProofVerSig, CommitSignature, CommitMessage,
    CommitVerKey) : ZKVerSig.


(* Alternating honest/attacker randomness for
   signature, message, verification key, zk proof *)

fun computeZkVerSig(RND, RND, RND, RND, RND, RND, RND, RND) : ZKProofVerSig.


(* key derivation *)

fun deriveVk(SigKey) : VerKey.
reduc forall sk : SigKey; isVerKey(deriveVk(sk)) = true.


(* proof opening, opening the actual ZKP is private *)

reduc forall zkp : ZKProofVerSig, cs : CommitSignature, cm : CommitMessage, cvk : CommitVerKey;
    openZkVerSig(zkVerSig(zkp, cs, cm, cvk)) = (zkp, cs, cm, cvk).


(* verkey message wrapper *)

fun vkMsg(VerKey) : Message.
reduc forall vk : VerKey; getVkMsg(vkMsg(vk)) = vk.

(* commitments *)

fun commitSig(Signature, RND, RND) : CommitSignature.
fun commitMsg(Message, RND, RND) : CommitMessage.
fun commitVk(VerKey, RND, RND) : CommitVerKey.

(* to compensate for the verkey wrapper *)
reduc forall vk : VerKey, r : RND, ar : RND; msgVkComEq(commitMsg(vkMsg(vk), r, ar), commitVk(vk, r, ar)) = true.

(* signing and verification*)

fun sign(Message, RND, SigKey) : Signature.

reduc forall m : Message, r : RND, sk : SigKey; verSig(sign(m, r, sk), m, deriveVk(sk)) = m.

(* zero-knowledge verification *)
```

```

reduc forall rs : RND, ars : RND, m : Message, rm : RND, arm : RND, sk : SigKey, rvk : RND, arvk : RND, r : RND, ar : RND, rsig : RND;
    verifyZkVerSig(zkVerSig(
            computeZkVerSig(rs, ars, rm, arm, rvk, arvk, r, ar),
      commitSig(sign(m, rsig, sk), rs, ars),
      commitMsg(m, rm, arm),
      commitVk(deriveVk(sk), rvk, arvk))) = true.

(*
   re-randomization
   this system assumes, that honest principals do not exploit algebraic properties of re-randomization

   in particular, if a principal hides a value, it must be re-randomized before outputting it again (this model enforces this);
   otherwise, the attacker can unhide the value (we also enforce this in the concrete implementation)
*)

reduc forall x : Signature, r : RND, ar : RND, r2 : RND; rerandCommitSig(commitSig(x, r, ar), r2) = commitSig(x, r2, ar) [private].
reduc forall x : Message, r : RND, ar : RND, r2 : RND; rerandCommitMsg(commitMsg(x, r, ar), r2) = commitMsg(x, r2, ar) [private].
reduc forall x : VerKey, r : RND, ar : RND, r2 : RND; rerandCommitVk(commitVk(x, r, ar), r2) = commitVk(x, r2, ar) [private].

reduc forall rs : RND, ars : RND, rm : RND, arm : RND, rvk : RND, arvk : RND, r : RND, ar : RND, rs2 : RND, rm2 : RND, rvk2 : RND, r2 : RND, s : Signature, m : Message, vk : VerKey;
    rerandZkVerSig(zkVerSig(computeZkVerSig(rs, ars, rm, arm, rvk, arvk, r, ar), commitSig(s, rs, ars), commitMsg(m, rm, arm), commitVk(vk, rvk, arvk)), rs2, rm2, rvk2, r2) =
        zkVerSig(computeZkVerSig(rs2, ars, rm2, arm, rvk2, arvk, r2, ar), commitSig(s, rs2, ars), commitMsg(m
, rm2, arm), commitVk(vk, rvk2, arvk)) [private].


reduc forall x : Signature, r : RND, ar : RND, ar2 : RND; att_rerandCommitSig(commitSig(x, r, ar), ar2) = commitSig(x, r, ar2).
reduc forall x : Message, r : RND, ar : RND, ar2 : RND; att_rerandCommitMsg(commitMsg(x, r, ar), ar2) = commitMsg(x, r, ar2).
reduc forall x : VerKey, r : RND, ar : RND, ar2 : RND; att_rerandCommitVk(commitVk(x, r, ar), ar2) = commitVk(x, r, ar2).

reduc forall rs : RND, ars : RND, rm : RND, arm : RND, rvk : RND, arvk : RND, r : RND, ar : RND, ars2 : RND, arm2 : RND, arvk2 : RND, ar2 : RND;
    att_rerandZkVerSig(computeZkVerSig(rs, ars, rm, arm, rvk, arvk, r, ar), ars2, arm2, arvk2, ar2) =
        computeZkVerSig(rs, ars2, rm, arm2, rvk, arvk2, r, ar2).


(* Function Macros *)

letfun rerandZkVerSigAll(p : ZKVerSig) =
    let (=true) = verifyZkVerSig(p) in
    let (zkp : ZKProofVerSig, cs : CommitSignature, cm : CommitMessage, cvk : CommitVerKey) = openZkVerSig(p) in
    new rs2 : RND;
    new rm2 : RND;
    new rvk2 : RND;
    new r2 : RND;
    (rm2, rvk2, rerandZkVerSig(p, rs2, rm2, rvk2, r2)).


(* sig creation *)

letfun createSig(m : Message, sk : SigKey) =
    new r : RND;
    sign(m, r, sk).

letfun createVkSig(vk : VerKey, sk : SigKey) =
    createSig(vkMsg(vk), sk).


(* Zero-Knowledge stuff *)

letfun createZkVerSig_int(s : Signature, m : Message, vk : VerKey, rs : RND, rm : RND, rvk : RND, r : RND) =
    let (=m) = verSig(s, m, vk) in
    let zkp = computeZkVerSig(rs, EpsRnd, rm, EpsRnd, rvk, EpsRnd, r, EpsRnd) in
    zkVerSig(zkp, commitSig(s, rs, EpsRnd), commitMsg(m, rm, EpsRnd), commitVk(vk, rvk, EpsRnd)).

letfun createZkVerSig(s : Signature, m : Message, vk : VerKey) =
    new rs : RND;
    new rm : RND;
    new rvk : RND;
    new r : RND;
    (rm, rvk, createZkVerSig_int(s, m, vk, rs, rm, rvk, r)).


(* Zero-knowledge stuff for chain proofs *)

letfun createZKVerChain(m : Message, sigm : Signature, vk1 : VerKey, sigVk1 : Signature, vk2 : VerKey, sigVk2 : Signature, vk3 : VerKey) =
    new rzkp1 : RND;
    new rzkp2 : RND;
    new rzkp3 : RND;
    new rSigm : RND;
    new rSigVk1 : RND;
    new rSigVk2 : RND;
    new rm : RND;
    new rVk1 : RND;
    new rVk2 : RND;
    new rVk3 : RND;
      (* first proof *)
    let proofMsg = createZkVerSig_int(sigm, m, vk1, rSigm, rm, rVk1, rzkp1) in
      (* second proof *)
    let proofVk1 = createZkVerSig_int(sigVk1, vkMsg(vk1), vk2, rSigVk1, rVk1, rVk2, rzkp2) in
```

```
      (* third proof *)
    let proofVk2 = createZkVerSig_int(sigVk2, vkMsg(vk2), vk3
       , rSigVk2, rVk2, rVk3, rzkp3) in
      (**)
    (m, rm, vk3, rVk3, proofMsg, proofVk1, proofVk2).

letfun verifyZkChain(m : Message, rm : RND, vk3 : VerKey,
   rvk3 : RND, proofMsg : ZKVerSig, proofVk1 : ZKVerSig,
   proofVk2 : ZKVerSig) =
    if verifyZkVerSig(proofMsg) = true then
    if verifyZkVerSig(proofVk1) = true then
    if verifyZkVerSig(proofVk2) = true then
    let (zkp1m: ZKProofVerSig, c1Sigm : CommitSignature,
       c1Msgm : CommitMessage, c1Vk1 : CommitVerKey) =
       openZkVerSig(proofMsg) in
    let (zkp2Vk1 : ZKProofVerSig, c2SigVk1 : CommitSignature,
        c2Vk1 : CommitMessage, c2Vk2 : CommitVerKey) =
       openZkVerSig(proofVk1) in
    let (zkp3Vk2: ZKProofVerSig, c3SigVk2 : CommitSignature,
       c3Vk2 : CommitMessage, c3Vk3 : CommitVerKey) =
       openZkVerSig(proofVk2) in
    let (=c1Msgm) = commitMsg(m, rm, EpsRnd) in
    let (=true) = msgVkComEq(c2Vk1, c1Vk1) in
    let (=true) = msgVkComEq(c3Vk2, c2Vk2) in
    let (=c3Vk3) = commitVk(vk3, rvk3, EpsRnd) in
        true.


(* AWoT Processes *)

let distinguishedSigning(sk : SigKey) =
    in(c, x : VerKey);
    out(c, createVkSig(x, sk));
    0.

let honestSigner =
    in(c, (=true));
    new skH : SigKey;
    out(c, deriveVk(skH));
    (!distinguishedSigning(skH)).

let distinguishedPrincipal(id : Identity, sk : SigKey) =
    let vk = deriveVk(sk) in
    out(c, (id, vk));
    (!distinguishedSigning(sk) | 0)
    .

let distinguishedChain(ska : SigKey, skb : SigKey) =
    in(c, (m : Message, sigVkA : Signature, sigVkB : Signature, vkB1 : VerKey, sigVkB1 : Signature, vkB2 : VerKey, sigVkB2 : Signature, vkC : VerKey));
       (* check first chain * )
    if deriveVk(ska) = verVkSig(sigVkA, deriveVk(ska), vkB1) then
    if vkB1 = verVkSig(sigVkB1, vkB1, vkC) then
       (* check second chain * )
    if deriveVk(skb) = verVkSig(sigVkB, deriveVk(skb), vkB2) then
    if vkB2 = verVkSig(sigVkB2, vkB2, vkC) then
       (**)
    let sigAm = createSig(m, ska) in
    let sigBm = createSig(m, skb) in
    let (=m, rm1 : RND, =vkC, rvkC1 : RND, chainA1 : ZKVerSig, chainA2 : ZKVerSig, chainA3 : ZKVerSig) =
        createZKVerChain(m, sigAm, deriveVk(ska), sigVkA, vkB1, sigVkB1, vkC) in
    let (=m, rm2 : RND, =vkC, rvkC2 : RND, chainB1 : ZKVerSig, chainB2 : ZKVerSig, chainB3 : ZKVerSig) =
        createZKVerChain(m, sigBm, deriveVk(skb), sigVkB, vkB2, sigVkB2, vkC) in
    let (=true) = verifyZkChain(m, rm1, vkC, rvkC1, chainA1, chainA2, chainA3) in
    let (=true) = verifyZkChain(m, rm2, vkC, rvkC2, chainB1, chainB2, chainB3) in
    out(c, choice[(m, rm1, vkC, rvkC1, chainA1, chainA2, chainA3), (m, rm2, vkC, rvkC2, chainB1, chainB2, chainB3)]);
    0.


process
    new ska : SigKey;
    new skb : SigKey;
    (
       distinguishedPrincipal(Alice, ska) |
       distinguishedPrincipal(Bob, skb) |
       (!distinguishedChain(ska, skb)) |
       (!honestSigner)
    )
```

Listing B.1: ProVerif model for anonymity in anonymous webs of trust.

B.2. ObliviAd

First, we list the ProVerif code for profile privacy Listing B.2. We list the code for profile unlinkability in Listing B.3. Finally, the code for billing correctness is shown in Listing B.4.

```
free c, corc.
free c1, c2.
free start.

fun kw1/0.
fun kw2/0.

fun sign/2.
fun enc/3.

fun vk/1.
fun sk/1.
fun ek/1.
fun dk/1.

fun id/1.
fun ad/1.

reduc dec(enc(m,r,ek(k)), dk(k)) = m.
reduc verify(sign(x,sk(y)), vk(y)) = x.

reduc retrieveAd(kw1(), (x, y)) = x;
      retrieveAd(kw2(), (x, y)) = y.

let TPMaccount =
    new TPMchannel;
    (
      (
      in(c, xs1);
      let (ct1,=counter1) = verify(xs1, VKTPM) in
      let pt1 = dec(ct1, DKTPM) in
      out(TPMchannel, (counter1, pt1))
      )
    |
      (
      in(c, xs2);
      let (ct2,=counter2) = verify(xs2, VKTPM) in
      let pt2 = dec(ct2, DKTPM) in
      out(TPMchannel, (counter2, pt2))
      )
    |
      (
      in(TPMchannel, (=counter1, xxpt1));
      in(TPMchannel, (=counter2, xxpt2));
      out(c, choice[xxpt1,xxpt2]);
      out(c, choice[xxpt2,xxpt1])
      )
    ).

let TPMaccountCor =
    (
      in(c, xs1);
      let (ct1,=corcounter) = verify(xs1, VKTPM) in
      let pt1 = dec(ct1, DKTPM) in
      out(c, pt1)
    ).

let TPMdistAdd =
    (
      (
      in(TPMReq1, xp1);
      new r1;
      let adn1 = retrieveAd(xp1, (xad1, xad2)) in
      let cct1 = enc( id(adn1), r1, EKTPM) in
      let scct1 = sign( (cct1,counter1), SKTPM) in
      out(TPMResp1, (adn1, scct1))
      )
    |
      (
      in(TPMReq2, xp2);
      new r2;
      let adn2 = retrieveAd(xp2, (xad1, xad2)) in
      let cct2 = enc(id(adn2), r2, EKTPM) in
      let scct2 = sign( (cct2,counter2), SKTPM) in
      out(TPMResp2, (adn2, scct2))
      )
    |
      (
      in(corc, xadvkw);
      let advad = retrieveAd(xadvkw, (xad1, xad2)) in
      new ulr1;
      let cul11 = enc(id(advad), ulr1, EKTPM) in
      let cul21 = sign( (cul11, corcounter), SKTPM) in
      out(c, (advad, cul21))
      )
    ).

let Client1 =
    in(c1, =start);
    (
      (!out(TPMReq1, choice[kw1(), kw2()]))
    |
      (
      in(TPMResp1, (xrad1, xc1));
      out(c1, xc1)
      )
    ).

let Client2 =
    in(c2, =start);
    (
      (!out(TPMReq2, choice[kw2(), kw1()]))
    |
      (
      in(TPMResp2, (xrad2, xc2));
      out(c2, xc2)
      )
    ).


process
    new TPMseed;
    let SKTPM = sk(TPMseed) in
    let VKTPM = vk(TPMseed) in
    let EKTPM = ek(TPMseed) in
    let DKTPM = dk(TPMseed) in
    out(c, VKTPM);
    out(c, EKTPM);
    (
      !(in(c, (xad1, xad2));
        new counter1;
        new counter2;
        new corcounter;
        new TPMReq1;
        new TPMReq2;
        new TPMResp1;
        new TPMResp2;
        (
          (!TPMdistAdd)
        |
          (!TPMaccount)
        |
          (!TPMaccountCor)
        |
          (!Client1)
        |
          (!Client2)
        )
      )
    )
```

Listing B.2: ProVerif model for profile privacy.

```
free c, corc.
free c1, c2.
free start.

private fun kw1/0.
private fun kw2/0.

fun first/0.
fun second/0.

fun akw1/0.
fun akw2/0.

fun sign/2.
fun enc/3.

fun vk/1.
fun sk/1.
fun ek/1.
fun dk/1.

fun id/1.
fun ad/1.

reduc dec(enc(m,r,ek(k)), dk(k)) = m.
reduc verify(sign(x,sk(y)), vk(y)) = x.

reduc retrieveAd(kw1(), first(),  (a,b,c,d)) = a;
      retrieveAd(kw1(), second(), (a,b,c,d)) = b;
      retrieveAd(kw2(), first(),  (a,b,c,d)) = c;
      retrieveAd(kw2(), second(), (a,b,c,d)) = d.

reduc retrieveAdvAd(akw1(), first(),  (a,b,c,d)) = a;
      retrieveAdvAd(akw1(), second(), (a,b,c,d)) = b;
      retrieveAdvAd(akw2(), first(),  (a,b,c,d)) = c;
      retrieveAdvAd(akw2(), second(), (a,b,c,d)) = d.

let TPMaccount =
    new TPMchannel;
    (
      (
      in(c, xs1);
      let (ct1,=counter1) = verify(xs1, VKTPM) in
      let pt1 = dec(ct1, DKTPM) in
      out(TPMchannel, (counter1, pt1))
      )
    |
      (
      in(c, xs2);
      let (ct2,=counter2) = verify(xs2, VKTPM) in
      let pt2 = dec(ct2, DKTPM) in
      out(TPMchannel, (counter2, pt2))
      )
    |
      (
      in(c, xs3);
      let (ct3,=counter3) = verify(xs3, VKTPM) in
      let pt3 = dec(ct3, DKTPM) in
      out(TPMchannel, (counter3, pt3))
      )
    |
      (
      in(c, xs4);
      let (ct4,=counter4) = verify(xs4, VKTPM) in
      let pt4 = dec(ct4, DKTPM) in
      out(TPMchannel, (counter4, pt4))
      )
    |
      (
      in(TPMchannel, (=counter1, xxpt1));
      in(TPMchannel, (=counter2, xxpt2));
      in(TPMchannel, (=counter3, xxpt3));
      in(TPMchannel, (=counter4, xxpt4));
      out(c, xxpt1);
      out(c, choice[xxpt2,xxpt3]);
      out(c, choice[xxpt3,xxpt2]);
      out(c, xxpt4)
      )
    ).

let TPMaccountCor =
      (
      in(corc, xadvc);
      let (xdrad, =corcounter) = verify(xadvc, VKTPM) in
      let xrad = dec(xdrad, DKTPM) in
      out(c, xrad)
      )
    |
      (
      in(corc, xadvc);
      let (xdrad, =corcounter2) = verify(xadvc, VKTPM) in
      let xrad = dec(xdrad, DKTPM) in
      out(c, xrad)
      ).

let TPMdistAdd =
    (
      (
      in(TPMReqA1, xp1);
      new r1;
      let adn1 = retrieveAd(xp1, first(), (xad1, choice[xad2,
          xad3], choice[xad3, xad2], xad4)) in
      let cct1 = enc( id(adn1), r1, EKTPM) in
      let scct1 = sign( (cct1,counter1), SKTPM) in
      out(TPMRespA1, (adn1, scct1));
```

```
     in(TPMReqA2, xp2);
     new r2;
     let adn2 = retrieveAd(xp2, second(), (xad1, choice[xad2, xad3], choice[xad3, xad2], xad4)) in
     let cct2 = enc( id(adn2), r2, EKTPM) in
     let scct2 = sign( (cct2,counter2), SKTPM) in
     out(TPMRespA2, (adn2, scct2))
     )
   |
     (
     in(TPMReqB1, xp1);
     new r1;
     let adn1 = retrieveAd(xp1, first(), (xad1, choice[xad2, xad3], choice[xad3, xad2], xad4)) in
     let cct1 = enc( id(adn1), r1, EKTPM) in
     let scct1 = sign( (cct1,counter3), SKTPM) in
     out(TPMRespB1, (adn1, scct1));
     in(TPMReqB2, xp2);
     new r2;
     let adn2 = retrieveAd(xp2, second(), (xad1, choice[xad2, xad3], choice[xad3, xad2], xad4)) in
     let cct2 = enc( id(adn2), r2, EKTPM) in
     let scct2 = sign( (cct2,counter4), SKTPM) in
     out(TPMRespB2, (adn2, scct2))
     )
   |
     (
     in(corc, xadvkw);
     new ulr1;
     let xadvad = retrieveAdvAd(xadvkw, first(), (xad1, xad2, xad3, xad4)) in
     let cul11 = enc(id(xadvad), ulr1, EKTPM) in
     let dcul11 = sign( (cul11,  corcounter), SKTPM) in
     out(c, (xadvad, dcul11));
     in(corc, xadvkw2);
     new ulr12;
     let xadvad2 = retrieveAdvAd(xadvkw2, second(), (xad1, xad2, xad3, xad4)) in
     let cul112 = enc(id(xadvad2), ulr12, EKTPM) in
     let dcul112 = sign( (cul112,  corcounter2), SKTPM) in
     out(c, (xadvad2, dcul11))
     )
  ).
```

```

let Client1 =
    in(c1, =start);
    (
      (!out(TPMReqA1, kw1))
    |
      (!out(TPMReqA2, kw1))
    |
      (
      in(TPMRespA1, (xrad1, xc1));
      out(c1, xc1)
      )
    |
      (
      in(TPMRespA2, (xrad1, xc1));
      out(c1, xc1)
      )
    ).

let Client2 =
    in(c2, =start);
    (
      (!out(TPMReqB1, kw2))
    |
      (!out(TPMReqB2, kw2))
    |
      (
      in(TPMRespB1, (xrad2, xc2));
      out(c2, xc2)
      )
    |
      (
      in(TPMRespB2, (xrad2, xc2));
      out(c2, xc2)
      )
    ).


process
    new TPMseed;
    let SKTPM = sk(TPMseed) in
    let VKTPM = vk(TPMseed) in
    let EKTPM = ek(TPMseed) in
    let DKTPM = dk(TPMseed) in
    out(c, VKTPM);
    out(c, EKTPM);
    (
      !(in(c, (xad1, xad2, xad3, xad4));
        new counter1;
        new counter2;
        new counter3;
        new counter4;
        new corcounter;
        new corcounter2;
        new TPMReqA1;
      new TPMReqA2;
      new TPMRespA1;
      new TPMRespA2;
        new TPMReqB1;
      new TPMReqB2;
      new TPMRespB1;
      new TPMRespB2;
        (
    (!TPMdistAdd)
        |
    (!TPMaccount)
        |
    (!TPMaccountCor)
        |
    (!Client1)
        |
    (!Client2)
        )
      )
    )
```

Listing B.3: ProVerif model for profile unlinkability.

```
free c.
free c1.
free start.

fun kw/0.

fun sign/2.
fun enc/3.

fun vk/1.
fun sk/1.
fun ek/1.
fun dk/1.

fun id/1.

reduc dec(enc(m,r,ek(k)), dk(k)) = m.
reduc verify(sign(x,sk(y)), vk(y)) = x.

reduc retrieveAd(kw(), x) = x.

let TPMaccount =
      in(c, xs1);
      let (ct1,=counter) = verify(xs1, VKTPM) in
      let pt1 = dec(ct1, DKTPM) in
      event CountCoin(pt1, counter);
      out(c, pt1);
      event liveness0().

let TPMdistAdH =
    (
      new r;
      in(TPMchannelReq, xprof);
      event liveness1();
      let xretad = retrieveAd(xprof, xad1) in
      let c11 = enc(id(xretad), r, EKTPM) in
      let c21 = sign( (c11, counter), SKTPM) in
      event IssueHonCoin(id(xretad), counter);
      out(TPMchannelResp, (xretad, c21))
    ).

let TPMdistAdC =
    (
      in(c, =start);
      new r;
      let c11 = enc(id(xad1), r, EKTPM) in
      let c21 = sign( (c11, counter), SKTPM) in
      event IssueCompCoin(id(xad1), counter);
      out(c, c21);
      event liveness2()
    ).

```

```
let Client =
    (
      in(c1, =start);
      out(TPMchannelReq, kw);
      in(TPMchannelResp, (x1, x2));
      let (xid, zctr) = verify(x2, VKTPM) in
      event SendCoin(id(x1), zctr);
      out(c1, x2)
    ).

query ev:liveness0().
query ev:liveness1().
query ev:liveness2().

query evinj:CountCoin(id(advertisement), timestamp) ==>
      ((evinj:SendCoin(id(advertisement), timestamp) ==>
         evinj:IssueHonCoin(id(advertisement), timestamp))
      | evinj:IssueCompCoin(id(advertisement), timestamp)).


process
    new TPMseed;
    let SKTPM = sk(TPMseed) in
    let VKTPM = vk(TPMseed) in
    let EKTPM = ek(TPMseed) in
    let DKTPM = dk(TPMseed) in
    out(c, VKTPM);
    out(c, EKTPM);
    (
      !(in(c, xad1);
        (* new counter and new tpm connections for every time
            clients do requests  *)
        (!(new counter;
       new TPMchannelReq;
       new TPMchannelResp;
  (* different counter to prove unlinkability *)
        (
    (TPMdistAdH)
        |
    (TPMdistAdC)
        |
    (TPMaccount)
        |
    (Client)
        )))
      )
    )
```

Listing B.4: ProVerif model for billing correctness.

Bibliography

[1] Martín Abadi, Bruno Blanchet, and Cédric Fournet. Automated Verification of Selected Equivalences for Security Protocols. In *Proc. IEEE Symposium on Logic in Computer Science (LICS'05)*, pages 331–340. IEEE Computer Society Press, 2005. (Cited on pages 5 and 62.)

[2] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for Access Control in Distributed Systems. *ACM Transactions on Programming Languages and Systems*, 15:706–734, September 1993. (Cited on page 48.)

[3] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proc. Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115. ACM Press, 2001. (Cited on pages 61 and 98.)

[4] Attribute-based Credentials for Trust EU Project. `https://abc4trust.eu`. (Cited on page 46.)

[5] Alfarez Abdul-Rahman and Stephen Hailes. A Distributed Trust Model. In *Proc. Wworkshop on New Security Paradigms (NSPW'97)*, pages 48–60. ACM Press, 1997. (Cited on page 64.)

[6] Masayuki Abe, Georg Fuchsbauer, Jens Groth, Kristiyan Haralambiev, and Miyako Ohkubo. Structure-Preserving Signatures and Commitments to Group Elements. In *Proc. Advances in Cryptology (CRYPTO'10)*, volume 6223 of *Lecture Notes in Computer Science*, pages 209–236. Springer-Verlag, 2010. (Cited on pages 11, 23, and 27.)

[7] Agence French-Presse. Study: Facebook 'likes' reveal personal information. *The Raw Story*, March 2013. `http://www.rawstory.com/rs/2013/03/11/study-facebook-likes-reveal-personal-information`. (Cited on page 1.)

[8] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Proc. International Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*. ACM Press, 2013. (Cited on page 106.)

[9] Michal Moskal Andreas Blass, Yuri Gurevich and Itay Neeman. Evidential Authorization. *The Future of Software Engineering*, pages 77–99, 2011. (Cited on pages 13 and 48.)

[10] `http://www.anonymizer.com`. (Cited on pages 84 and 85.)

[11] Andrew W. Appel and Edward W. Felten. Proof-Carrying Authentication. In *Proc. ACM Conference on Computer and Communications Security (CCS'99)*, pages 52–62. ACM Press, 1999. (Cited on pages 10 and 12.)

[12] Claudio A. Ardagna, Jan Camenisch, Markulf Kohlweiss, Ronald Leenes, Gregory Neven, Bart Priem, Pierangela Samarati, Dieter Sommer, and Mario Verdicchio. Exploiting Cryptography for Privacy-Enhanced Aaccess Control: A result of the PRIME Project. *Journal of Computer Security*, 18(1):123–160, January 2010. (Cited on page 46.)

[13] Charles Arthur and Keith Stuart. PlayStation Network users fear identity theft after major data leak. *The Guardian*, April 2011. `http://www.guardian.co.uk/technology/2011/apr/27/playstation-users-identity-theft-data-leak`. (Cited on page 2.)

[14] Donovan Artz and Yolanda Gil. A Survey of Trust in Computer Science and the Semantic Web. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):58–71, 2007. (Cited on page 64.)

[15] Dmitri Asonov and Johann Christoph Freytag. Almost Optimal Private Information Retrieval. In *Proc. Privacy Enhancing Technologies Symposium (PETS'02)*, pages 209–223, 2002. (Cited on page 103.)

[16] Giuseppe Ateniese, Jan Camenisch, Susan Hohenberger, and Breno de Medeiros. Practical Group Signatures without Random Oracles. `http://eprint.iacr.org/2005/385`, 2005. (Cited on page 23.)

[17] Jan Ateniese, Giuseppeand Camenisch, Marc Joye, and Gene Tsudik. A Practical and Provably Secure Coalition-Resistant Group Signature Scheme. In *Proc. Advances in Cryptology (CRYPTO'00)*, volume 1880 of *Lecture Notes in Computer Science*, pages 255–270. Springer-Verlag, 2000. (Cited on page 45.)

[18] Man H. Au, Apu Kapadia, and Willy Susilo. BLACR: TTP-Free Blacklistable Anonymous Credentials with Reputation. In *Proc. Network and Distributed System Security Symposium (NDSS'12)*. Internet Society, 2012. (Cited on page 47.)

[19] Kumar Avijit, Anupam Datta, and Robert Harper. Distributed Programming with Distributed Authorization. In *Proc. ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'10)*, pages 27–38. ACM Press, 2010. (Cited on pages 13 and 105.)

[20] Julian Backes and Stefan Lorenz Kim Pecina. peloba Zero-Knowledge Library. Available at `https://github.com/peloba/zk-library`. (Cited on page 40.)

[21] Michael Backes, Martin P. Grochulla, Cătălin Hriţcu, and Matteo Maffei. Achieving Security Despite Compromise Using Zero-knowledge. In *Proc. IEEE Symposium on Computer Security Foundations (CSF'09)*. IEEE Computer Society Press, 2009. (Cited on page 37.)

[22] Michael Backes, Cătălin Hriţcu, and Matteo Maffei. Type-checking Zero-knowledge. In *Proc. ACM Conference on Computer and Communications Security (CCS'08)*, pages 357–370. ACM Press, 2008. (Cited on page 37.)

[23] Michael Backes, Aniket Kate, Matteo Maffei, and Kim Pecina. ObliviAd: Provably Secure and Practical Online Behavioral Advertising. In *Proc. IEEE Symposium on Security & Privacy (S&P'12)*, pages 257–271. IEEE Computer Society Press, 2012. (Cited on pages vii, 20, and 81.)

[24] Michael Backes, Stefan Lorenz, Matteo Maffei, and Kim Pecina. The CASPA Tool: Causality-based Abstraction for Security Protocol Analysis (Tool Paper). In *Proc. Computer Aided Verification (CAV'08)*, volume 5123 of *Lecture Notes in Computer Science*, pages 419–422. Springer-Verlag, 2008. (Cited on page 37.)

[25] Michael Backes, Stefan Lorenz, Matteo Maffei, and Kim Pecina. Anonymous Webs of Trust. In *Proc. Privacy Enhancing Technologies Symposium (PETS'10)*, volume 6205 of *Lecture Notes in Computer Science*, pages 130–148. Springer-Verlag, 2010. (Cited on pages vii, 4, 11, 44, 45, 51, 61, 64, 65, 77, 105, and 128.)

[26] Michael Backes, Stefan Lorenz, Matteo Maffei, and Kim Pecina. Brief Announcement: Anonymity and Trust in Distributed Systems. In *Proc. Symposium on Principles of Distributed Computing (PODC'10)*, pages 237–238. ACM Press, 2010. (Cited on pages vii and 51.)

[27] Michael Backes, Matteo Maffei, and Cătălin Hriţcu. Union and Intersection Types for Secure Protocol Implementations. In *Proc. Conference on Theory of Security and Applications (TOSCA'11)*, Lecture Notes in Computer Science. Springer-Verlag, 2011. (Cited on pages 13, 36, and 37.)

[28] Michael Backes, Matteo Maffei, and Kim Pecina. A Security API for Distributed Social Networks. In *Proc. Network and Distributed System Security Symposium (NDSS'11)*, pages 35–51. Internet Society, 2011. (Cited on pages vii, 4, 11, 44, 45, 46, 51, 77, and 105.)

[29] Michael Backes, Matteo Maffei, and Kim Pecina. Brief Announcement: Securing Social Networks. In *Proc. Symposium on Principles of Distributed Computing (PODC'11)*, pages 341–342. ACM Press, 2011. (Cited on pages vii and 51.)

[30] Michael Backes, Matteo Maffei, and Kim Pecina. Automated Synthesis of Privacy-Preserving Distributed Applications. In *Proc. Network and Distributed System Security Symposium (NDSS'12)*. Internet Society, 2012. (Cited on pages vii, 9, 13, 48, and 105.)

[31] Michael Backes, Matteo Maffei, Kim Pecina, and Raphael M. Reischuk. G2C: Cryptographic Protocols From Goal-Driven Specifications. In *Proc. Conference on Theory of Security and Applications (TOSCA'11)*, Lecture Notes in Computer Science. Springer-Verlag, 2011. (Cited on page 48.)

[32] Michael Backes, Matteo Maffei, and Dominique Unruh. Zero-Knowledge in the Applied Pi-calculus and Automated Verification of the Direct Anonymous Attestation Protocol. In *Proc. IEEE Symposium on Security & Privacy (S&P'08)*, pages 202–215. IEEE Computer Society Press, 2008. (Cited on page 78.)

[33] Michael Backes, Matteo Maffei, and Dominique Unruh. Computationally Sound Verification of Source Code. In *Proc. ACM Conference on Computer and Communications Security (CCS'10)*, pages 387–398. ACM Press, 2010. (Cited on page 36.)

[34] Michael Backes and Dominique Unruh. Computational Soundness of Symbolic Zero-Knowledge Proofs Against Active Attackers. In *Proc. IEEE Symposium on Computer Security Foundations (CSF'08)*, pages 255–269. IEEE Computer Society Press, 2008. (Cited on page 131.)

[35] Tucker Bailey, Andrea Del Miglio, , and Wolf Richter. The rising strategic risks of cyberattacks. *McKinsey & Company*, 2014. Accessed March 2015. (Cited on page 2.)

[36] Lucas Ballard, Matthew Green, Breno de Medeiros, and Fabian Monrose. Correlation-Resistant Storage via Keyword-Searchable Encryption. `http://eprint.iacr.org/2005/417`, 2005. (Cited on page 23.)

[37] Endre Bangerter, Essam Ghadafi, Stephan Krenn, Ahmad-Reza Sadeghi, Thomas Schneider, Nigel Smart, Joe-Kai Tsay, and Bogdan Warinschi. Final Report on Unified Theoretical Framework of Efficient Zero-Knowledge Proofs of Knowledge. Technical report, CACE: Computer Aided Cryptography Engineering, 2009. `http://zkc.cace-project.eu/resources/ZKPoK_theory.pdf`. (Cited on page 24.)

[38] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. Probabilistic Relational Verification for Cryptographic Implementations. In *Proc. Symposium on Principles of Programming Languages (POPL'14)*, pages 193–205. ACM Press, 2014. (Cited on page 13.)

[39] Lujo Bauer, Scott Garriss, Jonathan M. McCune, Michael K. Reiter, Jason Rouse, and Peter Rutenbar. Device-enabled Authorization in the Grey System. In *Proc. Information Security Conference (ISC'05)*, Lecture Notes in Computer Science, pages 431–445. Springer-Verlag, 2005. (Cited on page 10.)

[40] Lujo Bauer, Michael A. Schneider, Eduard W. Felten, and Andrew W. Appel. Access Control on the Web Using Proof-Carrying Authorization. In *Proc. DARPA Conference on Information Survivability Conference and Exposition (DISCEX'03)*, pages 117–119, Washington, DC, USA, 2003. IEEE Computer Society Press. (Cited on page 10.)

[41] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. Design and Semantics of a Decentralized Authorization Language. In *Proc. IEEE Symposium on Computer Security Foundations (CSF'07)*, pages 3–15. IEEE Computer Society Press, 2007. (Cited on pages 12 and 48.)

[42] Mira Belenkiy, Jan Camenisch, Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Hovav Shacham. Randomizable Proofs and Delegatable Anonymous Credentials. In *Proc. Advances in Cryptology (CRYPTO'09)*, volume 5677 of *Lecture Notes in Computer Science*, pages 108–125. Springer-Verlag, 2009. (Cited on pages 12, 40, 47, 48, 140, and 158.)

[43] Mira Belenkiy, Melissa Chase, Markulf Kohlweiss, and Anna Lysyanskaya. P-signatures and Noninteractive Anonymous Credentials. In *Proc. Theory of Cryptography Conference (TCC'08)*, pages 356–374. Springer-Verlag, 2008. (Cited on page 46.)

[44] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations Among Notions of Security for Public-Key Encryption Schemes. In *Proc. Advances in Cryptology (CRYPTO'98)*, volume 1462 of *Lecture Notes in Computer Science*, pages 26–45. Springer-Verlag, 1998. (Cited on page 85.)

[45] Mihir Bellare, Juan A. Garay, and Tal Rabin. Fast Batch Verification for Modular Exponentiation and Digital Signatures. In *Proc. Advances in Cryptology (EUROCRYPT'98)*, pages 236–250, 1998. (Cited on page 97.)

[46] Mihir Bellare and Chanathip Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. *Journal of Computer Security*, 21(4):469–491, 2008. (Cited on page 85.)

[47] Mihir Bellare and Phillip Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *Proc. ACM Conference on Computer and Communications Security (CCS'93)*, pages 62–73. ACM Press, 1993. (Cited on page 24.)

[48] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: A System for Secure Multi-Party Computation. In *Proc. ACM Conference on Computer and Communications Security (CCS'08)*, pages 257–266, New York, NY, USA, 2008. ACM Press. (Cited on page 26.)

[49] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement Types for Secure Implementations. *ACM Transactions on Programming Languages and Systems*, 33(2):8, 2011. (Cited on pages 13, 35, 36, 37, 39, 109, 110, 112, 175, 181, and 182.)

[50] Josh Bernoff. Do people care about the data you collect? Now, more than ever, they do. *empowered*, 2012. Available at `http://forrester.typepad.com/groundswell/2012/01/do-people-care-about-the-data-you-collect-now-more-than-ever-they-do.html`. (Cited on page 1.)

[51] Alex Biryukov, Ivan Pustogarov, and Ralf-Philipp Weinmann. Trawling for Tor Hidden Services: Detection, Measurement, Deanonymization. In *Proc. IEEE Symposium*

on Security & Privacy (S&P'13), pages 80–94. IEEE Computer Society Press, 2013. (Cited on page 1.)

[52] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proc. IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 82–96. IEEE Computer Society Press, 2001. (Cited on pages 5, 37, 62, 98, and 106.)

[53] Dan Boneh, Ben Lynn, and Hovav Shacham. Short Signatures from the Weil Pairing. *Journal of Cryptology*, 17(4):297–319, 2004. (Cited on page 24.)

[54] Dan Boneh and Hovav Shacham. Group Signatures with Verifier-Local Revocation. In *Proc. ACM Conference on Computer and Communications Security (CCS'04)*, pages 168–177. ACM Press, 2004. (Cited on page 45.)

[55] Emmanuel Bouillon. Taming the Beast : Assess Kerberos-protected Networks, 2009. White paper presented at Black Hat E 2009. (Cited on page 10.)

[56] Stefan Brands, Liesje Demuynck, and Bart De Decker. A Practical System for Globally Revoking the Unlinkable Pseudonyms of Unknown Users. In *Proc. Australasian Conference on Information Security and Privacy (ACISP'07)*, volume 4586 of *Lecture Notes in Computer Science*, pages 400–415. Springer-Verlag, 2007. (Cited on page 46.)

[57] Anne Wells Branscomb. Anonymity, Autonomy, and Accountability: Challenges to the First Amendment in Cyberspaces. *The Yale Law Journal*, 104(7), 1995. (Cited on page 84.)

[58] Emmanuel Bresson, Jacque Stern, and Michael Szydlo. Threshold Ring Signatures and Applications to Ad-hoc Groups. In *Proc. Advances in Cryptology (CRYPTO'02)*, volume 2442 of *Lecture Notes in Computer Science*, pages 465–480. Springer-Verlag, 2002. (Cited on page 45.)

[59] Ernest F. Brickell, Jan Camenisch, and Liqun Chen. Direct Anonymous Attestation. In *Proc. ACM Conference on Computer and Communications Security (CCS'04)*, pages 132–145, 2004. (Cited on page 47.)

[60] Ernie Brickell and Jiangtao Li. Enhanced Privacy ID: a Direct Anonymous Attestation Scheme with Enhanced Revocation Capabilities. In *Proc. ACM Workshop on Privacy in the Electronic Society (WPES'07)*, pages 21–30. ACM Press, 2007. (Cited on page 47.)

[61] Michele Bugliesi, Stefano Calzavara, Fabienne Eigner, and Matteo Maffei. Resource-aware Authorization Policies in Statically Typed Cryptographic Protocols. In *Proc. IEEE Symposium on Computer Security Foundations (CSF'11)*, pages 83–98. IEEE Computer Society Press, 2011. (Cited on page 37.)

[62] Michele Bugliesi, Stefano Calzavara, Fabienne Eigner, and Matteo Maffei. Logical Foundations of Secure Resource Management in Protocol Implementations. In

Proc. Principles of Security and Trust (POST'13), volume 7796 of *Lecture Notes in Computer Science*, pages 105–125. Springer-Verlag, 2013. (Cited on pages 13 and 37.)

[63] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. Authenticity by Tagging and Typing. In *Proc. ACM Workshop on Formal Methods in Security Engineering (FMSE'04)*, pages 1–12. ACM Press, 2004. (Cited on page 37.)

[64] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. Compositional Analysis of Authentication Protocols. In *Proc. European Symposium on Programming (ESOP'04)*, volume 2986 of *Lecture Notes in Computer Science*, pages 140–154. Springer-Verlag, 2004. (Cited on page 37.)

[65] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. Analysis of Typed-Based Analyses of Authentication Protocols. In *Proc. IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 112–125. IEEE Computer Society Press, 2005. (Cited on page 37.)

[66] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. Dynamic Types for Authentication. *Journal of Computer Security*, 15(6):563–617, 2007. (Cited on page 37.)

[67] Elizabeth Butler, Elizabeth McCann, and Joseph Thomas. Privacy Setting Awareness on Facebook and Its Effect on User-Posted Content. *Human Communication*, pages 39–55, 2011. (Cited on page 1.)

[68] Giorgio Calandriello, Panos Papadimitratos, Jean-Pierre Hubaux, and Antonio Lioy. Efficient and Robust Pseudonymous Authentication in VANET. In *Proc. ACM International Workshop on Vehicular Ad Hoc Networks (VANET'07)*, pages 19–28. ACM Press, 2007. (Cited on page 46.)

[69] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. *Request for Comments 4880: OpenPGP Message Format.* Internet Engineering Task Force, 2007. (Cited on page 57.)

[70] Jan Camenisch, Rafik Chaabouni, and abhi shelat. Efficient Protocols for Set Membership and Range Proofs. In *Proc. International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT'08)*, volume 5350 of *Lecture Notes in Computer Science*, pages 234–252. Springer-Verlag, 2008. (Cited on pages 25 and 47.)

[71] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. An Accumulator Based on Bilinear Maps and Efficient Revocation for Anonymous Credentials. In *Proc. International Conference on Practice and Theory in Public Key Cryptography (PKC'09)*, volume 5443 of *Lecture Notes in Computer Science*, pages 481–500. Springer-Verlag, 2009. (Cited on page 47.)

[72] Jan Camenisch and Anna Lysyanskaya. A Signature Scheme with Efficient Protocols. In *Proc. International Conference on Security in Communication Networks (SCN'02)*,

volume 2576 of *Lecture Notes in Computer Science*, pages 268–289. Springer-Verlag, 2002. (Cited on page 45.)

[73] Jan Camenisch and Markus Michels. Proving in Zero-Knowledge that a Number is the Product of Two Safe Primes. In *Proc. Advances in Cryptology (EUROCRYPT'98)*, volume 1592 of *Lecture Notes in Computer Science*, pages 107–122. Springer-Verlag, 1998. (Cited on pages 11, 44, 45, and 61.)

[74] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally Composable Two-Party and Multi-Party Secure Computation. In *Proc. ACM Symposium on Theory of Computing (STOC'02)*, pages 494–503, New York, NY, USA, 2002. ACM Press. (Cited on page 26.)

[75] Marco Carbone, Mogens Nielsen, and Vladimiro Sassone. A Formal Model for Trust in Dynamic Networks. In *International Conference on Software Engineering and Formal Methods (SEFM '03)*, pages 54–64. IEEE Computer Society Press, 2003. (Cited on page 64.)

[76] Germano Caronni. Walking the Web of Trust. In *Proc. IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'00)*, pages 153–158. IEEE Computer Society Press, 2000. (Cited on page 64.)

[77] Dario Catalano and Dario Fiore. Vector Commitments and their Applications. In *Proc. International Conference on Practice and Theory in Public Key Cryptography (PKC'13)*, volume 7778 of *Lecture Notes in Computer Science*, pages 55–72. Springer-Verlag, 2013. (Cited on page 47.)

[78] Dario Catalano, Dario Fiore, and Mariagrazia Messina. Zero-Knowledge Sets with Short Proofs. In *Proc. Advances in Cryptology (EUROCRYPT'08)*, volume 4965 of *Lecture Notes in Computer Science*, pages 433–450. Springer-Verlag, 2008. (Cited on page 47.)

[79] Avik Chaudhuri and Deepak Garg. PCAL: Language Support for Proof-Carrying Authorization Systems. In *Proc. European Symposium on Research in Computer Security (ESORICS'09)*, pages 184–199. Springer-Verlag, 2009. (Cited on page 48.)

[80] David Chaum. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Communications of the ACM*, 24(2):84–88, 1981. (Cited on page 87.)

[81] David Chaum. Blind Signatures for Untraceable Payments. In *Proc. Advances in Cryptology (CRYPTO'82)*, pages 199–203, 1982. (Cited on page 86.)

[82] David Chaum. Security without Identification: Transaction Systems to Make Big Brother Obsolete. *Communications of the ACM*, 28(10):1030–1044, October 1985. (Cited on pages 11 and 46.)

[83] David Chaum, Jan-Hendrik Evertse, and Jeroen van de Graaf. An Improved Protocol for Demonstrating Possession of Discrete Logarithms and Some Generalizations. In *Proc. Advances in Cryptology (EUROCRYPT'87)*, volume 304 of *Lecture Notes in Computer Science*, pages 127–141. Springer-Verlag, 1987. (Cited on page 11.)

[84] David Chaum, Amos Fiat, and Moni Naor. Untraceable Electronic Cash. In *Proc. Advances in Cryptology (CRYPTO'88)*, pages 319–327, New York, NY, USA, 1990. Springer-Verlag. (Cited on page 86.)

[85] David Chaum and Eugene van Heyst. Group Signatures. In *Proc. Advances in Cryptology (EUROCRYPT'91)*, volume 547 of *Lecture Notes in Computer Science*, pages 257–265. Springer-Verlag, 1991. (Cited on page 45.)

[86] Adnan A Chawdhry, Karen Paullet, and David M Douglas. Data Privacy: Are We Accidentally Sharing Too Much Information? In *Proc. Conference for Information Systems Applied Research (CONISAR'13)*, 2013. (Cited on page 1.)

[87] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM*, 45:965–981, November 1998. (Cited on pages 4, 86, and 103.)

[88] Amit Chowdhry. Facebook Relaunches Atlas For Marketers To Serve Targeted Ads Across Multiple Devices. *Forbes*, 2014. Available at `http://www.forbes.com/sites/amitchowdhry/2014/09/29/facebook-relaunches-atlas-for-marketers-to-serve-targeted-ads-across-multiple-devices/`. (Cited on page 1.)

[89] Cristian Coarfa, Peter Druschel, and Dan S. Wallach. Performance Analysis of TLS Web Servers. *ACM Transactions on Computer Systems*, 24(1):39–69, 2006. (Cited on page 97.)

[90] Federal Trade Commission. FTC Staff Report: Self-Regulatory Principles For Online Behavioral Advertising. `http://www.ftc.gov/opa/2009/02/behavad.shtm`, Feb 2009. Accessed November 2013. (Cited on pages 2 and 102.)

[91] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols. In *Proc. Advances in Cryptology (CRYPTO'94)*, volume 839 of *Lecture Notes in Computer Science*, pages 174–187. Springer-Verlag, 1994. (Cited on pages 44 and 45.)

[92] Ronald Cramer and Victor Shoup. A Practical Public Key Cryptosystem Provably Secure against Adaptive Chosen Ciphertext Attack. In *Proc. Advances in Cryptology (CRYPTO'98)*, volume 1462 of *Lecture Notes in Computer Science*, pages 13–25. Springer-Verlag, 1998. (Cited on page 85.)

[93] Ivan Damgård. On Σ-protocols. `http://www.cs.au.dk/~ivan/Sigma.pdf`. (Cited on page 44.)

[94] Angelo De Caro. jPBC Library. http://libeccio.dia.unisa.it/projects/jpbc/download.html. (Cited on page 40.)

[95] Stéphanie Delaune, Steve Kremer, and Mark Ryan. Coercion-resistance and receipt-freeness in electronic voting. In *Proc. IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 28–42. IEEE Computer Society Press, 2006. (Cited on page 106.)

[96] Stéphanie Delaune, Steve Kremer, and Mark Ryan. Verifying Privacy-Type Properties of Electronic Voting Protocols. *Journal of Computer Security*, 17:435–487, 2009. (Cited on page 89.)

[97] Xuan Ding, Lan Zhang, Zhiguo Wan, and Ming Gu. De-Anonymizing Dynamic Social Networks. In *Proc. Global Communications Conference (GLOBECOM'11)*, pages 1–6, 2011. (Cited on page 1.)

[98] Xuhua Ding, Yanjiang Yang, Robert H. Deng, and Shuhong Wang. A new hardware-assisted PIR with O(n) shuffle cost. *International Journal of Information Security*, 9(4):237–252, 2010. (Cited on page 103.)

[99] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The Second-Generation Onion Router. In *Proc. USENIX Security Symposium (USENIX'04)*, pages 303–320. USENIX Association, 2004. (Cited on page 2.)

[100] Charles Duhigg. How Companies Learn Your Secrets. *New York Times*, 2012. Available at http://www.nytimes.com/2012/02/19/magazine/shopping-habits.html. (Cited on page 4.)

[101] Karen Easterbrook, Kevin Kane, Lan Nguyen, Christian Paquin, and Greg Zaverucha. U-Prove. http://research.microsoft.com/en-us/projects/u-prove. (Cited on page 46.)

[102] eBizMBA Inc. Top 15 Most Popular Search Engines, Februrary 2015. http://www.ebizmba.com/articles/search-engines, 2015. Accessed Februrary 2015. (Cited on page 1.)

[103] eBizMBA Inc. Top 15 Most Popular Social Networking Sites, January 2015. http://www.ebizmba.com/articles/social-networking-websites, 2015. Accessed January 2015. (Cited on page 1.)

[104] Fabienne Eigner and Matteo Maffei. Differential Privacy by Typing in Security Protocols. In *Proc. IEEE Symposium on Computer Security Foundations (CSF'13)*, pages 272–286. IEEE Computer Society Press, 2013. (Cited on pages 13, 37, and 106.)

[105] Taher El Gamal. A Public Key Cryptosystem and a Signature Scheme based on Discrete Logarithms. In *Proc. Advances in Cryptology (CRYPTO'84)*, pages 10–18, New York, NY, USA, 1985. Springer-Verlag. (Cited on page 22.)

[106] European Commission. General Data Protection Regulation. `http://ec.europa.eu/justice/data-protection/document/review2012/com_2012_11_en.pdf`. Accessed January 2015. (Cited on page 2.)

[107] David S. Evans. The Online Advertising Industry: Economics, Evolution, and Privacy. *Journal of Economic Perspectives*, 23(3):37–60, 2009. (Cited on page 103.)

[108] `https://www.facebook.com/advertising/how-it-works`. (Cited on page 1.)

[109] Federal Trade Commission. Protecting Consumer Privacy in an Era of Rapid Change: Recommendations For Businesses and Policymakers. `http://ftc.gov/os/2012/03/120326privacyreport.pdf`, 2012. Accessed January 2015. (Cited on page 2.)

[110] Juan Feng, Hemant K. Bhargava, and David M. Pennock. Implementing Sponsored Search in Web Search Engines: Computational Evaluation of Alternative Mechanisms. *INFORMS Journal on Computing*, 19:137–148, January 2007. (Cited on page 84.)

[111] Amos Fiat and Adi Shamir. How To Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Proc. Advances in Cryptology (CRYPTO'87)*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer-Verlag, 1987. (Cited on pages 24 and 45.)

[112] Riccardo Focardi and Matteo Maffei. *Types for Security Protocols*, volume 5, chapter 7, pages 143–181. IOS Press, 2010. (Cited on page 37.)

[113] Tynan Ford. Cyber Attacks and Their Impact on Business. *globalEDGE*, 2014. Accessed March 2015. (Cited on page 2.)

[114] Matthew Fredrikson and Benjamin Livshits. RePriv: Re-imagining Content Personalization and In-browser Privacy. In *Proc. IEEE Symposium on Security & Privacy (S&P'11)*, pages 131–146. IEEE Computer Society Press, 2011. (Cited on pages 4, 89, 103, and 104.)

[115] David Mandell Freeman. Converting Pairing-Based Cryptosystems from Composite-Order Groups to Prime-Order Groups. In *Proc. Advances in Cryptology (EUROCRYPT'10)*, volume 6110 of *Lecture Notes in Computer Science*, pages 44–61. Springer-Verlag, 2010. (Cited on page 41.)

[116] Steven D. Galbraith, Kenneth G. Paterson, and Nigel P. Smart. Pairings for Cryptographers. *Discrete Applied Mathematics*, 156(16):3113–3121, September 2008. (Cited on page 21.)

[117] Steven D. Galbraith and Victor Rotger. Easy decision-diffie-hellman groups. *LMS Journal of Computation and Mathematics*, 7:201–218, 2004. (Cited on page 23.)

[118] Deepak Garg and Frank Pfenning. A Proof-Carrying File System. In *Proc. IEEE Symposium on Security & Privacy (S&P'10)*, pages 349–364. IEEE Computer Society Press, 2010. (Cited on pages 10, 12, 48, and 105.)

[119] Craig Gentry and Zulfikar Ramzan. Single-Database Private Information Retrieval with Constant Communication Rate. In *Proc. International Colloquium on Automata, Languages and Programming (ICALP'05)*, volume 3580 of *Lecture Notes in Computer Science*, pages 803–815. Springer-Verlag, 2005. (Cited on page 103.)

[120] Oded Goldreich. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *Proc. ACM Symposium on Theory of Computing (STOC'87)*, pages 182–194. ACM Press, 1987. (Cited on page 86.)

[121] Oded Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, 2001. (Cited on page 22.)

[122] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that Yield Nothing but Their Validity or All Languages in NP have Zero-Knowledge Proof Systems. *Journal of the ACM*, 38(3):690–728, 1991. (Cited on pages 3 and 22.)

[123] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43:431–473, May 1996. (Cited on pages 93 and 102.)

[124] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. *SIAM Journal on Computing*, 17(2):281–308, April 1988. (Cited on pages 23 and 85.)

[125] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious RAM Simulation with Efficient Worst-Case Access Overhead. *WPES*, abs/1107.5093, 2011. (Cited on page 93.)

[126] `http://www.wired.com/2011/07/google-revenue-sources/`. (Cited on page 1.)

[127] Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 4(11):451–521, 2003. (Cited on page 37.)

[128] Jens Groth and Amit Sahai. Efficient Non-interactive Proof Systems for Bilinear Groups. In *Proc. Advances in Cryptology (EUROCRYPT'08)*, volume 4965 of *Lecture Notes in Computer Science*, pages 415–432. Springer-Verlag, 2008. (Cited on pages 11, 22, 23, and 133.)

[129] Saikat Guha, Bin Cheng, and Paul Francis. Privad: Practical Privacy in Online Advertising. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI'11)*, Mar 2011. (Cited on pages 85, 89, and 103.)

[130] Saikat Guha, Alexey Reznichenko, Kevin Tang, Hamed Haddadi, and Paul Francis. Serving Ads from localhost for Performance, Privacy, and Profit. In *Proc. Workshop on Hot Topics in Networks (HotNets)*, 2009. (Cited on page 103.)

[131] Carl A. Gunter. *Semantics of Programming Languages*. Foundations of Computing Series. MIT Press, 1993. (Cited on page 112.)

[132] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest We Remember: Cold-Boot Attacks on Encryption Keys. *Journal of the ACM*, 52(5):91–98, May 2009. (Cited on page 83.)

[133] http://www.healthgrades.com. (Cited on page 13.)

[134] Ryan Henry, Kevin Henry, and Ian Goldberg. Making a Nymbler Nymble Using VERBS. In *Proc. Privacy Enhancing Technologies Symposium (PETS'10)*, pages 111–129. Springer-Verlag, 2010. (Cited on page 47.)

[135] Javier Herranz. Identity-based Ring Signatures from RSA. *Theoretical Computer Science*, 389(1-2):100–117, 2007. (Cited on page 46.)

[136] Kashmir Hill. How Target Figured Out A Teen Girl Was Pregnant Before Her Father Did. *Forbes*, 2012. Available at http://www.forbes.com/sites/kashmirhill/2012/02/16/how-target-figured-out-a-teen-girl-was-pregnant-before-her-father-did/. (Cited on page 4.)

[137] Daniel C. Howe and Helen Nissenbaum. *TrackMeNot: Resisting Surveillance in Web Search*, chapter 23, pages 417–436. Oxford University Press, 2009. (Cited on page 103.)

[138] Jingwei Huang and David Nicol. A Calculus of Trust and its Application to PKI and Identity Management. In *Proc. Symposium on Identity and Trust on the Internet (IDTrust'09)*, pages 23–37. ACM Press, 2009. (Cited on page 64.)

[139] http://www-03.ibm.com/security/cryptocards. (Cited on pages 20 and 83.)

[140] Thomas Icart. How to Hash into Elliptic Curves. In *Proc. Advances in Cryptology (CRYPTO'09)*, pages 303–316, 2009. (Cited on page 24.)

[141] Identity Mixer. http://idemix.wordpress.com/category/projects. (Cited on page 46.)

[142] http://www.jameda.de. (Cited on page 13.)

[143] Rob Jansen, Florian Tschorsch, Aaron Johnson, and Björn Scheuermann. The Sniper Attack: Anonymously Deanonymizing and Disabling the Tor Network. In *Proc. Network and Distributed System Security Symposium (NDSS'14)*. Internet Society, 2014. (Cited on page 1.)

[144] Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. Aura: a Programming Language for Authorization and Audit. *ACM SIGPLAN Notices*, 43(9):27–38, 2008. (Cited on pages 48 and 105.)

[145] Peter C. Johnson, Apu Kapadia, Patrick P. Tsang, and Sean W. Smith. Nymble: Anonymous IP-address Blocking. In *Proc. Privacy Enhancing Technologies Symposium (PETS'07)*, Lecture Notes in Computer Science, pages 113–133. Springer-Verlag, 2007. (Cited on page 47.)

[146] Audun Jøsang. An Algebra for Assessing Trust in Certification Chains. In *Proc. Network and Distributed System Security Symposium (NDSS'99)*. Internet Society, 1999. (Cited on page 64.)

[147] Antoine Joux. The Weil and Tate Pairings as Building Blocks for Public Key Cryptosystems. In *Proc. International Conference on Algorithmic Number Theory (ANTS'02)*, volume 2369 of *Lecture Notes in Computer Science*, pages 20–32. Springer-Verlag, 2002. (Cited on page 21.)

[148] James H. Morris Jr. Protection in Programming Languages. *Communications of the ACM*, 16(1):15–21, January 1973. (Cited on pages 36, 109, and 124.)

[149] Ari Juels. Targeted Advertising ... And Privacy Too. In *CT-RSA'01*, pages 408–424, 2001. (Cited on page 103.)

[150] Przemyslaw Kazienko and Michal Adamski. AdROSA - Adaptive personalization of web advertising. *Inf. Sci.*, 177(11):2269–2295, 2007. (Cited on page 103.)

[151] Fatemeh Khatibloo, Dave Frankland, Amelia Martland, and Allison Smith. Personal Identity Management Success Starts With Customer Understanding. *Forrester Research*, 2012. Available at `http://www.forrester.com/Personal+Identity+Management+Success+Starts+With+Customer+Understanding/fulltext/-/E-RES61039`. (Cited on page 1.)

[152] Aggelos Kiayias, Yiannis Tsiounis, and Moti Yung. Traceable Signatures. In *Proc. Advances in Cryptology (EUROCRYPT'04)*, volume 3027 of *Lecture Notes in Computer Science*, pages 571–589. Springer-Verlag, 2004. (Cited on page 45.)

[153] Myungsun Kim, Hyung Tae Lee, and Jung Hee Cheon. Mutual Private Set Intersection with Linear Complexity. In *Proc. International Conference on Information Security Applications (WISA'11)*, volume 7115 of *Lecture Notes in Computer Science*, pages 219–231. Springer-Verlag, 2011. (Cited on page 47.)

[154] Nancy J. King. Why Privacy Discussions about Pervasive Online Customer Profiling Should Focus on the Expanding Roles of Third-Parties. *International Journal of Private Law*, 4(2):193–229, 2011. (Cited on page 102.)

[155] Saranga Komanduri, Richard Shay, Blase Ur Greg Norcie, and Lorrie Faith Cranor. AdChoices? Compliance with Online Behavioral Advertising Notice and Choice Requirements. Technical Report CMU-CyLab-11-005, Carnegie Mellon University, October 2011. (Cited on page 102.)

[156] Michal Kosinski, David Stillwell, and Thore Graepel. Private traits and attributes are predictable from digital records of human behavior. *Proc. of the National Academy of Sciences*, 110(15):5802–5805, 2013. (Cited on page 4.)

[157] Eduard Kovacs. Web Hosting Provider Hetzner Hacked, Users Advised to Change Passwords. *Softpedia*, June 2013. `http://news.softpedia.com/news/Web-Hosting-Provider-Hetzner-Hacked-Users-Advised-to-Change-Passwords-359368.shtml`. (Cited on page 2.)

[158] Balachander Krishnamurthy and Craig E. Wills. Cat and mouse: content delivery tradeoffs in web access. In *Proc. International World Wide Web Conferences (WWW'06)*, pages 337–346. ACM Press, 2006. (Cited on pages 1 and 82.)

[159] Balachander Krishnamurthy and Craig E. Wills. Privacy diffusion on the web: a longitudinal perspective. In *Proc. International World Wide Web Conferences (WWW'09)*, pages 541–550. ACM Press, 2009. (Cited on pages 1 and 82.)

[160] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A Verified Implementation of ML. In *Proc. Symposium on Principles of Programming Languages (POPL'14)*, pages 179–191. ACM Press, 2014. (Cited on page 13.)

[161] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems*, 10:265–310, November 1992. (Cited on page 48.)

[162] Issie Lapowsky. Apple Will Impose Tougher Security After Celeb Photo Hack. *WIRED Magazine*, 2014. `http://www.wired.com/2014/09/tim-cook-hack/`. (Cited on page 2.)

[163] Jin Li, Man Ho Au, Willy Susilo, Dongqing Xie, and Kui Ren. Attribute-based signature and its applications. In *Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS'10)*, pages 60–69, New York, NY, USA, 2010. ACM Press. (Cited on page 46.)

[164] Zi Lin and Nicholas Hopper. Jack: Scalable Accumulator-Based Nymble System. In *Proc. ACM Workshop on Privacy in the Electronic Society (WPES'10)*, pages 53–62. ACM Press, 2010. (Cited on page 47.)

[165] Stefan Lorenz, Manuel Reinert, Kim Pecina, and Julian Backes. tales: The Anonymous Lecture Evaluation System. Available at `http://tales.peloba.de`. (Cited on page 52.)

[166] Gavin Lowe. A Hierarchy of Authentication Specifications. In *Proc. IEEE Computer Security Foundations Workshop (CSFW'97)*, pages 31–44. IEEE Computer Society Press, 1997. (Cited on page 101.)

[167] Li Lu, Jinsong Han, Yunhao Liu, Lei Hu, Jin-Peng Huai, Lionel Ni, and Jian Ma. Pseudo Trust: Zero-Knowledge Authentication in Anonymous P2Ps. *IEEE Transactions on Parallel and Distributed Systems*, 19(10):1325–1337, 2008. (Cited on pages 11 and 46.)

[168] Ben Lynn. The Pairing-Based Cryptography Library. `http://crypto.stanford.edu/pbc`. (Cited on page 40.)

[169] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *Proc. ACM Conference on Computer and Communications Security (CCS'14)*, pages 311–324. ACM Press, 2013. (Cited on page 102.)

[170] Matteo Maffei and Kim Pecina. Position Paper: Privacy-aware Proof-Carrying Authorization. In *Proc. ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS'11)*. ACM Digital Library, 2011. (Cited on pages vii, 3, 9, 12, 48, and 105.)

[171] Matteo Maffei, Kim Pecina, and Manuel Reinert. Security and Privacy by Declarative Design. In *Proc. IEEE Symposium on Computer Security Foundations (CSF'13)*, pages 81–96. IEEE Computer Society Press, 2013. (Cited on pages vii and 9.)

[172] Delfina Malandrino, Vittorio Scarano, and Raffaele Spinelli. How Increased Awareness Can Impact Attitudes and Behaviors toward Online Privacy Protection. In *Proc. International Conference on Social Computing (SocialCom'13)*, pages 57–62, 2013. (Cited on page 1.)

[173] Leonardo A. Martucci, Sebastian Ries, and Max Mühlhäuser. Sybil-Free Pseudonyms, Privacy and Trust: Identity Management in the Internet of Services. *Journal of Information Processing*, 19:317–331, 2011. (Cited on page 46.)

[174] Ueli Maurer. Modelling a public-key infrastructure. In *Proc. European Symposium on Research in Computer Security (ESORICS'96)*, volume 1146 of *Lecture Notes in Computer Science*, pages 325–350. Springer-Verlag, 1996. (Cited on page 64.)

[175] Jonathan Mayer and Arvind Narayanan. Do Not Track: Universal Web Tracking Opt-Out. `http://donottrack.us`. Accessed February 2015. (Cited on pages 102 and 103.)

[176] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proc. International Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*. ACM Press, 2013. (Cited on page 106.)

[177] Jon McLachlan, Andrew Tran, Nicholas Hopper, and Yongdae Kim. Scalable onion routing with torsk. In *Proc. ACM Conference on Computer and Communications Security (CCS'09)*, pages 590–599. ACM Press, 2009. (Cited on page 102.)

[178] http://www.express-scripts.com. (Cited on page 19.)

[179] Sarah Meiklejohn. An Extension of the Groth-Sahai Proof System, 2009. Master's Thesis, Brown University, Computer Science Department. (Cited on page 25.)

[180] Silvio Micali, Michael Rabin, and Joe Kilian. Zero-Knowledge Sets. In *Proc. IEEE Symposium on Foundations of Computer Science (FOCS'03)*, page 80. IEEE Computer Society Press, 2003. (Cited on page 47.)

[181] Atsuko Miyaji, Masaki Nakabayashi, and Shunzou Takano. New Explicit Conditions of Elliptic Curve Traces for FR-Reduction. *Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 84(5):1234–1243, 2001. (Cited on page 21.)

[182] Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Kim Pecina. Privacy Preserving Payments in Credit Networks: Enabling trust with privacy in online marketplaces. In *Proc. Network and Distributed System Security Symposium (NDSS'15)*, 2015. (Cited on pages 4 and 106.)

[183] Arvind Narayanan and Vitaly Shmatikov. De-Anonymizing Social Networks. In *Proc. IEEE Symposium on Security & Privacy (S&P'09)*, pages 173 – 187. IEEE Computer Society Press, 2009. (Cited on pages 1 and 67.)

[184] Nathan Olivares-Giles and Mat Honan. After Epic Hack, Apple Suspends Over-the-Phone AppleID Password Resets. *WIRED Magazine*, 2012. http://www.wired.com/2012/08/apple-icloud-password-freeze/. (Cited on page 2.)

[185] Femi Olumofin and Ian Goldberg. Revisiting the Computational Practicality of Private Information Retrieval. In *Proc. International Conference on Financial Cryptography and Data Security (FC'11)*, volume 7035 of *Lecture Notes in Computer Science*, pages 158–172, 2011. (Cited on pages 4 and 103.)

[186] Rafail Ostrovsky and III Skeith, WilliamE. A Survey of Single-Database Private Information Retrieval: Techniques and Applications. In Tatsuaki Okamoto and Xiaoyun Wang, editors, *Proc. International Conference on Practice and Theory in Public Key Cryptography (PKC'07)*, volume 4450 of *Lecture Notes in Computer Science*, pages 393–411. Springer-Verlag, 2007. (Cited on page 103.)

[187] Kim Pecina and Manuel Reinert. ASPLADA Library. Available at http://www.lbs.cs.uni-saarland.de/spdd. (Cited on page 40.)

[188] Wei Peng, Feng Li, Xukai Zou, and Jie Wu. A Two-Stage Deanonymization Attack Against Anonymized Social Networks. *IEEE Transactions on Computers*, 63(2):290–303, February 2014. (Cited on page 1.)

[189] Adrian Perrig, Sean W. Smith, Dawn Xiaodong Song, and J. D. Tygar. SAM: A Flexible and Secure Auction Architecture Using Trusted Hardware. In *Proc. International Parallel & Distributed Processing Symposium (IPDPS'01)*, page 170, 2001. http://sparrow.ece.cmu.edu/~adrian/projects/SAM. (Cited on page 96.)

[190] Privacy by Design. http://privacybydesign.ca. (Cited on page 2.)

[191] Jean-François Raymond and Anton Stiglic. Security Issues in the Diffie-Hellman Key Agreement Protocol. *IEEE Transactions on Information Theory*, 22:1–17, 2000. (Cited on page 10.)

[192] Alexey Reznichenko, Saikat Guha, and Paul Francis. Auctions in Do-Not-Track Compliant Internet Advertising. In *Proc. ACM Conference on Computer and Communications Security (CCS'11)*, pages 667–676. ACM Press, 2011. (Cited on pages 84, 96, and 104.)

[193] Ronald Linn Rivest, Adi Shamir, and Yael Tauman. How to Leak a Secret. In *Proc. International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT'01)*, volume 2248 of *Lecture Notes in Computer Science*, pages 552–565. Springer-Verlag, 2001. (Cited on page 45.)

[194] Peter Schartner and Martin Schaffer. Unique User-generated Digital Pseudonyms. In *Proc. International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security (MMM-ACNS'05)*, volume 3685 of *Lecture Notes in Computer Science*, pages 194–205, Berlin, Germany, 2005. Springer-Verlag. (Cited on page 46.)

[195] Claus-Peter Schnorr. Efficient Signature Generation by Smart Cards. *Journal of Cryptology*, 4(3):161–174, 1991. (Cited on page 44.)

[196] Jason Schreier. PlayStation Network Hack Leaves Credit Card Info at Risk. *WIRED Magazine*, 2011. http://www.wired.com/2011/04/playstation-network-hacked/. (Cited on page 2.)

[197] Andrew Shallue and Christiaan E. van de Woestijne. Construction of Rational Points on Elliptic Curves over Finite Fields. In *Proc. International Conference on Algorithmic Number Theory (ANTS'06)*, pages 510–524, Berlin, Germany, 2006. Springer-Verlag. (Cited on page 24.)

[198] Claude Elwood Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 28(4):656–715, July 1949. (Cited on page 20.)

[199] Elaine Shi, T.-H.Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $\mathcal{O}(\log^3 N)$ Worst-Case Cost. In *Proc. International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT'11)*, volume 7073 of *Lecture Notes in Computer Science*, pages 197–214. Springer-Verlag, 2011. (Cited on pages 83, 86, 91, 93, 94, and 102.)

[200] Radu Sion and Bogdan Carbunar. On the Practicality of Private Information Retrieval. In *Proc. Network and Distributed System Security Symposium (NDSS'07)*. Internet Society, 2007. (Cited on page 103.)

[201] Sean W. Smith. Outbound Authentication for Programmable Secure Coprocessors. *International Journal of Information Security*, 3(1):28–41, 2004. (Cited on pages 83, 85, 87, and 89.)

[202] Sean W. Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8):831–860, 1999. (Cited on page 87.)

[203] Mudhakar Srivatsa and Mike Hicks. Deanonymizing Mobility Traces: Using Social Network As a Side-channel. In *Proc. ACM Conference on Computer and Communications Security (CCS'12)*, pages 628–637. ACM Press, 2012. (Cited on page 1.)

[204] Emil Stefanov and Elaine Shi. ObliviStore: High Performance Oblivious Cloud Storage. In *Proc. IEEE Symposium on Security & Privacy (S&P'13)*, pages 253–267. IEEE Computer Society Press, 2013. (Cited on page 102.)

[205] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards Practical Oblivious RAM. In *Proc. Network and Distributed System Security Symposium (NDSS'12)*. Internet Society, 2012. (Cited on page 102.)

[206] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proc. ACM Conference on Computer and Communications Security (CCS'14)*, pages 299–310. ACM Press, 2013. (Cited on page 102.)

[207] The GNU Privacy Guard Team. GnuPG. `http://www.gnupg.org`. (Cited on pages 128 and 174.)

[208] The National Institute of Standards and Technology. Recommendataion for Key Management – Part 1: General (Revision 3). *NIST Special Publications*, 800–57, July 2012. `http://csrc.nist.gov/groups/ST/toolkit/key_management.html`. (Cited on pages 20, 22, 23, 24, and 41.)

[209] The Tor Project. `https://www.torproject.org`. Accessed October 2011. (Cited on pages 84, 85, 102, and 104.)

[210] Vincent Toubiana, Arvind Narayanan, Dan Boneh, Helen Nissenbaum, and Solon Barocas. Adnostic: Privacy Preserving Targeted Advertising. In *Proc. Network and Distributed System Security Symposium (NDSS'10)*, 2010. (Cited on pages 89, 103, and 104.)

[211] Patrick P. Tsang, Man Ho Au, Apu Kapadia, and Sean W. Smith. PEREA: Towards Practical TTP-Free Revocation in Anonymous Authentication. In *Proc. ACM Conference on Computer and Communications Security (CCS'08)*, pages 333–344. ACM Press, 2008. (Cited on page 47.)

[212] Patrick P. Tsang, Apu Kapadia, Cory Cornelius, and Sean W. Smith. Nymble: Blocking Misbehaving Users in Anonymizing Networks. *IEEE Transactions of Dependable and Secure Computing*, 8:256–269, 2011. (Cited on page 47.)

[213] Jayakrishnan Unnikrishnan and Farid Movahedi Naini. De-Anonymizing Private Data by Matching Statistics. In *Proc. Conference on Communication, Control, and Computing (CCC'13)*, pages 1616–1623. IEEE Computer Society Press, 2013. (Cited on page 1.)

[214] Dominique Unruh. The Impossibility of Computationally Sound XOR. `http://eprint.iacr.org/2010/389`, 2010. (Cited on pages 63, 131, and 140.)

[215] Jeffrey A. Vaughan. AuraConf: a Unified Approach to Authorization and Confidentiality. In *Proc. ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'11)*, pages 45–58, New York, NY, USA, 2011. ACM Press. (Cited on page 48.)

[216] `http://www.verisign.com`. (Cited on pages 56, 87, 128, and 174.)

[217] Riley Walters. Cyber Attacks on U.S. Companies in 2014. *The Heritage Foundation*, 2014. Accessed March 2015. (Cited on page 2.)

[218] Huaqun Wang and Hong Yu. A Novel Signer-Admission Ring Signature Scheme from Bilinear Pairings. In *Proc. International Workshop on Education Technology and Computer Science (ETCS'09)*, pages 631–634. IEEE Computer Society Press, 2009. (Cited on page 45.)

[219] Shuhong Wang, Xuhua Ding, Robert H. Deng, and Feng Bao. Private Information Retrieval Using Trusted Hardware. In *Proc. European Symposium on Research in Computer Security (ESORICS'06)*, pages 49–64, 2006. (Cited on page 103.)

[220] Victor Wei, Tsz Hon Yuen, and Fangguo Zhang. Group Signature Where Group Manager, Members and Open Authority are Identity-Based. In *Proc. Australasian Conference on Information Security and Privacy (ACISP'05)*, volume 3574 of *Lecture Notes in Computer Science*, pages 468–480. Springer-Verlag, 2005. (Cited on page 46.)

[221] Yunzhao Wei and YanXiang He. A Pseudonym Changing-Based Anonymity Protocol for P2P Reputation Systems. In *Proc. International Workshop on Education Technology and Computer Science (ETCS'09)*, pages 975–980. IEEE Computer Society Press, 2009. (Cited on page 46.)

[222] Mike Wheatley. Facebook Hacked Again: Only This Time, Anyone Can Do It. *Silicon Angle*, May 2013. `http://siliconangle.com/blog/2013/05/16/facebook-hacked-again-only-this-time-anyone-can-do-it`. (Cited on page 2.)

[223] Peter Williams and Radu Sion. Usable PIR. In *Proc. Network and Distributed System Security Symposium (NDSS'08)*. Internet Society, 2008. (Cited on pages 4, 83, and 86.)

[224] Yinglian Xie, Fang Yu, and Martin Abadi. De-anonymizing the Internet Using Unreliable IDs. In *Proc. 2009 ACM SIGCOMM conference on Data communication*, pages 75–86. ACM Press, 2009. (Cited on page 1.)

[225] Tsz Hon Yuen and Victor Wei. Fast and Proven Secure Blind Identity-Based Signcryption from Pairings. In *Proc. RSA Conference (CTRSA'05)*, volume 3376 of *Lecture Notes in Computer Science*, pages 305–322. Springer-Verlag, Heidelberg, Germany, 2005. (Cited on page 46.)

[226] Kim Zetter. Sony Got Hacked Hard: What We Know and Don't Know So Far. *WIRED Magazine*, 2014. `http://www.wired.com/2014/12/sony-hack-what-we-know/`. (Cited on page 2.)

List of Figures

2.2 Proof overview: anonymity of service-specific pseudonyms. 31

2.3 Proof overview: unlinkability of service-specific pseudonyms. 33

2.5 The results for the computation of a service-specific pseudonym. 42

2.6 The results for the non-membership proof for ℓ lists, a total number of 1000 elements distributed over the lists, and a security parameter of $\eta = 112$ bits. 42

2.7 The results for the membership proof for ℓ lists, a total number of 1000 elements distributed over the lists, and a security parameter of $\eta = 112$ bits. 43

2.8 The results for the identity escrow protocol. 43

3.1 The results for anonymous evaluation proof for various security parameters. The number of threads is fixed to 8. 54

3.2 The results for anonymous evaluation proof for various numbers of concurrent threads. The security parameter is fixed to $\eta = 112$ bits. 55

3.3 The results for anonymous evaluation proof using different CPUs with different clock speed. The number of threads is fixed to 8. 55

3.4 The results for anonymously authenticating a message via a certificate chain of length 2 using various security parameters. The number of threads is fixed to 8. 59

3.5 The results for anonymously authenticating a message via a certificate chain of length 2 using various number of threads. The security parameter is fixed to $\eta = 112$ bits. 60

3.6 The results for the anonymous authentication of a message via a certificate chain of length 2 using different CPUs with different clock speed. The number of threads is fixed to 8. 60

3.7 Anonymity game. 62

3.8 Single chain annotated with trust values. 64

3.9 Web of trust with multiple paths. 64

3.11 Protocol Register: user J registers with user I. 69

3.12 Protocol for getHandles: user J anonymously requests user I's handles. . . . 70

3.13 Protocols for getResource, putResource, and getFriends: user J issues a request to user I. 70

3.14 Protocol for IndirectRegister: user J anonymously issues an indirect registration request to user I. After the successful completion of the protocol, J is in an indirect social relation with user K. 71

3.15 The results for proof used to initiate the registration protocol. The number of threads is fixed to 8. 73
3.16 The results for registration proof used to register a protocol participant. The number of threads is fixed to 8. 74
3.17 The results for proof used in the getHandles protocol. The number of threads is fixed to 8. 74
3.18 The results for the proof used in the getHandles protocol using various number of threads. The security parameter is fixed to $\eta = 112$ bits. 75
3.19 The results for the proof used to initiate the getHandles protocol using different CPUs with different clock speed. The number of threads is fixed to 8. 75
3.20 The results for proof used to initiate the indirect registration protocol. The number of threads is fixed to 8. 76
3.21 The results for the proof used to initiate the indirect registration protocol using various number of threads. The security parameter is fixed to $\eta = 112$ bits. 76
3.22 The results for the proof used to initiate the indirect registration protocol using different CPUs with different clock speed. The number of threads is fixed to 8. 77

4.1 Distribution phase. 88
4.2 Tallying phase. 88
4.3 ORAM Operations for a bucket size of 2 and a tree depth of 4. 91
4.4 Results of our microbenchmark. For experiments a) and b), the tree depth is fixed to 24. For experiment c), the bucket size is fixed to 30. 95
4.5 Overview of the observational equivalence relation for profile privacy. The left side of the picture corresponds to P and the right side to Q. 98
4.6 Overview of the observational equivalence relation for profile unlinkability. The left side of the picture corresponds to P and the right side to Q. 100
4.7 The process P, annotated with logical predicates, used in the verification of the trace property in Equation 4.8. 101

List of Tables

2.1 High-level API interface functions. 14

3.1 Grammar of access control lists. 66

A.1 RCF syntax of types. 110
A.2 ML data type declarations. 115
A.3 ML data type declarations continued. 116
A.4 E_{API}: API RCF interface functions. 121
A.5 Definition of the signing key type *sigkey* and verification key type *verkey*. . 122
A.6 Typed library functions used by the API methods. 124
A.7 Typed auxiliary functions used by the API methods. 125
A.8 Syntax of RCF 176
A.9 Syntax of kinds 176
A.10 Syntax of typing environments. 177
A.11 Rules for well-formedness and deduction. 177
A.12 Kinding rules: $E \vdash T :: \nu$ 178
A.13 Subtyping rules: $E \vdash T <: U$ 178
A.14 Rules for values: $E \vdash M : T$ 179
A.15 Rules for expressions: $E \vdash A : T$ 180
A.16 Judgments of the RCF type system. 180
A.17 Excerpt of the extended rules for expressions: $E \vdash A : T \rightsquigarrow E'$ 206
A.18 ML API methods. 266

List of Listings

2.1 Code for the patient. 15

A.1 Implementation of getOperation^r. 128
A.2 Implementation of getOperation^e. 128
A.3 Implementation of rand. 129
A.4 Implementation of sign. 129
A.5 Implementation of check$_{sig}$. 129
A.6 Implementation of storeSK and restoreSK. 130
A.7 Implementation of computeR. 130
A.8 Implementation of computePsd. 131
A.9 Implementation of computeIDR. 131
A.10 Implementation of commit and openCommit. 131
A.11 Implementation of commit$_{sk}$ and openCommit$_{sk}$. 132
A.12 Implementation of getSome. 132
A.13 Implementation of getRevealed. 132
A.14 Implementation of commitZK and openZK. 133
A.15 Implementation of stripStm. 133
A.16 Implementation of checkZK. 134
A.17 Implementation of fakestm. 135
A.18 Implementation of createZK$^{\mathsf{e}}$. 137
A.19 Implementation of createZK. 138
A.20 Implementation of rerand$_{\mathsf{stm}}$. 140
A.21 Implementation of checkEq^1. 142
A.22 Implementation of checkEq. 143
A.23 Implementation of verify$_{stm}$. 145
A.24 Implementation of the verify function for the adversary. 147
A.25 Implementation of hide$_{\mathsf{stm}}$. 148
A.26 Implementation of combineOr. 149
A.27 Implementation of mkId. 152
A.28 Implementation of mkSays. 152
A.29 Implementation of mkSSP. 154
A.30 Implementation of mkREL. 154
A.31 Implementation of mkLM. 155
A.32 Implementation of mkLNM. 155
A.33 Implementation of mkIDRev. 156
A.34 Implementation of mk$_{\wedge}$. 156

A.35 Implementation of $\mathsf{split}_\wedge$. 157
A.36 Implementation of $\mathsf{mk}_\vee$. 157
A.37 Implementation of hide. 158
A.38 Implementation of rerand. 158
A.39 verify top-level structure 164
A.40 Or-Macro($\mathsf{Or}(\underline{\mathcal{F}_1^\vee}, \underline{\mathcal{F}_2^\vee}), \mathit{zkv}, \mathit{stm}, f)$. 164
A.41 And-Macro($\mathsf{And}(\underline{\mathcal{F}_1^\wedge}, \underline{\mathcal{F}_2^\wedge}), \mathit{stm}, f, \omega_1, \omega_2)$. 165
A.42 Says-Macro($\mathsf{Says}(x_{vk}, P_k^F(x_1, \ldots, x_n)), \mathit{stm}, f, \omega)$ 166
A.43 SSP-Macro($\mathsf{SSP}(x_{vk}, x_s, x_{psd}), \mathit{stm}, f, \omega)$ 167
A.44 Rel-Macro($\mathsf{REL}(x, \mathit{op}, y), \mathit{stm}, f, \omega)$ 169
A.45 Eqn-Macro($\mathsf{EQN}(x, \mathit{op}, y, z), \mathit{stm}, f, \omega)$ 170
A.46 LM-Macro($\mathsf{LM}(x, b, \ell), \mathit{stm}, f, \omega)$ 171
A.47 LNM-Macro($\mathsf{LNM}(x, \ell), \mathit{stm}, f, \omega)$ 172
A.48 Escrow-Macro($\mathsf{EscrowInfo}(x_{EA}, x_{vk}, x_R, x_s, x_{idr}), \mathit{stm}, f, \omega)$ 172
A.49 Excerpt of the implementation of mkSays. 203
A.50 Excerpt of the implementation of mkSSP. 204
A.51 Excerpt of the implementation of $\mathsf{mkIDRev}$. 204
A.52 Excerpt of the implementation of verify. 258
A.53 Excerpt of the verification function for disjunctions. 259
A.54 ML verification function $\mathsf{verify}'_{\mathcal{F}}$. 266

B.1 ProVerif model for anonymity in anonymous webs of trust. 269
B.2 ProVerif model for profile privacy. 276
B.3 ProVerif model for profile unlinkability. 279
B.4 ProVerif model for billing correctness. 284