Saarland University
Department of Computer Science

TEE-based Designs for Network Gateways, Web Authentication, and VM Introspection

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften (Dr.-Ing.)
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

vorgelegt von
Fabian Frank Schwarz

Saarbrücken,
2024

Tag des Kolloquiums:	August 28, 2024
Dekan:	Prof. Dr. Roland Speicher

Prüfungsausschuss:

Vorsitzender:	Prof. Dr. Peter Druschel
Berichterstattende:	Prof. Dr. Christian Rossow
	Dr. Sven Bugiel
	Prof. Dr. Hojoon Lee
Akademischer Mitarbeiter:	Dr. Alexi Turcotte

Abstract

Over the last decades, the complexity of client, server, and network devices has drastically increased—and so has the number of sophisticated attacks against them. New vulnerabilities are steadily being discovered, and attacks are becoming more sophisticated, including full system compromises that undermine software defenses. As a potential solution, CPU vendors and researchers have designed several architectural extensions for trusted execution environments (TEEs). TEEs enable new security schemes using hardware-based isolation of critical software components from a compromised system. However, the benefits of TEEs have still not been considered for many critical network and web authentication services.

Therefore, this dissertation explores how security-critical network and web services can benefit from TEEs. In particular, we propose three TEE-based designs for network firewalls and web authentication. First, we show how client-side TEEs can enable gateway firewalls to enforce trusted per-application network policies. Second, we enhance the TEE protection to the gateway by designing a TEE-based router architecture with isolated network paths and policy enforcement. Third, we combine electronic IDs with TEE-protected cloud services to solve the recovery and cost issues of FIDO2 web authentication. Finally, we design secure remote forensics of services protected by TEE virtual machines while preserving their strong hardware protection.

Zusammenfassung

Über die letzten Jahrzehnte hat sich die Komplexität von Client-, Server-, und Netzwerk-Geräten drastisch erhöht—und ebenso die Anzahl an Angriffen gegen sie. Ständig werden neue Schwachstellen entdeckt und immer komplexere Angriffe entwickelt, einschließlich vollständiger Systemübernahmen. Als Gegenmaßnahme haben Prozessorhersteller und Forscher etliche vertrauenswürdige Ausführungsumgebungen (kurz: TEEs) entworfen. TEEs ermöglichen neue Sicherheitslösungen durch eine hardware-basierte Isolation kritischer Softwarekomponenten vom kompromittierten Restsystem. Die Vorteile von TEEs wurden jedoch für viele kritische Web-Authentifizierungs- und Netzwerk-Dienste noch nicht untersucht.

Diese Dissertation erforscht daher, wie sicherheitskritische Dienste von TEEs profitieren können. Insbesondere schlagen wir drei TEE-basierte Designs für Netzwerk-Firewalls und Web-Authentifizierung vor. Zuerst zeigen wir, wie clientseitige TEEs es Gateway-Firewalls ermöglichen können, vertrauenswürdige Netzwerkregeln je Anwendung durchzusetzen. Zweitens entwerfen wir eine TEE-basierte Router-Architektur mit isolierten Netzwerkpfaden und -regeln. Drittens kombinieren wir elektronische Ausweise mit TEE-geschützten Cloud-Diensten um die Wiederherstellungs- und Kostenprobleme von FIDO2-Web-Authentifizierung zu lösen. Schließlich entwerfen wir eine sichere forensische Fernanalyse von Diensten, die durch vertrauenswürdige virtuelle Maschinen (TEE VMs) geschützt werden.

Background of this Dissertation

This dissertation is based on three peer-reviewed papers, a fourth one currently under major revision (conditional accept), an extended technical report of one of the papers, and unpublished extensions to them. The three peer-reviewed papers have been published at USENIX Security 2020 [P1], RAID 2022 [P2], and CCS 2022 [P3]. The fourth paper is currently under major revision (conditional accept) at USENIX Security 2024 [P4]. The four papers form the basis for Chapters 2 to 5 of this dissertation. The technical report [T1] provides additional information on [P3] and is therefore part of Chapter 4, especially Section 4.6.2.1, 4.6.5.5, 4.7.5, and 4.7.6. A noteworthy extension to SENG [P1]—our firewall extension for trusted per-application policies—which is not part of the original paper, is presented in Section 2.12 of this dissertation. This extension provides an alternative implementation of our per-application firewall policies tailored to Linux's Netfilter packet filtering system.

Papers of the Author

[P1] **Schwarz**, **F.** and Rossow, C. SENG, the SGX-Enforcing Network Gateway: Authorizing Communication from Shielded Clients. In: *29th USENIX Security Symposium*. 2020.

[P2] **Schwarz**, **F.** TrustedGateway: TEE-Assisted Routing and Firewall Enforcement Using ARM TrustZone. In: *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*. ACM, 2022.

[P3] **Schwarz**, **F.**, Do, K., Heide, G., Hanzlik, L., and Rossow, C. FeIDo: Recoverable FIDO2 Tokens Using Electronic IDs. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2022.

[P4] **Schwarz**, **F.** and Rossow, C. 00SEVen – Re-enabling Virtual Machine Forensics: Introspecting Confidential VMs using privileged in-VM Agents. In: *major revision (conditional accept) at USENIX Security 2024*.

Technical Reports of the Author

[T1] **Schwarz**, **F.**, Do, K., Heide, G., Hanzlik, L., and Rossow, C. *FeIDo: Recoverable FIDO2 Tokens Using Electronic IDs (Extended Version)*. Technical Report. `https://publications.cispa.saarland/3894/`. 2023.

I contributed to all papers and the technical report as the main author. In two cases [P1, P4], I was the sole contributor next to a faculty-level co-author, and in the case of [P2], the sole paper author. For our paper on trusted virtual machine introspection [P4], I supervised two student helpers that supported me in implementing the client side of the prototype and its analysis policies: Fabian van Rissenbeck helped me preparing the performance measurements of the policies (Section 5.8.1), Erik Schmidt helped me implementing the remote network connection of the LibVMI-based client prototype (Section 5.8), and both provided detection policies for the rootkit experiments (Section 5.8.2). The FeIDo paper [P3] is the result of a collaboration between

my faculty-level supervisor Christian Rossow, me and Khue Do, Gunnar Heide, and their faculty-level supervisor Lucjan Hanzlik. I was the main contributor to the paper and its implementation, however, all authors contributed with their valuable ideas. Lucjan Hanzlik and Khue Do contributed particularly to the reduction-based security argument of FeIDo (Section 4.7.1) and the unlinkability argument of its anonymous credentials (Section 4.7.4.1). In addition, they provided background information on electronic IDs and helped in finding the data required to calculate the risk of name collisions (Section 4.7.2.3). Gunnar Heide contributed to the browser extension of FeIDo's prototype (Section 4.8.1.2), helped me in implementing the Android application of the prototype (Section 4.8.1.1), and conducted the experiments for the performance evaluation (Section 4.8.2) under my supervision. For the aforementioned Netfilter-specific extension of SENG [P1] (Section 2.12), Leon Trampert was responsible for implementing the prototype under my supervision.

Acknowledgments

First and foremost, I want to thank my supervisor Prof. Dr. Christian Rossow. Our first contact dates back to my bachelor's thesis in 2015, of which he was my supervisor. Ever since then, I had the pleasure to cooperate with him on many different topics and learn a lot from him. Throughout my bachelor's thesis, master's thesis, and various research projects during my PhD, he guided me and provided me with countless advice and insights on how to become a successful researcher. I particularly thank him for giving me the freedom to work on projects that really drove my interest, including the change of my PhD topic right at the beginning from machine learning-based anomaly detection to trusted execution environments. I'm really grateful that he always gave me the opportunity to come up with interesting research ideas on my own. That way, I was able to develop a high degree of independence and expertise, such that I am confident that I will be able to master whatever challenges arise on my next career step—thank you for making that possible. Also special thanks for the many insightful research discussions, and the possibility to attend several workshops and conferences throughout my PhD.

Next, I want to thank Dr. Sven Bugiel who sparked my interest in system security in 2014 during my bachelor studies. Since then, throughout my bachelor studies, master studies, and early PhD phase, I enjoyed joining his lectures and discussing on mobile and trusted computing topics with him. I learned a lot from him on the fundamentals of system security designs, trusted computing, and system research, which had a significant impact on this dissertation. It was also Dr. Bugiel who recommended Prof. Rossow to me as a supervisor for my bachelor's thesis, and thus initiated my successful academic research experiences of the last years. Thank you for all the inspiration and for introducing me to the field of system security research—contributing to research on new system designs that can benefit real-world systems and users has been one of my strongest initial motivations for conducting a PhD. Also thank you again for being a reviewer of this dissertation.

I also thank Prof. Dr. Hojoon Lee for being a reviewer of this dissertation and for our insightful research discussions on system-level defenses. It would be a pleasure for me to cooperate on some future projects.

I thank Dr. Giancarlo Pellegrino for being my supervisor during my first research attempts as a student helper in the group of Prof. Rossow. Special thanks for motivating me to follow the research direction that drives my interest. It was one of our discussions that was the final push to make me move forward and switch from my initial PhD topic on machine-learning based anomaly detection to the field of trusted computing.

A special thanks goes to my former colleagues at CISPA: Dr. Markus Bauer, Dr. Johannes Krupp, Dr. Giorgi Maisuradze, Jonas Bushart, Benedikt Birtel, and Michael Brengel. Thank you for all the discussions, the daily "mensa walks", our trips to Dagstuhl, and the support during my PhD. Thank you to Markus for being the best office mate possible, for sharing all the ups and downs, and all of our discussions—the funny and the professional ones. Special thanks also for supporting me with the formatting of this dissertation. Special thanks to Johannes for your proofreading of and advice on the TrustedGateway paper, which really helped me a lot at that time, and thank you for sharing your post-PhD experiences. Special thanks to Giorgi for teaching

me how to structure and write a research paper, your valuable feedback on my first paper (SENG), and your encouraging words throughout the first two years of my PhD. I'm really happy that I was able to share my PhD journey with so many brilliant but humble people, and am grateful that we have stayed in contact beyond our PhD time.

I also want to thank the following colleagues and experienced researchers that have supported me throughout my PhD journey:

- Prof. Dr. Bernd Finkbeiner for being my mentor during the preparatory phase of my PhD
- Prof. Dr. Cas Cremers for providing me with helpful advice and insights on network protocol security during my first project
- Dhiman Chakraborty for many interesting and funny discussions, and for providing me with practical tips on Arm TrustZone development boards at the beginning of my TrustedGateway project
- Dr. Lucjan Hanzlik, Khue Do, and Gunnar Heide for the great cooperation during our joined FeIDo project
- Leon Trampert for his support on the SENG-Netfilter extension
- Erik Schmidt and Fabian van Rissenbeck for their support on the 00SEVen prototype
- my recent colleagues Marcos Sánchez Bajo, Amit Choudhari, Yepeng (Eric) Pan, and Bogdan Cebere for giving me a warm welcome in Dortmund during my visit, and for the great time in Kopenhagen during the CCS 2023 conference. Special thanks to Marcos for being such a great, warm-hearted room mate during that time. I wish all of you all the best for your PhD journeys!

Thank you to Saarland University, Saarbrücken Graduate School of Computer Science, and the CISPA Helmholtz Center for Information Security for providing me with a great working environment and organizational support throughout the preparatory and dissertation phases of my PhD.

Finally, I want to thank my family for all their mental support, especially my mother who always had my back and encouraged me. Also thanks to my friends, especially to Mark Stutz for all the discussions on my PhD and for the relaxing spare time activities, and to Dr. Patrick Speicher and Dr. Johannes Herrmann for sharing their PhD experiences with me. The time with you helped me recharge my batteries.

All of you supported me in making this dissertation possible.
Danke für alles!

Contents

List of Figures

List of Tables

1 Introduction

Sophisticated attacks against client, server, and network devices are rapidly becoming more prevalent due to the drastically increasing complexity of these devices. With their growing software stacks and permanent network connectivity, protecting their huge attack surfaces has become significantly harder. Attackers are steadily discovering new vulnerabilities and are developing more sophisticated attacks, including full system compromises. As soon as attackers have compromised the system software, e.g., OS kernel or virtualization software, they can entirely undermine existing software defenses and security services, such as process isolation, access control schemes, or network firewalls [160]. The rise in adoption of cloud computing, in which client services are offloaded to virtual machines (VMs) on third-party servers, further adds to the need for protection against untrusted system software. The cloud infrastructures are fully controlled by the third-party vendors, putting malicious insiders or shady vendors in an ideal position for system-level attacks. Furthermore, past vulnerabilities in virtualization software enabled attackers to escape cloud VMs and compromise the host systems—including all co-located tenant VMs [70].

One promising solution suggested by CPU vendors and researchers to address the threat of system-level attacks are hardware-assisted platform extensions that provide trusted execution environments (TEEs). In contrast to sandboxing techniques [235, 127, 100], which aim at confining potentially vulnerable or malicious software, TEEs enable the isolation of security-critical components from the rest of the system. That way, even on a system software compromise, the TEE-isolated services can stay protected and functional. Over the last years, CPU vendors and researchers introduced several TEEs with varying security boundaries, including user space TEEs like Intel SGX [49], TEEs with full system resource partitioning as provided by Arm TrustZone [178], recent VM-level TEEs like AMD SEV-SNP [3] and Intel TDX [106], or customizable research TEEs like CURE [20]. While TEEs differ w.r.t. their protection boundaries, features, or implementation details, all TEEs typically aim at providing a common set of properties: runtime protection of the contained code and data based on access control and/or memory encryption, the capability to attest their initial state (incl. code and data) using signatures or measurements, and support for crypto operations, e.g., deriving TEE-bound encryption keys to protect data at rest. That way, TEEs enable new protection designs rooted in attestable, hardware-isolated components.

TEEs not only offer additional protection but can enable new security features for existing services. There has been wide research on the application of TEEs which gradually pushed their limits beyond the vendor's intended use cases, including tailored protection designs for specific services, such as storage, network, or machine learning [182, 92, 126, 113], TEE containers for general purpose applications [24, 219, 202, 17, 3], or additional runtime protection and monitoring of TEEs [243, 40, 244]. Furthermore, there has been a lot of offensive research on TEEs and TEE-based designs [133, 226, 210], which helped to gradually improve their security. However, despite promising work showing how TEEs can enable new security services [152, 135] or extend existing concepts, e.g., for control-flow attestation [2] or protected system logs [122], the focus has been mainly on the protection of user applications rather than security services. In particular, many security-critical network and web authentication services still do not benefit from TEEs, even though these services are fundamental for the protection of

our daily client, server, and cloud communication.

Therefore, in the first part of this dissertation, we address the meta research question: *(MQ1) How can security-critical network and web authentication services benefit from TEEs?* In order to answer this question, throughout Chapters 2 to 4, we look at three concrete network and web security settings and explore the benefits and feasibility of applying existing TEEs to them. In fact, we propose three TEE-based design extensions for network firewalls and web authentication, which enable new security guarantees, even when facing strong system-level attackers. Our prototypes not only show the practical feasibility of our designs, but additionally, by releasing them as open-source projects (see list on page 9), we support security researchers and practitioners in actually adopting or extending our proposals.

In Chapter 2, we focus on the ability of network firewalls to securely attribute and filter network traffic of client (or server) applications. Gateway firewalls are important gatekeepers that mediate all incoming and outgoing network communication of consumer and enterprise networks. Their ability to attribute traffic to applications is crucial in order to filter malware and attack traffic. However, widely used identifiers for app attribution (e.g., connection port numbers) can be easily spoofed by malware, and client-side attribution modules can be bypassed via process injections [21] or disrupted on a system-level compromise. In Chapter 2, we therefore address the research question: *(RQ1) Can we use TEEs to enable secure per-application traffic attribution and firewall policy enforcement?* Client-side TEEs can provide trusted application identifiers for network traffic that cannot be spoofed even if the client device is compromised. That way, TEEs can allow for secure traffic-to-application attribution, and thus enable gateway firewalls to enforce per-application policies, e.g., blocking communication by malware and restricting traffic to known, fully-patched applications.

In Chapter 3, we focus on the protection of the traffic firewalling and routing by gateway routers. Current consumer and small to medium-sized enterprise (SME) routers expose a huge attack surface prone to vulnerabilities as they build on commodity OSes and incorporate many non-hardened auxiliary services, e.g., web UIs and media services. Unfortunately, on a remote system compromise, attackers can bypass the firewall and routing policies, and record and tamper with all network communication. In fact, over the last years, there have been several actively exploited service vulnerabilities affecting routers. Therefore, motivated by the importance of routers as network gatekeepers and their high security risk, we investigate the research question: *(RQ2) Can we use TEEs to protect the network path and policy enforcement of routers against compromised user and kernel space services?* System-level TEEs like Arm TrustZone are widely deployed and enable full partitioning of the CPU, memory, and peripherals into a secure and non-secure part. That way, these TEEs can provide strong isolation of the network hardware and software services interacting with the network traffic, even if the router OS and auxiliary services are compromised. Therefore, a redesign of standalone gateway routers based on such a TEE can guarantee secure traffic forwarding and policy enforcement, thus increasing trust into the security of consumer and enterprise networks.

In Chapter 4, we move beyond gateway-enforced network traffic filtering by looking at user authentication towards external web services. Due to the plethora of secu-

rity weaknesses imposed by password-based authentication [191, 65, 166, 54], platform vendors introduced token-based authentication, currently standardized via the FIDO2 authentication protocols [84]. FIDO2 augments passwords with web credentials based on public-private key pairs generated by hardware-isolated token devices owned by the users. While the public keys are shared with the web service, the private keys never leave the token such that the credentials stay resistant against theft, including phishing attempts as well as client- and server-side compromises. However, token-based FIDO2 authentication still faces practical challenges that hinder wide user adoption: users are unwilling to buy dedicated token devices, and a loss (or theft) of a token can render all web accounts permanently inaccessible. In fact, most users already posses a token-like device for authentication tasks: electronic IDs (eIDs), such as passports or national IDs. eIDs provide verifiable user information that rarely changes throughout a lifetime and can be used for authentication. However, eIDs are not compatible with FIDO2 and risk leaking private information to third parties. Therefore, in Chapter 4, we answer the research question: *(RQ3) Can we use TEEs to overcome the deployment and account recovery challenges of FIDO2 web authentication based on user eIDs while preserving user data privacy?* TEEs like Intel SGX or AMD SEV are strongly isolated even if the underlying host system is compromised. They can therefore securely process personal user data and derive web credentials based on it, without leaking either to untrusted third parties. Furthermore, the remote attestation of TEEs allows to securely offload TEE-based services to external servers (e.g., cloud) by remotely verifying their protection, removing the necessity of TEE support on user devices.

Having shown the benefits of TEE-based service designs in the first part of this dissertation, in the second part, we look into secure runtime monitoring of TEE-protected services. TEEs can provide strong hardware-based isolation from system-level attacks. However, their contained services can still be vulnerable to software exploitation, e.g., memory corruption causing a malicious code execution inside the TEE. To minimize the risk, the code size of TEE-based services and its associated attack surface is supposed to be small or even verified. However, the complexity of TEE-based services increases, and so does the risk of vulnerabilities [210], which raises the need for additional runtime detection and prevention solutions inside TEEs. Due to their strong isolation, TEEs require new dedicated defenses that cannot rely on the untrusted system software. Therefore, in the second part of this dissertation, we contribute to the meta research question: *(MQ2) In how far can we securely enable attack detection or prevention techniques for TEE-based services?* Researchers have proposed several protection [243, 202, 40], prevention [47, 210], and monitoring [244] solutions that are tailored to a specific TEE. However, while there has been a lot of focus on process-based TEEs like Intel SGX, upcoming VM-level TEEs like AMD SEV-SNP [3] or Intel TDX [106] are still rather unexplored. In fact, TEE VMs expose a much bigger attack surface than small process-level TEEs, because they consist of a whole commodity OS with many user services. Therefore, CPU vendors have recently started announcing new hardware features to support intra-TEE VM protections [105, 3], and Hecate [83] has made a first step towards intra-VM security policies. However, solutions for many common non-TEE defenses, including runtime memory introspection for attack detection and analysis, have still not been explored for TEE VMs.

Therefore, in Chapter 5, we contribute to the runtime security of TEE VMs by addressing the research question: *(RQ4) Can we enable forensic remote introspection of (potentially) compromised TEE VMs without breaking their security guarantees?* Indeed, such forensic capabilities are inherently important in the security ecosystem. They enable attack detection and prevention within TEE VMs and thus add an additional layer of protection, especially against software exploitation. However, the strong memory protection of TEE VMs not only strengthens security against local attackers, but also prevents benign forensic monitoring attempts by the legitimate VM owners. Therefore, to enable versatile but secure remote introspection, we had to overcome this challenge without weakening the protection of the TEE VMs.

Outline and Contributions

This dissertation consists of four main chapters—Chapters 2 to 5—which contribute to two meta research fields (*MQ1,MQ2*) by each answering one of the above research questions (*RQ1-4*). As outlined on page vii, these chapters are mainly based on our research papers [P1, P2, P3, P4]. In the following, we provide a brief overview of the contributions presented in each chapter and how they relate.

SENG: Extending Gateway Firewalls with Per-Application Policies (*RQ1*)

In Chapter 2, we demonstrate how client-side TEEs can enable gateway firewalls to securely attribute network traffic to applications and thus enforce application-specific policies, addressing research question *RQ1*. We present SENG, an extension for gateway firewalls with TEE-based client-side components. SENG uses the Intel SGX process-level TEE to isolate client applications from system-level attackers and securely tunnel their network traffic to the gateway. By leveraging Intel SGX's hardware-based remote attestation, the gateway firewall can precisely identify the application corresponding to a given network tunnel and thus enforce per-application policies. That way, the gateway can block communication by unauthorized applications, including client malware, external attackers, or compromised services.

SENG's client and gateway components cooperate to enable the traffic protection and attribution. SENG's client-side runtime uses a library OS to transparently wrap existing applications inside SGX enclaves and incorporates a lightweight TCP/IP and DNS stack to protect the traffic against system-level attacks. SENG's gateway service remotely attests each SGX enclave, i.e., verifies their initial code and data, to identify the wrapped applications and assigns them unique IP addresses. The gateway firewall can then seamlessly enforce application-specific policies on the associated IP addresses.

We evaluate our open-source prototype and show that SENG provides reasonable network throughput and latency for practical usage. In addition, we implement additional SENG prototypes with different tradeoffs: The SENG-SDK is an alternative client-side component which provides even better performance by dropping the LibOS, but at the cost of requiring manual porting of applications into SGX. SENG-Netfilter is an alternative gateway-side service that is specifically tailored to the Linux iptables firewall for easy policy specification. That way, depending on the setting, SENG can be

tweaked for more application portability, performance, or specific firewall integration.

TrustedGateway: Isolating Network Policy Enforcement of Routers (*RQ2*)

SENG extends gateway firewalls using client-side TEEs, however, it leaves the gateway unprotected. In Chapter 3, we enhance TEE-based protection to the security-critical network services of gateway routers. We design TrustedGateway (TruGW), a new router architecture, that uses the Arm TrustZone system-level TEE to isolate the full network path and policy enforcement from the vulnerable OS and auxiliary user space services, e.g., web UIs. That way, TruGW can guarantee secure traffic I/O, including traffic routing and firewalling, even on a system-level compromise, thus providing a positive answer to *RQ2*. TruGW forwards traffic only selectively to the vulnerable services and enables admins to restrict any communication from and to them.

TruGW leverages the full system partitioning of TrustZone to isolate the critical core network services and drivers and provide them with exclusive access to the physical network interfaced card (NIC). The router OS and services—typically Linux-based with a huge attack surface—are sandboxed in the untrusted (non-secure) TrustZone domain and can interact with the isolated network path only via a new, restricted virtual network interface. TruGW protects all forward traffic against the non-secure services and enables administrators to remotely configure its secure routing and firewall policies.

We implement an open-source prototype of TruGW and show that it achieves reasonable throughput, latency, and firewall performance. TruGW has a network throughput performance of $\geq$90 % compared to an unprotected router, imposes an average latency overhead of 3.4 % on page loads times for client devices routing through TruGW, and faces a minimal firewalling overhead of 0.5 % to 1 % when leveraging stateful firewall rules. We conclude that TruGW protects gateway routers against recently exploited service vulnerabilities with reasonable performance using a widely-available TEE.

FeIDo: Recoverable FIDO2 Authentication Tokens for Everyone (*RQ3*)

SENG and TrustedGateway protect network communication at the perimeter (gateway) of consumer or enterprise networks. However, except for high-security settings, users frequently communicate with external services, especially web services, towards which they must securely authenticate. Therefore, in Chapter 4, we design FeIDo, a TEE-assisted virtual FIDO2 token, to foster widespread user adoption of secure FIDO2 web authentication over insecure, solely password-based authentication. FeIDo combines widely-deployed eIDs, e.g., passports or national IDs, with user-shared, TEE-protected cloud services. That way, FeIDo provides users with direct access to FIDO2 authentication without imposing extra costs, and, in contrast to existing FIDO2 hardware tokens, enables easy account recovery on an eID (token) loss. Thus, FeIDo addresses *RQ3* by showing how TEEs help to overcome the open challenges of FIDO2 using eIDs.

FeIDo uses a client device, e.g., phone, to read unique personal user information from a user's eID and securely derives FIDO2 credentials for web authentication inside the cloud services. As the personal information is independent of a specific eID, users can derive their web credentials with any of their (replacement) eIDs, enabling direct recovery on an eID loss. The client shares personal information with the cloud services

only after ensuring their protection and genuineness using TEE-provided remote attestation. That way, the user can ensure that the TEE-protected cloud services derive the credentials in a privacy-preserving way and guarantee that no personal information is leaked to third parties, including the cloud and web service providers. Optionally, FeIDo can derive anonymous extra user data for web services, e.g., "is of full age".

We implement an open-source prototype of FeIDo consisting of a Firefox extension, Android app, and Linux service based on German ePassports and the Intel SGX TEE. We show our prototype's direct compatibility with the well-known *webauthn.io* FIDO2 test page, and evaluate its authentication performance. With caching enabled, FeIDo requires $\leq$2 s per authentication operation on average, which is comparable to the existing SoloKey and Nitrokey FIDO2 hardware tokens.

00SEVen: Enabling Secure Remote Forensics of TEE VMs (*RQ4*)

While the previous chapters present TEE-based service designs with tailored protection schemes, in Chapter 5, we provide secure remote introspection for any service isolated by VM-level TEEs, thus addressing *RQ4*. We present 00SEVen, a VMI system for AMD's VM-level TEE, called SEV-SNP. 00SEVen leverages recent intra-VM isolation features to provide isolated in-VM agents that can securely perform memory and register introspection as well as memory access traps. That way, 00SEVen enables periodic or event-based analysis, detection, and prevention of in-VM attacks, including software exploits and even kernel-level rootkits. The in-VM agents enable users to securely monitor their SEV-SNP VMs for attacks by remotely exposing the introspection capabilities to forensic clients. To the best of our knowledge, we are the first to securely re-enable VMI for TEE VMs despite their memory encryption.

00SEVen combines SEV-SNP's intra-VM isolation primitives, called VM privilege levels (VMPLs), with new hypervisor extensions to protect the in-VM forensic agents against both, out-of-VM and in-VM system-level attackers, as well as network attackers. 00SEVen uses VMPLs and their memory permissions to deprivilege the VM OS and thus protect the agents against in-VM system-level attackers, e.g., rootkits. For secure remote communication, 00SEVen combines SEV-SNP's remote attestation with new hypervisor support for VMPL-aware device I/O to provide secure, dedicated network channels between the in-VM agents and forensic clients of the VM owners. 00SEVen therefore enables secure remote introspection of SEV-SNP VMs while preserving SEV-SNP's protection guarantees against out-of-VM attackers.

We implement an open-source prototype of 00SEVen for QEMU/KVM with a remote client based on the widely-known LibVMI introspection library. That way, our prototype can support all analysis scripts and tools build on LibVMI, e.g., Volatility's LibVMI integration. We evaluate the effectiveness and performance based on a set of analysis scripts and three open-source Linux rootkits. 00SEVen can successfully detect the rootkits and even prevent infection by them using access traps. Furthermore, our prototype shows a reasonable performance overhead compared to a local, non-TEE VM introspection framework (KVMi), mainly attributed to the network communication.

Concluding Discussion

In Chapter 6, we conclude this dissertation with a summary of the presented contributions and an outlook on future research directions.

Open-source Prototypes

As part of the research covered by this dissertation, we implemented several prototypes that we have published as open-source projects. From our perspective, openly sharing prototypes with the community helps foster new follow-up research projects by making it easier for researchers to enter the field and gather hands-on experience. We encourage the community to use our prototypes for new experiments, build extensions on them, or take them as a reference when implementing entirely new prototypes. Below we provide a list of our open-source prototypes covered by this dissertation:

[S1] *SENG Prototypes*. URL: `https://github.com/sengsgx`.

[S2] *TrustedGateway Prototypes*. URL: `https://github.com/trugw`.

[S3] *FeIDo Prototypes*. URL: `https://github.com/feido-token`.

[S4] *00SEVen Prototypes*. URL: `https://github.com/sev-vmi/00seven`.

2

SENG, the SGX-Enforcing Network Gateway

Authorizing Communication from Shielded Clients

2.1 Motivation

Network administrators face a security-critical dilemma. While they want to tightly contain their hosts, they usually have to relax firewall policies to support a large variety of applications. However, liberal policies like this enable data exfiltration by unknown (and untrusted) client applications. An inability to attribute communication accurately and reliably to applications is at the heart of this problem. Firewall policies are restricted to coarse-grained features that are easy to evade and mimic, such as protocols or port numbers.

Therefore, in this chapter, we address *RQ1* (see page 4), i.e., answer if we can use trusted execution environments (TEEs) to enable secure traffic-to-application attribution and thus per-application firewall policy enforcement. We present SENG, a network gateway extension that leverages client-side TEEs to achieve this goal. SENG shields an application in an SGX-tailored LibOS and transparently establishes an attestation-based DTLS channel between the SGX enclave[1] and the central network gateway. Consequently, administrators can perfectly attribute traffic to its originating application, and thereby enforce fine-grained *per-application* communication policies at a central firewall. Our prototype implementation demonstrates that SENG (i) allows administrators to readily use their favorite firewall to enforce network policies on a *certified* per-application basis and (ii) prevents local system-level attackers from interfering with the shielded application's communication.

2.2 Problem Description

Companies and sovereign institutions aggregate increasing amounts of sensitive digital information, while the number of attacks on them is proliferating steadily at the same time. Attackers regularly infiltrate systems to steal information and disrupt competitors, e.g., using social engineering (phishing) or advanced exploits (watering hole, zero days) [75]. As a response, organizations harden endpoints, deploy network-based attack detection systems, and train their employees. Yet, given the abundance and power of attacks, preventing any kind of information leakage has become practically infeasible, even in highly-secure settings and in absence of internal attackers.

Foremost among these problems is the fact that containing an organization's incoming and outgoing communication is almost impossible. On the one hand, network administrators deploy firewalls and Intrusion Detection Systems (IDS) to tightly control and contain information flows. On the other hand, they have to support a vast diversity of applications and access methods and lack a mapping between which application causes which traffic. This enables internal clients to (possibly unknowingly) leak data by executing untrusted or even malicious software. Furthermore, companies opening their servers to partners lack control over which remote client applications are

[1] note: Intel SGX version 1 served as a basis for SENG, and was widely available on client CPUs at that time. After the release of SENG, Intel deprecated SGX on client CPUs and currently restricts support to server and data center CPUs. However, SENG can still be used for server or cloud applications, and we assume that its concepts can be transferred to other client TEEs. We discuss this briefly in Section 2.13.3, Section 2.13.1, and in the future work section of this dissertation (Section 6.2).

used to access these servers.

One fundamental solution to this problem is a certified attribution of network traffic to its application, which would allow for app-specific communication policies. Existing attempts to attribute traffic fall short in their security guarantees, as they (i) rely on protocol identification and thereby can be evaded by traffic morphing [102], (ii) rely on host-based sensors that can be evaded or manipulated by local attackers, or (iii) are host-based only and cannot be used at central perimeter firewalls. In fact, reliable traffic-to-app attribution is challenging, as attackers can inject code into trusted processes [22] and abuse their identity. For example, if malware injects itself into browsers, it hides its functionality within an otherwise trusted process and thus inherits the browser's identity and privileges. Lacking a hardware-based trust anchor, existing attribution attempts can be fooled by system-level attackers.

To tackle this underlying core problem, we require a design that (i) shields processes from system-level attackers and (ii) gives stronger integrity protection of processes than just their name or any sort of other loose identifier. In fact, trusted execution environments (TEEs) like Intel SGX [49] ensure such hardware-enforced protections and have been the subject of endeavors to shield client applications [126, 88] and outsourced network services [32, 179, 218]. Library operating systems (LibOSes) tailored for SGX wrap and shield unmodified client and server applications, thus protecting legacy applications out of the box [24, 219, 17]. However, while they do enable transparent shielding and attestation, existing LibOSes fail to provide the following two guarantees. First, they rely on the untrusted host's network stack, s.t. local system-level adversaries can still manipulate and redirect traffic (e.g., DNS spoofing, IP/TCP header modification). Second, the network gateway is still entirely blind to the concrete application which is sending and/or receiving data. Gateways can therefore neither block unauthenticated, vulnerable senders (e.g., malware, shadow IT) nor restrict communication with security-critical servers to certain trusted client applications.

2.3 Contributions

In this chapter, we present *SENG*, a network gateway service coupled with a client-side runtime library, which aims to solve the above problems. SENG transparently protects the connections of applications that are shielded in an SGX-tailored LibOS to prevent packet manipulation and redirection attacks by local system-level attackers. Technically, SENG automatically establishes attestation-based, trusted DTLS channels between the SGX enclaves and the central network gateway. Traffic from and to an enclave is wrapped in the respective secure tunnel and thus inherits enclave-to-gateway confidentiality and integrity guarantees. Furthermore, this design allows the gateway to link traffic to the trusted application causing it. Consequently, the gateway can distinguish between traffic from shielded and unshielded applications and can ultimately enforce central fine-grained *per-application* policies. We have designed SENG in such a way that shielded apps are wrapped in an SGX-based LibOS without requiring any modifications. This allows us to shield legacy binaries without source code changes and completely independent of the underlying network protocols. We also provide an alternative SENG design, which operates *without* LibOS and provides SENG support

for enclaves based on Intel's SGX SDK [107] instead. While the latter does require application modifications, it outperforms the LibOS variant in terms of performance.

To demonstrate the general feasibility, we have developed SENG in an open-source (cf. Section 2.14) [S1] C++ prototype based on Graphene-SGX[2] [219]. Our proof-of-concept illustrates the security benefits of an SGX-enforcing gateway. To highlight the two most important merits, SENG (i) allows network administrators to readily use their favorite firewall implementation (e.g., `Netfilter/iptables` [163]) to enforce network policies on a *certified* per-application basis and (ii) prevents local system-level attackers from interfering with the shielded application's communication.

In summary, we make the following contributions:

- We design SENG, which transparently (i.e., without the need of code rewriting) shields applications to protect and attribute their network traffic.
- SENG enables tight control over network communication at the perimeter and thereby mitigates information leakage by untrusted applications. Consequently, central firewalls can enforce the use of particular trusted applications for traffic entering or leaving their network.
- We implement and release a prototype [S1] and thoroughly evaluate its performance based on network- and microbenchmarks as well as a set of real-world client (cURL, Telnet) and server (NGINX) applications.

2.4 Threat Model

Centralized network firewalls ("perimeter firewalls") are a core security instrument in any network [76]. Network administrators typically segment clients and servers into disjoint subnetworks that are interconnected via a central network gateway—a classical demilitarized zone (DMZ) firewall setup, as shown in Figure 2.1. They can then specify firewall policies based on source and destination addresses and protocol information to regulate communication between these segments. To retain security guarantees of perimeter firewalls, administrators usually aim to prohibit secondary WAN connections (e.g., 4G/5G) or other bridges that would subvert the gateway's centralized position.

Unfortunately, perimeter firewalls are restricted to coarse-grained policies. They filter traffic based on host information (IP addresses, port number) and transport protocol (e.g., TCP or UDP). Firewalls cannot filter communication *per application*, as the application source is unknown. Firewalls therefore lack mechanisms to block communication of undesired and/or potentially malicious software. Firewalls have been extended to learn about client programs using host-based sensors [45]. However, these existing app attributions can be undermined when attackers compromise client systems (cf. Section 2.5), as malware can inject into allowlisted processes [22], or escalate its privileges to subvert host sensors.

This challenging setting is exactly our use case. We aim to provide *app-grained traffic attribution* to organizations with stationary clients that are potentially compromised by malware and/or want to isolate untrusted apps. Identical to the firewall

[2]note: Graphene has been renamed to Gramine [97] after the release of SENG

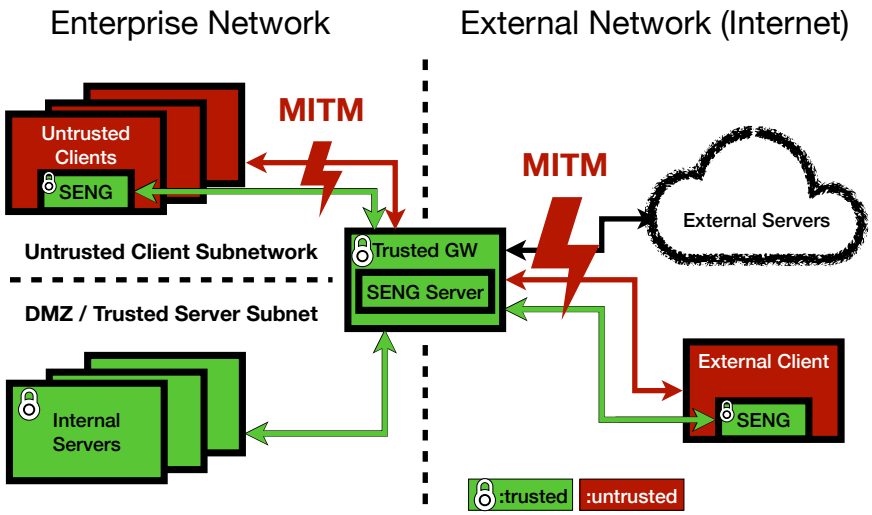

Figure 2.1: Overview of SENG's network topology and threat model

setting ("bastion host"), also in our threat model the firewall and its underlying system is fully trusted. In contrast to firewalls, however, we tolerate a system-level attacker fully controlling the client's software stack, including its OS and hypervisor(s). That is, we do not mistrust the user or its hardware, but allow its host system to be fully compromised. After compromise, attackers will attempt to leak sensitive host information either directly or indirectly by manipulating the network traffic of shielded apps.[3]

To tackle this problem, we leverage trusted hardware to enable firewalls to rely on app identifications for network traffic. Technically, we shield client apps inside an Intel SGX enclave with a trusted LibOS. Administrators can then maintain a list of trusted apps and use their identifiers to create firewall policies that govern which network resources a given app can access. For ease of discussion, we protect client systems and assume that internal servers are not compromised, while our methodology can also be applied to servers in principle.

For our work, we follow the classical SGX threat model. Denial-of-Service (DoS), side-channel attacks, and physical attacks against the CPU are out of scope [233, 225, 133] and can be tackled by orthogonal work [203, 168, 10, 192]. Similarly, enclaves are trusted and free of vulnerabilities. Any disk I/O by the application has to be protected (e.g., hashing files and transparent sealing as provided by existing file system shields and SDK functions [219, 17, 107]). Finally, we assume that all locally exposed enclave interfaces are shielded [204] to avoid an oracle-like API access that could be abused for information leaks based on confused deputy attacks.

2.5 Related Work

Table 2.1 summarizes related work and its deficiencies to cope with our threat model. For the discussion, we consider the following attackers: (a) user space malware

[3] We refer to related work to mitigate covert channels [35, 234] and focus on stopping explicit and malicious information exchange instead.

Table 2.1: Related work grouped into perimeter firewalls with host sensors, host-level firewalls, and secure middleboxes. Assessments follow the metrics, symbols and acronyms outlined in Section 2.5. (*note: QubesOS trusts OS of admin dom0, though)

Project	Components at...	Trust in... OS	Trust in... VMM	Trust in... CPU	Attackers	Central?	(SR2/3) C+I	(SR4) TA	(SR5) Attr	(SR6) ¬IL
SENG	Client, Gateway	no	no	✓	Sys_{cli}, mitm	yes	●	●	●	●
NVM *et al.*	Client, Gateway	yes	-	✓	MW_{user}	yes	○	-	◔	○
Assayer	Client, MBox, Srv	no	yes	✓	Sys_{cli}, mitm	yes	◔	●	○	○
Alcatraz	Cli/Srv, MBox, Gw	no	no	✓	$\text{Sys}_{\text{cli+mbox}}$, mitm	yes	◐	●	○	◐
EndBox	Client, Gateway	no	no	✓	Sys_{cli}, mitm	yes	◐	●	○	◔
iptables MAC	Client	yes	-	✓	MW_{user}	no	◐	-	●	●
ClipOS	Client	yes	-	✓	MW_{user}, mitm	no	◐	-	●	●
QubesOS	Client	no*	yes	✓	Sys_{cli}, mitm	no	◐	-	●	●
SafeBricks	Gateway, MBox	yes	yes	✓	$\text{Sys}_{\textbf{mbox}}$, mitm	yes	◐	-	-	-
LightBox	Gateway, MBox	yes	yes	✓	$\text{Sys}_{\textbf{mbox}}$, mitm	yes	◐	-	-	-

(MW_{user}), (b) system-level attackers at the client (Sys_{cli}) or middlebox (Sys_{mbox}), and (c) on-path MITM attackers ($mitm$). The last four columns rate if an approach fulfills (yes: ●, no: ○, n/a: -) the following requirements: (i) Confidentiality and integrity ($C+I$) of client traffic (incl. IP headers and DNS queries), (ii) traffic authentication (TA) of either protected client or host sensor traffic, (iii) secure (client) traffic-to-app attribution (Attr), and (iv) protection against information leakage ($\neg IL$)—defined as security requirements SR2–SR6 in Section 2.7.

Perimeter Firewalls with Host Information Perimeter firewalls with client-side sensors are most related to SENG. However, they fail to provide reliable traffic-to-app attribution (Attr:◔), which is our central design goal. Host sensors like the Cisco Network Visibility Module (NVM) [45] focus on firewall augmentation with per-flow host data, including app identifiers (e.g., hash of binary, process name). Unfortunately, malware can easily bypass such loose, static identifiers by injection into benign processes [22]. Furthermore, a system-level attacker can completely subvert host sensors such as NVM, as they fully rely on the OS. SOCKS [200] proxies and VPN [61] services also control traffic centrally, but, similarly, they cannot reliably link traffic to its applications.

Isolation-Based Traffic Auditing Assayer [176] uses a client-side hypervisor to augment app-level data of outbound client traffic with traffic statistics and signs it (C+I:◔, TA:●). However, Assayer has no insights into the app identities of annotated traffic (no introspection) and cannot prevent infected or malicious apps from submitting arbitrary traffic for annotation. Thus, Assayer can neither provide traffic-to-app attribution (Attr:○) nor prevent leaks by malware (¬IL:○).

Alcatraz [18] establishes secure tunnels between SGX enclaves integrated into network nodes (incl. clients and gateway). Traffic is securely tunneled between enclaves with hop-specific keys to provide traffic confidentiality and integrity as well as path integrity. While Alcatraz shields tunneled IP traffic from MITM attackers and compromised switches, Alcatraz doesn't protect traffic against client compromise (C+I:◓). Therefore, Alcatraz's client enclaves cannot link traffic to apps (Attr:○) and do not restrict access to the tunnel, s.t. local attackers can send arbitrary authenticated IP packets (¬IL:◓).

EndBox [87] outsources middlebox services to untrusted client systems for scalability. EndBox runs inside an SGX enclave and tunnels all app traffic through a VPN connection (C+I:◓) to the gateway, which blocks traffic that does not arrive through the enclave-terminated VPN tunnel (TA:●). However, similar to Alcatraz, EndBox cannot enforce app-grained policies (Attr:○), as all client apps are untrusted.

Container overlay networks like Slim OS [247] or Docker-based networks [60] assign virtual IP addresses to containers enabling per-container firewall policies at virtual switches. However, they cannot protect against system-level attackers, as they trust the client OS, have no HW-based container identifiers, and do not deal with information leakage.

Client-side Solutions with Host-level Firewalls Host-based firewalls enforce policies directly at the client host, but do not provide an enterprise-wide decision and enforcement point. They are often combined with compartmentalization frameworks which confine apps in sandboxes to mitigate system compromises, which lead to direct firewall subversion.

For example, iptables [163] is the de facto standard firewall configuration tool in Linux. A Debian extension allows policies per user and process ID [112], while mandatory access control (MAC) modules [197, 206] allow fine-grained policies (incl. app-grained). However, none of these approaches shares data with a central gateway firewall. While some firewalls support labeled IPsec, which can negotiate MAC contexts as traffic selectors [114], labeled IPsec faces major configuration and key management complexity. ClipOS [46] is a hardened Linux which sandboxes apps and plans to include multi-level compartmentalization support. However, system-level attackers can subvert all aforementioned approaches.

QubesOS [186] uses Xen to sandbox all apps into isolated VMs and provides per-app VM network policies. QubesOS could thus be modeled to enable app-grained, central policy enforcement by setting up separate VPN tunnels for each application VM and enforce rules on the unique per-app VPN IP addresses. However, this would require a complex client setup and requires trust in the hypervisor. In contrast, we want to root our app attribution in hardware and stay fully compatible with existing gateway firewalls.

SGX-Protected Middlebox Outsourcing Projects such as SafeBricks [179], LightBox [62] and ShieldBox [218] use SGX to protect middlebox services from untrusted cloud or middlebox providers. The approaches mostly differ w.r.t. the focus and implementation. SafeBricks, for instance, uses language-based methods to enforce least privilege on third-party middlebox functions and isolation across chained functions, while LightBox [62] focuses on support for stateful full-stack middlebox functions and high-performance. Gkantsidis et al. [85] additionally propose a middlebox-aware TLS variant (mbTLS) for secure inspection of encrypted client traffic. In contrast to our threat model, these projects trust the client hosts, and thus fail to provide app-to-traffic attribution (Attr:-) and to mitigate information leakage (¬IL:-). The middleboxes can directly benefit from our desired traffic attribution, as they integrate easily (cf. AR3 in Section 2.7.1).

2.6 Background

2.6.1 Intel SGX and Remote Attestation

TEEs provide an abstraction to run a process isolated from the remaining system. TEEs enforce hardware-based protection of the integrity and confidentiality of the contained code and data and have means to prove it to external entities [178, 49].

In the following, we focus on Intel SGX version 1, which forms a basis for our overall design and has been widely supported by client devices when SENG was released (cf. footnote 1 on page 13, and Section 2.13.3). SGX's TEE entities are *enclaves*, which

only rely on the security of the CPU. Enclaves provide a dedicated memory region called enclave page cache (EPC) which is isolated and transparently encrypted and authenticated. The enclave app code is limited to user space instructions, s.t. enclaves depend on the cooperation of the untrusted OS for system calls and interaction with hardware devices. Therefore, SGX provides direct access to untrusted memory and the notion of enclave calls (ECALLs) and outside calls (OCALLs), which allow controlled transitions between the trusted and untrusted world. Furthermore, SGX allows to store data encrypted on the disk via a sealing key derived by the CPU and only accessible to the respective enclave [49].

SGX enclaves can prove their identity and protection to local and remote entities. For local attestation, the CPU creates a cryptographic report of the enclave, which contains a measurement (secure hash) of the initial enclave state. The report is signed by the CPU with the key of the local challenger enclave and can then be passed to the challenger for verification. For remote attestation, the Intel-provided Quoting Enclave (QE) acts as local challenger. The QE then adds the platform state and forwards the resulting quote to a trusted remote attestation service, e.g., Intel Attestation Service (IAS), which checks the platform validity and returns a signed attestation report. Enclaves can bind user data (e.g., keys) to the attestation by embedding custom data into their reports [49, 128].

2.6.2 Enclave Development and Graphene-SGX

There are at least three major paradigms to develop TEE-enabled programs. First, applications can be explicitly designed for certain TEEs by using SDKs [107], which abstract the implementation details. SDKs usually provide APIs for attestation and interactions with the untrusted OS, e.g., for sealing files to disk. Second, semi-automated approaches rely on compiler support and developer-provided source code annotations to split code and data into sensitive and non-sensitive parts. The sensitive parts are then moved inside the isolated enclave and connected to the untrusted parts via shielding layers [146, 204]. Finally, as a third approach, SGX library operating systems securely execute unmodified applications inside enclaves [24, 17, 219, 202]. Due to the user space restriction of enclaves, these LibOSes handle system calls on behalf of the apps and transparently provide POSIX abstractions, e.g., multi-threading support. As the underlying OS is untrusted, the frameworks aim to shield system calls against so-called Iago attacks [39], in which the untrusted operating system manipulates system calls and their return values. However, while LibOSes typically provide shielding layers for secure disk I/O and file integrity, they do *not* protect network traffic and rely on the untrusted host network stack. While SCONE [17] includes transparent TLS proxy support for server apps, it fails to protect client traffic and DNS—both essential requirements of SENG.

In our design, we will follow the third approach, and use the Graphene-SGX LibOS (now called Gramine [97]), which is open-source and allows us to transparently execute unmodified applications in SGX enclaves [219]. Graphene-SGX transparently emulates some system calls internally, while others are delegated to the untrusted OS. A manifest file specifies the enclave size and number of threads, as well as the application and

corresponding dependencies that Graphene-SGX shields. The manifest is part of the enclave identity for attesting the shielded application. While Graphene-SGX provides multi-threading and a file system shield, it provides *no* secure network I/O for apps.

2.7 Design

2.7.1 Requirements

SENG's high-level goals are twofold: (i) prevent attacks against the traffic of SGX-shielded clients, and (ii) allow a central gateway to govern network access on a per-application basis. From these, we derive six security (SR) and three auxiliary (AR) requirements of our system, as shown next. These requirements hold equally for internal and external shielded clients. Five of these requirements (SR2–SR6) heavily rely on the new concepts introduced by our design.

SR1 Code and Data Protection During execution, the integrity and confidentiality of client code (binary, libs) and data (including files) must be protected.

SR2 Network Traffic Integrity and Confidentiality The integrity and confidentiality of network traffic between shielded apps and the gateway is guaranteed, which holds true both for internal *and* external clients.

SR3 Redirection Prevention Traffic from shielded clients must be protected against packet header manipulation by local system-level or on-path MITM attackers until it passes the gateway. Furthermore, local and on-path DNS redirection attacks must be prevented.

SR4 Protection-based Traffic Authentication The gateway must be able to distinguish between traffic of shielded applications and that of non-shielded ones. This property enables network policies that restrict the access to sensitive subnetworks to shielded apps only.

SR5 Accountability of Shielded Traffic The gateway must be able to link shielded traffic back to the respective shielded application to enforce per-app network policies.

SR6 Information Leakage and Remote Control Prevention Whenever SENG enforces that only shielded clients may communicate, local system-level and internal MITM attackers must not be able to leak information to external systems. In the opposite direction, attackers must not be able to send information (e.g., malware commands) from the outside to compromised clients.

AR1 No Client Code Changes. To ease adoption and to support closed-source and legacy applications, we seek for a solution that does not require any code changes in the client app and its dependencies.

AR2 Scalability of Gateway Server The overhead introduced to the gateway server per shielded app and per network connection must be low to allow for scaling.

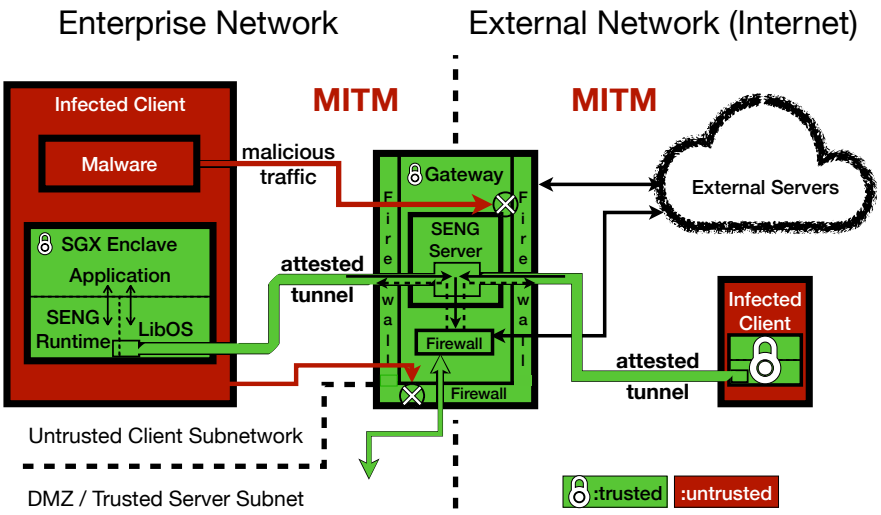

Figure 2.2: High-level overview of the SENG architecture

AR3 Compatibility with other Gateway Services The protection and authentication techniques used by SENG should not interfere with other services on the network gateway, such as middleboxes or firewalls.

2.7.2 Overview

We now provide an overview of the SENG architecture and explain how SENG shields network traffic of unmodified client applications and enables app-grained traffic control.

The SENG architecture consists of two main components: (i) a client-side shielding runtime, and (ii) a SENG server located at the gateway. Figure 2.2 provides an overview of the SENG components and communication channels. On the client side, the SENG runtime wraps a client application in a library OS (*LibOS*) and combines both in an SGX enclave. The dedicated SENG server is located at the central network gateway. It cooperates with the firewall and the SENG runtime instances to attribute and protect traffic of the shielded apps.

On the client, the LibOS and SENG runtime transparently shield the client applications from local system-level attackers. To this end, the LibOS loads and executes unmodified binary applications inside a hardware-protected SGX enclave. The LibOS transparently handles system calls of the app and shields them against Iago attacks [39] of the untrusted OS. For instance, the LibOS prepares its own file system to protect against disk I/O tampering. The SENG runtime adds to this in that it protects network I/O of shielded apps and establishes trust with the SENG server. Technically, the SENG runtime incorporates a lightweight user space TCP/IP stack to cope with the lack of trust in the host's network stack. This user space network stack manages the app's connections inside SGX and enables secure tunneling of whole IP packets—including the network and transport headers–to the SENG server.

The SENG server has to authenticate client apps and securely forward shielded traffic between SENG runtime and gateway. The SENG runtime and server establish an attested, secure communication channel to tunnel traffic. The SENG server listens

for incoming tunnel connections from shielded and trusted client apps. We use SGX's remote attestation to check the app's identity and verify that it runs inside a valid SGX enclave with SENG runtime. To this end, the SENG runtime generates a fresh public and private key pair and binds it to the enclave report—inspired by work of Knauth et al. [128]. The SENG runtime then uses the keys to establish a mutually authenticated, end-to-end protected connection to the SENG server and provides the attestation report during connection setup. Before accepting the connection, the SENG server checks that the attestation report is bound to the connection and belongs to a valid SGX enclave with a shielded application. After tunnel establishment, traffic of the shielded app can be securely tunneled to the SENG server and routed through the gateway (incl. firewall) while being protected from MITM attackers between enclave and gateway.

2.7.3 Application-Grained Firewall Policies

Placing the SENG server on the gateway allows for fine-grained traffic control at the perimeter firewall. With SENG, firewalls can precisely control *which shielded app may communicate where*. This adds a completely new degree of freedom that standard firewalls do not give, as they subsume all applications of a given system into a single address.

The SENG server maintains a central allowlist (database) of trusted applications, which links apps to their trusted attestation reports, and additionally, to an app-specific IP subnetwork. The SENG server assigns a unique IP address from this particular subnet to each shielded enclave instance of a given client app. Optionally, the host IP on which a shielded enclave is executing can additionally be taken into account when selecting the subnetwork. The enclave-unique addresses make the shielded app's identifier visible to all gateway services, including firewalls. Firewalls use this mapping to define app-specific policies, which are easily integrated into existing toolchains.

To demonstrate this, we introduce a typical corporate network setup, as shown in Figure 2.3. The network consists of a central, SENG-enabled gateway which interconnects an untrusted internal client subnetwork, a trusted internal server subnet, a DMZ, and external networks. The DMZ provides typical services for internal and external hosts, including a public web shop and a DNS server. The internal servers are only reachable by internal clients and host an intranet web server, as well as an LDAP and database server. The client workstations run a set of trusted client applications (e.g., browsers, mail clients) which require access to internal and external servers. The white columns in Table 2.2 show traditional firewall policies (e.g., configured using `iptables`) for this setup. Rules 1-2 allow workstations to connect to external hosts, rules 3 and 8-10 grant them connections to internal and DMZ servers, and rules 4-7 allow external clients to connect to servers in the DMZ. Rule 11 allows internal and DMZ servers to connect to external servers. Rule 12 allows all communication of such established connections, and rule 13 is the default policy that rejects any other traffic.

If client hosts are fully compromised by a system-level attacker (cf. Section 2.4), these traditional policies fall short. First, they allow malware on trusted hosts to communicate to external servers. Second, they do not refine which external clients

Table 2.2: Traditional firewall policies for the corporate sample network (Figure 2.3) and their app-grained SENG alternatives (gray column). The variables in column 2 and 4 represent subnets (e.g., `$WORKSTATIONS`) or server IP addresses (e.g., `$IMAP`). The new variables in the gray column represent SENG enclave subnets (`$WS` for workstations, `$ANY` for arbitrary IP addresses).

No.	Source (w/o SENG)	Source (with SENG)	Destination	Dst. Port	State	Action
1	$WORKSTATIONS	$WS_FIREFOX_72	$EXTERNAL	80, 443	NEW	ACCEPT
2	$WORKSTATIONS	$WS_PSQL_TLS_ONLY	$EXTERNAL	5432	NEW	ACCEPT
3	*	$ANY_THUNDERBIRD_68	$IMAP	143, 993	NEW	ACCEPT
			$SMTP	465, 587	NEW	ACCEPT
4	$EXTERNAL	$EXTERNAL	$SMTP	25	NEW	ACCEPT
5	*	*	$DNS	53	NEW	ACCEPT
6	*	$ANY_FILEZILLA	$FTPS	989, 990	NEW	ACCEPT
7	*	*	$WEBSHOP	80, 443	NEW	ACCEPT
8	$WORKSTATIONS	$WS_FIREFOX_72	$INTRANET	80, 443	NEW	ACCEPT
9	$WORKSTATIONS	$WS_PSQL	$DATABASE	5432	NEW	ACCEPT
10	$WORKSTATIONS	$WS_ENCLAVES	$LDAP	389, 636	NEW	ACCEPT
11	$SERVERS	$SERVERS	$EXTERNAL	*	NEW	ACCEPT
12	*	*	*	*	ESTABL.	ACCEPT
13	*	*	*	*	*	REJECT

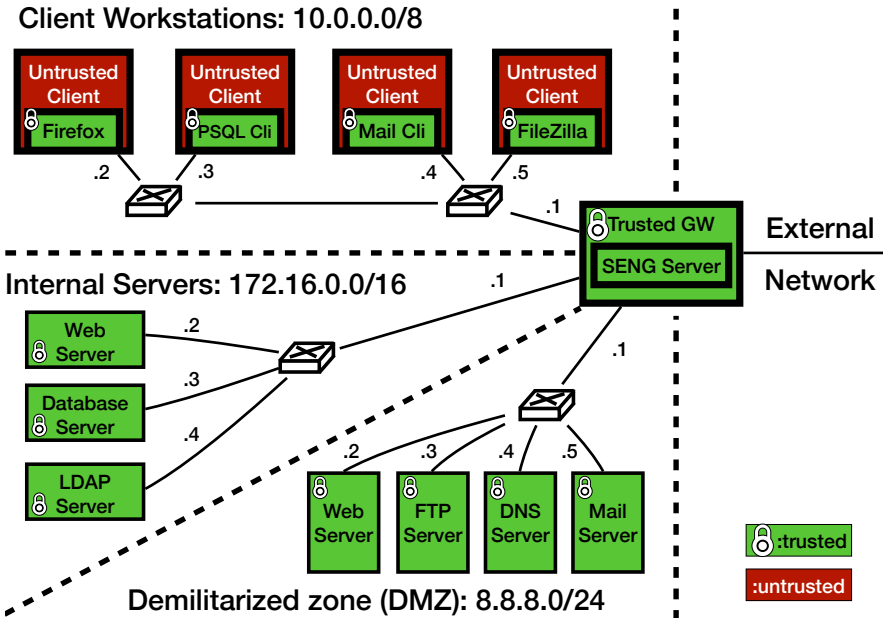

Figure 2.3: Sample topology of a corporate network consisting of a SENG-enabled gateway, a subnet of untrusted clients with shielded apps, an internal server subnet and a DMZ.

may use servers in the DMZ. To tackle these shortcomings, SENG grants only trusted apps network access. The gray column in Table 2.2 shows the policy modifications that SENG requires. Administrators just have to replace the coarse-grained source addresses with app-grained addresses. For example, in rule 1, the firewall can now control that only vetted Firefox clients from the workstation network can access external networks, and any untrusted software is blocked. This minor change significantly hardens the firewall setup. The SENG-enabled policies can be automatically derived when shielded apps specify which endpoints they need for communication.

Subsumed Enclave Subnetworks Optionally, network admins can group shielded apps sharing policies (e.g., all mail clients, or versions of same app) into privilege-based subnets. Table 2.2 exemplifies both cases: While rule 3 restricts access to an individual mail client version, rule 6 subsumes all FileZilla versions in a subnet. Rule 2 even restricts access to external databases only to PSQL clients configured with SSL mode enabled to protect against external MITM attackers.

Host IP Addresses We override the source IP address with an enclave-unique address to easily integrate SENG into existing firewalls (AR3). Note that the SENG server can still distinguish between enclaves running on different hosts and between enclaves running on different subnets. While rule 6 grants internal *and* external FileZilla enclaves access to the FTP server (DMZ), rule 8 restricts access to intranet web pages to shielded browsers on internal workstations only.

Avoiding Network Fragmentation While SENG's app-specific IP subnetworks enable easy integration into existing firewall and monitoring toolchains, there might be large-scale enterprises that already face high network fragmentation and have a high demand of IP addresses. In such a setting, with an increasing number of shielded apps and clients, SENG's enclave subnetworks might start reserving and fragmenting a noticeable portion of the available corporate IP space, potentially causing a shortage of available local IPv4 addresses. Therefore, in Section 2.12, we will describe an optional Netfilter/iptables-specific extension to the SENG server that trades in SENG's drop-in deployment (AR3) for less network fragmentation by enabling the use of a single enclave subnetwork if subsumed enclave subnetworks and source NATing are not enough.

2.7.4 Deployment of SENG

SENG raises questions regarding enclave deployment, key management and update handling, which we discuss next.

Enclave Deployment The SENG runtime and its dependencies are shipped to clients as a container image. Each shielded app needs a configuration file that lists the files the LibOS has to protect, which can be (partially) automated[4]. App bundles can then be offered, e.g., via corporate app stores.

New SENG client devices are enrolled by including their addresses as allowed host addresses in the policy database of the SENG server. Strong device bindings can optionally be established using orthogonal schemes such as IEEE 802.1X and strict mappings between hosts and IP addresses. Alternatively, one could bind a secret to the client CPU as part of the app installation process.

Mixed Environments / Gradual Deployment SENG can also be deployed in mixed environments, i.e., heterogeneous networks where not all hosts support SGX (and thus SENG). In this case, administrators can use network segmentation to separate SGX-enabled workstations from legacy workstations. Whereas the unprotected subnetwork of legacy hosts would be governed by traditional (and possibly more restrictive) firewall rules, the protected network could readily use SENG policies. In fact, given a particular workstation, this setup also allows to gradually migrate applications to SENG. Shielded apps would belong to the protected subnetwork, whereas all other legacy applications are bound to the unprotected subnetwork.

Key Management SENG requires minimal key management. The SENG server authenticates clients via remote attestation and the client key pair (K_{enc}, K_{enc}^{-1}) is generated on each startup, s.t. no key rollouts are required. The key pair of the SENG server (K_{srv}, K_{srv}^{-1}) must be securely managed and the public key K_{srv} is shipped to clients as part of the SENG runtime. See Section 2.9 for respective security considerations.

[4] e.g., using `https://github.com/oscarlab/graphene/tree/v1.0.1/Tools`, or an automated build chain for container generation [17]

Component Updates On each component update (incl. keys, app, libs, SENG and LibOS), the SENG runtime image is rebuilt, and a new attestation report is extracted and inserted into the allowlist. Thus, SENG can identify the exact software bundle of a given enclave (cf. Section 2.8.1) and allow, e.g., only specific app versions (Table 2.2, rule 1)—mitigating the risk of outdated software that exposes security vulnerabilities. While SENG provides new reports on each update, LibOSes commonly support dynamic loading [219, 24], s.t. SENG needs to reship *only the modified files*, the (small) configuration, and new enclave signature.

Critical Updates and Key Rollovers In case of critical security updates, the compromised reports must be removed from the allowlist to revoke network access. SENG can optionally terminate all established tunnels of such revoked apps, immediately disconnecting revoked apps from other network segments. A special case is the update of SENG's server key pair (K_{srv}, K_{srv}^{-1}) as part of a periodic or emergency key rollover. As the public key K_{srv} is pinned by each shielded app and part of their attestation, every app report changes and has to be revoked. However, note that when using a tunnel cipher with (perfect) forward secrecy, their session keys are unaffected by a server key breach (K_{srv}^{-1}). Thus, *all established tunnels and associated app connections can continue operation* (Table 2.2, rule 12).

2.8 Implementation

We now provide the details of the SENG architecture in chronological order of the shielded app's communication. That is, we first detail the setup phase, then how the app's network traffic is protected, and finally, how the perimeter firewall enforces app-grained communication policies.

2.8.1 Initialization and Tunnel Setup

Initialization Phase Before the SENG runtime can protect a client application, the SGX enclave must be set up. SENG uses the Graphene-SGX LibOS [219], as it supports dynamic loading of unmodified, multi-threaded Linux apps and shields system calls. First, Graphene-SGX sets up an SGX enclave and initializes the shielding layers. After finishing the setup, but before loading the application, the SENG runtime loader is called and launches a dedicated enclave thread for the user space TCP/IP stack and for the tunnel module. The TCP/IP stack is instantiated with the embedded lwIP stack [151], as it is lightweight and modular by design. The tunnel module manages the tunnel to the SENG server and registers itself as network driver for the default interface of lwIP, s.t. lwIP routes all IP packets of the client app through the tunnel module.

On the gateway-side, the SENG server creates a virtual IP-level network interface which it will later use for routing traffic of shielded apps and receiving packets destined for them. Afterwards, the SENG server sets up a welcome socket and waits for incoming tunnel connections by internal or external SENG runtime instances.

Tunnel Preparation After initialization, the SENG runtime generates credentials and the enclave report for the secure tunnel to the SENG server. The tunnel module uses DTLS (RFC 6347), which has well-documented end-to-end protection guarantees. We chose UDP-based DTLS over TLS as it requires less state and is faster, which improves scalability, and as the reliability and ordering guarantees of TLS are not required [80]. For tunneled TCP connections, the TCP/IP stacks of the communication endpoints—namely SENG runtime and target server—already guarantee reliable, in-order packet delivery. For tunneled UDP streams, both communication partners have to resolve packet reordering in the application protocol anyway, and the choice of DTLS thus does not weaken any security guarantees.

To couple remote attestation with the end-to-end protection of DTLS, the tunnel module generates a fresh RSA key pair (K_{enc}, K_{enc}^{-1}) and binds the public key K_{enc} as user data to the enclave report—following the idea of Knauth et al. [128]. The local Intel Quoting Enclave (QE) transforms the report into a verifiable, signed quote using the attestation key. After receiving the signed remote attestation report via an attestation service, the tunnel module uses the RSA keys (K_{enc}, K_{enc}^{-1}) to generate an X.509 client certificate and embeds the attestation report with corresponding signature as extra fields.

Note that the tunnel module must not be able to directly communicate with external Attestation Services, e.g., Intel Attestation Service (IAS), to request the signed remote attestation report. Local and on-path adversaries could exploit the unprotected headers of the IAS connection as covert channel and leak information (violating SR6). To solve this dilemma, we can (i) let the enclave send the signed quote to the SENG server, which in turn performs the IAS communication itself, or (ii) operate an internal attestation service in the DMZ, and let the enclave submit the quote to the AS via TLS [199].

Tunnel Establishment The SENG runtime now connects to the SENG server via a mutually authenticated DTLS connection. For server authentication, the runtime uses the pinned server public key K_{srv}. For client authentication and remote attestation, the SENG server checks the validity and signature of the attestation report and matches the embedded user data with the certificate key K_{enc}. The SENG server then verifies if the report data belongs to a shielded application in the allowlist. Technically, the enclave measurement contains the Graphene-SGX library and memory-mapped manifest (MF):

$$\text{mrenclave} \leftarrow \text{measure}_{\text{sgx}}(\text{graphene}, \text{MF})$$

The manifest contains secure hashes $h(\cdot)$ for all dependencies of the SENG runtime and shielded app, including the runtime library (sengrt), the pinned server key K_{srv}, the app's binary and libraries, as well as other protected files:

$$\text{MF} := \{h(\text{sengrt}), h(K_{srv}), h(\text{app}), h(\text{lib}_1), ...\}$$

The file system shield enforces file integrity based on the hashes [219]. The inclusion of the manifest in the measurement results in a unique enclave identity (mrenclave) for each bundle of LibOS, SENG, and client app. Therefore, the SENG server can directly link the report to the exact version of the shielded app. If the app was verified, the

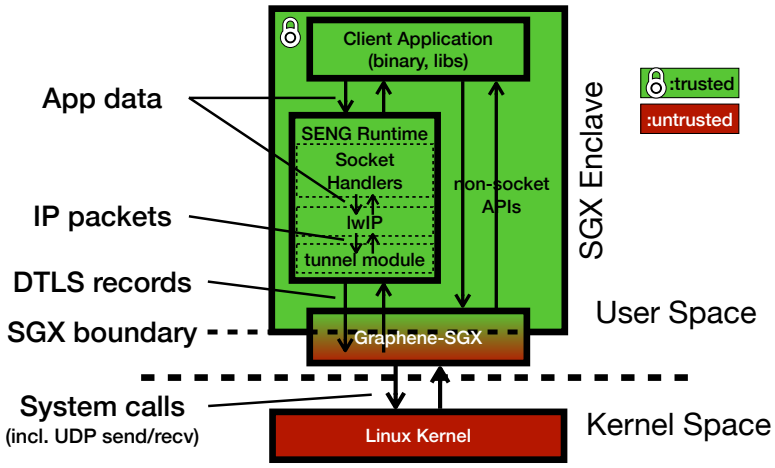

Figure 2.4: Overview of the SENG runtime components

SENG server knows that the DTLS tunnel is attested and established with a valid SGX enclave. Finally, the SENG server looks up the app-specific IP subnet in its internal database based on the app's identity (mrenclave) and, optionally, host IP, and assigns a unique IP address from the subnet to the SENG runtime instance (cf. Section 2.7.3). The SENG runtime takes over the reported IP configuration, and Graphene-SGX loads the app and transfers control to it.

2.8.2 Network Traffic Shielding

Redirecting IP Packets to the Tunnel SENG needs to protect the whole network traffic of shielded applications. Graphene-SGX links the client apps against a patched version of the standard C library where syscalls are replaced by calls to LibOS-internal handler functions. This allows us to fully-transparently wrap and shield system calls. The SENG runtime provides own handlers which shadow all network I/O functions, as shown in Figure 2.4. The SENG handlers transparently redirect all socket API functions of the client app to the respective lwIP functions, s.t. the app can perform network I/O only through the SGX-internal user space stack. lwIP manages all connections of the app and uses the tunnel module for receiving and sending the associated IP packets.

Sending Packets When the shielded app sends data, lwIP crafts corresponding IP packets and passes them to the tunnel module. The tunnel module wraps the IP packets with DTLS and forwards them through the attested tunnel to the SENG server. For transferring the DTLS records, the tunnel module uses the LibOS to perform the actual UDP send operation via the untrusted OS. Figure 2.4 shows the app's data flow and highlights that only the DTLS records cross the SGX boundary. The end-to-end security protection of DTLS prevents attacks by local or MITM attackers. The SENG server receives the DTLS records, decrypts contained IP packets and then passes them through the virtual network interface to the gateway network stack. The gateway then applies app-grained firewall rules (Section 2.8.4) and routes the packets to the target server.

Receiving Packets For inbound traffic, the SENG server receives the corresponding IP packets from the gateway through the virtual network interface. The SENG server uses the target address to look up the DTLS connection to the respective shielded client app and tunnels them back. The tunnel module receives and decrypts the IP packets and puts them into the lwIP inbox queue. lwIP then processes the packets and passes the contained app data to the shielded app.

2.8.3 DNS Resolution Shielding

Without further precautions, the enclave would fully rely on the host OS to resolve domains. Local system-level attackers could thus launch severe redirection attacks and redirect traffic of shielded apps to IP addresses of their choice. To tackle this problem, SENG shields DNS lookups of client applications via three complementary actions. First, the SENG runtime redirects the respective standard library functions (e.g., `getaddrinfo`) to lwIP and configures lwIP to use a trusted DNS resolver located at the gateway or in the DMZ. The trusted resolver can then securely query internal DNS servers or contact trusted external ones via integrity-protected DNS variants, e.g., DNSSEC, DNS over TLS (DoT) or DNS over HTTPS (DoH)[5]. Second, we provide trusted versions of configuration files used by third-party DNS libraries for looking up information like the name server IP ("resolv.conf") or protocol-specific port numbers ("/etc/services"). We leverage the file system shield of the LibOS to protect the integrity of the files. Third, all DNS queries sent via standard resolver functions or third-party libraries eventually pass lwIP and are therefore tunneled through the protected DTLS tunnel.

2.8.4 Application-Grained Policy Enforcement

SENG enables the perimeter firewall to apply app-grained network policies whenever shielded traffic is routed through the gateway. App traffic reaches the gateway only through SENG's virtual network interface and the SENG server forwards traffic to an app tunnel only if it matches the assigned enclave IP. Therefore, the gateway can identify outbound traffic as shielded iff received from SENG's network interface and routes inbound traffic destined for enclave IPs to the SENG server. In the process, the firewall on the gateway enforces *app-grained policies as network policies on the app-specific enclave IP subnets* (cf. Section 2.7.3). To prevent impersonation attacks via IP spoofing, the SENG server drops tunneled app traffic with mismatching enclave IP and the firewall drops enclave traffic not arriving through SENG's network interface.

2.8.5 Shielded Servers

So far, we took it for granted that all shielded apps are clients. However, SENG also supports shielded *server* apps. SENG server sockets work analogously to default server sockets. However, with SENG, the gateway can now fully control (i) if an enclave can expose server functionality, and if so, (ii) which clients are allowed to contact the

[5] RFC 4033, RFC 8484, and RFC 7858

enclave. Similar to client policies, server policies restrict communication to shielded clients or specific enclaves only (app-grained policies).

Once created, SENG server sockets are reachable through the gateway under the assigned enclave IPs. Recall that enclaves can either have public (globally routable) or private (RFC 1918) IP addresses. In case of public addresses, the enclave's server socket is immediately exposed. If the enclave's IP is private, yet should be reachable from external clients, the gateway uses destination NATing to expose the service.

2.9 Security Analysis

We now discuss how adversaries could attempt to attack SENG. Table 2.3 summarizes the attacks and respective defense mechanisms. We discuss why the protection from the above adversaries implies the fulfillment of the security goals of Section 2.7.1 and therefore solves the initial challenges.

Adversary Types With reference to Figure 2.1 (see page 16), SENG faces several types of adversaries: (i) a client-side system-level attacker ("*Sys*") who fully controls the enclave's OS interactions (including traffic), (ii) MITM attackers in the internal or external client subnetwork (depending on the client's location) who can fully control the traffic between the client and the SENG gateway ("M_{c2gw}"), (iii) MITM attackers on the path between the SENG gateway and the destination server (either internal or external) ("M_{gw2srv}"), (iv) an internal attacker inside the organization who aims to leak sensitive data ("*Internal*"), and finally, (v) an external attacker outside of the organization who aims to sneak data (or malware commands) into the network ("*External*"). We will use these attacker models to discuss how SENG protects against 16 security-critical attacks. In addition, we will discuss a relaxed threat model assuming a compromised enclave in A17. Furthermore, while SENG assumes a trusted gateway ("bastion host", see Section 2.4), in A18, we will discuss the impact of a system-level gateway compromise ("Sys_{gw}") on SENG and outline possible mitigation strategies.

A01: Code/Data Tampering (*SR1*) *Sys* may aim to hijack the shielded app code, tamper with the runtime data, or leak sensitive information like tunnel keys. The hardware-enforced protection of Intel SGX blocks all unauthenticated access to enclave memory and therefore prevents such attacks.

A02: File Tampering (*SR1*) Furthermore, the file system shield uses the manifest MF to check the integrity of the SENG runtime, pinned SENG server key K_{srv}, application binary and all its dependencies (e.g. libs, config files), such that any attempt to tamper with files is detected and blocked (cf. Section 2.8.1).

A03: LibOS Modification (*SR2-4*) Patching the LibOS binary or its manifest to replace loaded files, e.g., the client app, or the pinned SENG server key K_{srv}, is possible, but results in deviating enclave identities (mrenclave). During remote attestation, the SENG server will thus refuse the tunnel, as the unknown enclave is not in the allowlist.

Table 2.3: Assessment of attacks on SENG and its respective countermeasures, following the attacker models of Section 2.9.

Target / Goal	**Attack**	**Adversaries**	**SENG's Defense Mechanisms**	**Secure?**
Shielded App	A01: Code/Data Tampering	Sys	SGX Enclave	✓
	A02: File Tampering	Sys	File System Shield	✓
	A03: LibOS Modification	Sys	Attest + Allowlist	✓
SENG's Tunneling and Access Control	A04: Fake/Custom Enclave	Sys	Attest + Pinning + Allowlist	✓
	A05: Client Impersonation	Sys, M_{c2gw}	Key Binding + Traffic Auth.	✓
	A06: Server Impersonation	Sys, M_{c2gw}	Pinning + DTLS	✓
	A07: Attacking SENG's Keys	Sys, M_{c2gw}	SENG's Key Management	✓
	A08: Tunnel Tampering	Sys, M_{c2gw}	DTLS + Trusted TCP/IP Stack	✓
App Connections	A09: DNS Spoofing	Sys, M_{c2gw}, M_{gw2srv}	SENG's DNS Shield	✓
	A10: Internal Conn. Tampering	Sys, M_{c2gw}	DTLS Tunnel + DMZ	✓
	A11: External Conn. Tampering	M_{gw2srv}	(Enforce Apps w/ Sec. Comm.)	(✓)
Information Leaks and Remote Control	A12: Direct Info Leak	Internal	SENG's Shielding and Policies	✓
	A13: Direct Remote Control	External	SENG's Shielding and Policies	✓
	A14: Covert Channel (Header)	Internal + External	SENG's Tunneling + DTLS	✓
	A15: Covert Channel (Timing)	Internal + External	(Adopt Time Masking)	(✓)
App Interfaces	A16: Steering Shielded Apps	Sys	(Secure I/O + Caller IDs)	(✓)
SENG's Policies	A17: Privilege Escalation	Malicious Enclave	Traffic Auth. + Policies	✓
Central Gateway	A18: Gateway Compromise	Sys_{gw}	(TEE-protected Srv+FW+NIC)	(✓)

A04: Fake/Custom Enclave (*SR4*) An adversary could try to establish a tunnel to the SENG server directly, or from within a custom enclave. As the SENG server expects a valid, correctly-signed attestation report, it will refuse direct connections with attacker-crafted fake reports. When the adversary contacts the SENG server from within a custom enclave, the attestation report will be valid, but not in the allowlist. Therefore, the SENG server will refuse the connection by the unknown enclave as in the previous attack (A03).

A05: Client Impersonation (*SR4+SR5*) Attackers could try to impersonate a trusted client application. First, attackers could intercept an allowlisted attestation report and embed it into their own client certificates. However, the report will not be bound to the certificate and the SENG server will detect the mismatch and deny access. Second, attackers could spoof an IP from a trusted enclave subnetwork. However, the SENG-enabled gateway can identify the non-tunneled traffic as unauthenticated and drop the packets (see Section 2.8.4).

A06: Server Impersonation (*SR2*) The attacker can also try to impersonate the SENG server by intercepting connection attempts. If successful, the adversary could gain access to all connections of the shielded application, including unprotected legacy traffic. However, the SENG runtime pins the valid SENG server key K_{srv} and checks it during the DTLS handshake to detect such impersonation attacks.

A07: Attacking SENG Keys (*SR2*) SENG performs secure key management to prevent multiple attacks against the tunnel security: (i) Rollback attacks against SENG's server public key K_{srv} do not exist, as K_{srv} is not sealed to disk and is integrity protected (A02). A rollback of the whole app bundle (incl. K_{srv}, LibOS and all dependencies) results in a deprecated, blocked report (A03). (ii) If a private key of the SENG (or attestation) server is breached, SENG blocks all vulnerable reports and thus enclaves with stolen keys (cf. Section 2.7.4). As DTLS supports ciphers with perfect forward secrecy, established tunnels are not affected by a breach of the SENG server key K_{srv}^{-1}. (iii) The client RSA key pair (K_{enc}, K_{enc}^{-1}) is freshly generated for every new enclave instance and the *private key* K_{enc}^{-1} *never leaves* the enclave, s.t. it is protected against attackers (cf. A01).

A08: Tunnel Tampering (*SR2*) Tampering with established tunnel connections is not possible, because of the end-to-end security guarantees of DTLS. An adversary can reorder or drop tunnel packets, which is explicitly supported by DTLS. However, tunneled UDP connections do not expect reliable or in-order delivery and the endpoint network stacks still ensure reliability and ordering guarantees for TCP packets (Section 2.8.1).

A09: DNS Spoofing (*SR3*) An attacker can try to leak information by redirecting connections of shielded apps via DNS reply spoofing. SENG shields DNS traffic via multiple complementary methods as discussed in Section 2.8.3. First, spoofing the results of untrusted resolver functions is prevented by redirecting the function calls to lwIP. Second, DNS redirection to attacker-controlled nameservers via modification of

system configuration files is prevented by providing versions with trusted IP addresses and port mappings. The LibOS ensures the integrity of the files via the file system shield. Third, *Sys* and both types of MITM attackers (M_{c2gw}, M_{gw2srv}) can try to attack unprotected DNS traffic directly. Direct attacks are prevented by securely tunneling DNS traffic through the DTLS tunnel to trusted, internal resolvers which follow integrity-protected DNS protocols for name resolution (e.g. DNSSEC, DoH, DoT).

A10: Attacking Connections to Internal Servers (*SR2+SR3*) Attacking the communication between shielded apps and internal servers (incl. DMZ) is not possible. The traffic is protected from *Sys* and M_{c2gw} attackers by SENG's DTLS tunnels between the shielded apps and the gateway. As the internal servers are located in trusted networks, there are no M_{gw2srv} attackers between them and the trusted gateway.

A11: Attacking Connections to External Servers (*SR2+SR3*) SENG cannot protect the traffic between gateway and external servers. However, SENG enables network administrators to grant access to external networks only to shielded applications that securely establish end-to-end protected connections (e.g. Table 2.2, rule 2). If required, the file system shield can protect app-specific configuration files that define the security level of the shielded app. Therefore, SENG can indirectly enforce protection against M_{gw2srv} attackers.

A12: Direct Information Leakage (*SR6*) SENG enables the gateway to identify and block traffic coming from non-shielded senders, such as malware. Attackers cannot modify the behavior of shielded apps to leak information (A01–A03). They cannot get access to attested tunnel connections to authenticate malicious traffic for home-calling either (A04–A05, A07–08). Leaking non-encrypted traffic of shielded apps to the external network or to attacker-controlled external servers via DNS- or header-based redirection attacks are prevented as well (A09–A11). As a result, adversaries can neither connect to external servers, nor encode sensitive data in shielded traffic, nor redirect internal, shielded traffic to external networks.

A13: Direct Remote Control (*SR6*) SENG enforces access control also for incoming connections, which blocks direct connections from external adversaries to internal malware. Sneaking data into the internal network by attacking external shielded clients is prevented analogously to attacks against internal apps (see A12).

A14: Header-based Covert Channels (*SR6*) Any attempts to establish a covert channel via header manipulations is prevented by SENG. Information leakage by internal attackers via tunnel header manipulation is prevented, as the SENG server strips the headers at the gateway. Remote commands that external attackers may inject by manipulating communication headers is likewise prevented, as the gateway strips the link layer headers and the SENG server securely tunnels the IP packets to the shielded applications. Therefore, adversaries cannot observe information encoded in the internal headers.

A15: Timing-based Covert Channels (*SR6*) Attackers may aim to create side channels based on packet timings (e.g., encoding information by delaying packets). While we excluded such covert channels from our threat model, SENG could adopt techniques to mask timing channels [35, 234].

A16: Steering Shielded Programs for Info Leaks (*SR6*) Attackers could try to abuse shielded applications to exfiltrate data. Consider a shielded browser. Its interactive interface lets users navigate (e.g., enter URLs). While we trust the user, a system-level attacker could intercept keyboard input and inject malicious commands into the shielded app. This way, adversaries control network traffic even of shielded apps. Non-interactive interfaces allow for similar attacks. For example, if users click on links displayed in a shielded mail client, the mail client calls a non-interactive interface to steer a browser to open the link. Attackers can intercept or use the interface to control the browsing targets and query strings. The general underlying problem is that shielded applications have to verify if their inputs stem from shielded applications.

To mitigate these attacks, we can rely on trusted I/O for interactive applications in addition to the shielded interfaces we specified in our threat model (cf. Section 2.4). We regard the adoption of secure I/O in the form of upcoming HW extensions [131] or dongles [68, 117] as realistic for critical business environments which already deploy HW authentication dongles. The LibOS can leverage trusted I/O to use attested, secure I/O paths between enclave and I/O devices [68, 117]. The LibOS can then verify that user input comes from a trusted device before forwarding input to the shielded app. Shielded interfaces based on local attestation, like SGX-based RPC calls [204], allow shielded apps to securely interact and thereby protect non-interactive interfaces (e.g., trustworthy path from mail client to browser). Problems still persist, however, if the caller has different (lower) app-grained privileges than the callee. To avoid the resulting confused-deputy attacks, the callee would have to forward the identifier of the caller to the SENG server—a significant research endeavor we leave open to future work.

A17: Privilege Escalation by Backdoored or Compromised Enclaves (*SR6*) We now discuss a relaxed threat model, where attackers can gain control over shielded apps, e.g., via backdoors or runtime compromises. Once compromised, attackers can send malicious traffic through the app's attested tunnel as long as the traffic matches the app's policies. If the policies are restrictive and allow communication to few vetted destinations only (e.g., shielded mail clients may only contact the local mail server), the resulting harm is limited. Any attempt of the compromised enclave to spoof its IP addresses, e.g., to join a more privileged subnetwork, will fail, because the SENG server detects unauthenticated traffic (A05) and restricts tunneled traffic to the assigned enclave IP (cf. Section 2.8.4). Perspectively, the app-grained traffic separation enables app-specific classification models for network intrusion detection systems, which further ease the detection of anomalous behavior of shielded apps upon compromise.

A18: SENG Bypass via Gateway Compromise (*SR2-3, SR4-6*) Our threat model fully trusts the central gateway, following the widely popular "bastion host" setting of network firewalls. If system-level attackers gain full control over the SENG server,

firewall, and network card (NIC), they obtain full access to the network traffic (breaking SR2+SR3) and can bypass the firewall (breaking SR4-6). While one could move the SENG server and firewall into user-level TEEs (e.g., SGX enclaves) to protect the decrypted enclave traffic and firewall integrity, this approach can only protect enclave-to-enclave communication (breaking SR2+SR3). Yet as system-level attackers control the hardware, they can still bypass the firewall and tamper with the communication.

To tackle this extended threat model, the gateway could rely on a system-level TEE, which is isolated from the compromised OS and can additionally claim exclusive ownership of the network card. We regard Arm TrustZone-assisted TEE systems, e.g., OP-TEE [145], a reasonable choice for designing a secure gateway with SENG support. TrustZone extends CPUs, memory, and devices with the notion of a normal and secure mode (resp. "world") and allows for hardware-enforced access control based on the current CPU mode [178]. In Chapter 3, we design a new trusted gateway architecture that builds on TrustZone in order to isolate the full network I/O path and the firewall service from system-level attackers and compromised user space services. That way, our trusted gateway can guarantee secure traffic control and firewall enforcement on all network traffic. As a future extension, the SENG server could be integrated as an additional isolated application into our trusted gateway in order to protect the SENG server against gateway attackers while extending the gateway firewall with SENG's per-application policies.

2.10 Prototype Implementation

We have implemented a prototype for the SENG runtime and SENG server, as well as an alternative, library OS-independent runtime SDK based on Intel's SGX SDK [107].

SENG Client Runtime (with LibOS) Our client-side component is written in C/C++ and consists of Graphene-SGX[6] [219] and our SENG runtime library. As enclave exits cause huge performance overhead [173], we use experimental support for exit-less syscalls in Graphene-SGX [69]. The runtime is implemented in about 2400 lines of code (SLOC)[7] and uses lwIP 2.1.2 [151], OpenSSL 1.0.2g, and an adapted version of the sgx-ra-tls attester code[8] [128]. We only included the IPv4 modules of lwIP to minimize the code base, and patched the definitions in the header file to be compatible with POSIX/Linux. We chose OpenSSL as it is well-known and fast. If a smaller code base is preferred over performance, we can easily replace it with lightweight alternatives like mbedTLS. For the tunnel, we use DTLS 1.2 with the `ECDHE_RSA_WITH_AES_256_GCM_SHA384` cipher suite.

The SENG runtime is integrated as a middle layer between Graphene-SGX and the shielded app via the preloading functionality of the internal linker. The runtime exposes a secure socket API to the app which shadows the unprotected API of Graphene and forwards calls to lwIP. We configured Graphene-SGX and lwIP to use two distinct file

[6] commit: 58cb88d2c187358aad428b100d1ff444173e1a2b
[7] measured using cloc [52]
[8] commit: 10de7cc9ff8ffaebc103617d62e47e699f2fb5ff

descriptor ranges, s.t. we can distinguish between calls of the app and those of the tunnel module.

In our current version, the tunnel module directly communicates with the IAS and embeds the attestation report inside the X.509 client certificate. However, note that the attestation variants described in Section 2.8.1 could be easily integrated. While the tunnel module thread handles DTLS packet receipt, the lwIP thread handles the decrypted IP packets. For increased parallelization and syscall reduction, we currently use one DTLS socket per direction and replaced lwIP-internal locks with spinlocks.

SENG Client Runtime Without LibOS (SENG-SDK) Our standard client runtime uses a LibOS, which adds to the client app's complexity and overhead to ease SENG integration. In certain settings, it may be desired to deploy SENG for client apps that cannot sacrifice performance or memory overhead. We thus designed an alternative client-side runtime SDK that adds support for apps based on Intel's SGX SDK [107]. This so-called SENG-SDK does not include a library OS, which makes it more lightweight and enables flexible integration into other frameworks [204]. Furthermore, by dropping the LibOS, the SDK trades legacy support (AR1) in for higher performance (cf. Section 2.11.5) and support for native SGX apps with trusted-untrusted split design.

The SENG-SDK is fully compatible with the SENG server and all SGX SDK-based toolchains. While SENG-SDK cannot remove the effort of porting apps to SGX, the toolchain integration makes porting enclaves to the SDK straightforward. Furthermore, the SDK provides a single init function which handles the whole setup (network stack, tunnels, threads) and afterwards exposes a secure POSIX-style socket and DNS API for trusted enclave code. SENG-SDK is written in about 2300 lines of C/C++ code and uses lwIP, adapted sgx-ra-tls attester code, SGX SSL[9] v2.2, and the SGX SDK v2.7.1. We added timeout support to condition variables of SGX SDK for lwIP, included the SSL stack into SGX SSL, and added O/ECALLs for the DTLS tunnel management. We use switchless OCALLs to accelerate the tunnel socket I/O.

SENG Server Our server prototype is an event-based, single-threaded DTLS server written in C/C++ based on libuv 1.9.1 [141], OpenSSL 1.0.2g, and the challenger code of sgx-ra-tls. The core functionality consists of ≈1300 lines of code (SLOC), and support for SENG server sockets adds ≈1500 lines. The server uses a TUN device as IP-level virtual network interface to the gateway. The SENG server configures the TUN device as the default gateway for connected SENG runtime clients and links each DTLS tunnel to the client's enclave IP address. After the release of SENG, we implemented a multi-threaded rewrite of the SENG server in Go (not part of the evaluation), and published it as an additional open-source project for the community (cf. Section 2.14).

2.11 Evaluation

We now evaluate our prototype implementation regarding efficacy and overhead. We use iPerf3 [111] to measure the network throughput, and then show how the results

[9] Intel's SGX port of OpenSSL

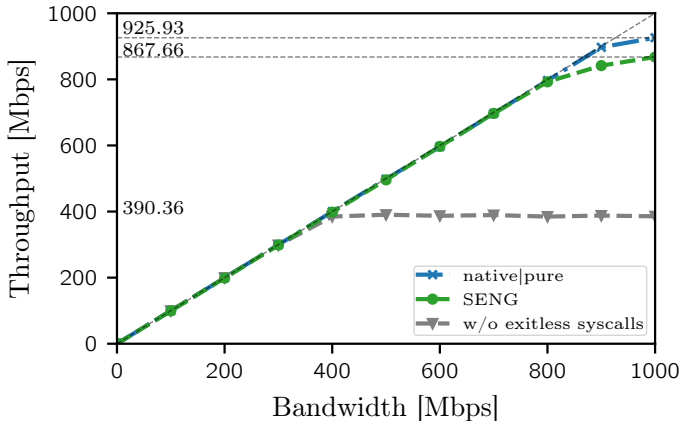

Figure 2.5: iPerf3 throughput of a single TCP connection measured for the native and pure setups (both blue), SENG (green), and SENG without exitless syscalls (gray).

transfer to real-world client (cURL, Telnet) and server (NGINX) applications. We then provide microbenchmarks to measure the setup phase of the SENG runtime. Afterwards, we revisit SENG's NGINX performance and significantly improve it by porting NGINX to the SENG-SDK. We conclude with a discussion on the SENG server scalability under an increasing number of enclaves and according tunnels.

In our experiments, the SENG server runs on a workstation with an Intel® Core™i5-4690 CPU with 4 cores, 32 GB of memory and Debian 9 with a 4.9 Linux kernel. The SGX-enabled client system has an Intel® Core™i7-6700 CPU with 8 cores, 64 GB of memory and runs the SGX enclaves inside a Ubuntu 16.04.4 LTS docker container with a 4.15 Linux kernel. The underlying host runs Ubuntu 18.04.2 LTS. Both systems are connected to the local network via 1 Gbps NICs (Intel I217-LM/I219-LM). We route the client's traffic via the SENG server to ensure that traffic from and to our SGX client system passes our virtual network gateway.

We take the native execution of the applications ("native") as baseline for our evaluation and compare it with the performance of Graphene-SGX ("pure") and of SENG ("SENG"). This way, we can attribute the overhead to either Graphene-SGX or the additional latency and overhead introduced by the SENG runtime and SENG server components.

2.11.1 Network Performance

We first report on the maximum downlink throughput of a single TCP connection using iPerf3. iPerf3 sends TCP packets to another iPerf3 instance and measures the resulting throughput. We generate the traffic on the gateway and receive traffic inside the enclave on the client system. We keep the default configuration of iPerf3, which calculates the average over 10 s, and we step-wise increase the bandwidth of the workload.

Figure 2.5 shows the average receive throughput over five iterations. The throughputs of all three approaches scale linearly with increased iPerf3 bandwidths, and SENG shows no overhead for bandwidths up to ≈800 Mbps. The native and pure Graphene-SGX setups both reach a maximum throughput of 925.93 Mbps, whereas SENG's peak

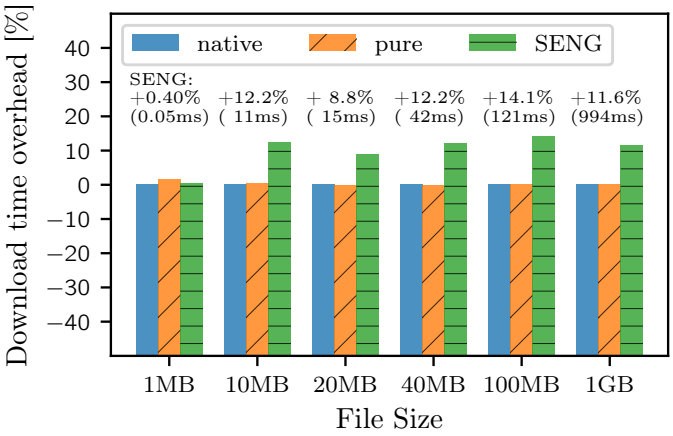

Figure 2.6: Time differences from cURL benchmark for native, pure, and SENG.

average throughput is 867.66 Mbps (≈6 % lower). Our 10 s measurements include TCP's slow start, and we observed higher temporal throughputs of ≈933 Mbps for native and pure, as well as ≈899 Mbps for SENG, reducing the peak loss to 3 % to 4 %. The slightly lower peak throughput of SENG is caused by the additional latency added by the SENG-internal TCP/IP stack and the DTLS tunnel. We included the results of SENG with enclave exits on every syscall (≈390 Mbps) to highlight that exitless designs are a key-enabler for I/O-intense enclaves [17, 173].

We conclude that the reduced throughput peak (3 % to 7 %) is acceptable, especially as clients and/or remote parties are typically bound to lower bandwidths, which showed no overhead.

2.11.2 Client Applications

cURL cURL is a popular tool/library to transfer data via several common protocols. In our setting, an external partner could use cURL to exchange files with internal servers. We have chosen cURL to check if SENG readily supports and scales to real-world client apps. To this end, we set up an Apache web server and measured how long cURL takes to download files via HTTP. Apache runs on the local gateway to capture the overhead with minimal impact from network jitter, analogous to iPerf3. We used the built-in measurements of cURL and took the 30 % trimmed mean over 50 iterations for each file size as a robust estimator [17].

Figure 2.6 shows the observed download time overhead relative to native execution. Graphene-SGX is again on par with the baseline as it shares the untrusted kernel network stack. For a file size of 1 MB, SENG shows minimal overhead due to the short download time. As the file size increases, SENG faces overhead of 8.8 % to 14.1 % which is higher than the one reported for iPerf3, but still reasonable. We observed TCP segmentation for every cURL payload, which was not present during iPerf3 and adds reassembly load and delay on lwIP as it cannot use HW offloading and has a lightweight design.

We conclude that SENG also shows reasonable performance for real-world client apps. Note that exitless syscalls in Graphene-SGX are still experimental and future

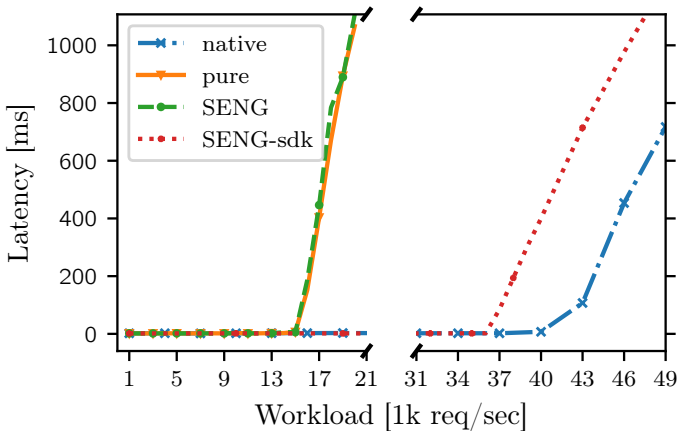

Figure 2.7: Average request latencies of NGINX measured for native (blue), pure (orange), SENG (green), and SENG with SENG-SDK instead of LibOS (red)

versions might stabilize and further reduce the network overhead.

Telnet Telnet (RFC 854) is widely used for remote terminal access and serves as our representative for remote login tools. SENG's built-in DTLS tunnel protects plaintext Telnet against local system-level *and* on-path attackers within the organization network. Furthermore, SENG can restrict remote access to trusted, TLS-based login clients and shield them from local user- or system-level attackers (e.g., hooks).

We used a Telnet server on a local workstation and measured over 10 iterations the average time it takes for a Telnet client to log in, execute a set of Bash commands for entering a directory, list the contained files, and finally, display the content of a 1 kB document. Telnet takes 269.38 ms during native execution and faces 0.17 % overhead for Graphene-SGX and 0.09 % for SENG, which is practically negligible.

2.11.3 Server Application (NGINX)

We next evaluate a server setting where we aim to shield an internal server from internal MITM and system-level attackers. We chose NGINX as a demonstrator which is a widespread event-based HTTP server. NGINX runs on the client host inside SGX and uses a single, poll-based worker thread to serve the 612 Byte demo page via HTTP. We used the wrk2 benchmark tool from an internal workstation to issue HTTP requests under step-wise increasing request frequency. For each workload, wrk2 spawned two threads with 100 connections and calculated the mean reply latency over ten seconds.

Figure 2.7 shows the average latencies over five iterations. Graphene-SGX and SENG can handle ≈15 k requests per second with a per-reply latency of 1.5 ms to 2.5 ms before performance degrades. Native execution clearly outperforms "pure" and SENG with ≈40 k. This is no surprise and follows the observations of Tsai et. al [219], because Graphene-SGX currently only supports synchronous syscalls, which cannot effectively overlap computation and I/O. We inspected the CPU utilization of NGINX under different loads and revealed that in the "pure" and "SENG" setting, the NGINX thread saturates the CPU via continuous polling and Graphene's I/O overhead.

Table 2.4: Client setup times of SENG and Graphene-SGX (LibOS)

	Microbench	Time [ms]	StdDev [ms]
SENG runtime	spawn lwIP thread	38.13	± 0.53
	OpenSSL init	**710.98**	±10.16
	RSA key gen (2048)	84.55	±66.25
	get SGX quote	35.67	± 2.20
	get IAS report	**639.05**	±16.46
	gen X.509 Cli-Cert	1.59	± 0.13
	DTLS tunnel setup	19.86	± 1.22
	spawn tunnel thread	42.64	± 1.20
	total SENG runtime	1578.03	±68.12
	without SSL init	867.05	-
	without SSL init + IAS	228.00	-
LibOS	(a) LibOS init (default)	868.00	±12.64
	(b) LibOS init (reduced)	728.27	± 8.06
	(c) LibOS init (minimal)	274.27	± 1.67

In conclusion, SENG cannot yet compete with native NGINX, but is on par with Graphene-SGX while providing more security guarantees and features on top of it. Furthermore, the bottleneck can be attributed to Graphene-SGX rather than to SENG and we therefore expect better performance under future asynchronous or batched I/O support. In Section 2.11.5, we will revisit this claim and show that we can significantly improve the performance of NGINX by porting it to the SENG-SDK (cf. "SENG-sdk" in Figure 2.7).

2.11.4 Setup Microbenchmark

We now measure the initialization overhead that the SENG runtime adds to Graphene-SGX, excluding the prototype-specific socket API handlers. As the setup time of Graphene-SGX depends on the enclave configuration, we measured the time for three configurations: (a) default values of LibOS-internal tests, (b) with reduced stack, heap and thread number, and (c) with minimal accepted size.[10] For SENG, we measured the different setup phases of the runtime.

Table 2.4 breaks down the average setup times over ten iterations. The total startup overhead of the SENG runtime is 1578.03 ms, i.e. it adds about 182 % overhead on top of the Graphene-SGX initialization under default configuration. However, the vast majority of this overhead stems from two steps: (i) the init routine of the OpenSSL library (710.98 ms) and (ii) the IAS communication (639.05 ms). The high OpenSSL startup time is partially attributable to the default seeding of the random number generator. It could be reduced by switching to the `RDRAND` engine to approach a setup time of 867.05 ms, which is comparable to the default LibOS time (a). As discussed in Section 2.8.1, the remote attestation could be handled by an internal AS server with

[10] default: 256MB size, 32MB heap, 4MB stack, 4 threads; reduced: 4MB heap, 256KB stack, 2 threads; min.: 128MB size + reduced; all: 2 rpc threads

caching support instead. Thus, the startup time added by the SENG runtime could be further reduced to ideally 228 ms, i.e. *about* 26 % *of the default* LibOS time (a).

We conclude that SENG adds a reasonable startup overhead which could be optimized to become comparable to the initialization time of Graphene-SGX under reduced (b) or minimal (c) enclave configuration.

2.11.5 Accelerating NGINX using SENG-SDK

We next revisit the NGINX results of Section 2.11.3 and show that SENG performs significantly better when replacing Graphene-SGX with a faster primitive. SENG performed on par with "pure" Graphene-SGX for NGINX with ≈15 k requests per second, but got clearly outperformed by the native baseline of ≈40 k (cf. Figure 2.7). To show that SENG can overcome the bottleneck caused by Graphene-SGX, we dropped the LibOS and instead ported NGINX[11] to our SENG-SDK. We ported only NGINX's platform-specific code to preserve comparability with previous results and added about 1100 lines of code (SLOC) for enclave setup and missing syscalls.

Figure 2.7 shows that SENG-SDK ("SENG-sdk") reaches ≈36 k request per second with a per-reply latency of 1.5 ms to 2.0 ms. SENG-SDK significantly outperforms the Graphene-based SENG runtime by factor 2.4 and reaches up to 90 % of native performance. Compared to Graphene-SGX, SENG-SDK provides more efficient OCALL interfaces tailored for the DTLS tunnel I/O and benefits from the more lightweight abstractions of Intel's SGX SDK. However, note that SENG-SDK loses legacy support and drop-in deployment (AR1).

We conclude that SENG can significantly benefit from performance improvements of the underlying primitives, letting it handle complex apps like NGINX with small overhead. Our rudimentary port to SDK-SENG achieved 90 % of native performance and could be further improved by adding NGINX-specific optimizations and an efficient file system shield. We are confident that the SENG runtime will likewise benefit from upcoming improvements of Graphene-SGX.

2.11.6 Server Scalability and Memory Overhead

We now discuss how the SENG server scales w.r.t. the number of clients and connections. The server has a small static memory footprint of which the TUN interface accounts for at most 750 kB under a full transmit queue[12]. The dynamic memory overhead is largely determined by the send and receive buffers of the per-enclave DTLS tunnels. In common settings, these would consume 8 KiB to 256 KiB per enclave and direction, plus about 32 KiB for the SSL frame buffer, but can be tuned to lower values. When considering the upper range, this still means that we could handle about 2000 clients per 1 GiB memory, with a huge potential for swapping large parts of the typically unused buffers. For SOCKS servers, the memory overhead increases with the number of connections they have to perform on behalf of the clients. In contrast, the SENG server is oblivious to the tunneled client connections and therefore faces constant per-client overhead.

[11] in single-process mode

[12] default length stores maximum 500 packets

The limiting performance bottleneck of the SENG server is the computational overhead of de- and encryption of DTLS packets and the general network I/O. In our experiments, the server easily coped with any client bandwidth, and given its 1 Gbps network card we cannot test higher loads. The CPU utilization (around 65 % on a single core, including waiting time) at maximum bandwidths suggests that the non-optimized server implementation will scale to 6+ Gbps on our hardware. This performance could be further optimized by improving the server code (e.g., using vectored sending, replacing the tunnel device with DPDK kernel NICs, etc.).

2.12 SENG Netfilter and iptables Extension

As discussed in Section 2.7.3 and Section 2.8.4, SENG uses app-grained IP subnetworks in order to enable existing gateway services, especially firewalls, seamless policy enforcement on enclave traffic. While this design requires no service changes and enables drop-in deployment (AR3), it might cause scalability issues in large-scale settings with increasing numbers of shielded apps and clients. While source NATing and grouping of apps with the same network privileges can decrease the IP fragmentation caused by app subnetworks (Section 2.7.3), large-scale enterprises with a high variety of apps and a high demand of IP addresses might still risk running out of available IPv4 addresses.

Therefore, in the following, we present *SENG-Netfilter*, a server-side extension to SENG, which intergrates per-application policy enforcement into the Netfilter/Xtables firewall and iptables tool. The integration enables the SENG server to expose shielded application identifiers *directly to the firewall* rather than indirectly via app-specific enclave subnetworks. Consequently, all enclave subnetworks can be subsumed into a single one to prevent shortage of IPv4 addresses and reduce network complexity. SENG-Netfilter enables administrators of large-scale enterprise setups to easily define per–application policies using new SENG iptables rule specifiers to express policies based on the shielded app's metadata. In contrast to vanilla SENG, the SENG Netfilter extension is firewall-specific and therefore trades in the drop-in deployment of SENG w.r.t. other gateway services (AR3) for more network scalability and usability.

2.12.1 Design of SENG-Netfilter

In SENG-Netfilter, the SENG server stays responsible for attesting shielded applications and assigning them IPs, however, in addition, it shares the enclave IPs and attestation metadata with the Netfilter/iptables firewall. The SENG Netfilter extension introduces three new components: (i) a SENG Netfilter kernel module, (ii) a netlink-based user space library, and (iii) an iptables extension library. The SENG Netfilter module (i) extends the Netfilter/Xtables firewall with new SENG application rule specifiers (e.g., app measurement) and handles the matching of network traffic against them. The SENG server uses the new user space library (ii) to expose the mapping between enclave IPs and their associated metadata (incl. measurement, host IP, and user-defined app categories) to the Netfilter module, such that it can perform the traffic matching. The iptables extension library (iii) registers against iptables to enable admins to easily use the SENG rule specifiers as part of their firewall policies. We have implemented

```
# 1. Thunderbird 115 enclaves may send to tcp/25 (smtp)
iptables -A INPUT -i tun --source 192.168.28.0/24 -p tcp \
  --destination-port 25 -m seng --src-app <thbird115> -j ACCEPT

# 2. browser enclaves may connect to tcp/443 (https)
iptables -A INPUT -i tun --source 192.168.28.0/24 -p tcp \
  --destination-port 443 -m seng --src-cat Browser \
  -m conntrack --ctstate NEW -j ACCEPT

# 3. allow communication from the gateway to NGINX enclaves
iptables -A OUTPUT -o tun --destination 192.168.28.0/24 -p tcp \
  --destination-port 443 -m seng --dst-app <nginx> -j ACCEPT

# 4. block enclave traffic from non-internal host IPs
iptables -A INPUT -i tun --source 192.168.28.0/24 \
  -m seng ! --src-host 10.0.0.0/8 -j DROP

# 5. allow established TCP connections from enclaves
iptables -A INPUT -i tun --source 192.168.28.0/24 -p tcp \
  -m conntrack --ctstate ESTABLISHED -j ACCEPT

# 6. allow established TCP connections to enclaves
iptables -A OUTPUT -o tun --destination 192.168.28.0/24 -p tcp \
  -m conntrack --ctstate ESTABLISHED -j ACCEPT
```

Figure 2.8: iptables rules using the SENG-Netfilter specifiers. `192.168.28.0/24` is used a the sole local enclave subnetwork. App filters use the SGX measurements.

and released an open-source prototype of SENG-Netfilter (cf. Section 2.14) [S1] in order to demonstrate the benefits of our tailored Netfilter and iptables integration.

2.12.1.1 SENG iptables Rules

The SENG iptables extension adds a set of new rule specifiers via the "seng" module (`-m seng`) that allows for filtering traffic based on the application metadata. Admins must define these app-grained firewall rules on SENG's virtual IP-level (tunnel) network interface (cf. Section 2.8.1). That way, the default iptables source and destination IP specifiers of these rules are interpreted as the enclave IPs rather than the host IPs on which they are executing. The new SENG rule specifiers enable admins to define rules that additionally match against the source or destination application measurements (`-src/dst-app`) and the host IPs of the applications (`-src/dst-host`). Optionally, admins can define application categories in the SENG server's database—which specifies the app allowlist and subnetwork/s—to group shielded apps (e.g., "trusted browsers") and enforce per-category firewall policies (using `-src/dst-cat`). Figure 2.8 shows examples of iptables rules using the new SENG rule specifiers.

2.12.1.2 Rule Enforcement and Metadata Sharing

The SENG Netfilter kernel module performs the traffic matching against SENG's iptables rules. The module maintains an internal hash table for mapping active enclave IPs to the shielded app's metadata exposed by the SENG server, including the app measurement (mrenclave), app category (if used), and the host IP on which the enclave is running. When SENG iptables rules are active, the module matches an IP packet against them by using the enclave IP (of the packet) to look up the corresponding application metadata in the hash table and compare the metadata against the rules.

For maintaining the hash table, the module defines a dedicated generic netlink channel on which it listens for update messages by the SENG server. The SENG server uses the new user space library to inform the module via the netlink channel whenever a new enclave IP has been assigned or an existing enclave (tunnel) has shut down. When a new shielded app has connected, the SENG server sends the assigned enclave IP together with the associated application metadata (from its database) to the SENG module for the rule enforcement. The module can then add a new mapping to its internal hash table with the new enclave IP and metadata. When an enclave (tunnel) shuts down, the SENG server sends the enclave IP to the module for deletion, and additionally deletes all connection tracking (conntrack) entries associated with connections from or to this enclave IP. That way, the SENG server prevents an exploitation of stale connection entries for bypassing firewall rules if the released IP gets re-assigned to an enclave with less-privileged network access.

2.13 Discussion

We conclude with a discussion on upcoming improvements and directions to overcome limitations of our SENG prototype.

2.13.1 Overcoming Memory Limitations of Enclaves

Client-side TEEs like SGX face two common challenges in practice: (i) performance impact of context switches and (ii) limited secure memory. In Section 2.11.1 and Section 2.11.5, we have already presented that careful switchless designs and improvements in existing LibOS primitives (incl. more recent ones like Occlum [202]) can significantly increase SENG's performance for complex apps like NGINX. In the following, we focus on the memory bottleneck (ii). SENG's prototype builds on the first version of SGX, which was widely available on client-side devices when SENG has been released. However, SGXv1 was limited to only 128 MB of total EPC memory (of which around 90 MB were usable by apps) and did neither support memory sharing across enclaves nor dynamic memory management. Thus, running many SGXv1 enclaves in parallel stressed memory and triggered expensive paging. After the release of SENG, Intel has further improved SGX and has published SGX version 2 for server and data center CPUs with support for dynamic memory management, e.g., lazy loading and page unloading, and with much larger EPC sizes of ≈64 GB to 512 GB. For the recent SGXv2, the memory bottleneck has thus been resolved except if a tremendously high number of services would be co-hosted in SGXv2 enclaves for thousands of cloud tenants.

Nevertheless, as future client-side TEEs might still face similar secure memory limitations as SGXv1, and to foster better enclave scalability for data centers, we propose multiple ways to decrease the memory utilization in SENG: (a) The majority of recent TEEs, including SGXv2, dynamic TrustZone, and VM-level TEEs, support dynamic memory management, i.e., support lazy loading and page unloading, which can be used to decrease the total memory pressure. In fact, recent studies on library debloating [185, 184] have shown that apps only use small fractions of the loaded code (incl. libraries), and tools like RAZOR [184] trim over 70 % of bloated binaries. With widespread dynamic paging support, SENG can integrate compiler- and loader-based schemes into the LibOS to reduce the enclave footprint. (b) SENG could follow the idea of Panoply by splitting the SENG runtime library and other shared libraries into separate enclaves that are shared by all shielded apps and used for attested RPC calls [204]. (c) More recent LibOSes like Occlum [202] apply HW-isolation mechanisms together with SW-based fault isolation to efficiently and securely run multiple processes in a single enclave. By integration of SENG inside Occlum rather than Graphene-SGX, multiple shielded apps with same privileges could directly share common libraries inside an enclave. We conclude that there are several mid-term and long-term directions for increasing the number of concurrent SENG-protected apps by decreasing the amount of secure memory required per enclave.

2.13.2 Frequent Measurement Updates

Any change to an app will cause a change to the enclave report and identity, too, thus resulting in a need to frequently update SENG's database of allowed app measurements. While alternative designs limit the number of updates by including only a loader inside the measurement [24], we highlight that our choice roots the app identity directly in the HW. We thus can directly specify app-grained policies on the exact app identity and do not need additional, potentially vulnerable, SW-based authentication schemes. As discussed in Section 2.7.4, we also regard integration of measurement updates into today's continuous build chains as practical and have shown in Section 2.7.3 that SENG is flexible enough to group multiple app versions into shared enclave IP subnetworks. A future direction might include exploration of shared "library enclaves" ("micron" in Panoply [204]) to compartmentalize enclaves while keeping HW-based identification.

2.13.3 Other TEEs and Platforms

While our current design uses SGX, it relies on common properties of other TEEs, namely trusted execution and remote attestation. Therefore, we can likely transfer SENG to other TEEs [31, 124, 241]. When designing SENG, we chose Intel SGX, as it has been widely available on commodity client systems, and posed additional challenges due to its restriction to user space code. After the release of SENG, Intel SGX support has unfortunately been deprecated for client devices. However, Intel is still continuing SGX support for server and data center CPUs, i.e., SENG can still be readily used to attribute network traffic to server or cloud applications. Furthermore, conceptually, SENG should be portable to the recent VM-level TEEs, e.g., AMD SEV [3] or Intel TDX [106], as they provide similar protection and attestation features. In particular,

the vSGX project [241], which virtualizes SGX enclaves using AMD SEV VMs, might be a promising system for adopting SENG on non-Intel platforms.

2.13.4 Prototype Limitations

Our current SENG runtime prototype does not support all system calls yet. We miss `fork` and `exec` in particular, for which support could be added following concepts of existing LibOSes [219, 204]. Furthermore, the SENG server currently supports only a simple SQLite3 database for defining the shielded application (enclave) allowlist, subnetworks, and host addresses rather than featuring a full database integration.

2.14 Artifacts

The prototypes of SENG are available as open-source projects at `https://github.com/sengsgx/` [S1], including the SENG runtime, SDK, and server, as well as the SENG Netfilter extension and a multi-threaded rewrite of the SENG server in Go. See page 9 for a list of all open-source prototypes covered by this dissertation.

2.15 Conclusion

Network administrators have lost control over which client apps communicate in their sensitive networks. Not being able to centrally, *precisely and reliably* govern network accesses regularly results in data exfiltration by malware or exploitation attempts against vulnerable client software. Unfortunately, existing attempts to prevent such incidents (anti-virus, malware sandboxes, IDS, etc.) are susceptible to evasion. Therefore, in this chapter, we have designed SENG which shows how client-side TEEs can help address this challenge—answering *RQ1* (see page 4). SENG's ability to specify app-grained policies enables for fine-grained and *application-aware* traffic control concepts. Moreover, SENG provides strong security guarantees that are rooted in hardware and even withstand client-side system-level attackers. SENG thus fills a need that has existed since the introduction of firewalls: per-app attribution of network traffic.

SENG does not focus on the protection of the network traffic and firewall policies against a *gateway* system-level compromise (see A18 of Section 2.9). Therefore, in the next chapter, we will propose a TEE-based gateway router architecture that guarantees secure network I/O and policy enforcement, even under such a system-level attacker.

3

TrustedGateway

TEE-Assisted Routing and Firewall Enforcement using ARM TrustZone

3.1 Motivation

Gateway routers are at the heart of every network infrastructure, interconnecting subnetworks and enforcing access control policies using firewalls. However, their central position makes them high-value targets for network compromises. Typically, gateways are erroneously assumed to be hardened against software vulnerabilities ("*bastion host*"). In fact, though, they inherit the attack surface of their underlying commodity OSes which together with the wealth of *auxiliary* services available on both consumer and enterprise gateways—web and VoIP, file sharing, remote logins, monitoring, etc.—undermines this belief. This is underlined by a plethora of recent CVEs for commodity OSes and services of popular routers which resulted in authentication bypass or remote code execution thus enabling attackers full control over their security policies.

Therefore, in this chapter, we address *RQ2* (see page 4), i.e., answer if we can redesign gateway routers based on TEEs such that their network I/O and policy enforcement stays protected even if the router OS or auxiliary user space services have been compromised. We propose TrustedGateway (TruGW), a gateway architecture, which isolates "core" networking features—routing and firewall—from error-prone auxiliary services and gateway OSes. TruGW leverages a TEE-assisted design to protect the network path and policies while staying compatible with commodity gateway platforms. TruGW uses Arm TrustZone to protect the NIC and traffic processing from a fully-compromised gateway and permits policy updates only by trusted remote administrators. That way, TruGW can readily guarantee the secure enforcement of trusted policies on commodity gateways. TruGW's small attack surface is a key enabler to regain trust in core network infrastructures.

3.2 Problem Description

Gateway routers interconnect networks and govern their communication using firewall policies. Therefore, gateways are attractive targets for adversaries seeking to abuse their central position for network infiltration and information leakage. While gateways were assumed to be hardened ("*bastion host*"), a series of recent CVEs has raised serious concerns over their security. As we will show in Section 3.4, these vulnerabilities typically arise out of non-hardened auxiliary services that execute on gateways. These services easily add up to a large, complex code base which is hard to audit. Therefore, many serious vulnerabilities lurk in these auxiliary services, which enable attackers to remotely compromise the gateways. Once compromised, they threaten also core services (routing and firewalling), because gateways nowadays build on commodity OSes (cf. Table 3.1) for which several vulnerabilities and privilege escalation attacks have been revealed. Consequently, remote attackers can chain service to system exploits to gain full control over gateways and their policies—putting the entire network infrastructure at serious risk. As we will discuss in Section 3.4, the root cause indeed seems to be the increased threat surface due to *auxiliary* services and commodity systems, because, in fact, the core gateway tasks represent just a small fraction of the entire software stack and attack surface on gateways. Yet vendors keep adding a plethora of auxiliary gateway services (e.g., VoIP, file sharing, web proxies, printing, IoT hubs,

Table 3.1: A sample of router network operating systems (NOS) and the respective commodity OSes they derive from. Many common router NOSes are based on commodity OSes and therefore inherit their security vulnerabilities. Many "old", hardware-specific NOSes nowadays run as user space services or VMs on recent platforms.

Vendor	Network OS	Underlying Commodity OS
AVM	Fritz!OS	Linux [149]
Cisco	IOS XR	Linux (Wind River), old: QNX [42]
Cisco	IOS XE	Linux [44]
DrayTek	—	Linux (now: DrayOS) [196]
Juniper	Junos OS	FreeBSD [119]
Juniper	Junos OS Evolved	Linux [118]
—	DD-/OpenWRT	Linux [33, 66]

content caching) to increase system utility and gain marketing advantages—at the cost of security.

Researchers and large companies have realized the need for more secure network gateways and try to re-establish trust by isolating their critical core functionalities. However, existing approaches fail to protect commodity gateways—leaving millions of home and smaller enterprise networks vulnerable (cf. Section 3.5). Commodity gateways relying on VMs or OS containers for service isolation [43, 120] suffer from their huge attack surface, while data center SmartNICs, which perform routing and filtering isolated from the host system, are too expensive, bulky, and complex for commodity devices. Research proposals using secure containers based on Intel SGX for routing and firewall protection [179, 62, 18] rely on future hardware support and cannot guarantee policy enforcement on standalone gateways due to SGX's missing hardware control over NICs, which enables a full policy bypass.

To foster widespread protection of network infrastructures of consumers and smaller enterprises, we require a design that (i) guarantees secure enforcement of a gateway's routing and firewall policies even under a system-level attacker while having (ii) a small trusted computing base (TCB) and (iii) compatibility with commodity hardware and software. However, the complexity of network subsystems, including NIC I/O and multiple layers of system software, makes it particularly challenging to come up with a design that balances *security*, *performance*, and *compatibility*. For example, a fully isolated network stack provides high protection, but at the cost of a bloated TCB and potential incompatibilities with separated commodity services, while a low TCB solution might face security limitations or high performance penalties on calls into the protected submodules. In addition, compatibility with consumer gateways is often in conflict with new efficient security technologies (e.g., SmartNICs) and might require TCB-increasing extra frameworks.

3.3 Contributions

In this chapter, we present *TrustedGateway (TruGW)*, a system architecture for *commodity* gateway routers, which aims to tackle this design challenge. TruGW builds on Arm TrustZone-assisted trusted execution environments (TEEs) which provide HW-enforced memory and I/O isolation, can be easily combined with existing OS and hypervisor-based designs, and are widely available in millions of edge devices [178]. TruGW provides a new trusted networking core with a low TCB, which provides secure network I/O and traffic processing isolated from system-level attackers who have compromised the router OS and auxiliary user space services. TruGW leverages TrustZone (TZ) to protect the core's memory and grant it exclusive NIC access. That way, TruGW's network core has full control over the gateway's ingress and egress path and can guarantee the enforcement of trusted network policies. In particular, TruGW shows how to solve several technical challenges: (i) enable fast, trusted network I/O in spite of TZ's high context switching overhead, (ii) after NIC isolation, re-establish network access for commodity services *without* breaking security or compatibility, and (iii) allow for trusted policy configuration—all while preserving a low TCB.

Technically, TruGW implements a minimal NIC I/O framework in TZ's secure world, which provides essential network and link layer abstractions, and realizes trusted routing and firewalling on top of it. Trusted policies are configured by authenticated remote administrators via a new trusted configuration service. TruGW's framework enables to incorporate only the essential I/O parts of physical NIC drivers into TZ, which preserves a low TCB. To overcome TZ's slow context-switches, TruGW designs a trusted, lightweight notifier and worker system for efficiently scheduling trusted NIC I/O, while keeping the system scheduler and threading in the untrusted world for a better compatibility and TCB. For supporting the untrusted router services, TruGW implements a virtio-based network device in TZ, which exposes a virtual NIC to the untrusted system for shared network access. However, to prevent network attacks by untrusted services (e.g. ARP spoofing), TruGW tightly controls and filters their traffic.

We implement an open-source prototype of TruGW (cf. Section 3.10) [S2] by extending an existing TEE with ≈10.5 k lines of TruGW-specific code. We evaluated this prototype on the Nitrogen6X dev board [30], which nicely resembles the hardware configuration of small commodity gateways. Our proof-of-concept illustrates how TruGW efficiently enforces trusted routing and firewall policies even under a system-level compromise, and thus re-establishes trust in commodity gateways and their millions of consumer and enterprise networks.

In summary, we make the following contributions:

- We raise awareness of the serious risk of remote system compromises of commodity network routers, how they undermine firewall policies, and why existing defenses fall short of efficiently protecting consumer and SME routers.

- We design TrustedGateway (TruGW), an architecture which efficiently enforces trusted routing and firewall policies under a system compromise on standalone commodity gateways. TruGW is tailored to balance *security*, *performance*, and *compatibility* for seamless consumer and SME deployment.

Table 3.2: Number of CVE entries (as of 02.03.2022) for OSes, hypervisors, and Linux networking components based on categories of `cvedetails.com` and keyword searches on `cve.mitre.org`. The CVEs show that (i) security kernels (e.g. OP-TEE) face a way smaller risk of exploitation than commodity OSes used by routers, and (ii) the Linux firewall and NIC drivers add only minimally to the risk of code execution (CE) vulnerabilities compared to the full kernel or hypervisors. Note that the Xen and KVM hypervisor require an additional host OS (e.g. Linux).

Product	**Total CVEs**	**CE**	**Search Keywords**
Linux kernel	2763	263	—
Windows 10 OS	2590	538	—
FreeBSD OS	455	54	—
OP-TEE OS	10	3	—
Xen	378	20	—
QEMU (KVM)	355	77	—
Linux Firewall	67	2	linux netfilter
Ethernet NIC Driver	25	1	linux drivers net ethernet
Wireless Driver	39	1	linux drivers net wireless

- TruGW provides a TEE-tailored networking framework, and implements a TEE-located virtio-net device to support controlled network access by untrusted auxiliary or OS services.
- TruGW provides a low TCB, trusted web service for remote policy management with a secure admin enrollment process.
- We implement a TruGW prototype [S2] and evaluate its attack surface, network performance, and secure memory overhead.

3.4 Setting: Gateway Routers are High-value Targets

Gateway routers play a critical role for the security of consumer and enterprise networks. They isolate and interconnect internal client and server subnetworks, and their network firewalls serve as central gatekeepers for all ingress and egress network traffic. The gateways' central role makes them attractive targets for a network infiltration putting intruders in an ideal position for attacks. While gateways are widely assumed to be trusted, their number of services has drastically increased over the years and so did their attack surface. In fact, gateways nowadays fulfill a plethora of auxiliary functionalities beyond secure traffic control, including proxies to cloud services, edge computing, and typical consumer services such as file sharing, VoIP, streaming, or network monitoring. Table 3.1 shows that popular gateway platforms therefore derive from large commodity OSes, typically Linux, to easily integrate such services.

This software stack composition opens up a huge attack surface. For instance, 12 popular auxiliary network services on DD-WRT [66] routers together already include a large, error-prone code base of ≈4517 kSLOC (see Section 3.8.2). Table 3.3 (page 56)

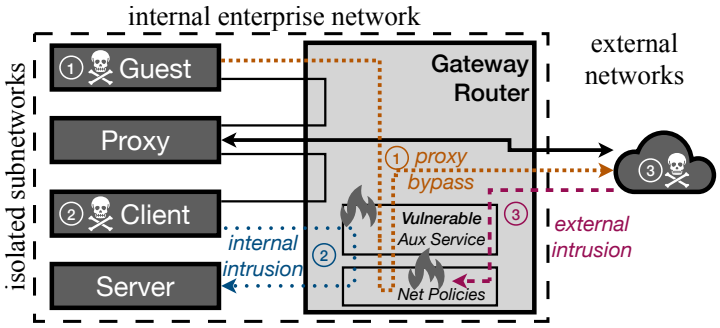

Figure 3.1: Three critical attacks enabled by vulnerable auxiliary services that undermine a gateway's network policies.

presents recent CVEs of popular network devices, that enable remote attackers control over a gateway's network policies or even the whole system—bypassing any kind of system-level defense. In fact, all these vulnerabilities lurk in auxiliary services and system software unrelated to the security-critical core networking components (e.g., firewalls). In addition, Table 3.2 shows that the widely-used Linux kernel has faced thousands of CVEs of which ≈10 % directly result in malicious code execution (CE)—with new ones getting steadily discovered [195, 232, 194]. In contrast, less than 100 CVEs have been reported for the Linux kernel firewall and Ethernet NIC drivers *together* with merely ≈3 direct CEs. However, the plethora of kernel and *remote* service vulnerabilities enable attackers to *fully* compromise gateways, and thus undermine also their security-critical components and policies.

Figure 3.1 shows exemplary consequences of such an insecure gateway in a small enterprise network. The central gateway interconnects an isolated guest, client, and multiple server subnetworks. The gateway firewall permits guests and clients to access external networks only through a traffic-filtering proxy. Furthermore, clients can access only servers of their work department, and the firewall heavily filters external ingress traffic. However, any vulnerable service on the gateway undermines these policies. Attackers can compromise the gateway using a remote code execution (RCE) against a service and perform a privilege escalation (e.g., kernel exploit) to gain access to the firewall policies. (1) A malicious guest could manipulate the firewall policies to bypass the proxy for direct external network access (e.g., to launch a spam campaign). (2) A malicious or malware-infected client can bypass the server isolation to sabotage or steal internal secrets. Lastly, (3) if vulnerable gateway services are exposed to the public (e.g., file sharing), they enable external attackers to infiltrate the enterprise network.

Our goal is to re-establish trust in gateways by designing TruGW, a new architecture for commodity network gateways, which enforces authenticated routing and firewall policies even when the auxiliary services or system software are compromised. That way, our envisioned gateway significantly hardens the security of enterprise networks by eliminating the discussed threats. Furthermore, TruGW strengthens millions of home networks by hardening consumer routers, which include a plethora of auxiliary features (e.g., media, IoT), by providing secure traffic isolation and filtering.

Table 3.3: A sample of recent security critical vulnerabilities in auxiliary services, OS kernels, and hypervisors (VMMs) used by popular network devices. The CVEs show that remote attackers can fully compromise such devices by chaining remote code execution exploits to OS and (if required) VMM exploits. Thus, attackers gain full control over a device's routing and firewall.

	CVE	Device	Target / Vulnerability	Attack Effect
User Space Network Services	2019-16028	Cisco Firepower Firewall	LDAP Bypass (via HTTP)	remote admin access
	2019-17621	D-Link DIR-859 Wi-Fi router	UPnP service (via HTTP)	remote code execution (LAN)
	2019-19494	Broadcom-based cable modems	buffer overflow (via JS)	remote kernel code execution
	2020-3115	Cisco SD-WAN	vManage (input validation error)	local privilege escalation (root)
	2020-11503	Sophos XG Firewall	awarrensmtp (heap overflow)	remote code execution
	2020-15635	Netgear WLAN Router R6700	acsd service (buffer overflow)	remote code execution
	2020-27600	D-Link Router DIR-846	HNAP service	remote command execution
	2021-0254	Juniper ACX/MX routers	overlayd (buffer overflow)	remote code execution
	2021-0260	Juniper net. devices (Junos OS)	snmpd (improper authorization)	remote SNMP read/write access
	2021-1287	Cisco Wireless VPN routers	web mngt. interface	remote code execution (root)
	2021-1539	Cisco ASR-5000 routers	TACACS auth. bypass (via SSH)	remote command execution
	2021-1602	Cisco RV160/260 routers	web mngt. interface	remote code execution (root)
	2023-20198	Cisco IOS XE routers/switches	web config. interface	remote access (non-admin)
	2023-20273	Cisco IOS XE routers/switches	web config. interface	remote privilege escalation (root)
OS	2020-7460	FreeBSD-based Routers (Table 3.1)	FreeBSD kernel	local kernel code execution
	2021-31440	Linux-based Routers (Table 3.1)	Linux kernel 5.11.15	local kernel code execution
	2023-32258	Linux-based Routers (Table 3.1)	multiple Linux v5/v6, in-kernel SMB	remote kernel code execution
VMM	2020-7467	FreeBSD-based Routers (Table 3.1)	bhyve (FreeBSD hypervisor)	VM escape (host code exec.)
	2020-14364	Linux-based Routers (Table 3.1)	KVM-QEMU (Linux hypervisor)	VM escape (host code exec.)
	2021-4206	Linux-based Routers (Table 3.1)	KVM-QEMU (Linux hypervisor)	VM escape (host code exec.)

3.4.1 Threat Model

TruGW relaxes the strong bastion host assumption for all gateway router software except for the "core networking" features—routing and firewall. We build on the common threat model which assumes a gateway located at the network perimeter and a set of internal (*int.*) and external (*ext.*) network clients trying to circumvent the gateway's security policies (cf. Figure 3.1). Motivated by the discussed plethora of gateway CVEs, we extend this model for TruGW in that we tolerate a system-level attacker (Sys_{gw}) which has gained major control over a gateway's software stack, including all auxiliary services, the OS, and, if available, the hypervisor (cf. Section 3.5). After a compromise, Sys_{gw} will attempt to leverage their central position to perform man-in-the-middle attacks, tamper with routing rules, and bypass firewall policies for full network access. We only trust verified admins which remotely manage the networking policies via secure configuration requests from trusted devices.

TruGW will root its security guarantees in hardware by leveraging CPU-provided secure containers (a.k.a. TEEs) for protecting the network traffic and policy enforcement. We therefore trust the gateway's CPU and all hardware bound to the TEE. Furthermore, we trust the software in the TEE—our trusted computing base (TCB)—and assume it to be free of vulnerabilities. While we regard Sys_{gw} in control of all non-TEE software, we exclude side-channel, denial-of-service (DoS), and all forms of physical attacks.

3.5 Towards Secure Network Gateways

We will now outline TruGW's design goals and requirements and discuss in how far alternative solutions fall short of fulfilling them.

3.5.1 Goals and Requirements

The goal of TruGW is the protection and guaranteed enforcement of a gateway's traffic routing and firewalling even under a full system compromise. In addition, we want TruGW to be easily integrable into commodity gateways without extra costs for wide adoption in home and (small) enterprise networks. TruGW therefore must build only on commodity hardware features and refrain from changes to a gateway's system software. At the same time, the interplay with existing gateway OSes and auxiliary services has to be efficient, and the architecture itself feature a small TCB that can be easily audited. We derive the following seven security (SR) and four auxiliary (AR) requirements that TruGW's design will fulfill:

SR1 Secure Network Setup. The setup phase must prevent unauthenticated network communication until the firewall has initialized a restrictive or restored a trusted state.

SR2 Routing and Firewall Isolation. The integrity of the routing and firewall components must be guaranteed.

SR3 Mandatory Policy Enforcment. The enforcement of the routing and firewall policies must be guaranteed.

SR4 Traffic Protection. The untrusted system must not be able to access (*confidentiality*) or tamper with traffic (*integrity*) not explicitly destined to it. This includes all forward traffic.

SR5 Spoofing Prevention. The untrusted system must not be able to spoof network addresses (e.g., MAC, IP).

SR6 Trusted Policy Changes. Only authenticated remote admins must be able to perform trusted policy changes.

SR7 Attack Surface. The trusted computing base (TCB) and exposed attack surface must be small.

AR1 Commodity hardware. The design must build only on cost-efficient commodity hardware applicable to network routers.

AR2 Service Compatiblity. The design must support existing untrusted gateway OSes and auxiliary (network) services.

AR3 Minimal Changes. The design must require only minimal changes to the untrusted commodity system software.

AR4 Network Overhead. The design must only introduce reasonably small network performance overhead to stay attractive to consumers and enterprises.

To achieve these goals, TruGW's idea is to leverage Arm TrustZone (TZ)—a widely available commodity TEE [178]—to isolate the network I/O path from the compromised system, and design new, trusted networking components. That way, even system-level attackers (Sys_{gw}) can neither tamper with network traffic or policies, nor bypass them. However, it is particularly challenging to come up with a design that fulfills multiple, partially conflicting goals, especially considering the complexity of network subsystems. For example, backwards compatibility (*AR1-3*) is often in conflict with new efficient security technologies (*AR4*) and might require additional, TCB-increasing frameworks (*SR7*), while a small TCB might limit the performance (*AR4*) or functionality (e.g. *SR3*). TruGW's main contribution is therefore to solve this design challenge and several additional challenges resulting from it (cf. Section 3.6 and 3.7).

3.5.2 Design Tradeoffs and their Shortcomings

Several related attempts follow similar objectives than our envisioned trusted gateway, yet fall short of fulfilling important security guarantees and/or deployment requirements. We now discuss these approaches and their shortcomings w.r.t. TruGW's properties, and motivate TruGW's decision in favour of a TrustZone-based design.

Dedicated Devices Moving core networking services to dedicated devices could be seen as an intuitive solution to our depicted problem. While such a physical separation removes potentially vulnerable auxiliary services from the core networking devices, even dedicated routers/firewalls still have a high attack surface, including a full commodity OS (~~*SR7*~~). In addition, the extra devices introduce additional prime, energy, and maintenance costs (~~*AR1*~~). Furthermore, the declined usability (lack of auxiliary services) and the resulting need for multiple devices destroys a core marketing argument of feature-rich routers (*related to* ~~*AR2*~~).

SmartNICs and P4 In-network firewalls have been proposed for scalable enforcement isolated from vulnerable gateway systems. FlowBlaze [180] enables stateful network functions on SmartNICs for high scalability, whereas Kang et al. [121] introduce context-aware policy enforcement on P4-programmable SDN switches. While these solutions promise great scalability and security, they are too expensive, complex (*related to* ~~*AR2/3*~~), and "bulky" (form factor) for consumers and smaller enterprises (~~*AR1*~~). In contrast, TruGW focuses on protecting exactly these millions of users by providing them with an affordable gateway design for commodity hardware.

Intel SGX Gateway designs based on Intel's commodity, hardware-isolated user space containers—so-called Intel SGX enclaves—suffer from their missing hardware control [49]. They cannot guarantee secure network policy enforcement on a standalone gateway, because they can neither directly access the NICs nor prevent attackers from doing so (~~*SR1/3/5*~~). Alcatraz [18] enforces firewall rules and traffic protection, but requires SGX support on every enterprise middlebox, switch, and host for per-hop tunnels (~~*AR1*~~). SafeBricks [179] and LightBox [62] securely offload middleboxes to an untrusted cloud provider using SGX, but must assume a trusted enterprise gateway to tunnel traffic to them (~~Sys_{gw}~~). SENG [P1] uses SGX on the client-side to enforce trusted per-application firewall policies on the gateway, but assumes the gateway to be trusted (~~Sys_{gw}~~), as discussed in Chapter 2 (especially in A18 of Section 2.9). TruGW's focus is on *providing* such a secure design for *standalone* gateways, i.e., we close a gap of existing orthogonal designs.

Virtualization Hypervisors enable a secure containment of compromised OSes and support secure I/O paths. Advanced gateway platforms by Cisco [43] and Juniper [120] already support VMs for running third-party user space services. However, for our envisioned gateway, hypervisors face two main limitations: a high attack surface (~~*SR7*~~), and compatibility issues (~~*AR3/4*~~). Following ideas of VMwall [208], a gateway could use a hypervisor to protect the network processing inside the host VM ("dom0") against a compromised gateway OS. However, Table 3.2 (page 54) shows that commodity hypervisors like Xen [216] or QEMU/KVM face a high attack surface, which is even further increased by the dom0 OS—by default a full-blown Linux. Even when splitting core services into multiple VMs (similar to QubesOS [217, 136]), the TCB stays large (~~*SR7*~~). Minimal, so-called *micro-hypervisors* have a low TCB but are by design functionally limited, e.g., to a single VM without isolated I/O, which makes efficient secure I/O difficult (~~*AR4*~~) [41]. Furthermore, the use of security micro-hypervisors is in conflict

with deployed commodity gateway hypervisors, and therefore either (a) requires slow, complex nested virtualization (~~*SR7,AR4*~~), (b) deep integration with gateway hypervisors (~~*AR2/3*~~), or (c) can only support gateways without hypervisors. McCormack et al. [153] have proposed such a micro-hypervisor-based secure gateway, however their concept fails to guarantee traffic protection and policy enforcement against system-level attackers (~~*SR3/4*~~). Zhou et al. [245, 246] used micro-hypervisors to build minimal TCB, trusted I/O paths from applications to specific device classes, but have not focused on NICs or network policies (~~*SR1–6*~~). TruGW's minimal TCB efficiently enforces and protects secure networking against Sys_{gw} even if they control a gateway hypervisor.

Confidential Virtual Machines (TEE VMs) Recent trusted virtualization technologies, e.g., AMD SEV-SNP, Intel TDX, and Arm CCA [3, 106, 16], transfer principles of TEEs (e.g. Intel SGX) to virtual machines. These confidential VMs are hardware-protected against the host platform (incl. hypervisor, peripherals) regarding the confidentiality and integrity of their memory and CPU registers. However, we are not aware of any research on trusted network routers using them for isolation. In fact, these technologies are currently not widely available yet (TDX, CCA) or mostly limited to high-end desktop or server CPUs (SEV-SNP) rather than cost-efficient router platforms (~~*AR1*~~*, related to* ~~*AR2/3*~~). While virtual routers or firewalls could be implemented inside confidential VMs, confidential VMs currently lack control over the hardware NICs and therefore cannot guarantee secure network policy enforcement on a standalone gateway (~~*SR3*~~), similar to SGX-based approaches [179, 62]. Furthermore, by default, such a VM would include a whole network OS with a huge attack surface (~~*SR7*~~). In Chapter 5, we use recent intra-VM isolation techniques of confidential VMs as part of our secure VM introspection system. Furthermore, we discuss future trusted I/O support of confidential VMs (Section 5.9.5) and briefly outline the idea of porting TruGW to them in Section 6.2.1. However, leveraging these features for VM-controlled secure network paths is non-trivial, might require SmartNIC (and CPU) support, and will therefore become available only in data centers rather than router platforms in the foreseeable future. In contrast, TruGW readily isolates network traffic and enforces routing and firewall policies on cost-efficient consumer and smaller enterprise routers.

Arm TrustZone TruGW builds on Arm TrustZone (TZ)[1], because TZ makes an ideal candidate for a secure network gateway due to its hardware-enforced memory and I/O isolation, and its widespread availability [178]. TZ provides hardware primitives for Arm-based TEEs, i.e., secure containers for hosting code and data isolated from all system software. Unlike Intel SGX, TZ is a system-level TEE and additionally features device isolation. TZ extends all system resources—including CPU, memory and devices—with a security state and supports HW-enforced access control rules based on the states [178, 175]. TZ's features enable standalone security architectures with trusted I/O similar to hypervisors, but with a potentially very small TCB (cf. OP-TEE's small number of CVEs in Table 3.2 on page 54, and code size in Section 3.9.2 and 3.8.2) and without being in conflict with deployed gateway hypervisors. In fact, TrustZone

[1] Our current focus is on Arm TrustZone for Cortex-A (TZ-A).

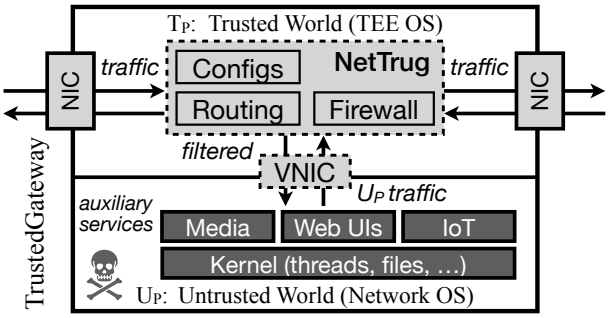

Figure 3.2: Design overview of TruGW with the new (dashed) trusted NetTrug and VNIC (dark: untrusted, light: trusted).

has been used for many domains like trusted user I/O [140, 236], trusted peripheral access [125, 148], and secure stream processing [175]. However, none of these approaches explore the protection of a gateway's network path and policy enforcement (~~*SR1-6*~~). Even though StreamBox-TZ [175] proposes exclusive NIC access by trusted components for stream processing performance, it simply assumes trusted networking stacks and NIC isolation as an available black box. In fact, StreamBox-TZ neither provides details about networking, nor considers network policies, nor access by untrusted services (~~*SR1-6, AR2-4*~~). To the best of our knowledge, *there is no such* trusted networking support fulfilling all requirements for secure gateways. Therefore, TruGW designs new trusted networking components as part of its secure gateway architecture.

3.6 TruGW's Design

We now describe TruGW's gateway design and mention challenges it had to solve. To highlight how TruGW fulfills the requirements outlined in Section 3.5.1, we refer to them at relevant passages. We provide additional details of TruGW's architecture and its implementation in Section 3.7.

TruGW's main idea is to isolate network I/O and critical "core" gateway functionalities from a gateway's error-prone auxiliary services and system software. That way, TruGW's network "core" keeps full control over the network traffic and can guarantee secure policy enforcement even on a service or system compromise. As shown in Figure 3.2, TruGW uses a TrustZone-assisted TEE to divide the gateway architecture into an untrusted (U_P) and a memory-isolated trusted (T_P) partition. The untrusted U_P runs a gateway OS—in the following called *network operating system (NOS)*—which hosts the auxiliary services and commodity kernel. The trusted T_P hosts the TEE OS and all core components of TruGW, i.e., our TCB. TZ-assisted TEEs [145, 90] have a minimal TCB (*SR7*) with few CVEs (Table 3.2, OP-TEE), however, at the cost of a very limited secure runtime dedicated to small, U_P-exposed RPC services, e.g., trusted key storage [92]. Current TEE OSes are *not* designed for fast, I/O-intense tasks and thus *neither* support trusted network I/O *nor* traffic processing. Therefore, TruGW designs a new TZ-tailored networking core in T_P called *NetTrug*. NetTrug includes new modules for trusted network I/O, routing, and firewalling isolated from U_P at-

tackers (cf. Figure 3.2). Policies are remotely configured via a new trusted interface (Section 3.6.3). To preserve compatibility with U_P services under an isolated network path, TruGW implements a new trusted virtual network device called *VNIC*, which together with NetTrug provides U_P with tightly-controlled network access (*AR2*).

We will now present how TruGW tackled the following major challenges: (i) achieve fast, trusted networking in spite of TZ's high context-switch overhead, (ii) securely share network access with U_P services *without* breaking security or compatibility, and (iii) provide trusted policy configuration—all while preserving a low TCB.

3.6.1 Trusted Networking

In a commodity gateway, the network I/O and processing is performed by drivers and services typically located in the NOS kernel. The NIC drivers form the I/O interface to the NICs (network hardware) while the services perform essential tasks, e.g., routing. However, their location makes them fully controllable by U_P system-level attackers enabling them to tamper with all traffic and bypass any security policy. To guarantee secure traffic and policy processing (*SR1-4*), NetTrug therefore revokes U_P's NIC access and provides trusted networking in T_P.

I/O and Scheduling First, NetTrug must enable trusted NIC I/O paths. NetTrug therefore protects the NICs against U_P and supports trusted NIC drivers in T_P. NetTrug protects a NIC's I/O interfaces in T_P: memory-mapped device registers, shared I/O rings, and interrupts. Device registers enable drivers to interact with a NIC and especially configure the memory location of the I/O descriptor rings. These rings contain information about processable network buffers and by default reside in unprotected system memory together with their buffers. Descriptor changes are signaled via NIC interrupts and device registers [48]. If these interfaces stay unprotected, U_P attackers can tamper with network traffic inside the I/O buffers or directly interact with the NICs and thus bypass any policy (~~*SR1-4*~~). Therefore, NetTrug leverages TZ's Protection Controller (TZPC) [178] to bind all NICs exclusively to the trusted kernel space (T_P^k) from boot on (*SR1*). That way, TZ blocks all U_P access attempts to the NIC registers and securely redirects all NIC interrupts to T_P^k.

To protect the I/O rings and enable trusted I/O operation, NetTrug requires trusted NIC drivers inside T_P^k. However, current TZ-assisted TEEs [145, 90] have *no* support for network I/O. Naïvely, we could try to port existing drivers to T_P, but this raises several technical challenges: a full port would massively bloat the TCB (~~*SR7*~~), because drivers heavily depend on large, kernel-integrated driver frameworks and include many management functions beyond I/O. Furthermore, driver frameworks assume *fast* interrupt and threading support, which is either (i) not available in T_P due to TZ TEEs [145] relying on U_P for scheduling, which suffers from costly TEE context-switches and limitations in interrupt contexts (cf. Section 3.7.2) (~~*AR4*~~), or (ii) requires secure hardware timers [90] and respective U_P system-level changes for a TZ-tailored system scheduler—violating TruGW's goal of a *low TCB* design for *commodity* gateways (~~*SR7,AR1+3*~~).

Instead, NetTrug designs two new trusted kernel frameworks in T_P: a NIC I/O framework with partial driver integration, and a notifier and worker framework for

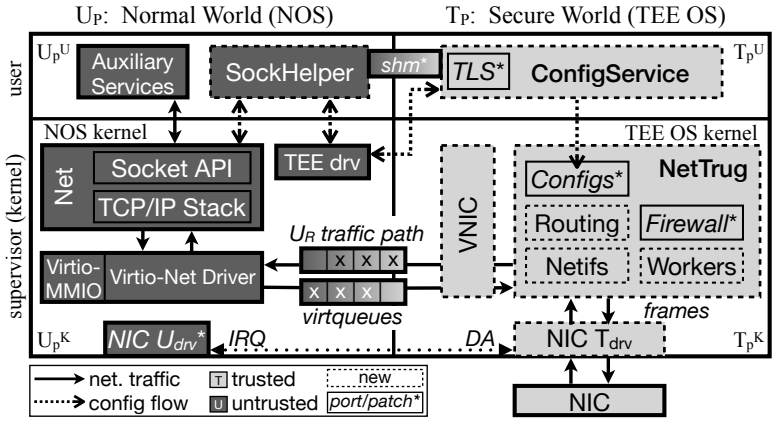

Figure 3.3: The TruGW architecture with untrusted (dark) and trusted (light) components. New components are marked with dashed lines; heavily ported or patched ones with stars.

efficient I/O scheduling. To keep the TCB small (*SR7*), NetTrug's I/O framework implements only the essential network and link layer abstractions required for I/O operations of NIC drivers, e.g., packet queues and NIC device interfaces. Furthermore, NetTrug splits each NIC driver in two parts: a trusted I/O part (T_{drv}) and an untrusted auxiliary part (U_{drv}). As shown in Figure 3.3, NetTrug integrates only the trusted part T_{drv} into T_P^k, but keeps U_{drv} in the untrusted network OS. T_{drv} protects the NIC I/O descriptor rings in T_P memory, handles NIC interrupts, and securely performs I/O isolated from U_P attackers (*SR4*). U_{drv} has no NIC access and only handles uncritical tasks on behalf of T_{drv} (split details in Section 3.7.3.1). To enable fast but compatible, low TCB I/O paths (*SR7,AR1+3-4*), NetTrug keeps the system scheduler and threading in the U_P NOS and instead designs new trusted NIC I/O workers. These workers build on lightweight, U_P-scheduled TEE OS threads, but are designed to minimize costly TZ context switches to U_P and be notifiable by trusted interrupt handlers. They are scheduled via a new trusted notifier on packet events, and run all NetTrug network tasks, incl. T_{drv} (details: Section 3.7.2). Combined, these frameworks ensure that NetTrug has exclusive control over the gateway's ingress and egress network paths and can efficiently perform secure NIC I/O even under a full U_P system compromise (*SR1+4,AR4*).

Routing and Firewall For secure networking, NetTrug additionally requires trusted traffic routing and filtering—features entirely missing in current TEE OSes. However, we cannot directly port existing network stacks into TZ. Similar to driver frameworks, they consist of large modules which would bloat the TCB (~~*SR7*~~) and heavily rely on threading and synchronization primitives not efficiently available in compatible TEE OSes (~~*AR1+3-4*~~). In addition, these stacks are not designed to defend against U_P system-level attackers (~~*SR1-5*~~). Therefore, NetTrug designs new TZ-aware networking modules on top of its I/O and worker frameworks. In contrast to existing stacks, NetTrug focuses on the security-critical "core services"—routing and firewall—and explicitly excludes client protocol and socket stacks (e.g., TCP/IP) from T_P to minimize

the TCB [248] (*SR7*), as shown in Figure 3.3. NetTrug's exclusive NIC control guarantees secure traffic processing by its networking modules (*SR3-4*). NetTrug introduces trusted and untrusted network interfaces on which its workers enforce trusted routing and firewall policies. NetTrug maps all physical NICs to trusted interfaces by default, and can enforce extra routing and filter rules on untrusted interfaces. In Section 3.6.2, we will explain how NetTrug and its virtual VNIC device securely enable tightly-controlled network access to U_P services via an untrusted network interface. For traffic filtering, NetTrug incorporates a network firewall into its trusted networking frameworks. NetTrug assumes at least a stateful L3/L4 firewall for secure, efficient traffic protection, but is conceptually oblivious to the concrete firewall capabilities (cf. Section 3.7.2). In contrast to commodity gateways, NetTrug securely manages trusted routing and firewall policies in T_P and guarantees mandatory policy enforcement. U_P attackers can neither bypass nor tamper with these policies (*SR1-3*).

3.6.2 Securely Sharing Network Access

As NetTrug isolates all NICs and processes their traffic securely in T_P^k, any direct network access by U_P attackers is blocked (*SR4*). However, as NetTrug focuses on trusted network I/O and security-critical services (*SR7*), all remaining gateway services stay in U_P and become unreachable. To resolve this compatibility issue, TruGW designs VNIC, a new virtual network device that performs secure traffic forwarding between U_P and T_P. In contrast to commodity virtual network devices [165], VNIC is tailored to TZ and integrated into NetTrug's networking frameworks. Together with VNIC, NetTrug enables tightly-controlled network access for U_P services (*AR2*).

From U_P's point of view, VNIC exposes a memory mapped virtio-net device, i.e., a virtual Ethernet card with a low TCB memory interface (virtio-mmio) following the virtio standard [165] (*SR7*). That way, the U_P NOS can use its builtin virtio drivers (*AR3*) to initialize a network interface to VNIC, which serves as U_P's default interface for all network I/O (cf. Section 3.7.3.2). The interface is configured with all IP addresses of the gateway. As a result, U_P services need not be modified and can use the standard socket API and TCP/IP stack of the NOS for their network communication (*AR2*).

From T_P's point of view, VNIC is a special trusted NIC driver associated with an untrusted NetTrug network interface. VNIC performs the buffer I/O between the untrusted U_P virtio-net driver and NetTrug. VNIC's virtual I/O rings are located in untrusted memory shared with the U_P driver (virtqueues, Figure 3.3). Therefore, VNIC must securely copy network buffers between the rings and T_P and check that untrusted buffers never overlap with trusted T_P memory. That way, VNIC prevents traffic tampering and memory attacks by U_P system-level attackers and enables NetTrug to securely process the network buffers in protected T_P memory (*SR3, SR4*).

VNIC provides an explicit, secure path to U_P services and thus enables NetTrug to make them reachable again. However, NetTrug must enforce additional security measures on the VNIC-associated untrusted network interface to protect forward traffic against U_P (*SR4*) and prevent address spoofing attacks by U_P-located attackers (*SR5*). By default, NetTrug routes traffic only to U_P if it is explicitly destined to one of TruGW's IPs. That way, the forward traffic stays isolated from U_P (*SR4*) and avoids

additional I/O overhead (*AR4*). To prevent spoofing by U_P (*SR5*), NetTrug replaces the source MACs of all egress traffic with those of the output interfaces, drops packets from U_P with spoofed source IPs, and handles U_P's host discovery messages locally in T_P (Section 3.7.4.1). In addition, TruGW enables trusted admins to define firewall policies directly on the VNIC interface to tightly control network access from and to U_P, as discussed next.

3.6.3 Trusted Policy Configuration

Administrators are used to manage network policies using a NOS-provided U_P web application. However, any configuration service inside U_P gives system-level attackers full control over a gateway's policies (~~*SR3+6*~~). Naïvely, we could isolate a configuration service in T_P and make it remotely reachable directly via NetTrug. Yet this would require a full network and web stack in NetTrug (incl. a full-fledged web server, TCP/IP stack, and socket API), leading to a stark increase in its TCB size and attack surface (~~*SR7*~~).

Instead, TruGW offers a web-based configuration that does not require these complex software stacks. To this end, we introduce *ConfigService*, a new tiny trusted user space (T_P^u) service for secure remote configuration of NetTrug's trusted network policies (cf. Figure 3.3). ConfigService provides authenticated admins with a trusted web application for policy management (*SR6*) while offloading web resources and the connection handling securely to U_P for a low TCB (*SR7*). ConfigService includes a new minimal (≈2.1 kSLOC, plus TLS library) HTTPS endpoint to handle TLS sessions with admins and ship a web interface to their browsers. To minimize the TCB (*SR7*), ConfigService securely offloads the TCP socket management to U_P; a new untrusted user space service (SockHelper) handles the TCP sockets for ConfigService and forwards the protected TLS records between the U_P network stack and ConfigService (Figure 3.3). That way, ConfigService requires no TCP/IP or socket stack inside T_P.

SockHelper makes ConfigService remotely reachable via VNIC. However, U_P attackers become strong on-path MITM attackers as they control the shared U_P TCP/IP stack. While TLS provides end-to-end protection between admins and ConfigService, ConfigService must additionally prevent impersonation and web attacks by U_P (*SR6*). Therefore, TruGW introduces a dedicated trusted web address (domain or IP) for ConfigService and supports a secure enrollment process for establishing credentials for mutual TLS authentication. The trusted address guarantees a different web origin than U_P services even though the TCP/IP stack is shared and thus enables ConfigService to prevent web attacks by U_P (cf. Section 3.7.5). During the enrollment process, TruGW generates a TLS server certificate with ConfigService's trusted address and registers a TLS client certificate for a master admin. Admins then submit their own TLS client certificates to ConfigService and get them approved by the master admin. By leveraging TLS client certificates, TruGW avoids password-related security issues [84], reduces the risk of phishing attacks, and can benefit from TPM-based storage backends [91].

Policy Translation TruGW avoids inventing new policy languages to ease adoption. ConfigService uses a standard routing syntax (similar to ip-route [123]) and the vanilla

firewall syntax (cf. Section 3.7.2) for configuration. In addition, ConfigService enables the reconfiguration of TruGW's IPs (cf. Section 3.7.4.2). Admins can reuse existing policies and further restrict services running on the gateway by defining new routing and firewall rules for the VNIC interface. The VNIC interface enables admins to explicitly control traffic from and to untrusted U_P services and ConfigService. Comparing to a commodity firewall configuration tool like iptables, firewall rules on NetTrug's physical NIC interfaces roughly translate to the pre-/postrouting and forward chains of iptables while rules on the VNIC interface roughly translate to the input and output chains.

3.7 TruGW Details and Implementation

We will now present the details of our TruGW architecture. We picked Linux as the U_P OS, given that many commodity gateway NOSes are derivatives of Linux (cf. Table 3.1, page 52). For the TEE, we chose OP-TEE [145] as it is a well-known, open-source TEE for TZ with upstream Linux support and a low TCB (*SR7*, cf. Table 3.2 on page 54 and code size in Section 3.9.2 and 3.8.2). OP-TEE provides a small trusted kernel component running in TZ's secure kernel mode (T_P^k) and supports trusted user applications (TAs) in the secure user mode (T_P^u). Our implementation targets an i.MX6 SoC [164], which features a TZ-compatible Central Security Unit (CSU) for device isolation and a TZ Address Space Controller (TZASC) [178] for the memory partitioning. Without sacrificing generality and for ease of discussion, we assume an Ethernet-based router that operates in an IPv4 network.

3.7.1 Technical Background

We now provide background information on Arm's boot security and OP-TEE's threading architecture as relevant for TruGW's secure TEE integration and I/O workers.

3.7.1.1 Arm Secure and Trusted Boot

Arm secure boot provides mechanisms to verify that only trusted images are loaded during system boot. To implement this, the boot images are signed and each loader verifies the image of the next stage before transferring control to it. That way, secure boot establishes a chain of trust which is rooted in a trusted root signing key. While the details are implementation-specific, the concepts of secure boot are well known [25, 14]. They typically include: (i) a trusted root key k_{rot} stored (typically by the manufacturer) in tamper-proof non-volatile storage (e.g., OTP) inaccessible by system software, (ii) a trusted boot ROM which uses k_{rot} to verify the signature of the first stage bootloader image, and (iii) a set of public key hashes used to verify subsequent boot images (e.g., TEE image, U_P bootloader) [25, 14]. The TrustZone-specific trusted board boot [14] follows these principles to enable verification of the TEE (T_P) image(s) and the U_P bootloader. TruGW integrates its trusted kernel components into OP-TEE's TEE image such that they are securely verified on boot, as will be discussed in Section 3.7.2.

3.7.1.2 OP-TEE's Thread Scheduling

OP-TEE [145] is not in control of the CPU scheduling and relies on the Linux scheduler in U_P for running secure tasks. Linux applications must explicitly call into OP-TEE for service. This design suites OP-TEE's service-oriented design and principle of least privilege and contributes to OP-TEE's small TCB (*SR7*). OP-TEE implements the GlobalPlatform TEE Client API [86] which enables applications to create a session with an OP-TEE trusted application (TA) and then invoke TA-exposed RPC interfaces. OP-TEE's Linux driver and secure kernel handle the resulting thread context switches between Linux (in U_P) and OP-TEE's TAs (in T_P) based on TrustZone's SMC CPU instruction [178]. OP-TEE's kernel stores the execution contexts of the Linux threads in trusted TEE thread structures while they perform TEE tasks. In Figure 3.3 (page 63), the resulting control flow is shown for TruGW's SockHelper and ConfigService TA.

However, a design like OP-TEE's causes high performance overhead and limitations. Keeping the thread scheduler in U_P significantly increases the required number of expensive context switches between T_P and U_P on common tasks, e.g., thread synchronization. Furthermore, trusted T_P interrupt handlers cannot schedule TEE tasks as the U_P scheduling APIs used for TEE-associated threads are context-switching and therefore incompatible with interrupt contexts [215]. For that reason, such scheduling designs (incl. OP-TEE's) are unsuitable for fast, trusted network I/O as required by TruGW. This motivated us to design TruGW's new worker framework (cf. Section 3.6.1) which overcomes these limitations and enables efficient NIC I/O in T_P while keeping the scheduler in U_P, as we will discuss in detail in the next section.

3.7.2 TEE Integration and Networking

TruGW's security is rooted in the integrity of its T_P components and boot process. Therefore, TruGW leverages secure boot to guarantee that only trusted bootloader and TEE images are loaded (cf. Section 3.7.1.1). TruGW's trusted kernel (T_P^k) components (cf. Figure 3.3) extend OP-TEE's kernel and are therefore verified as part of the TEE images (SR2). Optionally, TruGW can include all U_P OS images in the secure boot chain, e.g., by combining trusted boot with UEFI secure boot [36]. The trusted bootloader includes a device tree (DT) blob [144] which describes all hardware components of the system. On TEE boot, NetTrug parses the DT to bind all NICs to T_P by configuring them as secure in i.MX6's CSU[2] [135] and thus protect them against U_P (cf. Section 3.6.1). To prevent early boot attacks by U_P, TruGW transfers control to the U_P bootloader only after all protections have been successfully set up (*SR1*).

Trusted Networking NetTrug is TruGW's central extension to the trusted TEE kernel. NetTrug mediates all gateway traffic and securely performs trusted network I/O and policy enforcement in T_P^k (cf. Section 3.6.1). On TEE boot, NetTrug initializes one trusted network interface for each NIC and one untrusted interface for VNIC and allocates an egress queue, ARP cache (cf. Section 3.7.4.1), I/O workers, and a configurable, static IP address (cf. Section 3.7.4.2) to each of them. NetTrug tags untrusted

[2] a TZPC or an other SoC-specific technology can replace the CSU

interfaces, s.t. its routing and firewall modules can enforce special restrictions on them, e.g., to isolate trusted forward (and broadcast) traffic and prevent spoofing attacks by U_P (*SR4-5*; cf. Section 3.6.2 and 3.7.4.1). For packet filtering, NetTrug incorporates the stateful, BPF-based L3/L4 firewall NPF [188]. To this end, we ported NPF to OP-TEE and NetTrug's worker framework, and integrated it as a callable firewall module into NetTrug's networking loop, where NPF enforces trusted filter rules on given IP packets. We picked NPF as it is well-known (NetBSD's firewall) and feature-rich. However, conceptually, NetTrug could adopt additional firewall modules (e.g., application level) as trusted kernel or user modules.

NetTrug's new I/O workers perform the actual traffic processing for each interface securely in T_P^k using a polling-based I/O model. On setup, NIC drivers (incl. VNIC) request I/O workers for their interfaces and allocate device-specific I/O callbacks to them. On a packet event (e.g., signaled by an interrupt handler), workers poll and process all current RX (or TX) packets of their assigned NIC, before reentering a sleep state. They perform a typical I/O loop: (i) Ethernet RX via driver, (ii) link layer processing, (iii) ingress filtering and IP routing, (iv) egress filtering and ARP resolution, (v) egress enqueueing, and (vi) packet transmission via driver.

NetTrug's Workers For TruGW to be practical, it is crucial that TruGW's trusted networking causes only a small performance penalty compared to commodity gateways (*AR4*). While TruGW and OP-TEE both follow the idea of keeping full scheduling and threading stacks in U_P to preserve compatibility and a low TCB (cf. Section 3.6.1), OP-TEE's approach is not suitable for efficient NIC I/O. OP-TEE relies on U_P threads to call into the TEE for service and assigns them lightweight TEE tasks (a.k.a. threads) on entry (cf. Section 3.7.1.2). This design causes high overhead on thread switches and synchronization—both omnipresent in networking cores—due to costly context switches between T_P and U_P. In addition, it is *not* possible to schedule TEE tasks from trusted interrupt handlers as required for NIC I/O, because the U_P APIs are context-switching and thus not callable from interrupt contexts [215].

NetTrug's trusted workers build on lightweight (OP-)TEE threads, but overcome their limitations. NetTrug exposes a new, minimal worker registration interface to U_P, which a helper service uses to provide a pool of U_P threads. One thread registers as NetTrug's notifier and the others as workers. NetTrug's networking modules (e.g., drivers) can request scheduling of a worker using a new dedicated T_P^k API (similar to NAPI [48]). The API directly flags a worker without any context switch and is thus also callable from trusted NIC interrupt handlers, e.g., on a packet event. NetTrug's notifier periodically checks for flagged workers and if sleeping, wakes up their associated threads using U_P's scheduler. As the worker's sleep and wake-up operations fall back to costly context switches to U_P, NetTrug minimizes their number using several optimizations, e.g., I/O batch processing, notification coalescing on multiple packets or full queues, and a grace period of idling before putting worker threads to sleep. That way, NetTrug keeps the performance penalty low (cf. Section 3.9.3) while preserving a U_P-compatible, low TCB design.

3.7.3 Trusted Network Device I/O

3.7.3.1 Split NIC Driver Operation

NetTrug's network I/O and worker frameworks provide the essential support required for secure and efficient NIC driver I/O in T_P. As full NIC drivers would bloat the TCB (*SR7*), we split them and port only the critical, I/O relevant driver parts to OP-TEE and NetTrug while keeping the uncritical rest in U_P (cf. Section 3.6.1). On T_P boot, the secure subdriver T_{drv} registers a trusted network interface and I/O workers on NetTrug for the NIC and securely allocates the NIC I/O descriptor rings in T_P. Combined with the NIC's T_P-binding established by NetTrug (cf. Section 3.7.2), the NIC is in a clean and protected state before the untrusted NOS starts booting (*SR1*). On U_P boot, the untrusted subdriver U_{drv} is responsible for performing uncritical configuration tasks (e.g., power management) [246] and starting the physical Ethernet device of the NIC (PHY).[3] However, the NIC protection blocks any access attempts by U_{drv} to a NIC, s.t. they result in a data abort (DA). Therefore, T_{drv} registers a secure DA handler. That way, if an uncritical U_{drv} task requires a one-time NIC access (e.g., PHY startup), T_{drv} can trap the access fault in T_P, decode it [135], and securely perform the access on behalf of U_{drv}. After boot, the trusted NIC workers securely perform the NIC I/O and the packet forwarding between the NICs and NetTrug. T_{drv} securely handles the NIC's I/O interrupts in T_P and forwards uncritical ones to U_{drv} if required. *U_{drv} is not involved in the I/O phase*, which enables a secure, low overhead operation (*SR2-4, AR4*).

3.7.3.2 VNIC Device I/O

We designed VNIC's U_P-interface based on virtio-net and virtio-mmio [165] to make it compatible with commodity NOSes and drivers (*AR3*) while having a small TCB (*SR7*). On T_P boot, VNIC registers an untrusted network interface on NetTrug and extends the device tree [144] (cf. Section 3.7.2) to expose itself as a simple (*SR7*), memory-mapped device to U_P (virtio-mmio). On U_P boot, Linux detects the VNIC device and uses its virtio default drivers to set up a network interface for U_P. To enable U_P interaction, VNIC exposes virtual device registers to U_P using a dedicated memory region. VNIC protects the region from U_P via the TZASC (cf. Section 3.7), s.t. access attempts by U_P trap as data abort exceptions into T_P. On a trap, VNIC decodes the respective physical target address [135] and maps it to its virtual device registers. That way, VNIC can transparently detect and handle configuration requests and I/O ring notifications by U_P. On network I/O, VNIC's NetTrug worker receives Ethernet frames from U_P or NetTrug, securely processes and routes them, and forwards traffic between T_P and U_P (cf. Section 3.6.2 and 3.7.2).

[3] a potential splitting of the PHY drivers is left as future work

3.7.4 Address Resolution and Assignment

3.7.4.1 ARP

TruGW must guarantee secure MAC address resolution to prevent redirection and spoofing attacks by attackers in U_P (*SR5*). Therefore, NetTrug includes a trusted ARP stack inside T_P and performs extra checks on U_P traffic. For the physical NIC interfaces, the ARP stack handles MAC address resolution and ARP requests securely in T_P^k. For the untrusted VNIC interface, NetTrug performs special steps to prevent ARP spoofing attacks by U_P: (a) U_P's ARP requests are directly answered by NetTrug with a virtual MAC and (b) U_P's ARP replies are dropped. That way, NetTrug transparently handles U_P's ARP resolution and prevents U_P from poisoning the ARP caches of any NIC interface or of any internal or external host (*SR5*). When forwarding traffic to U_P, NetTrug knows VNIC's U_P-exposed MAC and can directly use it as the destination MAC.

3.7.4.2 DHCP and DNS

By default, TruGW does not assign IP addresses or handle DNS queries to keep its TCB small (*SR7*). TruGW has a set of static, preconfigured (yet configurable) IP addresses (cf. Section 3.7.2). We assume that network admins reconfigure these to fit their setup and operate a dedicated DHCP server to assign addresses to clients. Conceptually, NetTrug could incorporate a *basic* DHCP stack for smaller networks, e.g., providing gateway, client, and DNS server IPs. However, a full DHCP server would require a UDP/IP and socket stack inside T_P, which significantly increases TruGW's TCB (cf. Section 3.6.1). Regarding DNS, the current design of TruGW assumes DNS to be outside of the gateway, such as a dedicated DNS resolver or an external DNS resolver (e.g., provided by ISPs or other entities such as Google). Either way, NetTrug protects the confidentiality and integrity of DNS and DHCP communication against U_P system-level attackers using its restrictive routing and anti-spoofing measures on the VNIC interface (*SR4-5*; cf. Section 3.6.2 and 3.7.4.1).

3.7.5 Trusted Policy Management

TruGW must prevent unauthenticated network communication by U_P and network attackers until a trusted policy has been provided. On startup, NetTrug therefore sets up a "restrictive boot policy". This policy only allows local HTTPS connections *to* TruGW's configuration ports, but neither outgoing U_P connections nor traffic forwarding across network clients. That way, NetTrug restricts network traffic to local configuration sessions until a policy gets configured via ConfigService or securely restored from disk (*SR1*).

ConfigService is implemented as an OP-TEE trusted user application (TA). Its binary is signed, integrity checked by OP-TEE on load, and protected against version rollbacks [145]. On an admin connection, ConfigService ships only an initial tiny, integrity-checked (hash) root HTML file. All other web resources are loaded from an untrusted U_P Apache server. That way, ConfigService can keep its latency and memory footprint low (cf. Section 3.9.3 and 3.9.4) and does not depend on external resources

which are blocked on startup (*SR1+7,AR4*). ConfigService uses subresource integrity (SRI) [11] to guarantee the integrity of the U_P-offloaded resources (*SR6*). Furthermore, it verifies custom HTTP request headers to protect against cross-site request forgery (CSRF) [23]; attacker-induced requests from different origins, e.g., by rogue untrusted services (cf. Section 3.6.3), cannot add such custom headers. To support NPF's policy language, we ported NPF's client tool to WebAssembly (Wasm) [67]. It parses the NPF policies inside the trusted admin browsers and sends BPF filters via ConfigService to NetTrug, where they are securely parsed, compiled, and enforced.

ConfigService's server and client authentication is based on TLS server and client certificates, respectively. On initial boot, NetTrug securely issues a self-signed TLS server certificate C_{cnf} for ConfigService's trusted web address (cf. Section 3.6.3) and stores it on rollback-protected storage. For initial enrollment, the master admin then connects via an exclusive physical network access to ConfigService and uploads a securely generated TLS client certificate C_{mst}. In addition, the master admin distributes C_{cnf} to all admins for certificate pinning (cf. Section 3.7.6) to prevent phishing and CSRF attacks against ConfigService's trusted address, especially by U_P attackers. The master can trust the initial C_{cnf} on first use (TOFU) as the secure boot (cf. Section 3.7.2), factory state of U_P, and exclusive network access of the master admin rule out any device or network attacker. On completion, ConfigService securely stores C_{mst} and starts enforcing access control based on the TLS client certificates of the HTTPS client connections. Clients without a registered TLS client certificate can only upload a TLS client certificate C_{adm} to request admin access, which then has to be explicitly granted by the master. Only admins and the master have access to the trusted routing and firewall policies. The master can additionally revoke admin certificates or request server key rollovers, e.g., on a key breach. An explicit trusted factory reset (e.g., via button) can wipe *all* certificates for a full re-enrollment.

3.7.6 Deployment

TruGW has been designed with the goal to be compatible with commodity Arm gateway routers (*AR1*). TruGW currently requires Arm TrustZone with memory and device isolation (TZASC, TZPC) and support for rollback-protected storage (e.g., eMMC with RPMB). T_P's secure memory demands are about 16–32 MB and therefore easily met by many router platforms (cf. Section 3.9.4). Regarding software, TruGW is compatible with commodity Linux and its upstream OP-TEE and virtio drivers (*AR3*). The untrusted NIC drivers (U_{drv}) are slightly adapted versions of the Linux drivers. Manufacturers can easily deploy TruGW, because its T_P^k components are direct extensions of the OP-TEE image(s) and its ConfigService TA and U_P services can be packed into OP-TEE's Linux software package. TruGW is non-intrusive in that its TEE extension does not affect other applications (*AR2*) and its U_P helper services (e.g., SockHelper) do not require any special permissions.

Manufacturers can update TruGW using standard methods. The untrusted and trusted user space components (incl. ConfigService) can be updated via regular Linux package updates. Attackers cannot manipulate the trusted components as OP-TEE only accepts vendor-signed TAs (cf. Section 3.7.5). TruGW's trusted kernel components

(e.g., NetTrug) require an update of the TEE image using existing (or device-specific) methods for firmware updates [15]. TruGW does not affect the way commodity U_P software is updated.

Admins can follow common best practices for managing TruGW's TLS server and client certificates. The master admin distributes ConfigService's server certificate C_{cnf} to all admins for certificate pinning (e.g., via group policies). U_P manages TLS server certificates of untrusted U_P services. Admins must vet these U_P certificates to *not include* the trusted web address of ConfigService before distributing them to guarantee distinct web origins (cf. Section 3.6.3). To ease the U_P vetting, TruGW could integrate a T_P certificate authority restricted to untrusted addresses (cf. `RFC5280`), whose certificate could then be distributed instead. Key breaches and rollovers are securely handled by master or via a full re-enrollment (cf. Section 3.7.5).

3.8 Security Analysis

We now analyze TruGW's security design by discussing its countermeasures against critical attacks and assessing how it contains real-world vulnerabilities of commodity gateways.

3.8.1 Attacks and their Countermeasures

We now summarize attacks against TruGW. Many of them are directly related to the requirements defined in Section 3.5.1.

Adversary Types Following our defined threat model (cf. Section 3.4.1), TruGW's main focus is on system-level attackers (Sys_{gw}) which gained full control over U_P via a remote service and system exploit. Furthermore, we assume malicious network clients located in internal (*int.*) or external (*ext.*) networks with the goal of bypassing access restrictions. Beyond our threat model, we assume that adversaries might control a web page visited by an admin (*web*). Finally, while we regard admins and their systems as trusted, we also discuss the implications of a system-level attacker on the systems of the admins (Sys_{adm}) or master (Sys_{mst}). Based on these attacker roles, we now discuss how TruGW protects against 14 security-critical attacks shown in Table 3.4 (page 73).

A01: Image/Binary Tampering (*SR2*) The integrity of TruGW's T_P images (e.g., NetTrug) and device tree are guaranteed by secure boot (cf. Section 3.7.2). Tampering with ConfigService's binary is prevented as OP-TEE verifies TA binaries on load and prevents rollbacks.

A02: Code/Data Tampering (*SR2*) Sys_{gw} cannot tamper with TruGW's T_P components using memory writes or direct memory access. From boot on, TruGW protects T_P memory and NICs from Sys_{gw} using TrustZone and securely allocates all data in T_P (cf. Section 3.7.2).

Table 3.4: Overview of TruGW's defense measures against security-critical attacks by the adversaries defined in Section 3.8.1.

Target / Goal	**Attack**	**Adversaries**	**TruGW's Defense Mechanisms**	**Secure?**
Component Integrity	A01: Image/Binary Tampering	Sys_{gw}	secure boot + signed TAs	✓
	A02: Code/Data Tampering	Sys_{gw}	TZ memory/NIC protection via NetTrug (+ T_{drv})	✓
Policy Enforcement	A03: Policy Enforcement Bypass	Sys_{gw}, *int.*, *ext.*	NetTrug's NIC I/O and policies + firewall's perimeter location	✓
Address Spoofing	A04: Direct MAC/IP Spoofing	Sys_{gw}, *int.*	NetTrug's filtering (+ port pinning + subnetwork isolation)	✓
	A05: ARP Poisoning/Spoofing	Sys_{gw}, *int.*	NetTrug's trusted ARP handling (+ cf. A04)	✓
Traffic Protection	A06: Traffic Tampering/Sniffing	Sys_{gw}	T_{drv} + NetTrug's NIC I/O + restrictive routing	✓
Trusted Policy Configuration	A07: Policy Tampering	Sys_{gw}	TZASC + NetTrug + secure storage	✓
	A08: Policy Change via Auth. Bypass	Sys_{gw}, *int.*(, *ext.*)	ConfigService's enrollment and certificate management	✓
	A09: Config Connection Tampering	Sys_{gw}, *int.*	ConfigService's protected TLS endpoint	✓
	A10: ConfigService Spoofing	Sys_{gw}, *int.*, *web*	C_{cnf} pinning + trusted domain or IP vetting	✓
	A11: CSRF against ConfigService	*web* (Sys_{gw})	custom request header + trusted domain or IP	✓
	A12: ConfigService File Tampering	Sys_{gw}	SRI + hashing (+ signed TAs)	✓
Admins / Master	A13: Admin/Master Compromise	Sys_{adm}, Sys_{mst}	TPM + TEE browser + secure user I/O	(✓)
Leakage	A14: Covert Channel (Hdrs,Time)	*int.*, *ext.* (Sys_{gw})	header filters + traffic tunnels + time masking	(✓)

A03: Policy Enforcement Bypass (*SR1/3*) Sys_{gw} cannot bypass NetTrug's trusted policies, because NetTrug has full control over the NIC I/O paths from TEE boot (cf. A02) and can therefore guarantee their enforcement. *int.* and *ext.* attackers cannot bypass TruGW's policies due to TruGW's deployment at the perimeter.

A04: Direct MAC/IP Spoofing (*SR5*) TruGW prevents Sys_{gw} from sending traffic with spoofed source MAC or IP address by replacing the source MAC with the MAC of the resp. output NIC and by dropping U_P packets with spoofed source IP on VNIC (cf. Section 3.6.2). To defend against *int.* adversaries, TruGW can securely enforce port-based MAC pinning schemes and subnetwork isolation in T_P.

A05: ARP Poisoning/Spoofing (*SR5*) TruGW performs ARP request and response handling securely in NetTrug. To prevent ARP poisoning and spoofing by Sys_{gw}, NetTrug isolates U_P ARP messages by directly replying to U_P ARP requests and not forwarding U_P ARP replies (cf. Section 3.7.4). For *int.* attackers, NetTrug can securely enforce static routes or other common schemes (cf. A04).

A06: Traffic Tampering/Sniffing (*SR4*) Sys_{gw} can neither read nor manipulate any forward traffic or any network packet stored in TruGW's trusted I/O buffers. NetTrug and its secure NIC drivers (T_{drv}) protect the NIC I/O paths (incl. I/O rings) in T_P (cf. Section 3.6.1). In addition, NetTrug routes only U_P-destined traffic to U_P (cf. Section 3.6.2).

A07: Policy Tampering (*SR6*) Sys_{gw} cannot directly tamper with trusted policies in memory or on disk. NetTrug isolates the policies in T_P memory and allows changes only by ConfigService. Disk backups are protected via OP-TEE's secure storage API.

A08: Policy Change via Auth. Bypass (*SR6*) Sys_{gw} and *int.* cannot modify trusted policies (or IPs) via ConfigService, because only master and admins have access. In addition, the initial master enrollment is secure, because the gateway (incl. U_P) is in a secure boot state and the master has exclusive device access (cf. Section 3.7.5). Afterwards, master grants only trusted admins access to ConfigService and blocks any malicious requests by Sys_{gw} or *int.* TruGW restricts access to ConfigService to internal clients, which blocks *ext.*

A09 Tampering with Config Session (*SR6*) Sys_{gw} and *int.* cannot tamper with connections between trusted admins and ConfigService, because they are TLS-protected and end in T_P.

A10: ConfigService Spoofing (*SR6*) Neither Sys_{gw}, nor *int.*, nor *web* can impersonate ConfigService, because admins securely pin its server certificate (C_{cnf}) for the trusted web address (cf. Section 3.7.5). Furthermore, admins distribute U_P service certificates only for untrusted addresses (cf. Section 3.7.6).

A11: CSRF against ConfigService (*SR6*) TruGW prevents *web* attackers from launching CSRF attacks against admins of ConfigService by requiring custom HTTP request headers [23] (cf. Section 3.7.5) which are only settable from the same web origin. As ConfigService has a trusted web domain (or IP) and thus different origin than U_P services (cf. Section 3.6.3), Sys_{gw} cannot launch CSRF either.

A12: ConfigService Resource Tampering (*SR6*) Sys_{gw} cannot tamper with ConfigService's root HTML or U_P-hosted web resources, because ConfigService uses secure hashing and subresource integrity (SRI) to check their integrity on load (cf. Section 3.7.5).

A13: Admin/Master Compromise (*SR6*) While we assume the master, admins, and their systems as trusted (cf. Section 3.4.1), we now discuss the implications of a full compromise of their systems. Sys_{adm} cannot steal the admin private key K_{adm}^{-1} if it has been securely generated and stored in a TPM. However, Sys_{adm} can use the admin credentials to maliciously reconfigure TruGW's trusted policies via ConfigService. If such a breach is detected, the master must immediately revoke C_{adm}. To prevent such an attack, TruGW can deploy orthogonal solutions on the admin-side, which establish a secure I/O channel between the admin and a TEE-protected browser [68, 117], and then enforce the use of the trusted browser for ConfigService access, e.g., via SENG's per-application policies (cf. Section 2.7.3). That way, Sys_{adm} can neither steal K_{adm}^{-1} nor use it. The situation is similar for Sys_{mst}, however, on a master key breach K_{mst}^{-1}, a full enrollment reset is required (cf. Section 3.7.5). On re-enrollment, the master requires a clean system to prevent hijacking attempts by Sys_{mst}.

A14: Covert Channels (Headers, Timing) TruGW's current focus is *not* on an active prevention of covert channels. However, TruGW could adopt existing techniques to contain or prevent covert channels. For instance, TruGW could heavily filter all packet headers (incl. U_P's) to remove storage channels [234], deploy client-side solutions for protected traffic tunnels to TruGW to entirely strip untrusted headers by *int.* (as presented by SENG in A14 of Section 2.9), or adopt time masking schemes to prevent timing channels by *int.* and *ext.* [35]. As TruGW currently relies on U_P for scheduling (cf. Section 3.7.2), Sys_{gw} controls the scheduler and can exploit it for additional timing channels. TruGW could prevent them by switching to a T_P-controlled scheduler [90].

3.8.2 Real-World Vulnerabilities

Recent CVEs in network gateways have raised serious security concerns and motivated our design of TruGW. We now asses how TruGW addresses these real-world vulnerabilities. As discussed in Section 3.4, critical CVEs of network gateways mainly lurk in auxiliary services, e.g., SNMP or web interfaces, and system components unrelated to core network functionalities (cf. Table 3.2 on page 54, and Table 3.3 on page 56). They enable remote attackers full control over the system, i.e., attackers effectively gain the privileges of Sys_{gw}, e.g., via a remote code execution. By design, TruGW contains exactly these types of services and system components in U_P and securely isolates the

Table 3.5: Code sizes of auxiliary network services on DD-WRT routers (rev. 47201) in thousands of source lines of code (kSLOC).

		Lines of Code [kSLOC]		
Service	**Short Description**	**Total**	C/ASM	Hdrs
asterisk	VoIP server	766.6	673.7	92.9
dropbear	sys utils (incl. sshd)	95.3	87.5	7.8
freeradius3	authentication service	116.1	109.6	6.5
krb5	authentication service	308.3	256.9	51.4
lighttpd	web server	82.9	71.5	11.4
minidlna	streaming server	691.0	553.3	137.7
nginx	web server	140.8	132.3	8.5
proftpd	FTP daemon	227.7	220.9	6.8
samba4	file sharing server	1515.3	1431.8	83.5
snmp	sys/net monitoring	288.2	257.9	30.3
squid	web proxy	51.3	15.0	36.3
zabbix	sys/net monitoring	233.5	221.1	12.4
SUM		4517.0	4031.5	485.5

core network functionalities (incl. firewall) in T_P against Sys_{gw}. Therefore, TruGW successfully protects gateways against recent attacks.

TruGW's security can only be undermined if vulnerabilities lurk in the remaining attack surface within T_P services themselves. However, TruGW only includes core network services in T_P (e.g., firewall, NIC drivers), which have faced very few CVEs, especially compared to commodity OSes (cf. Section 3.4). Furthermore, our current TruGW T_P^k prototype (cf. Section 3.9.1) only has ≈110 kSLOC, whereas 12 popular auxiliary services of DD-WRT routers already include an attack surface which is one order of magnitude larger (≈4517 kSLOC, as shown in Table 3.5). Commodity U_P OSes are even larger and have faced 2-3 orders of magnitudes more CVEs than OP-TEE OS, which faced only ≈10 CVEs (cf. Table 3.2, page 54). Therefore, TruGW drastically decreases the TCB size of commodity routers and thus risk of critical vulnerabilities.

3.9 Evaluation

We now describe our prototype implementation and evaluate it in terms of code size (TCB), performance, and memory overhead.

3.9.1 Open-source Prototype

We implemented an open-source TruGW prototype (cf. Section 3.10) on a Nitrogen6X development board [30] with an i.MX6Q Arm CPU (32bit, 4 cores), 2 GB RAM, and the Gbps Freescale Fast Ethernet Controller (FEC) as our secure NIC. FEC is known to be technically limited to ≈470 Mbps maximum (cf. errata 004512), and indeed showed only ≈400 Mbps for incoming/outgoing traffic in a vanilla setting in our experiments.

However, we nevertheless chose this board due to its Ethernet support and as its TrustZone support was well documented and successfully used in research projects by others [135]. We run Debian 10 with a 4.14 Linux kernel[4] as untrusted U_P OS and OP-TEE 3.8.0 [145] as the secure T_P OS. We use U-Boot 2018.07 as the (trusted) bootloader.[4]

To implement NetTrug's ARP, routing, and NPF integration, we ported NPF-Router [187] to OP-TEE and significantly extended it with trusted workers (incl. notifier), device driver callbacks, packet buffers and queues, and VNIC support. NetTrug's worker registration interface (cf. Section 3.7.2) is exposed to U_P via OP-TEE's TA client API. We have implemented VNIC mostly from scratch, but use the vqueue implementation of Trusty OS [90] for the I/O rings. T_{drv} follows the Linux FEC driver and registers separate Rx and Tx workers on NetTrug for increased performance. We integrated T_{drv} into NetTrug's driver framework and enabled interrupt sharing with U_{drv} (cf. Section 3.7.3.1). For trapping and decoding U_P NIC and VNIC access faults, we have extended and integrated parts of SeCloak [135] into OP-TEE and NetTrug.

ConfigService is implemented as an OP-TEE trusted application (TA). We use the tiny picohttpparser [167] for HTTP parsing and a small subset of mbedTLS for TLS. The untrusted SockHelper is a small C program which handles the TCP server sockets and calls into ConfigService via OP-TEE's TA API. SockHelper exchanges TLS records with ConfigService using a new ringbuffer based on OP-TEE's shared memory API. ConfigService's web application is a simple web page which communicates via GET and POST XMLHttpRequests with ConfigService and uses our WebAssembly (Wasm) port of NPF's client tool for the firewall policy parsing (cf. Section 3.7.5).

3.9.2 Code Size Analysis

We now analyze to what extent TruGW's components increase the TCB size compared to vanilla OP-TEE. We measured the source lines of code (SLOC) of OP-TEE's and TruGW's trusted kernel components using cloc [52]. As OP-TEE and NPF choose files depending on the platform configuration, we only counted the actually included source/header files. OP-TEE's core has ≈52 kSLOC plus ≈24.4 kSLOC for its crypto library LibTomCrypt [213]. TruGW adds ≈8.4 kSLOC for NetTrug, VNIC, T_{drv}, and device access trapping combined plus ≈25.1 kSLOC for the NPF firewall with all its libraries. That means, TruGW's core increases OP-TEE's core only by ≈16 % and the addition of NPF is roughly on a par with OP-TEE's crypto library (*SR7*). Moreover, NPF makes up ≈75 % of the current TruGW code base, and there is a substantial shrinking potential as NPF's design is not tailored to TrustZone. ConfigService adds only ≈2.1 kSLOC to T_P^u, plus a subset of mbedTLS. Altogether, TruGW has a reasonably small impact on OP-TEE's TCB size and significantly decreases the overall attack surface compared to commodity U_P OSes and services (cf. Section 3.8.2).

3.9.3 Performance Evaluation

We now report on the network performance of TruGW. We evaluate (i) the network throughput of TruGW, (ii) the overhead of its firewall, (iii) TruGW's impact on network

[4] we use the imx6 forks provided by the board vendor (Boundary Devices)

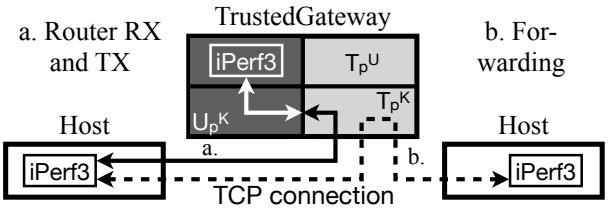

Figure 3.4: Throughput evaluation setup. a. Throughput between an untrusted gateway router service and an internal host (both directions). b. Forwarding throughput.

latency, and (iv) the page load time of ConfigService. To this end, we interconnected TruGW with two client hosts using a 5-port gigabit switch. The first host ($Host_M$) is a Macbook Pro (Mac) with an Apple Thunderbolt-to-Gigabit Ethernet Adapter. The second host ($Host_L$) is an HP Z1 workstation with an Intel I219-LM NIC running Ubuntu. Each host is in a separate IP subnetwork and configured to use TruGW as its default gateway, s.t. all traffic is forwarded through TruGW.

3.9.3.1 Network Throughput

We use iPerf3 [111] to evaluate the TCP network throughput of TruGW in three ways: (i) the downlink of an untrusted router service ("Router RX", e.g., file upload to a local file server on the gateway) and (ii) uplink throughput of an untrusted router service ("Router TX", e.g., file download from the gateway's file server), and (iii) the client throughput when routing all traffic through TruGW ("Forwarding"). iPerf3 sends TCP traffic via a single connection to another iPerf3 instance and measures the resulting throughput performance over 10 s. Figure 3.4 illustrates our test cases and corresponding network flows. For (i) and (ii), iPerf3 runs on one client host and as an untrusted router service in TruGW's U_P. TruGW serves as the (i) receiver and (ii) sender respectively. For (iii), we run iPerf3 on both client hosts and consider both sending directions. We compare TruGW to the plain Linux setup of the Nitrogen6X board without TrustZone as the baseline ("vanilla"). We disable NIC offloading features as our current driver implementation does not yet support them, and map device registers uncached due to OP-TEE's limited mapping support [135]. For both setups, we perform 20 iterations for each test case.

Figure 3.5 shows the performance results for all six tests. (i) TruGW reaches a receive throughput of about 385 Mbps, which is about 90 % of the vanilla throughput. The observed overhead is likely caused by the current implementation of VNIC's U_P interface (virtio-mmio). VNIC currently uses (legacy) interrupts for buffer notifications to U_P (and access traps in the opposite direction) which can frequently interrupt the U_P iPerf3 thread. We could further improve the results by refining VNIC's batch processing, but we regard the performance hit as acceptable for rare bulk uploads to router services. (ii) TruGW reaches a transmission throughput of 392 Mbps for $Host_M$, which is about 6.5 % higher than that of vanilla (368 Mbps avg.). For $Host_L$, TruGW reaches 99 % of vanilla's average throughput (369 Mbps). While TruGW's throughput to $Host_M$ is currently higher than vanilla, we have observed comparable maximum throughput values for $Host_L$ and vanilla, too. VNIC currently performs aggressive

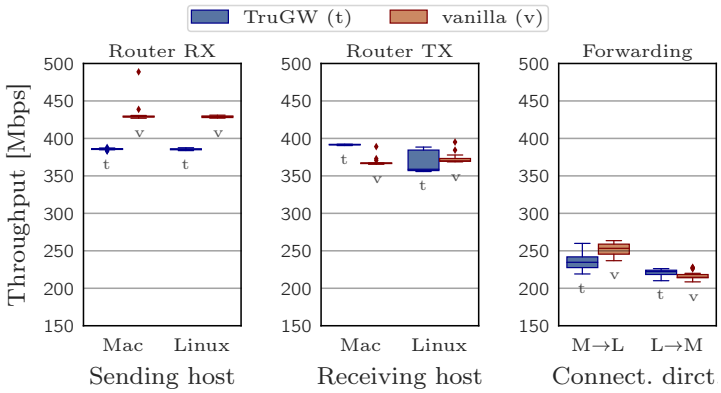

Figure 3.5: iPerf3 TCP throughput when the TruGW gateway router is used as a receiver (left), sender (middle), and forwarder (right); each for two clients (Mac/Linux).

packet forwarding retries on a NIC congestion, which seem to benefit from $Host_M$'s ACK sending behavior. (iii) Lastly, TruGW's forwarding performance reaches 92.6 % to 93.8 % (236 Mbps avg.) of the vanilla throughput when $Host_M$ is the sender and 101.9 % to 103.5 % (221 Mbps avg.) when $Host_M$ is the receiver. In summary, TruGW shows an overall high throughput ≥90 % (*AR4*) and performs similar to the vanilla system (92.6 % to 103.5 %) when forwarding.

3.9.3.2 Firewall Overhead

We now measure if adding NPF firewall rules causes overhead. We repeated the three iPerf3 measurements with $Host_M$ while applying filter rules, i.e., the "Mac Router RX/TX" and the "$M \rightarrow L$" benchmarks of Figure 3.5. For RX/TX, we defined rules on the VNIC interface, which check for TCP connections to TruGW's IP on iPerf3's port. For forwarding, we defined analogous rules on the NIC interface to match iPerf3's connections to $Host_L$. We performed 20 iterations of each test with (a) stateful rules (connection tracking) and (b) bidirectional, stateless rules.

We observed a small overhead of about 0.5 % to 1 % in each test. This is not surprising, because NPF enforces rules using just-in-time compiled BPF code and has a fast path for connection tracking, which enables efficient allowlisting policies. While the overhead will naturally increase with large rulesets, the observed overhead comes from NPF's static code. As long as stateful policies capture most of the traffic—which is the norm for most networks—the overhead is thus marginal.

3.9.3.3 Latency Overhead

We now evaluate how TruGW affects latency during web browsing and on a per-packet basis.

Web browsing To follow a typical user scenario, we measure the client-side load times of web pages. We selected the ten stable pages from the top 13 of the Tranco list [132] for the evaluation. We excluded "tmall.com" and "qq.com" as they blocked the page

Table 3.6: Overview of Chrome page load times and overhead when routing through TruGW as an intermediate router.

	avg. load [ms]		
Web Page	**Baseline**	**TruGW**	**Overhead**
instagram.com	1298.5	1362.8	4.95%
linkedin.com	654.0	685.8	4.86%
google.com	563.0	590.1	4.81%
youtube.com	560.8	587.2	4.71%
microsoft.com	823.1	856.3	4.03%
baidu.com	6642.9	6895.1	3.80%
facebook.com	813.2	843.6	3.74%
apple.com	963.8	993.0	3.03%
wikipedia.org	701.5	704.5	0.43%
twitter.com	1125.7	1126.5	0.07%

load or faced a high baseline variance (multiple seconds) and "windowsupdate.com" as Chrome refused to load it. For each page, we measured the average load times over 10 iterations from $Host_M$ using a Chrome extension [224]. We kept all DNS entries cached, but cleared the web caches after each page load. We compare the baseline without TruGW (using a home router as $Host_M$'s direct gateway to a ≈60 Mbps line) to a setup with TruGW as an additional intermediate router between them.

TruGW incurs an average load time overhead of ≈3.4 % reaching from 0.07 % to 4.95 % peak, as shown in Table 3.6. The latency is low when most packets arrive while TruGW's I/O workers are still polling, and is slightly higher when TruGW's notifier must wake them up (cf. Section 3.7.2). The workers partially compensate this by having an idle grace period before entering the sleep state. We regard the observed average overhead as reasonably small. Most of the overheads translate to page load delays of about 30 ms, which is not noticeable by average users.

Packet Latency To gauge how latency-critical applications (e.g., gaming) are affected by TruGW, we also evaluate the per-packet latency using `ping`. We measure the average round trip time (RTT) from $Host_M$ to an external server for 1000 packets over 10 iterations. We use the same baseline as in the page load test. TruGW shows an average RTT of 14.22 ms, which is a tiny per-packet slowdown of ≈0.37 ms (≈2.67 %) compared to the average baseline of 13.85 ms.

3.9.3.4 Trusted ConfigService Load

We now briefly report on the page load time of ConfigService's master admin page. We follow the approach of the previous section (Section 3.9.3.3). The load time includes the server and client TLS authentication and the fetching of all ConfigService web resources. We have observed an average load time of about 1385 ms, which fulfills current user expectations of 1 s to 2 s [230]. We can further optimize ConfigService if required.

3.9.4 Secure Memory Overhead

Since routers are usually resource-constrained devices, we now discuss the secure memory overhead of TruGW. TruGW currently shares OP-TEE's default configuration and claims 30 MB of the system RAM exclusively for the trusted partition T_P and 2 MB for shared memory. VNIC additionally claims 268 B for its virtual device registers [165], but its network buffers are allocated in untrusted memory instead (cf. Section 3.6.2). These memory requirements are easily met by commodity router platforms. For instance, OpenWrt [33] recommends $\geq$ 128 MB of RAM for routers, which is fulfilled by the majority of its supported Arm devices.[5] In addition, the TruGW prototype currently leaves ≈20 MB of the 32 MB for trusted user apps, such that we could further reduce TruGW's secure memory requirements, likely to ≈16 MB. The exact secure memory demand depends on the number of NICs and their I/O buffer sizes. For instance, the FEC NIC uses two rings à 512 entries for Ethernet frames (≈1.5 MB). However, TruGW relocates NIC I/O buffers, egress queues, and firewall states from U_P to T_P, i.e., they do not increase the overall system RAM demands.

ConfigService has a small memory footprint inside the TA memory. The demands are defined by the per-client TLS ringbuffers (≈4 kB), HTTP buffers (≈4 kB), and internal TLS buffers (≈3.6 kB to 32 kB). The TLS admin certificates and the HTML file are securely stored on disk. Other web resources are offloaded to U_P (cf. Section 3.7.5).

3.10 Artifacts

The prototypes of TrustedGateway are available as open-source projects at `https://github.com/trugw/` [S2]. See page 9 for a list of all open-source prototypes covered by this dissertation.

3.11 Conclusion

The increasing attack surface introduced by commodity gateway OSes and auxiliary services enables remote attackers to easily compromise gateway routers and bypass their security-critical network policies. Unfortunately, network infrastructures still widely rely on the assumption that gateways are trusted ("bastion hosts"). Existing ad-hoc protection attempts result in large attack surfaces or are not suitable for the protection of standalone consumer and SME gateways. TruGW bridges this important gap by leveraging a system-level TEE to guarantee trusted policy enforcement with a small attack surface even on a fully compromised gateway—answering our second research question *RQ2* (see page 4). TruGW's design builds on the widely-available Arm TrustZone TEE and combines it with well-supported software features (e.g., virtio) to enable an affordable and readily deployable secure gateway architecture. TruGW thus restores the trust in the security of gateway routers.

Together with SENG (see Chapter 2), TruGW increases the security of consumer and enterprise networks by enforcing strong, fine-grained firewall and routing policies. Even if a local client or gateway device is compromised by a system-level attacker,

[5] list of respective devices: `https://openwrt.org/toh/views/toh_available_16128`

SENG and TruGW can guarantee secure traffic attribution, forwarding, and policy enforcement. However, both approaches have limited control on the way benign users interact with external services, e.g., web services. In particular, they cannot protect the user authentication towards such services, e.g., against password theft via phishing attacks or a server-side compromise. Therefore, in the next chapter, we will explore if TEEs can help provide users with secure web authentication.

4

FelDo: Recoverable FIDO2 Tokens Using Electronic IDs

Solving Token Loss and User Data Privacy via TEE-protected Attribute-based Credentials

4.1 Motivation

Two-factor authentication (2FA) mitigates the security risks of passwords as sole authentication factor. FIDO2—the *de facto* standard for interoperable web authentication—leverages strong, hardware-backed second factors. However, practical challenges hinder wider FIDO2 user adoption for 2FA tokens, such as the extra costs ($20-$30 per token) or the risk of inaccessible accounts upon token loss/theft.

To foster wider adoption of secure web authentication, in this chapter, we leverage TEEs to overcome the above deployment and account recovery challenges of FIDO2 web authentication, thus answering research question *RQ3* (see page 5). We propose FeIDo, a *virtual* FIDO2 token that combines the security and interoperability of FIDO2 2FA authentication with the prevalence of existing eIDs (e.g., electronic passports) and the attestable confidentiality of TEEs. Our core idea is to derive FIDO2 credentials based on personally-identifying and verifiable attributes—name, date of birth, and place of birth—that we obtain from the user's eID. As these attributes do not change even for refreshed eID documents, the credentials "survive" token loss. Even though FeIDo operates on privacy-critical data, all personal data and resulting FIDO2 credentials stay unlinkable, are never leaked to third parties, and are securely managed in TEEs, i.e., remotely attestable hardware containers (e.g., SGX enclaves). In contrast to existing FIDO2 tokens, FeIDo can also derive and share verifiable meta attributes (*anonymous credentials*) with web services. These enable verified but pseudonymous user checks, e.g., for age verification (e.g., "is adult").

4.2 Problem Description

Passwords still represent the most popular type of credentials in web authentication—despite their widely-studied deficiencies [84], such as the risks of password database breaches [191], shoulder surfing [65], phishing [166], or low entropy passwords [54]. To mitigate these issues, a growing number of web services offer *two-factor authentication* (2FA) to increase authentication security. 2FA schemes usually ask the user for some proof of possession, such as one-time passwords (OTP) sent to the user's mobile phone. However, common second factors, e.g., SMS OTP [134] or OTP apps, are vulnerable to client-side and server-side leaks of the OTPs and their secret seeds. Furthermore, they require interceptable user input for entering the OTPs, and users must manage them for each service and client device. To enable stronger 2FA, the FIDO2 standard defines how to use protected hardware tokens (also called *authenticators*), especially for web authentication [150]. FIDO2 follows a challenge-response protocol where for each origin (e.g., web service), the hardware token securely generates and stores a public key pair. Upon authentication, the FIDO2 client first verifies the server origin, after which the token digitally signs a server-chosen challenge using its private key, which is never accessible outside the token.

Given the above security benefits, FIDO2 has become the *de facto* standard for strong, interoperable authentication supported by many popular services. However, while FIDO2 *support* is increasing, the actual *user adoption*—i.e., users leveraging FIDO2-compatible hardware tokens—lags behind. Two fundamental downsides of hard-

ware tokens hinder a wider adoption. **(1) Costs:** Users are reluctant to buy dedicated hardware tokens as they incur extra costs. Even basic FIDO2 tokens cost around \$20-\$30, which is a non-negligible investment. **(2) Token loss:** Hardware tokens are subject to loss or theft. Users might no longer be able to log in, as token-based credentials cannot be backed up. Indeed, token vendors recommend registering at least two different tokens to the same user account [240]—further increasing the costs and hampering the usability of hardware tokens overall. Alternatively, users must fall back to less secure alternative authentication or account recovery schemes (e.g., recovery codes) if provided by the web service [238].

The barrier of additional costs has motivated vendors to offer FIDO2-compatible "virtual" tokens. These tokens do not require extra hardware but root their signing security in trusted hardware of client devices. For instance, Android and Windows 10 have FIDO2-certified authenticators: Android's *Keystore* backed by Arm TrustZone [92], or the *Windows Hello* authentication service relying on the Trusted Platform Module (TPM). On the one hand, this allows virtual tokens to securely store authentication-relevant secrets in trusted hardware so attackers cannot steal them. On the other, this advantage comes at the cost that users can still not back up their authentication secrets. Furthermore, not all user devices offer the trusted hardware (e.g., TPM) required by the virtual FIDO2 tokens.

4.3 Contributions

In this chapter, we propose *FeIDo*, a fully FIDO2-compliant virtual token that tackles the challenges of costs *and* token loss. FeIDo is mainly designed for—but *not* limited to—providing private users with a strong second factor for two-factor web authentication. FeIDo is a virtual FIDO2 token utilizing electronic identifications (eIDs) such as electronic passports or ID cards. FeIDo uses the communication interface of eIDs as standardized by the International Civil Aviation Organization (ICAO) to extract personal data from the documents. eIDs can prove the authenticity of personal attributes, such as the user's name, place of birth, and date of birth. These personal attributes then form the basis for FeIDo's user authentication. eIDs nicely address the above authentication challenges in FIDO2: **(1) No extra costs:** eIDs obsolete the need for dedicated security tokens for eID holders. Over 1 billion citizens already own electronic IDs or passports [162, 214] (cf. Section 4.4.2). Our setup thus does not impose additional hardware requirements—users can leverage eID readers such as NFC readers that ship with off-the-shelf phones (cf. Section 4.4.2). **(2) Token recovery:** The authentication in FeIDo is not bound to a particular eID but only to its verifiable personal attributes. These attributes do not change even if a lost/stolen eID is replaced by a new one, which enables direct credential and thus account recovery.

While personal attributes could be directly shared with and verified by authenticating services, most attributes stored on eIDs are privacy-sensitive (e.g., name, place of living) and *not* required for authentication. There are many cases in which users wish to remain pseudonymous, such as in adult websites, forums (e.g., health forums), learning platforms for kids, or even social media. Therefore, we design a FIDO2-compatible attribute-based authentication scheme in which third parties do *not* learn personal de-

tails and credentials are unlinkable. Our core idea is to use trusted remote *credential services* that validate and vet—but never share or leak—the personal data. The credential services feed the personal user data as input to a key derivation function to derive *attribute-based FIDO2 credentials*. The resulting credentials depend on personal attributes (name, date of birth, and place of birth) and a secret chosen by the credential services but are unlinkable. To guarantee that the credential services can be trusted to protect personal data against third parties (including the hosting providers), they execute in attestable Trusted Execution Environments (TEEs). Users can remotely attest the protection and authenticity of a given credential service, verifying its validity before sharing their personal data.

In addition, our design comes with an attractive extension that is not in the scope of existing FIDO2 tokens. FeIDo enables *anonymous credentials* that allow web service providers to learn pseudonymous meta user attributes and verify their authenticity. For example, adult websites may have to ensure that their users are of legal age, or governmental websites may want to restrict services to residents. FeIDo's credential service derives such meta attributes (e.g., "is adult") from the raw eID data (e.g., *dob* = May 14, 1981) in an attestable way and, at the same time, guarantees that the raw user data is never exposed outside of the credential service's TEE.

Our design on the user side is agnostic to the choice of a concrete OS, hardware, and eID. Users can use standard interfaces such as NFC to read personal data from their eID and prove its authenticity to a credential service (cf. Section 4.4.2). The credential service requires a TEE providing remote attestation, data encryption, and integrity checks, e.g., as available on public cloud platforms—and one instance can easily handle tens of thousands of users. FeIDo clients perform remote attestation before forwarding data to the credential service—notably *without* requiring TEE support themselves. Clients also do not need to back up their credentials to withstand device loss, as credentials are always freshly derived from eID data. FIDO2-capable web services can readily use FeIDo-backed credentials, and can leverage anonymous credentials using FIDO2 extension fields. In our evaluation, we instantiated this general design in a concrete setting without losing generality. We securely implement an open-source prototype (cf. Section 4.10) [S3] that consists of an Android app reading personal data from an ePassport and a TEE-protected credential service receiving and vetting this personal data to derive signing keys for authentication. We evaluate our prototype by measuring the FIDO2 authentication duration on the well-known *webauthn.io* test page [64]. We show that FeIDo is comparable in efficiency to existing FIDO2 hardware tokens while tackling their shortcomings.

In summary, we make the following contributions:

- We design FeIDo, a virtual FIDO2 token that enables account recovery on a token loss and is readily available without extra costs to the large population of users owning a compliant eID such as ICAO-standardized ePassports.
- As the crucial enabler for account recovery, we present the concept of *attribute-based* FIDO2 credentials, protecting personal data within an attestable TEE.
- FeIDo enables anonymous credentials (e.g., age group) that web providers can

verify without having access to privacy-infringing raw user attributes (e.g., date of birth).

- We analyze the security and prototype [S3] of FeIDo.

4.4 Background and Related Work

We first provide the reader with background information on FIDO2 and electronic IDs and describe their current shortcomings.

4.4.1 FIDO2

The FIDO2 standard (and its predecessor U2F) describes how to use hardware tokens for authentication. Such tokens rely on public-key cryptography (PKC). To form authentication credentials, the client creates a dedicated key pair per service as credentials with the help of a hardware token ("*authenticator*"). The hardware token generates and stores these key pairs. To register their credentials with an account, users send their service-specific public key to the authenticating server ("*relying party*"). The corresponding private key never leaves the hardware token, even if the user's system ("*agent*") is compromised. To authenticate a user, client and server follow a challenge-response protocol, as shown in Figure 4.1 for a web authentication setting. The client signs a server-chosen challenge using their authenticator (token). To this end, the agent verifies the origin (i.e., domain and port number) of the relying party (web service). The authenticator then uses the origin-specific private key to sign the challenge. The relying party uses the public key associated with the authenticating account to verify the signed client response. In essence, possession of the private key serves as proof for authentication.

This authentication workflow is realized in two core FIDO2 components, *WebAuthn* and *CTAP2*. WebAuthn [150] standardizes the interface of PKC-based authentication on the web. Basically, WebAuthn defines the communication between the client device and the web server, including the challenge-response protocol mentioned above. The standard defines an API for the agents that defines how to generate fresh credentials (*MakeCredential*) and get the response to the relying party's challenge (*GetAssertion*). The Client to Authenticator Protocol (CTAP2) complements WebAuthn. CTAP2 defines the communication between the authenticator and the agent.

The FIDO2 standard supports using authenticators in several private and enterprise use cases for secure authentication [72]. In this work, we set the prime focus on FIDO2 authenticators as secure second factors for web authentication by private users. However, we will later explain how our proposed scheme can be used as a single factor (Section 4.9.1) or in enterprise-focused use cases (Section 4.9.2).

4.4.1.1 Open Challenges for FIDO2

FIDO2 mitigates password cracking via database breaches, phishing attacks, and insecure storage of credentials. FIDO2 also makes users unlinkable across services, as the

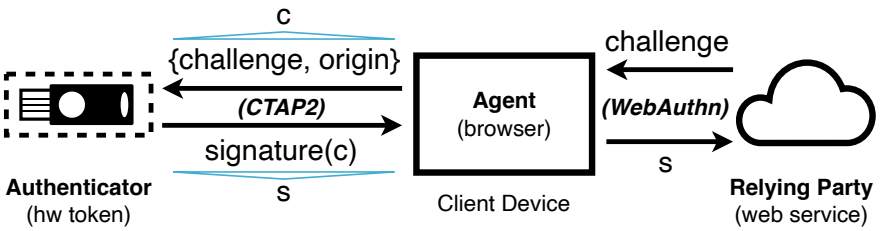

Figure 4.1: Simplified flow of a FIDO2 web authentication.

authenticator generates a fresh public key for every relying party. However, FIDO2 introduces a new set of problems.

Costs The extra costs for dedicated hardware are one of the main obstacles to scaling FIDO2 to the masses. Even the most basic hardware-based authenticators cost around 20-30 USD, a non-negligible expense for many. To tackle this challenge, vendors offer FIDO2-compliant "virtual" authenticators that use trusted computing features (e.g., TPM or secure enclaves) of the user's hardware. Google introduced an API to Android OS that allows the user to store FIDO2 keys inside the Arm TrustZone-backed Android Keystore [92]. To use this virtual FIDO2 authenticator, the user relies on the authentication mechanism used to access the phone, i.e., a PIN code or biometric features. The Windows Hello authentication service provides a similar solution based on TPMs. FIDO2 is also available for macOS and iOS using Apple's secure enclave for key management and FaceID and TouchID as user access control [28]. Chakraborty and Bugiel proposed an authenticator called *simFIDO* [37] based on simTPM [38]. They use a cheap SIM card running the Java Card OS as TPM to implement a FIDO2-compliant authenticator, mitigating the cost issue. On the downside, this solution requires mobile service providers to support and install extra applications (simFIDO and simTPM) on their SIM cards. While all these approaches mitigate the cost issue, they require special trusted computing features on the client side, which are not present in many consumer devices. Furthermore, unless users choose alternative authentication methods, they will lose access to their credentials if the device gets lost or stolen (discussed next).

Recently, support for roaming FIDO2 authenticators has been integrated into all major browsers, which enables users to use a phone as a virtual FIDO2 second factor for other devices [74]. While this scheme decreases the need for trusted hardware on *some* user devices, this scheme still faces many disadvantages (incl. credential loss) compared to our new scheme, as we will discuss in Section 4.6.2.1.

Token Loss Another open challenge of FIDO2 is to provide a cost-efficient and practical solution for account recovery in case the authenticator is lost. In principle, as the private credential keys are protected on the authenticator and can not be extracted, losing an authenticator implies losing access to all authenticator-protected accounts. The naive yet recommended solution [239, 238] is registering an additional authenticator to the same account as a backup, doubling the costs. Alternatively, if supported by the relying party, additional authentication factors or service-specific recovery strategies, such as SMS OTP or recovery codes, can be used, which, however, degrade security

and face several drawbacks [134, 130].

Recently, some platform vendors have partially rolled out support for multi-device FIDO2 credentials, also called *passkeys* [74]. In Section 4.6.2.1, we will compare our new scheme to passkeys and show that while they approach the token loss challenge, passkeys are still suffering from many disadvantages and undesired tradeoffs.

4.4.2 eIDs for Authentication

This chapter proposes a FIDO2-compatible virtual authenticator that uses electronic IDs (eIDs). eIDs and their user-based associations are inherently well-suited for authentication. A combination of personal data such as the full name, day of birth, and place of birth is reasonably specific to be used in strong authentication mechanisms. For many eIDs, such personal data can be (i) securely read electronically, (ii) verified to stem from a valid eID, and (iii) accompanied by freshness guarantees that show if a user currently possesses the eID. eIDs typically include RFID chips with NFC interfaces, for which respective NFC support is widely available on consumer phones, including all iPhones released since 2016 (iOS $\geq$ 13.2) [205, 95] and most Android phones since '15, e.g., from LG and HTC, all Samsung phones since '15, and all Huawei phones since '17 [29, 95]. To prevent eID spoofing and ensure data authenticity, the personal data is signed by a trusted issuing authority, e.g., a state. Furthermore, eIDs usually physically protect memory for secret storage, prevent unauthorized data access, and detect eID cloning.

The eID interfaces are either standardized (in the case of electronic passports) or subject to the issuing country (in the case of national IDs). Having said this, an ever-increasing number of countries deploy national eIDs or ePassports that provide the above capabilities. For example, every eID following the ICAO standard [109] for electronic machine-readable travel documents (eMRTDs) satisfies these requirements. As of mid-2019, over 150 countries issued ePassports [214], including the top 10 most populated countries constituting more than half of the earth's population. Moreover, also many national IDs implement this standard. For example, starting from August 2021, all European Union member states are required to issue ICAO-compliant national ID cards [189]. Compliant IDs are also provided by Panama, Uruguay, Algeria, Saudi Arabia, Ukraine, Kyrgyzstan, and mainland China ID cards for Hong Kong, Macao, and Taiwan [209]. Several other national eIDs implement similar interfaces, often defined in regional or country-specific standards [162]. Taking into account the accessibility of ICAO-compliant eIDs, throughout this chapter, without losing generality, we assume the eID is an eMRTD. This way, we can explain the general concept by the example of concrete communication protocols.

An eMRTD interacts with a client through an NFC reader via well-defined protocols. Three of these protocols [108], which map to the eID capabilities mentioned before, are relevant in our context:

(i) PACE Password Authenticated Connection Establishment (PACE) is a password-based access protocol to protect the communication between an eMRTD and a reader. PACE assures that the reader is authorized to access certain data groups

on the eID [1, 27]. The chip asks the reader for a (static) password as access control, typically printed on the eID. The user provides this eID-specific secret to the reader. This password can optionally be cached in software to ease future accesses. The chip and reader derive a Diffie-Hellman session key from this weak password to grant/obtain read access and to establish a secure channel for subsequent data exchange.

(ii) PA During Passive Authentication (PA), the reader verifies the data authenticity of the retrieved personal data. Next to raw data, an eMRTD chip can also send a document security object (DSO), i.e., signed hash values of the personal data stored on the eID. The DSO is also signed and can be verified, ensuring that only a trusted eID authority can create valid DSOs. Readers can therefore validate the eMRTD data by comparing data hashes.

(iii) CA While PA ensures data authenticity, Chip Authentication (CA) prevents data and eID cloning. When initiating CA, the chip shares a static public key with the reader. By checking this public key against the previously hash-protected eID data received via PACE, one can verify the binding between the present eID and user data. In addition, a successful CA channel establishment proves that the eID is in possession of the *unclonable* CA private key and therefore authentic. After the reader sends an ephemeral public key to the chip, both the chip and the reader derive a shared key between the two parties from these keys—technically, a Diffie-Hellman key exchange. They later use this shared key to derive session keys (encryption key and MAC key) for secure communication.

4.4.2.1 Risks of eID-Based Authentication

The previous discussion showed that eIDs provide well-defined and easy-to-access interfaces to extract verifiable personal data securely. However, while exposing eID data to third parties for authentication is technically possible, it infringes a user's privacy as the personal data stored on eIDs is highly privacy-sensitive (e.g., date of birth, place of living). The data goes much beyond what is necessary for authentication purposes. Furthermore, there are many use cases in which users wish to remain pseudonymous (e.g., health forums), as described in Section 4.3. Therefore, we see the potential for an eID-based authentication scheme that can be based on an eID's verified user data *without* revealing any user data to the authenticating service.

State authorities already identified authentication as another eID use case. Some proposals even make eID-based identification compatible across countries, such as eIDAS in the EU [158, 89, 71]. However, they either infringe privacy per the above argument or rely on a country-specific pseudonym functionality whose authentication credentials are invalidated every time an eID is renewed.

4.5 FeIDo: Design Goals and Threats

To overcome the roadblocks of FIDO2 adoption, this work aims to design a new authenticator, called *FeIDo*. In the following, we present FeIDo's goals and threat model.

4.5.1 Goals and Requirements

The goal of FeIDo is to form a FIDO2 authenticator for secure web authentication, which overcomes current limitations of hardware and virtual authenticators. We design FeIDo to combine concepts of virtual FIDO2 authenticators with that of smartcards owned by billions [162] of citizens: eIDs such as electronic passports or ID cards. To guide the design of FeIDo, we have defined six requirements:

R1 Compatibility FeIDo must be compatible with the standard FIDO2 protocols for authenticator-based web authentication.

R2 Economic FeIDo must build on commodity hardware and eIDs that are readily available for eID holders *without* extra costs.

R3 Account Recovery The user must be able to recover access to their accounts on a FeIDo authenticator loss *without* having to register additional authenticators.

R4 User Privacy Third parties (incl. relying party, FeIDo provider) must neither be able to access personal user data directly nor link personal data to FeIDo's web credentials.

R5 Platform Independence FeIDo must not bind users to a specific software or hardware vendor (e.g., OS, client device).

R6 Anonymous Credentials The relying parties should be able to learn authenticated yet pseudonymous meta information on a user for verification, but *without* violating user privacy.

FeIDo provides a strong FIDO2 second factor for private users' web authentication (2FA). However, FeIDo is *not* limited to this application and can support additional FIDO2 use cases, such as passwordless and enterprise schemes discussed in Section 4.9.1 and Section 4.9.2.

4.5.2 Threat Model

We follow the threat model of FIDO2 web authentication [73] and extend it to include FeIDo-specific components and entities. Analogous to the FIDO2 setting, a trusted user wants to register and log into a web service via their browser using the FIDO2 protocols and the FeIDo authenticator. We trust the web service and assume a secure, authenticated connection between the browser and service. From a FIDO2 perspective, the user's browser forms the *agent* while the web service forms the *relying party*. We assume the agent and client device to be trusted—following the trust setting of token-based authentication. While we will discuss the implications of client-side attackers and how we could relax our trust assumptions using TEEs on client devices in Section 4.7.5, we also want to support users *without* access to special client-side features.

Figure 4.2 shows that FeIDo adds new components not present in existing authenticators: a user eID, a remote credential service, and a client-located middleware. We consider several attacks against each of the three components and active and passive network attackers targeting the communication between them. We trust the individual components in the following ways:

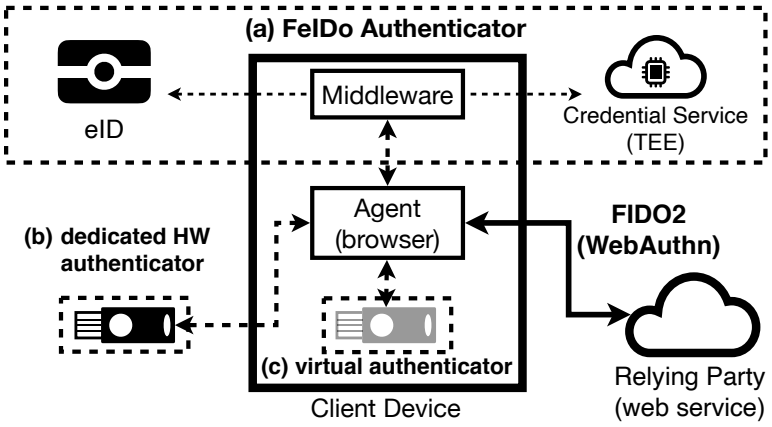

Figure 4.2: Comparison between (a) FeIDo and (b) existing hardware and (c) virtual FIDO2 authenticators.

eID Similar to hardware authenticators, we trust the eID's hardware-based protections and security protocols (Section 4.4.2) and the issuing document authority such that we can verify if a user's eID is genuine. That way, FeIDo can rule out attackers trying to spoof or clone arbitrary eIDs that impersonate others. We assume an attacker trying to steal and abuse the user eID for account hijacking (see Section 4.7.3). Finally, we discuss the security of eIDs and explain the impact of malicious authorities on FeIDo in Section 4.7.2.2.

Credential Service Service instances run on untrusted remote systems such as public cloud platforms. We assume that remote attackers, including the hosting providers and platforms, try to manipulate, steal, or clone credential services. Therefore, we operate each service in a secure, hardware-based container [154] protecting them against the above threats. Furthermore, we assume the containers support remote attestation to rule out attacks that impersonate the service.

Client Middleware We do not trust the middleware to perform any eID validation but merely use it as a proxy between the eID and the credential service. Following our assumption that we trust the client device, the middleware shares eID data only with a valid, protected credential service, verified via remote attestation.

We assume FeIDo's components to be free of exploitable vulnerabilities. To prevent physical and micro-architectural side-channels, we rely on existing smartcard features such as memory and constant-time cryptographic operations, similar to existing FIDO2 authenticators. We assume orthogonal defenses for the hardware-based containers protecting the credential services [168, 56, 20].

4.6 FeIDo: Concepts and Design

In the following sections, we explain the concepts and design of the FeIDo authenticator, including its optional support for anonymous credentials. The section then concludes with a deployment analysis. We refer to FeIDo's design requirements (Section 4.5.1) in relevant passages.

4.6.1 Big Picture

FeIDo's core idea is to derive FIDO2 web credentials based on a user's unique, personal attributes that FeIDo securely retrieves from the user's eID (*R1*). At its core, FeIDo foresees hardware-protected and attestable *credential services* that convert privacy-critical personal attributes—name, place of birth, and day of birth—to pseudonymous authentication credentials. The credential service acts as a sort of "pseudonymizing proxy" that ensures unlinkability and preserves user privacy against both the authenticating services and the untrusted credential service hosters. FeIDo combines the advantages of token-based authentication (strong security and privacy) and eID-based authentication (easy token recovery, no extra costs), tackling the open challenges of both schemes (cf. Section 4.4).

In FIDO2 terminology, FeIDo forms a virtual FIDO2 authenticator that combines TEE-protected credential services with user-owned eIDs. The credential service derives *attribute-based credentials* that are user-specific yet unlinkable (*R4*) and depend on a user's attributes rather than a specific physical device or eID. These attribute-based credentials have clear advantages when it comes to account recovery because any user eID carrying the *same* personal attributes can serve as a replacement eID (*R2+3*). In order to retain user privacy and credential security, each credential service instance is protected in a remotely verifiable hardware TEE and prevents attribute and credential leakage to third parties (*R4*). In addition, the service supports so-called *anonymous credentials* (*R6*), an extension that allows a relying party to learn meta user attributes, e.g., for age verification, *without* violating a user's privacy (see Section 4.6.4)—a feature not present in current eIDs and authenticators. FeIDo's service-based design requires neither secrets nor trusted hardware features on client devices, which benefits account recovery, user costs, and cross-platform support (*R2+3, R5*).

Figure 4.2 shows how FeIDo forms the virtual FIDO2 authenticator by combining an off-the-shelf user eID with two new software components: a TEE-protected remote *credential service* and a client-located *middleware*. The client middleware is a *secretless* component that interfaces the FeIDo authenticator with the FIDO2 agent (typically part of the browser) and securely mediates internal communication between the eID and credential service (see Section 4.6.5).

4.6.2 Comparison to Existing Authenticators

FeIDo combines concepts of virtual FIDO2 authenticators with eIDs to overcome the drawbacks of existing hardware and virtual authenticators and enable new features, as shown in Table 4.1. FeIDo removes the need (and costs) for buying extra hardware authenticators dedicated "only" to authentication purposes. Instead, FeIDo can

Table 4.1: Feature comparison between FeIDo, existing hardware and virtual FIDO2 authenticators, and eID schemes.

	FeIDo	**HW Auth**	**Virt. Auth**	**eID**
R1 – FIDO2 Compatible	✔	✔	✔	×
R2 – *No* Extra Costs	✔	×	✔	✔
R2 – Widely Deployed	✔	×	✔	✔
R3 – Device Loss Recovery	✔	✔	×	✔
R3 – Token Loss Recovery	✔	(✔)	(✔)	✔
R4 – User Privacy	✔	✔	✔	×
R5 – Cross-Platform	✔	✔	×	✔
R6 – Anon. Credentials	✔	×	×	×

leverage the wide deployment of eIDs across billions of citizens and their compatibility with off-the-shelf phones (see Section 4.4.2). In contrast, hardware authenticators have only limited user adoption due to their risk of token loss and their incurring extra costs (Section 4.4.1.1). In contrast to virtual authenticators, FeIDo relies on secure hardware containers on the credential service side rather than client devices. The credential services can use TEEs widely available on public cloud platforms (e.g., Intel SGX, AMD SEV-SNP [154]) and are thus easily shareable by thousands of users with negligible costs, similar to Tor nodes [58]. FeIDo poses no requirements on client devices and uses the client middleware only as a *secretless* proxy between eID and credential service. Therefore, similar to hardware authenticators, FeIDo is independent of the specific client device, which enables easy porting of the middleware to other client platforms. Furthermore, a client device loss does not affect FeIDo or hardware authenticators, while virtual authenticators and their credentials are tightly bound to a specific client platform and device.

FeIDo retains account access on a token loss because any replacement eID carrying the same user attributes enables access to a user's credentials and thus accounts (Section 4.6.3). Therefore, in contrast to existing authenticators, FeIDo can recover from a token loss without requiring backup tokens (cf. hardware authenticators), insecure vendor-specific cloud backups of token secrets (cf. some virtual authenticators), or insecure service-specific recovery methods (e.g., recovery codes). A potential downside can be a longer renewal time for eIDs (a few weeks) compared to buying a new hardware authenticator if no replacement eID is available (e.g., driver's license, ePassport). Finally, FeIDo enables anonymous credentials during the authentication process, e.g., for pseudonymous age verification—in contrast to existing FIDO2 authenticators.

Finally, when comparing FeIDo to naïve eID-based authentication *without* the proposed middleware and credential services (Table 4.1, last column), we see that they are not FIDO2 compatible and fully violate a user's privacy by exposing personal data to the authenticating services, as described in Section 4.4.2.1. Furthermore, they do not support anonymous credentials.

In the next section, we will compare FeIDo to two additional FIDO2 authentication variants for which support has increasingly rolled out: phone-based roaming authen-

ticators and the recent multi-device credentials, also called *passkeys* [74]. We discuss them separately as they mainly build on the discussed FIDO2 authenticators rather than representing new authenticator types and thus partially inherit their weaknesses. While these two approaches share *some* of FeIDo's goals, we will see that FeIDo still offers different tradeoffs and several additional benefits over these schemes.

4.6.2.1 (Phone-based) Roaming Authenticators and Passkeys

The FIDO Alliance and platform vendors are increasingly supporting two recent FIDO authentication variants: phone-based roaming authenticators and multi-device credentials, also called *passkeys*. The first enables using Bluetooth-enabled phones as a virtual FIDO2 token for 2FA while the latter tries to tackle the problem of device loss for FIDO credentials [74]. Due to their similarities to FeIDo regarding their goals, we will now briefly compare these recent schemes to FeIDo.

Phones as Roaming Authenticators The extended WebAuthn specification enables the use of phones as roaming, i.e., external, authenticators. During a web authentication operation, users can connect a phone via Bluetooth to their workstation and then leverage the phone as a virtual FIDO2 token as the second factor for a 2FA scheme. That way, users without a physical FIDO2 token get access to a virtual, cross-device FIDO2 2nd factor for client devices without a trusted built-in virtual token. However, this approach still suffers from the disadvantages of virtual tokens as described in Section 4.6.2 (also cf. Table 4.1) and is mainly tackling deployment issues. In particular, phone-based roaming authenticators are still prone to token loss, similar to dedicated hardware tokens, and require trusted hardware on the phone-side for protecting the FIDO2 credentials against a device compromise. In contrast, FeIDo solves the token and device loss challenges and provides a secure cross-device FIDO2 token that is compatible with any user device supporting NFC. Furthermore, FeIDo can securely support the use of an *untrusted* phone or auxiliary device as an eID proxy for workstations without NFC, as we will describe in Section 4.7.6.

Multi-device FIDO2 Credentials (Passkeys) To tackle device loss, the FIDO Alliance has introduced multi-device FIDO2 credentials, so-called passkeys, that can be synchronized across multiple devices [74]. Platform vendors are increasingly rolling out support for passkeys on client devices and hardware authenticators. Furthermore, they provide cloud services that store encrypted backups of the passkeys (FIDO2 credentials) and share them across user devices. That way, the credentials can be used across multiple devices and recovered on a new device after a device loss. The on-device protection of the credentials is platform-specific and typically relies on trusted client hardware and authentication mechanisms, e.g., TEEs and biometrics, or dedicated hardware tokens, i.e., on virtual or hardware authenticators (cf. Section 4.6.2).

In contrast to FeIDo, passkeys depend on trust in third-party service providers for the credential recovery, still require trusted hardware for client-side credential protection, and are still not fully supported. Full support for passkeys across all platforms and common services is still pending, though steadily increasing, whereas FeIDo can be

readily used if at least one NFC-compatible phone is available. Furthermore, while the cloud-based solution for passkey synchronization and backups is convenient for users, users depend on the (platform-specific) service providers for credential availability and the security of their backups. FeIDo *never* stores any user credentials (or attributes) but dynamically re-derives them for each authentication operation. In addition, the security of FeIDo's TEE-protected credential service can be remotely attested by the client middleware before sharing any sensitive user information, whereas the user has to blindly trust the passkey service providers. The open-source design and envisioned deployment model of FeIDo's credential service (further discussed in Section 4.6.3.3) also enable for shared hosting and therefore a high availability of the user credentials. While passkeys optionally allow for more secure single-device credentials (device-bound) that never leave a device and might be rooted in external hardware authenticators, these passkeys reintroduce the problems of device and token loss (cf. Section 4.6.2). Finally, FeIDo supports anonymous credentials (cf. Section 4.6.4) which are neither supported by existing FIDO2 tokens nor passkeys.

4.6.3 Attribute-based Credentials

FeIDo's credential service performs the actual FIDO2 *authenticator operations* for web registration and login, i.e., credential creation and assertion [150]. We have designed the resulting virtual token as a user-independent network service. This design mitigates the risk that users can lose secrets. Instead, the credentials become remotely usable for users from any device (*R2+3, R5*). The credential service keeps no persistent user or credential state. Instead, to distinguish users and bind their FIDO2 credentials exclusively to them, the credential service *dynamically derives* a user's credentials using a key derivation function (KDF). The unique, personal attributes from the eID are fed to the KDF, resulting in *attribute-based credentials*.

4.6.3.1 Reading eID User Attributes

When FeIDo authenticates a user, the credential service first securely reads a user's personal attributes from their eID. The service cooperates with the client middleware on each authentication request to remotely access the user's eID. By explicitly requiring a user's eID on each request, the credential service can verify the freshness of the attributes and eID and thus guarantee that credentials are only derived if the eID is valid and present. Otherwise, attackers could provide a different user's data for an impersonation attack or use bogus eID clones. To enable secure eID access and verification, FeIDo requires eIDs to provide support for (1) establishing a secure end-to-end connection to them and (2) verifying the authenticity and integrity of the eID and its stored user attributes. In our eMRTD-compatible prototype, the credential service retrieves user attributes—tunneled via the client middleware—leveraging the eID protocols described in Section 4.4.2.

4.6.3.2 Credential Derivation

The credential service now feeds the personal user attributes into a KDF to derive user-specific credentials. The KDF input binds the resulting credentials to (a) the eID owner and (b) the relying party. The credential service receives the relying party's effective domain (or origin) rp_{id} from the client middleware. For the user binding, the credential service chooses a unique set of attributes from the personal user data included in eIDs: the user's full name (*name*), date of birth (*dob*) and place of birth (*pob*), and state (*state*) issuing the eIDs. For now, we assume that the above combination of user attributes (i) remains constant for a given user and (ii) provides sufficient uniqueness guarantees. We discuss these properties in more detail in Section 4.6.6.2 and Section 4.7.2.3, and explain the prevention of cross-state attacks in Section 4.7.2.2.

To guarantee the security of the attribute-based user credentials, the KDF input additionally needs to include a secret s_{kdf} that is only accessible by a genuine, hardware-protected credential service. Lacking this secret, attackers cannot recalculate credentials purely based on the correct personal user attributes and relying party information (rp_{id}). Using the above inputs, we can now derive the credentials using a KDF. The credential service leverages an HMAC as the KDF, uses s_{kdf} as the HMAC's key [129], and all user attributes and relying party information as HMAC inputs. More formally:

$$h_{cred} = HMAC(s_{kdf}, rp_{id}|name|dob|pob|state) \tag{4.1}$$

where "|" is the byte string concatenation. The credential service interprets the resulting hash h_{cred} as the private key $\mathsf{sk}_{\mathsf{cred}}$ of the user's WebAuthn credential according to the specified signature algorithm and then calculates the respective public key $\mathsf{pk}_{\mathsf{cred}}$.

This reproducible public key pair can now be used in the FIDO2-typical way: During a credential registration (*MakeCredential*), the credential service includes the public key $\mathsf{pk}_{\mathsf{cred}}$ in the response. For a login (*GetAssertion*), the credential service uses the private key $\mathsf{sk}_{\mathsf{cred}}$ to sign the relying party's assertion challenge [150]. Personal data such as user attributes are *not* transmitted to the relying party. Furthermore, the credential service does neither persistently store user data nor credentials but instead deletes them at this point.

4.6.3.3 Cloud-Based Design

FeIDo foresees multiple redundant credential service instances that operate on public cloud platforms, which mainly serves two purposes. First, users do not risk losing the key derivation secret—all they need for authentication is stored on replaceable eIDs. Second, as public credential services can be hosted and shared by many, the costs become negligible. For a more detailed discussion on deployment scenarios, we refer to Section 4.6.6.1.

Given that the credential service is offloaded to the cloud, users want to ensure that a contacted service instance is trustworthy, i.e., it is a genuine service that is protected and does not leak s_{kdf} or personal data. Furthermore, the communication between the credential service and the middleware must be secured. Therefore, each credential service instance operates in a TEE that allows achieving all these properties by offering integrity, attestation, and confidentiality. Several TEE implementations provide these

principles [154], like Intel SGX, AMD SEV-SNP, or RISC-V Keystone—of which some are widely available on cloud platforms.

In our prototype, we have chosen an Intel SGX-based TEE. Intel SGX is a widespread, commodity server-grade CPU extension that provides user-level TEEs, so-called *enclaves*. While Intel has deprecated SGX for consumer CPUs, it continues to support SGX for cloud CPUs—exactly our setting. SGX enclaves run in dedicated, confidentiality- and integrity-protected memory regions and expose a minimal trusted computing base (TCB), including only their code, data, and the CPU (cf. background information on SGX in Section 2.6.1) [49]. Using Intel SGX's remote attestation, FeIDo users can verify the exact code and data running inside an enclave based on hardware-issued cryptographic proofs and bootstrap authenticated connections to them (similar to SENG in Chapter 2) [P1, 128]—*without* requiring access to SGX-capable hardware on the client side.

The TEE-based design is key to securing a cloud-based KDF and shielding the secret from attackers. The credential service randomly generates s_{kdf} on its *initial* startup. To be able to derive the same user credentials after a restart, the credential service must either previously use Intel SGX's sealing capability to store s_{kdf} on disk securely for recovery [49] or securely receive a copy by another credential service instance after mutual remote attestation (cf. Section 4.6.6.3). Either way, the secret can never be read outside of an enclave.

4.6.4 Anonymous Credential Extension

The design of the credential service can be extended towards *anonymous credentials*—a feature neither provided by eID schemes nor FIDO2 authenticators (Section 4.6.2). In certain settings, the relying party might be interested in a curated form of user data. For example, the relying party may want to verify if users are adults or children (user age above/below X years) before granting them access to adult content or kids-only chats or may want to restrict its service to residents of a particular country. The credential service can derive such anonymous credentials from the trusted user attributes *without* leaking the raw data to the relying party (R6). Instead of receiving the raw user attributes (e.g., date of birth), the relying party *only* learns the anonymous credentials (e.g., "is adult"). In contrast to hardware-only tokens, FeIDo can add such meta attributes easily in software. In order to provide users full control over their data, the credential service shares only anonymous credentials that the user has explicitly permitted for a given relying party.

The relying party must be able to verify anonymous credentials. Otherwise, attackers could spoof meta attributes to bypass additional access policies, such as age restrictions. Therefore, the credential service, executing in a TEE, enables a relying party to remotely attest that a genuine credential service has generated the provided meta attributes (see Section 4.6.5.4).

4.6.5 FIDO2 Integration

We now describe how our general idea of attribute-based credentials blends into the FIDO2 authentication workflow. The middleware mediates this integration as the cen-

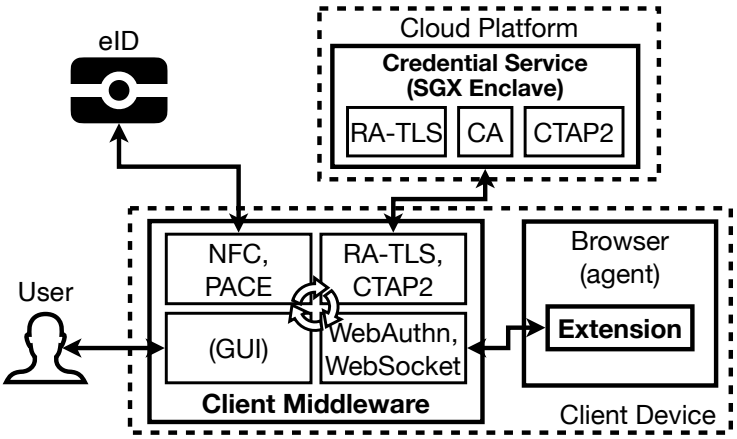

Figure 4.3: The client middleware is the central mediator and consists of multiple modules for communication with the browser extension, credential service, eID, and user.

tral entity. In the following, we describe: (i) the middleware, (ii) the message flow and how it ensures FeIDo's FIDO2 compatibility, (iii) how FeIDo can support the revocation of stolen (or lost) eIDs, (iv) how it can extend the authentication process with anonymous credentials, (v) and how these can be combined with remote attestation to thwart offline attacks on the exceptional case of an s_{kdf} breach.

4.6.5.1 Client Middleware

The client middleware is the central communication mediator of FeIDo. It fulfills two roles regarding FIDO2: on the one hand, it serves as a WebAuthn/CTAP2 agent for the browser, which contacts the FeIDo authenticator and replies with WebAuthn responses for the relying party. On the other, the client middleware is part of the authenticator itself and mediates the communication between the eID, credential service, and optionally, the user. That way, the client middleware bridges the gap between multiple internal domains while ensuring seamless compatibility with the FIDO2 infrastructure. The middleware in our prototype uses an integration solution fully compatible with commodity browsers: a browser extension-based proxy agent. As shown in Figure 4.3, the browser extension cooperates with the client middleware to integrate FeIDo into a browser's WebAuthn framework.

4.6.5.2 FIDO2 Authentication Workflow

A FeIDo authentication session follows the message flow in Figure 4.4. On a web authentication request, the agent (browser) notifies the client middleware. The middleware then communicates with the credential service and eID via authenticated, end-to-end protected connections to come up with the respective WebAuthn credential (upon registration) or assertion response (upon authentication) for the relying party. The middleware mediates to give the credential service implicit access to the user's eID. We now iterate over these steps in more detail:

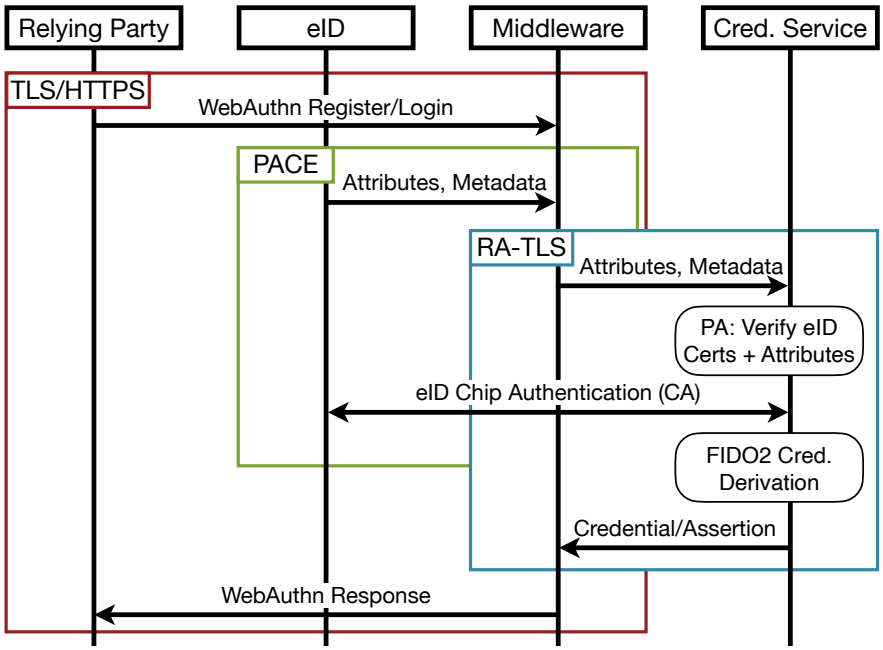

Figure 4.4: Message flow in a FeIDo authentication session.

(1) FIDO2 Authentication Initialization To initiate a FIDO2-based authentication, the relying party sends a WebAuthn registration or authentication request to the agent (typically, a user's browser). The agent then reaches out to the appropriate authenticator as described in Section 4.4.1. In our prototype, the browser extension monitors browser sessions for calls to the WebAuthn request APIs. On a web authentication request, the browser extension notifies the client middleware via a local WebSocket connection and forwards the WebAuthn request. That way, the browser extension integrates the client middleware as a WebAuthn agent into the browser. The client middleware can then start processing the request as part of the FeIDo authenticator. The choice of WebSocket has the advantage that it is compatible with commodity browsers and enables an easy relocation of the FeIDo authenticator to a trusted auxiliary user device, e.g., a smartphone, in case the client device has no eID reader. In Section 4.7.6, we will explain how FeIDo can retain its security guarantees even if the user leverages an *untrusted auxiliary* device (e.g., a borrowed phone that is compromised) as the eID reader.

(2) Reading User Attributes Next, the client middleware establishes a PACE-secured connection with the eID to read the personal attributes, according to the description in Section 4.6.3.1. It defers the CA-based eID validity check to a later stage. Depending on the eID and agent, the client middleware may prompt for user interaction, such as putting the eID on the reader or entering its PIN, if required.

(3) Credential Service Interaction The client middleware then establishes a secure network connection to the credential service to share the user attributes. To this end,

the client middleware must verify that the connection is end-to-end protected and established with a genuine, TEE-protected credential service instance. Therefore, the client middleware uses RA-TLS [P1, 128], which combines a TLS connection with a TEE-provided remote attestation (similar to SENG in Chapter 2, which however uses DTLS). That way, the client middleware has a hardware-assured guarantee that it communicates with a credential service instance that is TEE-protected, well-behaving, and does not leak credentials or attributes to others.

As shown in Figure 4.3, the credential service incorporates an RA-TLS server endpoint [128] for attested, end-to-end protected communication with the client middleware. A TLS server key pair is freshly generated by the credential service enclave on startup or securely shared across different credential service instances (cf. Section 4.6.6). RA-TLS binds the hash of the TLS server's public key to the SGX attestation report by incorporating it as authenticated user data and then integrates the resulting signed attestation quote inside the TLS server certificate (cf. details on SENG's DTLS variant in Section 2.8.1). This integration enables the client middleware to verify the hardware-assured identity of the credential service and its binding to the established TLS connection as part of the TLS server certificate validation [128, P1]. The client middleware can then securely request WebAuthn operations from the credential service through the RA-TLS connection.

(4) eID and Attribute Validation After receiving the user attributes, the credential service has to validate them using Passive Authentication (PA; cf. Section 4.4). Furthermore, it uses Chip Authentication (CA; cf. Section 4.4), initiated through the middleware's PACE channel, to verify that the service can access the eID that shipped the attributes. If both succeed, it uses the KDF to derive the user-specific authentication credentials: $\mathsf{sk}_{\mathsf{cred}}$ and $\mathsf{pk}_{\mathsf{cred}}$. Note that it is vital that the credential service performs this validation and relies on the middleware only to tunnel the respective communication. Otherwise, a malicious middleware could spoof arbitrary personal attributes and eIDs and thus perform impersonation attacks.

(5) Authentication Termination Finally, to conclude the authentication, the credential service sends the reply for the requested authentication operation to the client middleware [150], i.e., the signed credential public key $\mathsf{pk}_{\mathsf{cred}}$ on a registration and the assertion challenge signed by the private key $\mathsf{sk}_{\mathsf{cred}}$ on a login request, respectively (see Section 4.6.3.2). The middleware converts this into a *WebAuthn Response* and shares this with the relying party. To the relying party, this workflow is entirely transparent—it will not even notice that eID data was read/used at any point in time.

4.6.5.3 eID Revocation

FeIDo is designed as a *second* factor for 2FA. Therefore, stealing a user's eID does *not* suffice to gain access to the user's accounts (Section 4.7.3.2). Having said this, FeIDo supports revocation of stolen eIDs to prevent account hijacking attacks if attackers have *additionally* compromised the primary factor or if FeIDo serves as a sole authenticator (Section 4.9.1). FeIDo foresees three complementary eID revocation strategies,

providing different tradeoffs.

First, the credential service can consult existing trusted databases for stolen eIDs on eID validation (Section 4.6.5.2, step 4), such as Interpol's I-Checkit service [110], and deny their use for authentication. The eID lookup does *not* leak personal user data as it only requires pseudonymous eID identifiers (number, type, issuing state). This approach does not add state or complexity to the credential service but requires trust in the service provider (e.g., Interpol) not to re-enable or DoS eIDs and might require a *small* fee [110].

Second, the credential service can derive and share *eID-specific* revocation certificates with the client middleware, which users can leverage on demand (e.g., upon eID loss) to denylist a particular eID for all credential service instances. The middleware can protect the certificates in client-side keystores that require physical device access and user authentication [92]. On eID validation, each credential service checks the denylist and aborts authentication. That way, FeIDo can securely enable eID denylisting without requiring a trusted third party but client-side certificate storage and secure state synchronization of the eID denylist by the credential services.

Third, web services can block users on a per-eID basis. The credential service provides web services with service-specific, pseudonymized eID identifiers id_{srv} that the web services can store together with a user's FIDO2 public key. That way, web services can deny specific eIDs to enable users to revoke old eIDs when logging in with new ones—all without requiring additional state on the credential service or client-side, but at the cost of small changes to the web services. The credential services can dynamically derive unlinkable eID identifiers id_{srv} for each service based on the eID number, type, and issuer:

$$id_{srv} = HMAC(s_{kdf}, rp_{id}|eid\_num|eid\_type|state)$$

This approach faces a potential attack window until a replacement eID has been acquired (cf. Section 4.6.2) and registered on the web services. However, users can immediately close it if they own a second eID (e.g., eID and ePassport) or leverage one of the previous two approaches, i.e., report the eID as stolen or denylist it.

4.6.5.4 Anonymous Credentials in FIDO2

FeIDo's credential service enables a relying party to query pseudonymous meta user attributes, so-called anonymous credentials, on every authentication (cf. Section 4.6.4). The relying party can query these meta attributes (e.g., "is adult") using WebAuthn extensions [150] defined by FeIDo. On an authentication request, the middleware asks the user which of the requested meta attributes they want to share with the given relying party and caches the decision for future requests. Only meta attributes explicitly permitted by the user are requested of the credential service, so users stay in full control over their data. The credential service calculates the meta attributes based on the verified personal user attributes as part of the regular authentication process and includes them (e.g., "is adult: true") inside the signed WebAuthn response data (*authenticator data* [150]).

It is crucial that the relying party can verify that the provided meta attributes were not spoofed but computed by some genuine FeIDo authenticator based on verified user

attributes. Therefore, the credential service enables a relying party to link every user credential to the service's SGX attestation report. This linking enables a relying party to verify that some genuine credential service has securely generated a given credential. Thus, the relying party transitively knows that all meta attributes associated with the credential have also been generated by the (*code-wise*) same credential service and are therefore trustworthy.

Technically, the credential service binds the public key hash of the WebAuthn key pair used for signing the credential generation response, called the "attestation key pair" [150], to the SGX attestation as authenticated user data—following the approach of RA-TLS (cf. Section 4.6.5.2, step 3). The respective SGX attestation report is then integrated as an X.509 extension [128] inside the public key's certificate that is included in the WebAuthn credential response [150]. That way, the relying party can verify the response signature and associated attestation to check that some genuine credential service has generated the included credential. The key pair and attestation report are freshly generated for each new credential and preserve FIDO2's credential unlinkability (cf. Section 4.7.4.1).

4.6.5.5 Attesting Logins to Thwart Offline Attacks on a Secret Breach

Beyond the just discussed anonymous credential attestation, FeIDo can leverage remote attestation in order to tackle the unlikely situation of a KDF secret s_{kdf} breach (cf. Section 4.7.3.3) caused by, e.g., temporary vulnerabilities in the hardware TEE itself—even though we currently regard such TEE vulnerabilities as out of scope and to be handled by orthogonal defense mechanisms (cf. Section 4.5.2). By requesting FeIDo's credential service to include a fresh SGX attestation report linked to the current login process (technically: the challenge data from the relying party) as part of the meta attributes (Section 4.6.5.4), the relying party can verify that a genuine credential service protected by a *patched* TEE is used for this login attempt. That way, the relying party can prevent attackers from exploiting the stolen KDF secret to perform authentication operations using offline-calculated user credentials until a new s'_{kdf} is rolled out, because the attacker cannot forge a valid attestation of FeIDo's credential service bound to the fresh login challenge.

4.6.6 Deployment and Failover

FeIDo's design enables a cost-efficient, flexible, and scalable deployment on commodity hardware. However, FeIDo also faces failover challenges that it must address to assure credential access.

4.6.6.1 Component Deployment

The FeIDo authenticator is compatible with the FIDO2 standard for web authentication and can therefore be transparently used for existing relying parties (cf. Section 4.8). FeIDo leverages off-the-shelf eIDs on the client-side and relies on commodity TEEs for its credential service. That way, FeIDo becomes easily deployable and does not

demand additional user hardware as required by dedicated hardware or virtual FIDO2 authenticators.

FeIDo's credential service can be securely deployed on any remote server. We envision individuals or organizations volunteering to operate multiple credential service instances in a cloud, similar to software mirrors or Tor [58] nodes. That way, the credential services become widely accessible and can be shared by many users, so deployment costs become negligible. As the attribute-based credentials require only an *initial* state on the credential service (the secret and TLS server key), several service instances can be deployed in parallel without state synchronization—enabling wide availability and load balancing. Users can then freely choose any of the services for authentication as long as the service shares the KDF secret required for deriving the requested user credential. We discuss the credential service's secret bootstrap and failover in Section 4.6.6.3 and the possibility of local enterprise deployments in Section 4.9.2.

FeIDo's client middleware can be deployed as a regular application on any client device that supports an eID reader interface. The typical eID interfaces include NFC, USB, and BLE and are thus widely available (Section 4.4.2). If no reader is supported, e.g., on a workstation, the eID interactions can be offloaded to a trusted or untrusted auxiliary device, such as a smartphone (cf. Section 4.6.5.2 and Section 4.7.6). The presented WebAuthn integration of the middleware via a browser extension is compatible with commodity browsers, as demonstrated by our prototype (Section 4.8.1).

4.6.6.2 Update Management

FeIDo's design as a virtual FIDO2 authenticator enables flexible update management. In contrast to dedicated hardware authenticators, all components of FeIDo can be updated via software or microcode updates. Therefore, the components can be quickly patched without extra (hardware) costs for users or credential service providers. FeIDo's attribute-based credentials enable users to seamlessly use any replacement eID issued by the same state that includes the same verifiable user attributes (Section 4.6.3) because the KDF will derive the same credential keys (*R3*).

One typical but infrequent corner case occurs if user attributes change, such as names after marriage. Such changes result in different WebAuthn credentials and thus require a user to re-register new credentials for their accounts. The same argument holds for users immigrating to other countries. Having said this, as FeIDo seamlessly accepts *any* valid, non-revoked user eID, users can use their "old" documents for migrating to newer attributes without risking account loss. We will discuss this approach further in Section 4.9.3.

4.6.6.3 Secret Management

The credential service only requires minimal state on bootstrap: a KDF secret s_{kdf} and a TLS server key pair. The credential service can freshly generate the TLS server keys on each startup. If required for load balancing, the credential service can send a certificate signing request to a trusted certificate authority via TLS or even securely share the TLS server keys with other credential service instances via RA-TLS channels.

Secure management of s_{kdf} is crucial because s_{kdf} is part of the KDF input and binds the user credentials to the credential service instances (cf. Section 4.6.3). A credential service can recover user credentials *only* if using the same s_{kdf} used for creation. Analogously, multiple credential service instances can derive the same user credentials *only* if they share the same s_{kdf}. Therefore, FeIDo and the credential service providers must ensure that s_{kdf} can be restored. At the same time, s_{kdf} must never be disclosed outside of the TEE protection domain to guarantee that it remains unknown to attackers.

We realize this guarantee as follows. The *initial* credential service instance generates a random s_{kdf}. The credential service then uses the Intel SGX sealing functionality which binds s_{kdf} to the credential service enclave and stores s_{kdf} encrypted and signed on the untrusted disk for recovery on a restart [49]. Sealing allows data recovery only on the same physical CPU. Should multiple credential service instances be scaled to different CPUs, they must cooperate to synchronize on the key. The credential service providers can configure new service instances to securely fetch s_{kdf} from other instances via mutually-attested RA-TLS channels. This way, s_{kdf} is *only* accessible by verified, TEE-protected credential services and is resilient against partial service failures or data loss. In the unlikely case of an s_{kdf} breach, e.g., caused by a vulnerability in the TEE (out of scope), FeIDo can bind login operations to the credential service's attestation report to prevent offline credential usage by attackers, as discussed in Section 4.6.5.5.

4.7 Security Analysis

In this section, we give a security analysis of FeIDo (Section 4.7.1 and Section 4.7.2), discuss the implications of different component thefts (Section 4.7.3), and conclude with an analysis of FeIDo's anonymous credentials (Section 4.7.4). In addition, we will discuss a relaxed threat model assuming client-side attackers in Section 4.7.5 and explain how to securely offload the eID interface to an untrusted proxy device in Section 4.7.6.

4.7.1 FeIDo's FIDO2 Security

We now provide arguments about the security of FeIDo and leave a more formal security proof of our scheme open to future work. We show that based on the security of the building blocks of FeIDo, it can be seen as a standard virtual authenticator that uses a keyed pseudorandom function to generate FIDO2 credentials. As shown by Hanzlik, Loss, and Wagner [101], such a virtual token is formally secure against a man-in-the-middle attacker residing between the agent and the relying party. The authors also prove the unlinkability of the generated FIDO2 credentials. We will consider two types of attackers: (A1) a man-in-the-middle attacker residing between the agent and the relying party, and (A2) an attacker that additionally controls the communication between the components of FeIDo.

Man-in-the-middle attackers (A1) cannot distinguish whether they interact with FeIDo or a standard virtual authenticator. It is in line with the design goals specified in Section 4.5.1, where we state that FeIDo must comply with the FIDO2 protocols (*R1*). This also means that we can directly apply the result from [101], assuming the function used to derive the credentials is pseudorandom. In our prototype, we use

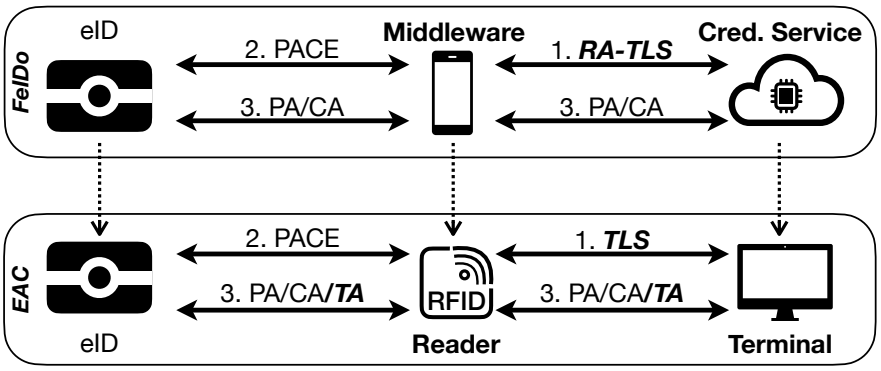

Figure 4.5: Mapping FeIDo to EAC eID security model (50). In contrast to EAC, FeIDo uses RA-TLS instead of TLS and TA.

HMAC (cf. Section 4.6.3.2, Equation 4.1), which Bellare showed to be a pseudorandom function [26]. Therefore, the security of FeIDo follows from [101], i.e., FeIDo provides unlinkable FIDO2 credentials and is secure against a man-in-the-middle attacker residing between the agent and the relying party.

4.7.1.1 Reduction Security of FeIDo

We now discuss the case where an attacker can additionally interact with the components of FeIDo (A2). We argue that assuming the credential service enclave is a secure TEE (Section 4.7.2.1), the KDF secret is not accessible outside valid credential services (Section 4.7.2.1), attributes used as input are unique (Section 4.7.2.3), and the eID is secure and the authority issuing the eID is trusted (Section 4.7.2.2), it is not easier to break the security of FeIDo than in the just discussed case where the attacker cannot interact with the components of FeIDo (A1). Trusting the issuing authority also assumes that an insider attacker cannot apply for an eID with personal data of a different person, e.g., to launch a mimicry attack. In Section 4.7.2, we show that those assumptions are reasonable.

Under those assumptions, the only difference between FeIDo and a virtual token is the communication between components. Therefore, we argue the security of FeIDo by reduction, starting from the secure Extended Access Control (EAC) protocol of ICAO-compliant eIDs—an authenticated key exchange protocol between an eID and a terminal. EAC has been proven to be secure by Dagdelen and Fischlin [50]. The protocol combines eID authentication using CA and PA (Section 4.4.2) with certificate-based authentication of the terminal, called *Terminal Authentication* (TA). TA is another standard protocol in ICAO-standardized eIDs [108]. It is a challenge-response protocol during which the eID validates the authenticity of the terminal communicating with it based on a certificate chain rooted in the eID authorities. TA serves as an additional protection mechanism that requires the terminal to be specifically certified by an eID authority [108], e.g., when access to sensitive biometric data is requested. In EAC, an eID reader communicates with a terminal using TLS to proxy CA, PA, and TA for mutual eID and terminal authentication. The initial authentication between the reader

and eID can be based on PACE (cf. Section 4.4.2). We now show how FeIDo's components correspond to the ones used by the EAC security model in [50]. In both cases, the role of the eID remains the same. We simplify the function of the client middleware to that of an RFID/NFC reader. Finally, the credential service is the terminal verifying the eID. We will now briefly argue that the minor changes in FeIDo do not influence security. We begin by showing that RA-TLS provides the same security guarantees as a TLS connection and implies a trusted credential service. These two results allow us to argue the security of FeIDo without executing Terminal Authentication and directly apply the results from [50] since the other subprotocols used (CA and PA) are the same, as shown in Figure 4.5.

RA-TLS security The RA-TLS protocol combines the guarantees of TLS with the remote attestation of a TEE (cf. Section 4.6.5.2, step 3). Therefore, it provides an end-to-end protected connection between the client middleware and a verified credential service instance—protecting against network attackers and spoofed credential services. It follows that the RA-TLS connection satisfies the security properties of the TLS connection between the eID reader and terminal as required by the EAC model.

EAC security The EAC protocol considers authentication of the terminal and the eID. The credential service is protected inside a TEE against system-level attackers at the service provider platform. In addition, the middleware remotely attests the protection and exact code of the credential service as part of the RA-TLS connection establishment. In combination with the credential service's open-source design—allowing for code audits (cf. Section 4.8.1)—users can validate the service's correctness and integrity. Under the assumption that the credential service executes in a secure TEE (Section 4.7.2.1), it follows that the credential service, i.e., the terminal, is trusted. It also means that the security of the whole EAC protocol holds even if, in FeIDo, we do not execute Terminal Authentication. Finally, the Passive Authentication protocol (PA, cf. Section 4.4.2) provides the authenticity of the public key used by the eID during Chip Authentication (CA, cf. Section 4.4.2), which is required by the EAC proof.

4.7.2 Security Assumption Verification

In the following, we discuss and verify our security assumptions made in FeIDo's security argument (cf. Section 4.7.1.1).

4.7.2.1 Credential Service TEE and Secret Security

The TEE protection of the credential service is crucial for FeIDo to protect the KDF secret and thus the users' FIDO2 credentials against malicious service providers and spoofed services. Furthermore, remote attestation ensures that all communication parties can validate whether they communicate with a genuine credential service.

The TEE-assisted protection of the secret guarantees that an attacker who knows all user attributes cannot forge credentials. To derive the same secret key as the authenticator, the attacker would need to retrieve the credential service's secret key s_{kdf} for the KDF. The credential service protects s_{kdf} using the runtime protection and

secure storage mechanisms of its TEE. The credential service shares s_{kdf} *only* with other verified credential services via mutually-attested, end-to-end protected connections (cf. Section 4.6.6.3). Furthermore, relying parties can detect abuse of leaked secrets (cf. Section 4.7.3.3).

4.7.2.2 eID Security and Malicious eID Authority

FeIDo relies on secure eIDs and a trusted eID authority to securely bind credentials to users based on their eIDs. FeIDo assumes the eIDs are free of backdoors and deploy typical smartcard protections, such as unclonable memory, constant-time cryptographic operations, and authentication mechanisms (see Section 4.4.2, Section 4.5.2). That way, FeIDo can rule out attackers trying to clone valid user eIDs for account hijacking. Thus, FeIDo follows similar assumptions as dedicated hardware authenticators, where trust is put into the authenticator devices and their manufacturing vendors. Note that for eIDs, cross-authority attacks are impossible, i.e., a malicious state trying to create valid eIDs of another. eIDs include the issuing country and are signed by a country-specific key checked during Passive Authentication. However, a malicious country M can still issue an eID that matches all attributes of an existing eID of a target country T *except* the issuing country code T. Therefore, FeIDo explicitly includes the issuing country as input to the KDF (*state*, Equation 4.1), resulting in *different* credentials. Thus, a malicious state (a.k.a. country) cannot issue eID clones to hijack accounts of residents of a different state.

4.7.2.3 Uniqueness of Personal Attributes

FeIDo's KDF uses a person's full name, date of birth, place of birth, and state as input (cf. Section 4.6.3, Equation 4.1). If two persons of the same state are born on the same date, in the same city, *and* share the full name, they share the same credentials. The likelihood of such a collision highly depends on the city's respective naming convention and size. Without perfect global data to precisely measure this risk, we can only give approximations. For example, consider the US. Sweeney showed that place, gender, and date of birth could uniquely identify half of the US population [212]. Surprisingly, this is *without* considering the name and surname, which introduce significant entropy. More generally, assume n users are namesakes born in the same state, city, and year. Then, assuming $d = 365$ days per year, based on the birthday paradox, the probability that none of these users share the same birthday is:

$$P = \frac{d-1}{d} * \frac{d-2}{d} * \ldots * \frac{d-(n-1)}{d}$$

The resulting probabilities generally follow a Gaussian-like curve, but for smaller groups of namesakes, the chance for a collision decreases quadratically with the number of namesakes. To give upper bounds for the likelihood of a *full* collision in the US, we use the data from the US birth names inventory [181]. Consider the most common female (Olivia) and male name (Liam), combined with the most common surname (Smith). On average, even in the largest city of the census (NYC), there were just around $n = 8$ persons born with this name combination in 2019—resulting in a 92 %

chance even these US "worst-case combinations" of name/place/date are unique. Given the example of the US, one can argue that most users have unique personal data and hence KDF inputs. Admittedly, this argument may differ in naming conventions with highly-skewed name distributions. However, even in the unlikely event of full collisions, the security implications are limited in a 2FA setting. Section 4.9.1 discusses how uniqueness can be extended for a single-factor setting.

4.7.3 FeIDo Component Theft (or Loss)

We now discuss how the theft (or loss) of the client device, eID, or KDF secret affects the security of FeIDo.

4.7.3.1 Client Device Theft

By design, FeIDo only runs the *secretless* client middleware on the client devices, which is not involved in the attribute-based credential derivation but only serves as a message proxy between the eID and credential service (Section 4.6.5). Therefore, theft of a client device has no security impact on FeIDo and, in contrast to client-side virtual FIDO2 tokens (cf. Section 4.4.1.1 and Section 4.6.2), does not stop a user from accessing their credentials.

4.7.3.2 eID Token Theft

At first glance, FeIDo seems to share the security guarantees of hardware tokens upon token/eID theft: Attackers could use the stolen eID for authentication operations. While in a 2FA setting—FeIDo's main use case—a stolen eID (or token) is *not* sufficient for attackers to hijack user accounts, attackers *could* gain access if they have *additionally* compromised the primary authentication factor or if FeIDo is used as a sole authenticator (Section 4.9.1). Similar to hardware tokens, the credential services could restrict service to eID types that support a PIN known only to the owner to prevent abuse by attackers. Then again, FeIDo enables for additional protections that off-the-shelf hardware tokens do not offer. In contrast to hardware tokens (Section 4.6.2), FeIDo *does support* central and per-service revocation of stolen eIDs, which blocks account hijacking attempts (Section 4.6.5.3). Furthermore, on an eID loss, users neither need backup tokens, less secure login alternatives, nor re-registration of new credentials. Instead, FeIDo's attribute-based credentials enable secure credential recovery (and thus account recovery) directly via replacement eIDs.

4.7.3.3 KDF Secret Theft

FeIDo's KDF secret s_{kdf} is crucial for deriving the user credentials. Therefore, the credential service uses its TEE to protect s_{kdf} against theft by restricting access to s_{kdf} only to secure, verified service instances (cf. Section 4.7.2.1). Furthermore, FeIDo protects s_{kdf} against data loss and partial failures by combining TEE-protected backups with secure credential service replication (cf. Section 4.6.3.2). That way, FeIDo can guarantee credential availability and avoid s_{kdf} changes that would require credential re-registrations. Note that even if the secret should ever leak, offline-calculated

credentials will still not be accepted by the relying parties—remote attestation of the valid credential service can ensure that the credentials were genuinely derived (cf. Section 4.6.5.5).

4.7.4 Security of Anonymous Credentials

We now discuss the security guarantees of FeIDo's anonymous credentials (Section 4.6.4) and their implications on an eID theft (Section 4.7.3.2).

4.7.4.1 Anonymity / Unlinkability Guarantees

FeIDo's anonymous credentials enable a relying party only to learn pseudonymous meta user attributes. The credential service only allows for meta attributes that provide a sufficiently large anonymity set such that the anonymous credentials provide reasonable group anonymity—even when used in conjunction. In addition, users have full control over the attribute sharing because they can allow/block any meta attribute on a per-relying party basis and must explicitly perform an authentication to allow for queries by the relying party (cf. Section 4.6.4 and Section 4.6.5.4). This rules out (mass) query attempts with the goal of user deanonymization. In the following, we assume the meta attributes to be anonymous and defer their full specification to future work.

Even though FeIDo's anonymous credentials are bound to the FIDO2 authentication process (Section 4.6.5.4), they do not enable linking attacks. While a relying party can notice via an SGX attestation report that the meta attributes are coming from some credential service, i.e., FeIDo authenticator, different relying parties can still not link multiple authentications to the same FeIDo instance. First, we showed that FeIDo is a FIDO2 authenticator (cf. Section 4.7.1), i.e., its credentials and authentication operations are unlinkable. Second, the anonymous credential integration introduces *no* linkable information because (1) the meta attributes are anonymous, (2) SGX's attestation can provide unlinkability [49], and (3) credential services use fresh per-credential attestation keys (cf. *AnonCA* [150]).

4.7.4.2 Impact of eID Theft

The anonymous credentials preserve FeIDo's security guarantees and easy account recovery on an eID theft (Section 4.7.3.2). However, while FeIDo's eID revocation prevents *any* abuse of stolen eIDs, *unnoticed* theft, which does not lead to an eID revocation, can be a real threat in practice for bypassing access policies that are based on a user's anonymous credentials. For instance, non-adult attackers might *temporarily* steal an eID from an adult (e.g., a parent) to bypass age gates. To prevent such attacks, FeIDo's credential service could restrict service *only* to eIDs that support an access PIN only known to the eID owner (Section 4.7.3.2).

4.7.5 Client Device Compromise

So far, we followed FIDO2's threat model, assuming no attackers on the agent and client device (cf. Section 4.5.2). In the following, we assume such client-side attackers.

We compare FeIDo's security guarantees to those of hardware tokens and eIDs in this setting.

Strong User Confirmation Hardware-backed FIDO2 authenticators (incl. FeIDo) protect user credentials against direct access by client-side attackers. However, when the hardware tokens—or in FeIDo's case, user eIDs—are connected to the client device, *by default*, they cannot prevent attackers from *using* them to authenticate to a user's accounts (ignoring 2FA for the moment). Attackers can perform arbitrarily many authentication operations as long as the token/eID is connected—without the user noticing. In order to tackle this issue, some hardware tokens require explicit user confirmation on each operation, e.g., via a physical button on the token. However, even with such tokens, attackers can trick users into unwillingly confirming an authentication request to an attacker-controlled relying party by manipulating the target origin and displaying a phished confirmation request to the user. To prevent such attacks, *costly* high-end hardware authenticators feature an integrated display [193], which shows the target origin and thus enables target verification before user confirmation.

FeIDo can provide the *same* protection as costly high-end hardware tokens, but *without* incurring additional costs, by leveraging client-side TEE support if present on consumer phones. The idea is to let the credential service require that the WebAuthn request data has been signed as part of protected user confirmation. FeIDo's client middleware can then leverage technologies like Android's protected confirmation [53], which supports a TEE-protected secure UI and button I/O, to display the target relying party in an unforgeable way to the user. That way, the user can vet the target relying party of the WebAuthn request and cancel the authentication when detecting manipulation by a system-level attacker. That way, FeIDo users gain the same strong confirmation guarantees as provided by FIDO2 tokens with dedicated displays and buttons, but *without requiring expensive* token devices—clearly increasing the level of security compared to standard tokens.

User Attribute Leakage FeIDo's current design shares the property of eID schemes that client-side attackers can intercept the personal attributes read from a user eID—or the password required to read from the eID via PACE directly (cf. Section 4.4.2)—and therefore leak the attributes. While the attacker cannot use the attributes to access or recalculate FeIDo credentials (cf. Section 4.7.2.1), it can be considered a privacy disadvantage compared to dedicated FIDO2 authenticators. However, we argue that client-side attackers have several alternative opportunities to steal user information—even beyond what is readable via PACE from an eID. For instance, users typically store several personal documents on their devices, and attackers can intercept any user communication (incl. emails and chats). Furthermore, sensitive biometric eID information (except a low-resolution profile picture) is only accessible by special authorities.

4.7.6 Using an Untrusted Reader Device

For ease of discussion, so far, we have assumed that the client middleware is directly running on a client device with NFC support. In Section 4.6.5.2 and Section 4.6.6.1,

we have briefly mentioned the possibility of offloading the eID interactions to an auxiliary reader device, e.g., a phone, in case the primary client device (agent) has no NFC support. While FeIDo's WebSocket-based browser integration enables a direct relocation of the client middleware to a different trusted user device (cf. Section 4.6.5.2), this approach cannot be applied to fully untrusted reader devices. In the following, we elaborate on the latter setup and discuss required security considerations.

Following FeIDo's threat model (Section 4.5.2), a user wants to authenticate from a trusted client device to a web service using FeIDo. However, the client device does not support NFC, e.g., being an older workstation, and therefore cannot directly communicate with the user eID as required by FeIDo. To resolve this issue, the user can leverage an auxiliary device, e.g., a phone, as an eID-interacting proxy device. However, the auxiliary device might be untrusted and therefore must not be able to arbitrarily communicate with the user eID. Even though a malicious auxiliary device cannot steal a user's client-side passwords in a 2FA setting, the attacker could leverage eID access to perform authentication operations or read personal user information from the eID, as discussed in Section 4.7.5.

To prevent such attacks, FeIDo can keep the client middleware on the trusted client device and leverage the auxiliary reader device only as a message forwarder to the user eID. The secure end-to-end PACE and Chip Authentication (CA) connections (cf. Section 4.4.2) remain established between the trusted client device and user eID as was shown in Figure 4.4. To interact with the eID, the client middleware forwards the end-to-end protected messages through a TLS channel to the auxiliary device which passes them to the user eID. The connection establishments of PACE and CA are based on a Diffie-Hellman key exchange which is secure against man-in-the-middle attackers, including the untrusted auxiliary device. The auxiliary device cannot tamper with the communication as the communication keys are securely located on the eID and trusted client device. Furthermore, the auxiliary device cannot establish own communication channels with the user eID, because the attacker does not know the eID password required for PACE. The PACE password cannot be intercepted by the attacker during the secure connection establishment by the client device.

The attacker could try to brute force the PACE password because it has only around 50 bits of entropy in total and includes some personal user data (e.g., name) that might be easier to guess. However, our measurements have shown that each PACE establishment attempt requires $\approx$1 s resulting in a very long attack time for 2^{50} combinations. In addition, the eID needs to be steadily connected to the auxiliary reader device during this online attack. We therefore deem this attack scenario infeasible in practice, even for scenarios where the entropy would be partially decreased, e.g., by an attacker knowing the victim's full name upfront.

We conclude that users can securely use an untrusted auxiliary device, e.g., a borrowed phone, as an eID proxy for FeIDo. That way, FeIDo can easily support workstations even without NFC support.

4.8 Evaluation

We now describe our FeIDo prototype and evaluate its performance.

4.8.1 Prototype

Our current prototype consists of an Android app, a browser extension, and an SGX enclave. For the sake of demonstration, without losing generality, we picked a German ePassport as eID—but could have used any ICAO-standardized eID. Our prototype is open-source (cf. Section 4.10) [S3] and currently focuses on the core concepts for web authentication. It does not yet support anonymous credentials and does not yet cover the extended settings described in Section 4.7.5 (client compromise) and Section 4.7.6 (untrusted reader).

4.8.1.1 Client Middleware

An Android app represents the client middleware. It provides a UI for manually entering the eID's static password as required for PACE—a one-time process that can be replaced by taking a photo of the user's eID (cf. Section 4.4.2). The client middleware registers an NFC intent filter for detecting eIDs and uses the JMRTD library [170] for NFC-based communication with them, e.g., for PACE and data group queries. The client middleware communicates via protobuf messages with the credential service and browser extension through an RA-TLS (credential service) and WebSocket connection (browser extension), respectively. The client middleware enables the CA channel between the eID and the credential service by forwarding raw Application Protocol Data Unit [174] commands. As a performance optimization, the client middleware can cache data groups read from the user's eID to avoid re-reading them via NFC. Having said this, PACE, PA, and CA are re-executed on every authenticator operation to verify the eID (Section 4.6.3.1, Section 4.6.5.2).

4.8.1.2 Browser Extension

The browser extension uses a content script to overwrite the create() and get() functions of a browser's navigator.credentials API. That way, the browser extension intercepts WebAuthn requests by a web page and can forward them to the FeIDo authenticator through the client middleware. This approach enables easy integration of FeIDo without emulating a physical device (e.g., USB). The browser extension manually crafts respective return values for the overwritten functions based on the client middleware's WebAuthn response data to interact with the web pages seemingly. While we tested the browser extension with Firefox, the core APIs are also available in other browsers.

4.8.1.3 Credential Service

The credential service is implemented based on the Intel SGX SDK v2.15 [107] in ≈2.8 k lines of C/C++ source code (excl. libraries). Upon an incoming RA-TLS connection by a client middleware, the credential service sequentially processes the WebAuthn operation request and eID information before replying with a WebAuthn response. We have integrated an RA-TLS server endpoint based on the SGX SSL patches of the SENG-SDK (Section 2.10) [P1]. For parsing the eID data groups and running the PA and CA protocols, we patched the OpenPACE library v1.1.2 [159] to add support for German ePassports, SGX, and RA-TLS. To demonstrate eID revocation (Section 4.6.5.3), we

have implemented a service that mimics Interpol's I-Checkit database service of stolen eIDs [110]. After CA, the credential service queries the blocklist via TLS using the eID's document number, type, and country of issuance (following [110]) and aborts authentication on a database hit. For the attribute-based FIDO2 credential derivation, the credential service uses a SHA-256 HMAC to hash the attributes, forms a private key based on the hash, and finally calculates the corresponding public key. The credential service prototype currently supports only German ePassports and the ES256 algorithm for the WebAuthn signatures.

4.8.2 Performance Evaluation

We now evaluate the performance of our prototype implementation during a FIDO2 web authentication process.

4.8.2.1 Methodology

In our experiment, we host a local instance of the *webauthn.io* test page [63] and measure the time it takes to (i) register and (ii) log into an account using single factor FIDO2 authentication. We run the web service and the credential service on a Dell XPS 9560 laptop with an Intel® i7-7700 HQ and Ubuntu 18.04 LTS. For the agent, we use two devices: we run Firefox 96 and FeIDo's browser extension on the Dell XPS but run the middleware on a Pixel 4a phone with Android 12. The laptop and phone are interconnected via a local 1 Gbps network using Ethernet and Wifi.

To assess the practical feasibility of FeIDo, we compare its performance against two FIDO2 authenticators. For our measurements, we take a SoloKey Hacker v2.1 with an unlocked 4.1.2 firmware and a Nitrokey with a 2.4.0 firmware, which are connected via USB to the laptop, as baselines. We measure the average time over ten iterations for each operation (registration, login) for each authenticator. We keep the WebAuthn attestation feature of *webauthn.io* disabled for the measurements and assume the web page and client middleware app (with cached PACE password) to be preloaded. We assume that the ePassport is already placed on the NFC reader.

4.8.2.2 Evaluation Results

There were no significant time differences between the registration and login operations for all authenticators. We have measured the performance of our FeIDo prototype two times: once the initial, uncached performance (*uncached*) and once with the data groups of the ePassport cached by the client middleware (*cached*) as described in Section 4.8.1. The uncached FeIDo operations took ≈2980 ms on average, which is close to the average performance of Nitrokey of ≈3183 ms. The overall median duration of Nitrokey has been ≈2327 ms and the average duration of SoloKey ≈1813 ms. The standard error of the means (SEM) of both FeIDo setups was below 28 ms and those of SoloKey below 101 ms. Nitrokey faced bigger SEMs of ≈669 ms (register) and ≈844 ms (login) due to outliers caused by its unreliable user confirmation based on squeezing the authenticator case in contrast to SoloKey's button.

By caching the ePassport data groups in the client middleware, FeIDo has significantly improved its average operation duration by ≈19.3 % to ≈1878 ms, which is close to that of SoloKey. This shows that reading ePassport data via PACE contributes a major part. In fact, our measurements have shown that the overall communication processing between the client middleware and ePassport currently makes up ≈68 % to 78 % of FeIDo's operation duration. The total operation duration could probably be further improved by parallelizing the prototype, e.g., by setting up the PACE and RA-TLS connections concurrently or deriving the WebAuthn credentials while waiting for the PA and CA results. The current measurement excludes the eID revocation lookup as we have no access to the Interpol I-Checkit service. However, [110] states a lookup time of ≈30 ms plus network latency, which is an insignificant extra overhead. We conclude that the performance of our current FeIDo prototype is already in the range of existing hardware FIDO2 authenticators.

4.9 Discussion

In the following, we discuss FeIDo's applicability to single-factor (Section 4.9.1) and enterprise authentication schemes (Section 4.9.2) and how FeIDo operates in settings where personal user attributes change (Section 4.9.3).

4.9.1 FeIDo as Sole Authenticator (Passwordless)

While we assume FeIDo to be used as an additional factor in a 2FA scheme, in the following, we want to discuss in how far FeIDo could serve as a *sole* authenticator for FIDO2 web authentication. As FeIDo is a FIDO2-compliant virtual authenticator (cf. Section 4.7.1), it can be directly used as the sole token. However, to provide high security in a non-2FA setting, FeIDo requires additional protection.

On an eID theft, just as hardware tokens, FeIDo must prevent account hijacking by attackers. To handle this, as proposed in Section 4.7.3.2, FeIDo's credential services can restrict service to eIDs that support an access PIN known only to the genuine owner—analogous to PINs provided by some hardware tokens—and implement a suitable subset of FeIDo's eID revocation mechanisms (cf. Section 4.6.5.3). That way, stolen eIDs become unusable by attackers not knowing the PIN or even entirely revoked for FeIDo authentication operations.

While a collision of our chosen set of user attributes is unlikely in practice (Section 4.7.2.3), when considering FeIDo for usage as the *sole* authenticator, we must *guarantee* user-unique KDF inputs to derive distinct credentials. To this end, we could add a user-specific secret *salt* to the KDF input set (Section 4.6.3.2). This secret must be a strong, client-side random password accessible by the client middleware, which is forwarded to a credential service as part of the authentication request. That way, even on a full attribute collision of two users, their KDF input and thus derived credentials stay distinct. While the secret can be cached on the client device, e.g., in a phone's TEE-protected key storage [92], the user has to back up the salt against client device loss. As a positive side effect, the salt also protects against eID theft and cloned eIDs issued by malicious authorities.

The depicted sole-factor usage of FeIDo allows to draw a comparison to sole-factor passwords for web authentication. All FIDO2 schemes, including FeIDo, are resilient against several attacks that passwords do not withstand, such as phishing, shoulder surfing, password database leaks, or cracking/guessing attacks [191, 65, 166, 54]. Moreover, whereas password leaks give attackers immediate access to the associated account(s), the FeIDo/FIDO2 credentials cannot be extracted to impersonate users even if the attacker has physical access to the token (or eID). While attackers can *reuse* stolen/lost tokens to impersonate users, access codes (or the PIN/salt depicted above) mitigate this threat. Admittedly, passwords have lower requirements: they are widely supported, do not have to be carried, and require no special or dedicated hardware. For a more detailed comparison between sole-factor FIDO2 and passwords, we refer to Lyastani et al.'s systematization paper [84].

4.9.2 Enterprise Authentication Use Cases

So far, we have focused on FeIDo being used by private users. We now discuss FeIDo's applicability to enterprise-focused authentication use cases, loosely following those given in [72] by the FIDO Alliance. As FeIDo is fully FIDO2-compliant, in principle, FeIDo can be used for any enterprise setting where FIDO2 hardware or virtual authenticators are in use. Employees can directly use their eIDs with FeIDo for enterprise two- or single-factor web authentication. Alternatively, if a company issues electronic employee ID badges that provide the required personal attributes and compatible authentication protocols (Section 4.4.2), FeIDo could support them.

FeIDo can also be used for *local* enterprise service or device authentication if network connectivity is available. FeIDo requires access to a credential service and an attestation service for validating the TEE protecting the credential service. If internet connectivity is available (default), both services can be hosted remotely, e.g., in a public cloud and by the TEE vendor. Alternatively, to enable *local* intranet settings, a company can host a private credential service and attestation service (e.g., Intel SGX DCAP [103]) on a local enterprise server. That way, FeIDo can even be used for other settings such as local client device (domain) logins, remote logins, or SSH logins. FeIDo can even support physical access, e.g., via smart door locks, as smart locks often support NFC and intranet access for checking credentials against the enterprise database.

FeIDo's requirements are thus comparable to those of existing FIDO2 authenticators, except authenticators do not require access to a TEE-protected credential service, which simplifies local setups. In addition, they can better support offline use cases.

4.9.3 eID Migration on Attribute Changes

FeIDo's KDF-based credentials rely on the fact that user attributes do not change. While this is true for most attributes such as date and place of birth, names or nationality *may* change. For example, users may change their surname upon marriage. Since the credential derivation is based on a user's attributes, the user loses access to the relying parties once attributes change. This migration problem can be solved by temporarily switching to other authentication schemes and eventually (re-)linking the (new) user data to the account. Alternatively, users can leverage the fact that they may

own multiple valid eIDs issued by the same state (e.g., a national ID *and* an ePassport). FeIDo transparently accepts *any* valid, non-revoked eID of a user. The user can first apply for just one of the two documents with their new data. They can then still use the "old" document for a final authentication based on their outdated data and then link their new document to their accounts. Once all accounts are migrated, the user can finally replace the other eID.

4.10 Artifacts

The prototypes of FeIDo are available as open-source projects at `https://github.com/feido-token` [S3], including the SGX-based credential service, Android middleware app, Firefox browser extension, and demo eID database service. See page 9 for a list of all open-source prototypes covered by this dissertation.

4.11 Conclusion

FeIDo represents the first attribute-based FIDO2 virtual authenticator. The system uses an eID as the physical component that the user possesses and a TEE-protected credential service that interfaces the eID's authentication mechanism with the FIDO2 protocol. FeIDo addresses two major open challenges in FIDO2: cost efficiency, and recovery in case of authenticator loss. We also showed that it is secure under reasonable assumptions. In contrast to existing FIDO2 tokens and eIDs, FeIDo additionally enables anonymous credentials, which the credential service can provide as a FIDO2-compatible extension. Therefore, FeIDo not only shows that TEEs can help tackle the open challenges of FIDO2—thus positively answering *RQ3* (see page 5)—but can additionally enable new security features, e.g., FeIDo's anonymous credentials.

We released an open-source prototype of FeIDo and compared its efficiency with existing hardware authenticators. In particular, we showed that the execution time of our prototype implementation is comparable with standard FIDO2 authenticators. We conclude that FeIDo is a practical, cost-efficient, and secure alternative to existing hardware and virtual authenticators.

This chapter concludes the first part of this dissertation which focused on presenting TEE-based designs for security-critical network and web authentication services. Chapters 2 to 4 contributed to meta research question *MQ1* (see page 4) by showing how such services can benefit from TEEs, e.g., via protection against strong system-level attackers or enabling of new security features. In the next chapter, we will present the second part of this dissertation which contributes to meta question *MQ2* (see page 5), i.e., explores in how far we can enable additional runtime defenses for TEE-protected services. We will focus on secure runtime monitoring for VM-level TEEs to detect, analyze, or prevent in-VM attacks.

5

00SEVen – Re-enabling Virtual Machine Forensics

Introspecting Confidential VMs using privileged in-VM Agents

5.1 Motivation

The security guarantees of confidential VMs[1] like AMD's SEV are a double-edged sword: Their protection against undesired VM inspection by malicious or compromised cloud operators inherently renders existing VM introspection (VMI) services infeasible. However, considering that these VMs particularly target sensitive workloads (e.g., finance), their customers demand secure forensic capabilities.

Therefore, in this chapter, we address *RQ4* (see page 6), i.e., answer how we can enable VM owners to remotely inspect their *confidential* VMs without weakening the VMs' protection against the cloud platform. We analyze the technical and threat model-related challenges hindering the adoption of existing inspection techniques and present a new VMI design tackling them, called 00SEVen. In contrast to naïve in-VM memory aggregation tools, our approach is *isolated* from strong in-VM attackers and thus resistant against kernel-level attacks, and it provides VMI features *beyond* memory access. 00SEVen leverages the recent intra-VM privilege domains of AMD SEV-SNP—called VMPLs—and extends the QEMU/KVM hypervisor to provide VMPL-aware network I/O and VMI-assisting hypercalls. That way, we can serve VM owners with a *protected in-VM* forensic agent. The agent provides VM owners with *attested remote* memory and VM register introspection, secure pausing of the analysis target, and page access traps and function traps, all isolated from the cloud platform (incl. hypervisor) and in-VM rootkits.

5.2 Problem Description

The security of virtual machines (VMs) is a crucial factor that can determine if customers are willing and regulatorily permitted to offload processes to a cloud platform. In order to increase trust into cloud platforms, researchers and vendors have developed several VM security and monitoring solutions rooted in the privileged VM manager, i.e., the hypervisor. These solutions range from kernel code integrity schemes and virtualization-based enclaves to forensic VM introspection (VMI) services [198, 13, 246, 115]. An impactful recent virtualization extension has introduced so-called *confidential VMs*.[1] This security extension protects the confidentiality and integrity of a VM's memory and registers against the cloud platform, including the hypervisor and peripherals. Confidential VMs aim at enabling cloud adoption for highly sensitive customers that need to satisfy high security regulations and might distrust third-party cloud providers, e.g., the finance and health sector. AMD SEV is the pioneering solution, which has so far been extended multiple times (SEV-ES, SEV-SNP) [3] and is widely available at cloud platforms [96, 12, 93]. Due to the high demand, other major CPU vendors provide their own confidential VM designs, e.g., Intel TDX and Arm CCA [106, 16].

However, while confidential VMs provide attestable hardware protection against the cloud platform, they are still prone to runtime compromises. Remote attackers can still exploit vulnerable network services within the VM or perform software supply

[1] also called Trusted Execution Environment VMs, TEE VMs, or TVMs

chain attacks to gain control over a confidential VM. Considering the sensitivity of the confidential VMs' target group, it is therefore crucial to deploy additional security mechanisms and monitor the VM for indicators of compromise. Hypervisor-based VMI is a well-explored forensic method to analyze a VM's memory and execution state, allowing to scan for such indicators, e.g., traces of rootkits [115]. Unfortunately, the threat model of confidential VMs is inherently in conflict with VM security schemes that root their trust in the cloud hypervisor, including VMI. Furthermore, the protection of confidential VMs blocks major access methods required by these schemes—rendering them unfeasible.

5.3 Contributions

Our goal therefore is to analyze these limiting factors and then propose a new design for secure *remote* introspection of *confidential* VMs. That way, we fill an important gap by re-enabling VM owners, i.e., the cloud customers, to inspect the runtime states of their VMs (memory, registers) *without* sacrificing security guarantees against both the hypervisor (e.g., cloud vendors) and in-VM attackers (e.g., kernel-level rootkits). We aim to enable VM owners to, e.g., periodically scan for attacks or perform post-mortem digital forensics without leaking any inspection data to the hypervisor or in-VM attackers. This goal aligns with recent research ambitions trying to re-enable hypervisor-based security schemes for confidential VMs. Hecate [83] is a noticeable recent work that focuses on protecting legacy OSes inside confidential AMD VMs and re-enabling in-VM network and static code integrity policies. Hecate shows the need to redesign existing techniques for the threats and specifics of confidential VMs.

To the best of our knowledge, we are the first to focus on the challenges of re-enabling secure remote VMI for confidential VMs. Previous work on inspecting confidential VMs is limited to an attacker perspective. Approaches like SEVered [157] have exploited the missing integrity protection of early SEV versions to leak memory and register content of a VM [155, 156]. However, recent confidential VMs, e.g., SEV-SNP, feature integrity protection that fixes the root causes of these attacks. Existing out-of-VM forensic systems are blocked by SEV's memory protection [78, 242, 201]. VM owners currently must fall back on deploying forensic tools *inside* the confidential VMs, e.g., LeechAgent [79] or GPR [94]. However, these tools lack VMI features, e.g., VM pausing and traps. Even worse, they are *not* isolated from privileged in-VM malware (e.g., rootkits) providing system-level attackers full control over them and thus rendering them *insecure*.

In this chapter, we present *00SEVen*, a design for secure remote introspection of confidential AMD VMs, even under a strong in-VM attacker. Our design introduces a *privileged in-VM agent* that exposes introspection capabilities via the network to the VM owner, e.g., a cloud customer. The VM owner can securely connect an analysis client (e.g., based on LibVMI [142]) to our agent to start a remote introspection session of the VM. By deploying our agent *inside* the confidential VM, the agent can access the *private* VM memory without being blocked by the memory protection while still being isolated from the untrusted hypervisor. Furthermore, our agent is protected against in-VM system-level attackers, offers hardware-based attestation of our VMI

infrastructure, and supports VMI features *beyond* pure memory forensics, e.g., register access, VM pausing, and memory access traps.

However, these goals are not trivial to achieve for confidential VMs: The cloud hypervisor is *untrusted*, i.e., we can *not* rely on it to protect the agent against the in-VM OS. Furthermore, the hypervisor is still involved in important VM tasks, e.g., the scheduling of virtual CPUs, memory setup, and device I/O (e.g., networking). We therefore cannot simply apply existing isolation and introspection techniques [242, 201]. Instead, 00SEVen builds on hardware-based in-VM isolation mechanisms and adds new secure VMI-assisting hypercalls. Our implementation targets confidential SEV-SNP VMs [3], which are widely available in server CPUs and at cloud platforms (in contrast to, e.g., Intel TDX) and provide primitives for intra-VM isolation [96, 12, 93]. 00SEVen leverages SEV-SNP's intra-VM privilege domains, called VM privilege levels (VMPLs) [3], to protect our in-VM VMI modules (incl. agent) and grant them full VM memory and register access. Our modules run in a bare-metal environment *independent* of the untrusted in-VM OS. 00SEVen deprivileges the in-VM OS (incl. user space services) by placing it in a less privileged VMPL with memory restrictions that protect our VMI against in-VM attackers [4]. 00SEVen ensures that VMI results are shared *only* with the authenticated VM owner. The VM owner can securely send VMI requests to the in-VM agent using a new *attested* end-to-end encrypted network channel. We enhance the QEMU/KVM hypervisor to support binding a virtual channel device and its I/O directly to our agent's privileged VMPL to operate the channel independent of the in-VM OS. Finally, we extend the VM-to-hypervisor interface of SEV with new VMI-assisting hypercalls used by our agent to securely offload sub-tasks to the hypervisor, e.g., to pause the untrusted in-VM OS during a consistent analysis.

We implemented an open-source prototype of 00SEVen (cf. Section 5.11) [S4] with support for the common LibVMI client library. That way, 00SEVen becomes compatible with all analysis scripts and tools building on LibVMI, e.g., Volatility's VMI plugin [222]. Our prototype includes the bare-metal in-VM VMI agent, our extensions to the Linux KVM/QEMU hypervisor, and an extended version of LibVMI usable by the remote clients. Our evaluation shows a reasonable performance and effectiveness based on practical analysis tasks and real-world rootkits.

We regard 00SEVen as a first step towards secure VMI for confidential VMs. Instead of covering all existing state-of-the-art VMI features and optimizations, our focus is on providing solutions for challenges specific to VMI for confidential VMs. We encourage future work to build on top of our foundation to explore new optimizations [51, 242], use future features of SEV-SNP, or transfer our concepts to other platforms such as Intel TDX and Arm CCA [106, 16].

In summary, we make the following contributions:

- We analyze the incompatibilities of *confidential VMs* with existing VMI techniques and derive the resulting challenges imposed on confidential VMI designs.

- We design 00SEVen, a remote VMI for confidential SEV-SNP VMs. 00SEVen's in-VM agent enables clients to control and inspect their VMs *while preserving* security.

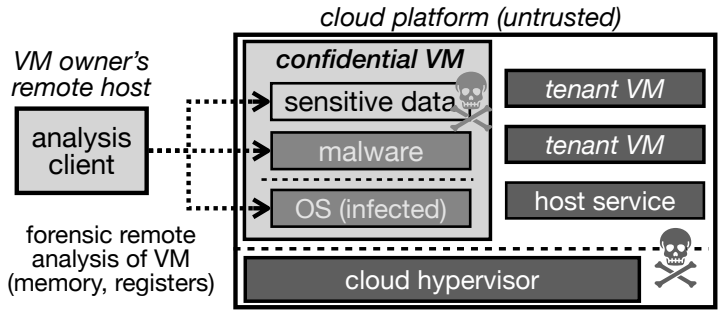

Figure 5.1: A client deploys a confidential VM at an untrusted cloud platform *(dark gray)*. As the VM might become compromised *(mid gray)*, the client wants to perform a remote analysis of the protected VM to scan for attack traces. *(light gray: trusted, dark: untrusted cloud, mid: untrusted in-VM)*

- We implement 00SEVen for KVM/QEMU, including its in-VM agent, hypervisor extensions, and remote client/s.
- We analyze the security of our implementation and evaluate our LibVMI-compatible prototype (open-source) [S4] based on macro-benchmarks and real-world rootkits.

5.4 Setting: Confidential VM In(tro)spection

As shown in Figure 5.1, we envision an organization that wants to *securely* offload services to a third-party cloud platform, e.g., to benefit from the cloud's resource scalability and availability. These services operate on *highly sensitive* data, e.g., health or financial data, customer data, or IP assets (intellectual property). The client ("*VM owner*") therefore decides to deploy these services in confidential VMs, in our case based on AMD SEV-SNP. However, as the VM might become compromised *at runtime*, the VM owner wants to perform a remote forensic analysis to search for attack traces. Memory forensics typically covers the aggregation of the target's memory, the identification of data structures, and finally the analysis tasks [81, 21, 143]. The VM owner plans to augment forensics with *VM introspection (VMI)*, which leverages the hypervisor's control over the VMs to expand memory forensics with additional techniques, including on-demand pausing of the VM, CPU register inspection, and event-based or live analysis [142, 201, 115]. Such runtime attack detection in confidential VMs implies a new threat model, as discussed next.

5.4.1 Threat Model

Our threat model combines aspects of the model of remote VMI for non-confidential VMs and that of confidential VMs. We refer to the former as "classical VMI". We trust the VM owner that deploys confidential SEV-SNP VMs in a third-party cloud and wants to remotely inspect them using a trusted client system *(light gray, Figure 5.1)*. We regard the network path between clients and VMs as untrusted. In contrast to

the classical VMI model, the VM owner does *not* fully trust the cloud platform. We therefore follow the model of confidential VMs in that we assume all system-level software of the cloud platform (incl. the hypervisor) and co-located VMs to be untrusted, e.g., potentially compromised by an attacker *(dark gray, Figure 5.1)*, and trust the cloud's CPU(s) and confidential VM implementation *(light gray)* to be secure and free of exploitable vulnerabilities, i.e., in our case, the SEV-SNP extension and attestation service [3].

We follow a stronger model than confidential VMs regarding the VM security in that we do *not* regard the *whole* confidential VM as trusted. Instead, similar to the classical VMI model, we assume a potential in-VM attacker *(mid gray)* that has compromised the VM OS (e.g., malware, rootkits) that the VM owner wants to remotely detect or analyze [81, 115]. An attacker might have gained control over the VM for instance by exploiting in-VM network services or software supply chain attacks against the VM's package managers.

We assume the in-VM attacker to *be distinct from* the cloud operator and *not* in control of the cloud platform (Figure 5.1). That is, we explicitly exclude collusion attacks of the in-VM attacker *(mid gray)* and the untrusted cloud platform *(dark gray)*. This assumption is in line with the threat model of SEV-SNP and other existing confidential VMs as they do *not* consider in-VM attackers in the first place [3, 106]. We regard our assumption as reasonable in practice, because a collusion attack would significantly increase the chance of an attack attribution to the cloud provider, which would result in a serious reputation loss and lawsuit. Furthermore, we regard a compromise of a specific confidential VM *and* the cloud platform by an external attacker *at the same time* as rather unlikely and prone to detection by the cloud security infrastructure. Nevertheless, in Section 5.7.3, we will briefly discuss the security impact of collusion attacks and the limiting factors of a full mitigation using SEV-SNP.

Finally, we exclude all side-channel and hardware attacks that go beyond the guarantees of SEV-SNP and refer to orthogonal research on these topics, e.g., Cipherfix [226]. Similarly, we regard the *prevention* of denial of service (DoS) attacks issued by the cloud platform as out of scope, because current confidential VMs cannot prevent them either [3]. However, our VMI design will explore if we can detect or limit DoS attacks by the hypervisor against our VMI operations. Furthermore, we will discuss in-VM attackers trying to detect and delay (or even DoS) analysis attempts to hide from them.

5.4.2 Design Goals and Requirements

To guide the design of our secure remote introspection of confidential VMs, called 00SEVen, we define *eight* major design requirements (*R1–R8*) that capture the functional and threat model-specific security demands for a practical solution. In addition, we define two desirable extra goals (*E1+E2*) going slightly *beyond* our threat model achievable with the current hardware support, which we will *not* focus on but still discuss in the chapter.

R1 Remote Memory and Register Access: 00SEVen must provide the VM owner full remote access to the VM's memory and virtual CPU registers (vCPU).

R2 Consistent Analysis 00SEVen must support secure pausing of the VM for a consistent memory and register analysis. Cloud attackers must not be able to resume VM execution without an explicit approval by 00SEVen.

R3 Event Traps 00SEVen must enable event-based analysis by supporting secure traps on VM read/write accesses to monitored memory pages or calls to kernel functions.

R4 Isolation from In-VM OS-level Attackers 00SEVen's VMI components and analysis results must be protected against in-VM OS-level attackers to enable secure analysis of user malware and kernel rootkits.

R5 Isolation from Cloud Attackers 00SEVen's VMI components and analysis results must be protected against the cloud platform (incl. the hypervisor) in line with the threat model of confidential VMs (here: SEV-SNP).

R6 Secure Communication Channel The network communication between 00SEVen's in-VM VMI components and the VM owner must be protected against passive (sniffing), active (tampering), and impersonation attacks by in-VM, cloud platform, and network attackers.

R7 Small TCB 00SEVen should keep the TCB and attack surface small to minimize the risk of a compromise.

R8 Small Overhead on VM Workload 00SEVen should minimize the extra overhead imposed on the confidential VM's workload while no introspection is active.

(E1 Detect Analysis DoS) 00SEVen should enable the VM owner to detect DoS attempts (e.g., scheduling-based) by cloud attackers (cf. *R5*) against VMI operations.

(E2 Hide Analysis from In-VM Attackers) 00SEVen should support hiding incoming remote analysis requests from in-VM attackers, e.g., to prevent attackers from hiding attack traces just in time (cf. *R4*).

00SEVen provides the foundation for secure remote VMI of *confidential* VMs, using the example of AMD SEV-SNP. We encourage future work to build on top of 00SEVen in order to securely explore further VMI features and optimization techniques [51, 242, 115, 77] or transfer our concepts to other platforms, as discussed in Section 5.9.1 for Intel TDX and Arm CCA.

5.4.3 (Un)Applicability of Existing VMI

Existing hypervisor-based VMI systems are inherently in conflict with our threat model. These systems rely on a *trusted* hypervisor to control the target VMs and securely access their memory or register content for the analysis [142]. However, in our setting, the VM owner assumes the cloud platform (incl. hypervisor) to be *untrusted* (cf. Figure 5.1 and *R5*). In fact, the hardware protection of SEV-SNP renders *any out-of-VM* approach unfeasible. SEV-SNP's memory and register protection blocks any cross- or out-of-VM

access attempts, including those by the hypervisor or peripherals (*R1*). SEV-SNP generates and uses a unique de-/encryption key in hardware for each confidential VM to encrypt a VM's private memory pages when storing them in system RAM. All VM code and page tables, as well as data pages marked for encryption in the in-VM page tables, are treated as *private* pages, i.e., are encrypted by SEV-SNP. Access to them only succeeds from *within* the respective confidential VM, thus protecting their confidentiality [7]. In addition, SEV-SNP protects the integrity of private pages and their address mappings. That way, SEV-SNP blocks unauthorized tampering attempts and enforces one-to-one host-to-VM address mappings [3, 155]. Thus, SEV-SNP breaks VMI relying on code injections [98] or page (re)mapping at the hypervisor's nested (second-level) page tables (NPTs), e.g., to access VM memory from a co-located VM [242] or isolate in-VM agents [201]. Furthermore, in contrast to non-confidential VMs, the hypervisor cannot access the vCPU registers of a confidential VM. SEV-SNP has relocated the VM's save areas (VMSAs)—storing the general purpose and control CPU registers on a VM exit—into private VM memory. Therefore, the hypervisor can no longer inspect vCPU registers or modify them (*R1*), e.g., to redirect the VM's control flow to a VMI implant [98]. Instead, if register access is required, the hypervisor requires explicit cooperation by the untrusted VM OS via *shared*, i.e., unencrypted, memory [9].

Existing *in-VM* VMI approaches suffer from functional and security issues. While in-VM forensic tools (cf. Section 5.3) can successfully access the private VM memory, they are *unprotected* against in-VM OS-level attackers. Therefore, in-VM attackers can easily tamper with the tools and their results, violating *R4* and *R6* (Section 5.4.2). In addition, they lack several VMI features, e.g., secure VM pausing for a consistent analysis (*R2*) and VM traps (*R3*). Unfortunately, it is non-trivial to protect and securely extend existing in-VM tools within our threat model. In SEV-SNP (and other confidential VMs), several important resources required by existing VMI systems are still handled by the untrusted hypervisor, which prevents a direct transfer of out-of-VM techniques [242, 98, 201]. The hypervisor controls the scheduling of vCPUs and the second-level memory mappings (incl. permissions) via nested page tables (NPTs). Furthermore, the untrusted hypervisor is still in the sole control of vCPU event interception [7]. Therefore, in-VM tools can neither use NPTs to isolate their memory space from in-VM attackers [201, 98, 242] (*R5*), nor pause vCPUs for a consistent introspection (*R2*), nor directly monitor page accesses or intercept VM executions (*R3*) [83].

5.5 Design of 00SEVen

Next, we present the design of 00SEVen, our remote VMI solution. In Section 5.6, we focus on its implementation details.

5.5.1 Design Overview

As shown in Figure 5.2, 00SEVen combines a secure *in-VM* agent with VMI-specific hypervisor extensions. Together they provide the VM owner with remote analysis capabilities (*R1*), securely overcoming the limitations imposed by SEV-SNP. Our new VMI agent forms 00SEVen's in-VM TCB, while the rest of the VM and the cloud plat-

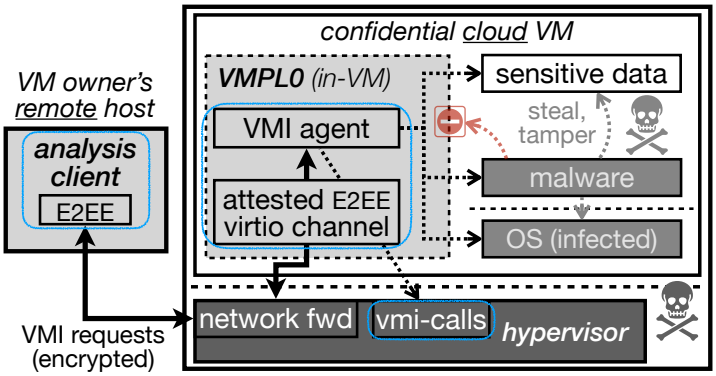

Figure 5.2: Design of 00SEVen: Secure in-VM agents enable remote VMI of confidential VMs. VM owners query the agents via attested end-to-end encrypted (E2EE) network channels. *(light gray: trusted, dark: untrusted cloud, mid: untrusted in-VM)*

form stay untrusted. In order to start a remote VMI session, the VM owner executes a 00SEVen-compatible forensic client application on a trusted system. The client connects via a network (reverse) proxy running at the cloud platform to our agent inside the SEV-SNP VM and uses SEV's hardware-assured remote attestation to authenticate it [5]. The VM owner then uses the client to issue VMI requests to our in-VM agent as required for the forensic analysis. The agent performs the requested VMI operations on behalf of the VM owner, e.g., accessing memory or register content (described in Section 5.5.2), and returns the results. The workflow of 00SEVen will be familiar to users of the common LibVMI [142] framework. However, in contrast to LibVMI, which interacts with a local hypervisor, our client library *remotely* communicates with 00SEVen's securely attested *in-VM agent*, not with the untrusted cloud hypervisor.

In-VM Agent Our in-VM agent forms the core of 00SEVen's VMI. The agent is responsible for processing the introspection commands of the analyst by implementing the respective VMI operations (cf. Section 5.5.2). While SEV-SNP protects the in-VM agent against out-of-VM attackers (incl. the hypervisor), it is crucial for a secure VMI that the agent is also protected against in-VM attackers, e.g., a compromised VM OS ($R4$–$R6$). Therefore, we leverage SEV-SNP's VM privilege levels (VMPLs) for in-VM isolation [3]. VMPLs are orthogonal to the kernel and user mode and enable multiple separate execution contexts per vCPU that share the same physical VM address space. In the current revision, SEV-SNP provides four hierarchically-ordered VMPLs, called VMPL0 to VMPL3, with VMPL3 being the least privileged. For each vCPU, the hypervisor requires one VMSA per VMPL, i.e., up to four separate register sets per vCPU. A vCPU cannot switch between VMSAs (incl. VMPLs) by itself, in contrast to the kernel and user mode. On scheduling, the hypervisor selects a vCPU's current VMSA, i.e., register set and VMPL. However, the hypervisor cannot directly access or modify the VMSAs and their associated VMPLs (cf. Section 5.4.3), and the boot-time VMSAs can be remotely attested. By default, VMPLs are unused and all VM software executes in VMPL0.

We deploy the modules of our VMI agent inside VMPL0, which forms a special

in-VM management domain. VMPL0 provides our agent with full VM memory access (*R1*) and the capability to define per-VMPL memory permissions (read, write, execute) that restrict access of less privileged VMPLs to a memory subrange. In addition, VMPL0 securely manages the VMSA pages (cf. Section 5.4.3) and therefore enables our agent full access to all registers of each vCPU (*R1*). 00SEVen deprivileges the untrusted VM OS and user services inside a less privileged VMPL, following a proposal by AMD [4]. That way, at boot time, our agent can define VMPL memory permissions for the less privileged VMPLs that isolate our agent and the vCPU registers, i.e., VMSA pages, from in-VM attackers (*R4*). For ease of discussion and without loosing generality, we assume the VM OS and services to be relocated only into VMPL1, ignoring the even less privileged VMPL2 or VMPL3. The VM OS running in VMPL1 can still execute every privileged CPU instruction except those restricted to VMPL0. This affects only the `pvalidate` instruction which is used to commit memory pages to the SEV VM at boot time (by default) [8]. Therefore, our VMPL0 agent takes care of the VM memory setup, as detailed in Section 5.6.1. Our agent executes in a bare-metal environment without an OS kernel, *independent of* the potentially compromised VM OS (*R8*). By refusing the use of a full-blown OS inside VMPL0, (in contrast to, e.g., Hecate [83]), we keep 00SEVen's TCB and attack surface significantly smaller (*R7*).

Hypervisor integration We extend the untrusted hypervisor to assist 00SEVen's in-VM agent with scheduling, remote communication, and VM control primitives. Together with new in-VM security checks, that way, our agent can securely enable remote VMI despite the untrusted hypervisor's VM control. During regular workloads, we want the hypervisor to execute the VM OS in VMPL1 without additional overhead by our VMI system (*R8*). On remote VMI requests by the VM owner, we expect the hypervisor to schedule our VMPL0 agent to perform VMI operations. However, existing hypervisors (e.g., KVM) do not yet distinguish between different VMPLs when scheduling vCPUs or delivering I/O events. We therefore extend the hypervisor with *VMPL-aware* scheduling and I/O operations. That way, we can bind a virtual I/O device for our VMI remote channel exclusively to VMPL0, i.e., our agent's domain. The channel device demands scheduling of VMPL0 from the hypervisor on VMI requests and permits channel I/O only by our agent. All other I/O devices (e.g., disk, NIC) stay associated with the VM OS in VMPL1, keeping their performance unaffected by our agent (*R8*). When finishing the requested VMI operations, our agent hyper-calls into the hypervisor to re-schedule VMPL1 execution. In Section 5.5.2, we will present 00SEVen's hypervisor interfaces for VMI control primitives: VM pausing and trapping.

Secure client-to-agent communication The VMI channel between 00SEVen's in-VM agent and the VM owner's remote client requires additional protection. In the current SEV-SNP design, all hardware device I/O must pass through the untrusted hypervisor. Therefore, we must rely on a packet forwarding service at the hypervisor-level to forward our VMI channel messages via the network to the remote client (cf. Figure 5.2), affecting their security (*R6*). In order to protect our VMI messages, the remote client and the in-VM agent use an end-to-end encrypted (E2EE) connection (*R6*). Otherwise, attackers could tamper with the messages to hide attack traces or

leak private VM memory by exploiting inspection requests. 00SEVen's connection endpoint is isolated from out-of-VM (cloud, network) and in-VM attackers by placing the protocol stacks directly inside VMPL0—letting packets leave VMPL0 only in E2EE form. As the agent operates only on the (E2EE) application-layer messages rather than full network packets, we preserve a small in-VM TCB (*R7*). Being located in VMPL0, the VMI channel operates independent of the untrusted VM OS in VMPL1 (*R8*)—not passing any packets through the VMPL1 network stack. To prevent impersonation attacks, we combine certificates with SEV-SNP's remote attestation for mutual authentication, as discussed in Section 5.6.3.

5.5.2 VMI Work Flow

We now describe 00SEVen's VMI (*R1*) and its hypervisor extensions for secure VM pausing (*R2*) or event traps (*R3*).

5.5.2.1 Modus Operandi

00SEVen supports multiple forms of remote analysis triggers and modes. A typical use case is the scanning for attack traces or active malware by the VM owner as part of an incident response process, e.g., triggered by an intrusion detection system or as part of a periodic security check. Beyond manually or periodically triggered analysis, 00SEVen re-enables more advanced use cases by supporting event-based triggers based on page access monitoring or code execution traps (cf. Section 5.5.2.5). That way, 00SEVen can notify the client-side analysis script, e.g., if an in-VM attacker tries to tamper with a memory page of a sensitive service (*R3*). 00SEVen's main analysis mode then enables the remote analyst (VM owner) to perform a *consistent* analysis of the VM by providing secure pausing of the untrusted VMPL1 services, i.e., the VM OS and user processes (cf. Section 5.5.2.4). In contrast to existing forensic tools that require downloading a full VM memory dump for a client-side offline analysis, 00SEVen enables interactive and selective remote memory and register introspection of the VM state and thus better scalability by avoiding gigabyte-size memory dump transfers. Depending on the analysis results, the VM owner can then quarantine or resume the VM. Even though not being our focus, 00SEVen also has limited support for live memory analysis, i.e., without pausing VMPL1 execution, as we will describe in Section 5.5.2.6.

5.5.2.2 Remote VMI Interface

00SEVen's remote interface enables flexible analysis tasks by the VM owner. 00SEVen's in-VM agent implements fundamental operations required for VMI and exposes them via an RPC-like interface to the remote client. These operations form the basis for high-level VMI tasks by providing memory and register access (cf. Section 5.5.2.3). Furthermore, the agent exposes secure VM control primitives for pausing (cf. Section 5.5.2.4) or trapping (cf. Section 5.5.2.5) the VM. That way, analysts have full control of the VMI and can implement flexible analysis scripts tailored to their use cases. Our operations are similar to the features of the common LibVMI framework [142], which makes it possible to adopt many existing LibVMI client analysis scripts and tools built on it,

```
1  connect_to_agent(target_vm)
2  pause_vmpl(OS) // consistent, like LibVMI
3
4  task_entry = ksym_va("init_task") + tasks_offset
5
6  // scan the list of processes
7  while (true) {
8      proc = task_entry - tasks_offset
9      exec = read_str_va(proc + comm_offset)
10     if (exec == "malware") { ... }
11     ...
12 }
13
14 resume_vmpl(OS) // VMPL1 OS
15 disconnect_from_agent()
```

Figure 5.3: Simplified excerpt of a client script remotely scanning the process list of a Linux SEV-SNP VM for malware. *(_va ≙ VM virtual address, comm ≙ command name)*

e.g., Volatility's VMI plugin [222]. As demonstration, our current remote analysis client is based on LibVMI. We extend LibVMI with a new 00SEVen driver, i.e., API backend, which sets up the attested E2EE connection to our agent and sends VMI operation and VM control requests to our agent, instead of interfacing with a local hypervisor.

Figure 5.3 shows a simplified client script for remotely inspecting the process list using 00SEVen. In the preamble (lines 1+2), the VM owner's client connects to 00SEVen's in-VM agent and requests secure pausing of the VM OS for a consistent analysis (cf. Section 5.5.2.4). Afterwards, the client resolves symbols to get the address of the process list of the Linux-based VM OS (line 4) and issues multiple memory read requests to 00SEVen's agent to scan the list (lines 7–12). The agent receives the requests via the E2EE channel and performs the respective memory accesses inside the VM. As we will discuss in Section 5.5.2.3, the client might perform some steps locally to speed up the analysis process, e.g., by caching page table information [51, 142]. Finally, in the epilogue (lines 14+15), the client resumes the VM execution if no malware has been found and disconnects from the agent.

5.5.2.3 VMI Operations

We now outline the basic VMI operations 00SEVen supports.

Physical Memory Access 00SEVen's basic memory access operation takes a physical memory address of the VM (PA_{vm}) as input. Our agent maps the PA_{vm} and then uses the resulting virtual address (VA_{vmpl0}) to access the page, returning the requested content to the remote analyst. In contrast to out-of-VM VMI, there is no need to explicitly translate the VM address to a host-level virtual address, as it will be automatically handled by the hardware on the in-VM access. Before each access, the agent must check that the requested physical address range does not overlap with the

VMPL0-exclusive memory region containing our agent's code and data (Section 5.6.1). Otherwise, in-VM attackers might maliciously modify kernel pointers to let them point into the VMPL0 region to cause a corruption of 00SEVen on incautious VMI write requests. Furthermore, the agent must ensure that the PA_{vm} is correctly mapped as private (encrypted) or shared page as registered in SEV-SNP (Section 5.4.3). This information must be encoded as a bit in each PA_{vm} but might be unavailable or untrusted when the PA_{vm} is taken from VMPL1. As VMPL0 registers ("validates" [3]) private pages in SEV-SNP, the agent can keep track of each page bit to correctly map them, requiring only 1 MiB for a VM with 32 GiB RAM and 4 KiB page size. Optionally, the agent can pre-map all pages linearly ($VA_{vmpl0} = PA_{vm} + \text{offset}$) for a direct VA_{vmpl0} lookup (calculation) rather than a slower on demand mapping and/or page table walk.

Virtual Memory Access For accessing virtual kernel or process addresses of the VMPL1-located VM OS, 00SEVen requires an address translation step. The VMPL0 agent uses separate page tables (PTs) that are isolated from the untrusted VM OS to prevent malicious remapping attacks by in-VM attackers. For accessing a virtual address of the VM OS (VA_{vmpl1}), we therefore must translate the VA_{vmpl1} to a PA_{vm} before the agent can access it via a VA_{vmpl0}, as described before. The translation requires a (software-based) page table walk through the respective PTs of the VM OS [143]. Depending on the virtual address space of the VA_{vmpl1}, the physical address (PA_{vm}) of the respective root PT (directory table base) can be located using different existing methods [143]: for kernel VAs based on the Linux 'init_top_pgt' kernel symbol, for process VAs inside the OS process list (Figure 5.3), or for the current address spaces in the `CR3` registers accessible in the vCPUs' VMSAs (Section 5.4.3). We refer to the forensic literature for more details, e.g., [143].

With our remote client being based on LibVMI (Section 5.5.2.2), the client performs kernel symbol translations locally, e.g., based on the symbol table of the compiled VM Linux kernel [177]. For the page table walks, the client issues the respective physical memory and `CR3` read requests to our agent as required for the $VA_{vmpl1} \Rightarrow PA_{vm}$ translation. After translation, the agent can then map and access the target virtual address via the resolved PA_{vm}.

Client-side Memory Caches 00SEVen's remote client adopts LibVMI's client-side caching [177] to enhance the memory access performance. We leverage LibVMI's page-level data caching to avoid additional network overhead for follow-up accesses to the same physical VM page, e.g., when reading multiple page table entries or fields of a structure [177, 142]. That is, the client requests whole page reads (4 KiB) of the agent and caches $\{PA_{vm} \rightarrow \text{page buffer}\}$ mappings, i.e., from a physical page address to the respective page content, in order to enable client-local follow-up accesses to the same memory page. That way, we avoid extra communication overhead. Furthermore, we accelerate the address translation process ($VA_{vmpl1} \rightarrow PA_{vm}$) required when accessing virtual addresses of the VMPL1 VM OS. LibVMI supports multiple related client-side caches which maintain $\{\text{kernel symbol} \rightarrow PA_{vm}\}$[2], $\{VA_{vmpl1} \rightarrow PA_{vm}\}$, and

[2] actually $\{\text{ksym} \rightarrow VA_{vmpl1}\}$, but PA_{vm} is statically derived for Linux [143]

{process ID $\rightarrow PA_{vm}$(root PT)} mappings and thus decrease the number of required VMPL1 page table walks and remote memory access requests. In Section 5.9.2, we will outline future directions to accelerate 00SEVen's virtual memory accesses even further by proposing new agent-side offloading and access optimizations strategies.

Virtual CPU Register Access 00SEVen's agent securely manages the register save states (VMSAs) of the vCPUs. The VMPL0 setup code allocates and registers one VMSA page per VMPL for each vCPU [9]. When a vCPU of an SEV-SNP VM is yielded, the CPU stores the general purpose, control, and virtualization registers of the vCPU in the respective private VMSA page (cf. Section 5.4.3). Therefore, our agent can directly inspect the register state of paused vCPUs (Section 5.5.2.4). To prevent in-VM attackers from tampering with vCPU registers, our agent protects VMSA pages using VMPL permissions.

5.5.2.4 Secure Pausing for a Consistent Analysis

00SEVen supports secure pausing of in-VM attackers for a consistent memory and register introspection (*R2*). That is, 00SEVen can pause in-VM attackers such that they cannot tamper with the memory or register content anymore, which stops data manipulation attacks during the analysis, e.g., hiding of attack traces. In contrast to non-confidential VMI, which trusts the hypervisor to fully pause all vCPUs during the analysis, 00SEVen must follow a new approach. 00SEVen must not fully stop vCPU execution because the in-VM agent must still perform the analysis. Instead, only the execution of the untrusted VM OS and user space services should be paused, i.e., the VMPL1 domain of the vCPUs. Furthermore, in our threat model, 00SEVen cannot trust the hypervisor to keep VMPL1 paused throughout the analysis. Therefore, we temporarily disable virtualization support in the `EFER` CPU control registers of all VMPL1 VMSAs to prevent their execution while performing the analysis in VMPL0. First, we extend the hypervisor with two new hypercalls from VMPL0 to the hypervisor—one to request pausing of all VMPL1 contexts, yielding them if active, and one for resuming them. Second, on a pause request, we let our VMPL0 agent iterate all VMPL1 VMSAs, i.e., saved register states (Section 5.4.3), and atomically unset the virtualization-enable CPU register bit `EFER.SVME`. If the register updates succeed, VMPL1 has been paused by the hypervisor, and we have *locked* VMPL1 execution, i.e., all attempts by the hypervisor to resume it will be blocked by the CPU. If the hypervisor ignores a pause request (DoS attempt), the register updates will fail as the VMSAs are not paused [4], causing the agent to retry. The remote analyst will detect the attack, as the analysis will not proceed (*E1*). On a resume request, the agent re-enables all VMPL1 VMSAs by setting their `EFER.SVME` and requests their scheduling by the hypervisor. As shown in Figure 5.3 (lines 2 and 14), we expose pause and resume APIs to the remote analyst.

5.5.2.5 Event-based VMI

00SEVen provides support for event-based VMI (*R3*): SEV-enabled memory access traps and kernel function traps.

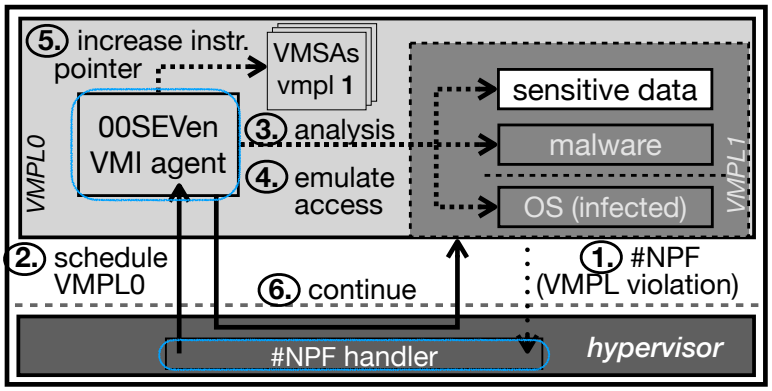

Figure 5.4: 00SEVen's VMPL0 agent combines VMPL permissions with instruction emulation for secure page monitoring. *(light gray: trusted, dark: untrusted cloud, mid: untrusted in-VM)*

Page R/W-Monitoring 00SEVen enables the remote analyst to request the monitoring of read and/or write accesses to private VM pages. That way, the analyst can for instance write-monitor a function pointer table in the VM OS, e.g., the system call table, to detect malicious tampering attempts by a rootkit. Existing VMI implements page monitoring using NPT permissions which we cannot trust [227]. Instead, to monitor a page, our agent securely modifies the VMPL1 memory permissions for that page to non-readable or non-writable using SEV's `RMPADJUST` CPU instruction, restricting access of the VM OS. On an access trap, the agent informs the analyst and waits for VMI requests while keeping the VM OS paused.

Figure 5.4 shows 00SEVen's control flow on an access trap. VMPL1's access is blocked by the VMPL permission and results in a nested page fault (NPF) at the untrusted hypervisor [7] (*step 1*). We extend the hypervisor's #NPF-handler to forward such VMPL violations to our agent by scheduling VMPL0 (*step 2*). The agent inspects the #NPF details (given in the vCPU's VMSA [7]) and, if a violation is associated with a monitored page, securely pauses the vCPU's VMPL1, notifies the remote analyst, and waits for analysis requests (*step 3*). After the analysis, our agent must proceed execution of VMPL1. Existing hypervisor-based VMI grants VMPL1 *temporary* page access and uses hardware single-stepping to securely perform the trapped access [227]. However, only the untrusted hypervisor can intercept the single-stepping exception, and the hypervisor could simply resume VMPL1 execution without informing our agent, resulting in relaxed VMPL permissions, i.e., a disabled trap. Therefore, instead, our agent reads VMPL1's instruction pointer register (RIP) to decode and emulate the violating memory access, i.e., performing it on behalf of VMPL1 (*step 4*) [135]. That way, the VMPL restrictions are never relaxed such that we do not risk disabled monitoring traps. During emulation, our agent translates the VMPL1 memory address using the vCPU's VMPL1 PTs, checks that it does not overlap with VMPL0 memory, and then maps it temporarily into VMPL0. We must extract the page offset of the target address from the instruction itself, as SEV masks the offset in the #NPF details for security reasons [139, 228]. Finally, after access emulation in VMPL0, the agent updates the VMPL1 registers in the VMSA, including the RIP to step over the emulated instruction

```
1 trap:
2     jmp <fct>   ; nop when trap is enabled
3     wrmsr <MSR_GHCB>, 0x200   ; info for KVM
4     mov rax, 4711   ; argument for VMPL0 agent
5     vmgexit   ; hypercall to schedule VMPL0
6     xor rax, 1234   ; check return value ...
7     jnz <trap>   ; ... by agent
8 fct:
```

Figure 5.5: Disabled VMPL1 kernel function trap trampoline of 00SEVen (simplified).

(*step 5*), clears the #NPF to prevent a fault replay [83], and calls into the hypervisor to resume execution (*step 6*).

The monitoring is protected against the untrusted hypervisor and in-VM attackers. VMPL0 has exclusive control over the VMSAs and VMPL permission. That is, only VMPL0 can access the vCPU registers of VMPL1 in the respective VMSA to perform an instruction emulation, and only VMPL0 can modify the VMPL1 permissions of the trap using the `RMPADJUST` CPU instruction [3]. Therefore, neither the hypervisor nor VMPL1 can resolve the trap, and the hypervisor must schedule 00SEVen's agent to successfully resume the trapped instruction. If the hypervisor instead directly re-schedules VMPL1, the NPF will be re-raised. Attempts to directly modify the SEV-SNP page attributes, e.g., making a private page shared or trying to directly modify the VMPL1 permissions, will either result in page data corruption or in the page becoming invalid for the VM, requiring a re-registration ("validation" [3]) only resolvable by VMPL0. The only option left for the attackers would be to try skipping the trapped memory access entirely, e.g., by modifying the trapped instruction. However, 00SEVen could prevent this by making the code page non-writable using VMPL1 permissions.

Kernel Function Traps Conceptually, 00SEVen also supports trapping the execution of VMPL1 kernel code. That way, an analysis can be triggered on execution of a certain VM OS function, e.g., a system call [229]. While we could adapt our page read/write-monitoring idea to trap page execution by marking pages as non-executable in the VMPL1 permissions, this approach would introduce significant emulation complexity. On a read/write-trap, 00SEVen's VMPL0 agent must emulate a single memory-accessing VMPL1 instruction to resume VMPL1. However, on a code execution trap, the agent would need to emulate all instructions located at the monitored memory page, including all kinds of control flow instructions, e.g., function calls. Alternatively, we could fall back on hardware single-stepping, but as discussed before, we then could not guarantee that our agent can re-enable the trap afterwards.

Instead, 00SEVen can inject VMPL0 trampolines at the beginning of VMPL1 kernel functions using compiler-assistance, similar to how ftrace is implemented in Linux [190] or hot patching in Windows [231]. The injected code (cf. Figure 5.5) serves as pseudo-breakpoints that, if enabled by our agent, call into VMPL0 for analysis. We inline a loop that performs a hypervisor call (`VMGEXIT` instruction) telling the hypervisor to schedule our VMPL0 agent and a check of the return register. SEV-SNP provides

communication interfaces between VMs and the hypervisor based on the GHCB specification (Guest-Hypervisor Communication Block) [9] using a MSR (model-specific register) or dynamically allocated shared memory. 00SEVen's trampoline passes the scheduling request number to the hypervisor using the statically-known GHCB MSR interface (line 3), which is compatible with static, compiler-assisted trampoline injections. After the agent's analysis, the agent sets the return register inside the VMPL1 VMSA of the trapped vCPU in order to confirm a successful trap handling to the trampoline. Otherwise, the loop retries the call to prevent the hypervisor from ignoring our scheduling request (lines 6+7), similar to how AMD SVSM handles service calls [4]. That way, we can reliably inject code execution traps. While we cannot hide our injected code, we can reliably set the VMPL1 permissions as non-writable to prevent tampering by in-VM attackers, because all code pages are treated as private VM pages in SEV-SNP and are therefore affected by VMPL permissions (cf. Section 5.4.3). The injected code is disabled by default by a prepended jump instruction skipping the call loop (line 2; *R8*). The remote analyst can select the functions to be monitored, and our agent will enable the respective trampolines by replacing their initial jumps in memory. However, note that the context switches between VMPL1 and VMPL0 through the hypervisor cause non-negligible overhead, limiting frequent execution tracing (cf. Section 5.9.4).

5.5.2.6 Live Analysis

Conceptually, 00SEVen supports live introspection for actively monitoring the memory state of a VM. To perform live analysis, the remote analyst would simply skip pausing the VM OS (Figure 5.3, line 2). That way, the remote analyst can perform its memory analysis without requiring the VM OS and user space services in VMPL1 to pause execution, enabling introspection of their memory changes at runtime. However, live analysis has multiple implications: vCPU register live inspection is not possible (cf. Section 5.5.2.3)—this limitation is shared with existing non-confidential VMI. In addition, shared with LibVMI [142], entries of translation and data caches (cf. Section 5.5.2.3) might become stale and the slower software-based virtual address translation might not catch up with fast memory changes [242], which might allow for undetected data or page table manipulations hiding attack traces.. In order to address stale cache entries, 00SEVen's agent could use page write-traps (Section 5.5.2.5) to actively monitor and detect page changes by the VM OS (e.g., page table entries) to update our caches—similar to [51]. However, in SEV-SNP, we still cannot intercept swaps of the active page table (`CR3`) without depending on the untrusted hypervisor (Section 5.4.3), potentially missing attacker-hidden ones [242]. While an alternative approach could be to try accessing virtual VM OS addresses (VA_{vmpl1}) directly without software translation to a VA_{vmpl0}, existing approaches are not applicable to 00SEVen due to threat model violations and restrictions by SEV-SNP, as explained in Section 5.9.2 (also cf. Section 5.5.2.3). Therefore, as described in Section 5.5.2.1, 00SEVen's current focus is on paused consistent analysis triggered manually or by page traps. Designing a secure solution for consistent live analysis in SEV-SNP is non-trivial, and we leave it for future work. If future SEV versions provide VMPL0 with support for securely intercepting VM exits (e.g., writes to `CR3`), a consistent live analysis design for 00SEVen might become easier to achieve, as discussed in Section 5.9.4.

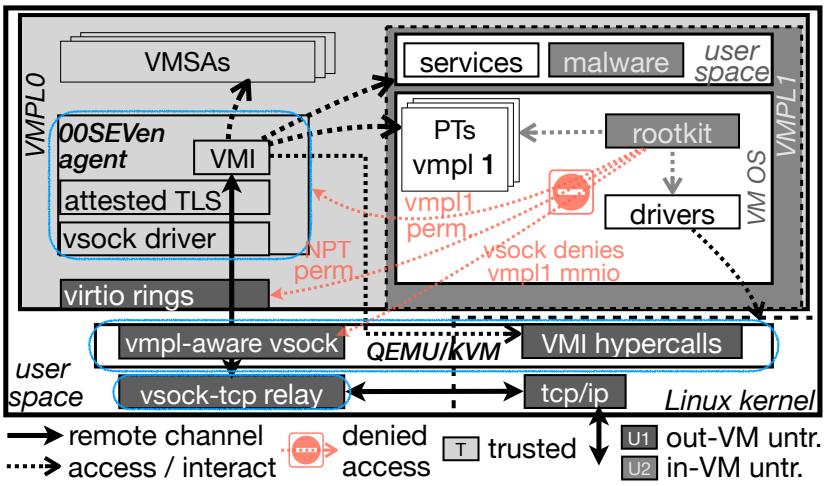

Figure 5.6: 00SEVen's implementation: an in-VM VMI agent, a network relay, and VMPL-aware QEMU/KVM extensions *(blue)*. *(light gray: trusted, dark: untrusted cloud, mid: untrusted in-VM)*

5.6 Implementation

We now describe details of our 00SEVen implementation for the QEMU/KVM hypervisor and an SEV-SNP VM with Linux OS.

5.6.1 Agent Integration and Startup

00SEVen's in-VM agent and its modules are an extension to AMD's secure virtual service module (SVSM) infrastructure [4]. SVSM provides a bare-metal VMPL0 environment written in Rust, which handles the vCPU (VMSA) and memory setup that is specific to SEV-SNP. In particular, as the VM OS is deprivileged inside VMPL1 and thus cannot execute the `pvalidate` instruction anymore (cf. Section 5.5.1), SVSM exposes service APIs to VMPL1 that re-enable the VM OS boot code to request the registration of memory pages to the VM on startup. 00SEVen's agent uses SVSM's memory and VMSA management facility to provide remote analysts with our secure VMI and communication channel infrastructure. For the SVSM, the QEMU/KVM hypervisor partitions the physical VM memory into a large VMPL1 region and a VMPL0-exclusive region at the top of the guest memory. The hypervisor maps the 00SEVen-extended SVSM into the VMPL0 region and the regular BIOS/UEFI into the VMPL1 region. On startup, the SVSM sets up the vCPUs and VMPL memory protections of the VMPL0 region, starts 00SEVen's VMI agent, and finally transfers control to VMPL1 for the Linux boot process. Figure 5.6 shows our implementation, excluding SVSM's modules.

During startup, 00SEVen's agent prepares the network communication with the remote analyst. The agent first initializes the dedicated virtual channel device with the hypervisor and then sets up a server socket that asynchronously listens for a remote connection by the VM analyst. On a successful connection, the agent starts a VMI session controlled by the remote analyst's operation requests, as described in Section 5.5.2.

5.6.2 Channel Device and Scheduling

We extend QEMU/KVM's VM setup process to prepare 00SEVen's VMPL-aware remote channel (Section 5.5.1). We adapted QEMU to allocate a dedicated virtual MMIO (memory-mapped I/O) page next to the VMPL0 memory region and associate a virtual MMIO-based I/O bus with it (virtio-mmio [165]). We bind the bus to VMPL0 (cf. next) and attach a virtual socket device (vsock) [165] to it as our remote channel device. A vsock device enables hypervisor services—in 00SEVen's case, a network relay—to connect to a socket-like interface in the VM and exchange messages without requiring complex network stacks (*R7*). We ported the required virtio drivers [221] into VMPL0 (Figure 5.6) and implemented SEV-SNP support for them in line with SEV's shared memory-based GHCB interface [9] (Guest-Hypervisor Communication Block), such that our agent can interact with the channel device via read and write GHCB requests to the MMIO page.

VMPL-aware MMIO We extend QEMU/KVM to support VMPL-aware virtual MMIO devices. On each virtual MMIO access by a VM, QEMU virtio-mmio devices can now inspect the accessor's VMPL and deny the access based on that. 00SEVen uses this mechanism to bind its remote channel device to VMPL0 (i.e., our VMI agent) by making its bus permit MMIO operations only by VMPL0, preventing any MMIO access by VMPL1 attackers. On a VM exit of a vCPU, our extension augments KVM's `kvm_run` structure ($\widehat{=}$ interface to QEMU) with the current VMPL of that vCPU. Furthermore, we extend QEMU's MMIO device callbacks to propagate the VMPL as a new access attribute to the MMIO target device.

VMPL-aware Scheduling As described in Section 5.5.1, we extend QEMU/KVM with VMPL-aware scheduling requests. That way, 00SEVen's channel device can demand explicit scheduling of VMPL0 on VMI requests by the remote analyst, such that the agent can read the channel and process the requests (Section 5.5.2). On a scheduling request, QEMU/KVM yields the vCPU, lets it switch from its VMPL1 register set (VMSA) to its VMPL0 set, and resumes the vCPU. Note that by default, a vsock device would instead inject an interrupt (IRQ) into the VM on a new message. However, QEMU/KVM does not yet support VMPL-aware IRQ delivery. Furthermore, in SVSM, VMPL0 executes with masked IRQs and enabled SEV-SNP restricted injection mode which prevents hypervisors from arbitrarily injecting interrupts into VMPL0, decreasing 00SEVen's attack surface [3]. In fact, when scheduling VMPL0, there must be no event injections by the hypervisor into the VM, e.g., device interrupt requests (IRQ) or non-maskable interrupts (NMI). Otherwise, SEV-SNP's restricted injection mode will cause the CPU to raise a hardware error when trying to resume VMPL0 execution. Therefore, we implement the following additional steps in QEMU/KVM: On a VMPL0 scheduling request, we temporarily disable NMIs for the vCPU in the hypervisor. Furthermore, we check for a pending IRQ injection for the VM OS by other devices. If so, we stash the pending (VMPL1) IRQ in a new field inside KVM's vCPU data structure. On the back-schedule to VMPL1, KVM then restores the stashed IRQ to deliver it to the VM OS and re-enables NMIs for the vCPU. That way, we prevent

NMIs (e.g., caused by Linux's CPU stall detection) and pending (device) IRQs from causing errors or getting dropped (potentially causing a vCPU hang). Note that there will be no new IRQ injection attempts during VMPL0 execution beyond the stashed one as VMPL0 executes with masked IRQs.

5.6.3 Attested Remote Communication

The remote communication of 00SEVen's agent with the VM analyst is built on top of the vsock channel. 00SEVen uses the channel to pass their messages via shared virtio rings between VMPL0 and a network relay service at the hypervisor which forwards them via TCP/IP through the network, as shown in Figure 5.6. To protect the messages, we integrate a TLS server endpoint into our agent and a TLS client endpoint into the remote analysis client. That way, the communication is end-to-end protected against in-VM and out-of-VM attackers even though it passes through the untrusted hypervisor. Technically, the agent and client exchange VMI requests and results (Section 5.5.2) using two separate transport layers: VSOCK for the agent–relay interconnect and TCP/IP for the relay–client one.

Network Relay The relay is a Linux user space service that manages a VSOCK client and a TCP server socket (e.g., socat). The relay waits for an incoming remote client TCP/IP connection by the VM analyst. On a connection, the relay connects to our in-VM agent via the vsock channel device and then starts forwarding packet payloads, i.e., TLS messages, between the agent and remote client. The relay listens on a per-VM dummy network interface that is configured on VM launch and can be interconnected via a bridge interface to the NIC of the cloud server. That way, we can (dynamically) assign the VMPL0 remote channel of each VM a dedicated IP address distinct from the IP of the untrusted VMPL1 OS and expose the channel via the Internet to the VM analyst.

Authentication and Attestation 00SEVen's TLS channel combines mutual certificate-based authentication with AMD SEV's remote attestation [5]. We use a client TLS certificate that is pinned by 00SEVen's agent to verify the VM owner's remote client, e.g., shipped to the agent inside an encrypted VM disk image. To enable authentication of the agent, we bind the TLS connection to the hardware-assured attestation report of SEV. The attestation measurement covers the VM's load-time state including 00SEVen's SVSM image with all agent modules. In addition, the SEV hardware adds the VMPL of the report-generating VM component to the attestation report. Therefore, the VM owner can remotely verify that it is in fact communicating with the VMPL0-protected agent of the owner's VM, not an attacker-controlled VM or an impostor agent in VMPL1. To bind the TLS connection to the attestation, the agent adds the hash of a fresh TLS server public key to the attestation report and sends the report via TLS to the client [128] (similar to SENG's and FeIDo's approach for Intel SGX, cf. Chapter 2 and 4). The client verifies the binding by checking if the keys in the agent's TLS certificate and the report match.

5.6.4 VMI-assisting Hypercalls

00SEVen adds new hypercalls to QEMU/KVM that securely support the agent's VM control primitives (cf. Section 5.5.1). The hypercalls are commands without arguments: pausing or resuming the VMPL1 contexts (Section 5.5.2.4), switching back to VMPL1 after an r/w-trap (Section 5.5.2.5) or to VMPL0 on a function trap (Section 5.5.2.5). The calls are implemented as new GHCB requests, i.e., extend SEV's facility for guest-to-hypervisor communication [9]. The VMPL switches change a vCPU's active VMSA (Section 5.5.1). On a VMPL1 pause request, we prevent the vCPU/s running the VMI agent from switching back to VMPL1 and let QEMU pause (later resume) all other vCPUs.

5.7 Security Analysis

We now summarize the security measures of 00SEVen and then discuss collusion attacks by the hypervisor and in-VM attackers, which are beyond our and SEV's threat models.

5.7.1 Adversary and Goal Recap

00SEVen's threat model (Section 5.4.1) covers two non-colluding types of adversaries: in-VM and out-of-VM attackers. The in-VM attackers are located inside VMPL1 and fully control the VM OS, including all user space services. Their main goal is to evade detection by 00SEVen, e.g., by compromising the agent. The out-of-VM attackers include the system software of the cloud platform hosting the confidential VMs (incl. hypervisor) and network attackers—the latter being a subset of the stronger cloud attackers. Their goal is to gain access to the private memory and register values of the SEV-SNP VMs or their VMI results, e.g., by tampering with VMI operations.

00SEVen's main goal is to enable secure remote VMI of the VMs while preserving SEV-SNP's security guarantees (Section 5.4.2). That is, 00SEVen protects the VMI operations and their requests from both types of attackers and enables the detection, prevention, or analysis of in-VM attacks.

5.7.2 00SEVen's Security Design

00SEVen's VMPL0-located agent forms its in-VM TCB. The agent's security is built on top of SEV-SNP's hardware-enforced memory and register protection [3]. Inside VMPL0, the agent is protected against out-of-VM cloud attackers and can leverage VMPL permissions to block access attempts by in-VM VMPL1 attackers. As SEV VMs rely on support by the hypervisor, the same holds for 00SEVen. Beyond scheduling and memory setup, 00SEVen offloads new VM control tasks to the hypervisor via new hypercalls (e.g., pausing, VMPL switch). However, the hypercalls expose only a minimal attack surface, and 00SEVen is designed to actively prevent malicious hypervisor behavior on these tasks (e.g., VMPL1-locking on pausing) or remotely observe it as anomalous VMI freezes. 00SEVen's pausing of VMPL1 can therefore securely enable consistent memory forensics, i.e., in-VM attackers cannot manipulate or remap any data during the analysis, thus preventing attempts to hide attack traces. Live analysis

can instead use memory traps to actively block suspicious data or PT changes (Section 5.5.2.6). On memory introspection, the range and C-bit checks of the VMPL0 agent (Section 5.5.2.3) additionally rule out attacks that map pages or pointers to VMPL0 or unprotected memory. The network forwarding of 00SEVen's remote channel is also offloaded to the hypervisor, but the communication is TLS-protected and authenticated using certificates and SEV's remote attestation. Therefore, neither out-of-VM nor in-VM attackers can tamper with or leak VMI operation requests or results. The channel's interface to the untrusted hypervisor is based on MMIO-based virtio [165] such that it exposes a minimal attack surface—in contrast to PCIe-based device I/O.

00SEVen currently relies on cooperation by the untrusted hypervisor to protect MMIO and shared pages against access by in-VM attackers, because SEV-SNP's VMPL permissions are enforced only for *private* VM pages [7]. 00SEVen's QEMU/KVM extension for VMPL-aware MMIO blocks MMIO accesses by VMPL1 to the channel device to prevent reconfiguration attacks. Optionally, the device configuration could be write-protected after device setup. 00SEVen's agent requires shared pages for the channel's virtio rings and for GHCB buffers used to perform hypercalls and MMIO requests (Section 5.6.2). In-VM attackers might tamper with these pages to change GHCB requests or perform DoS attacks against the remote channel. In addition, they might try to detect if a new analysis is pending by observing virtio ring changes (*E2*). That way, attackers could try to hide attack traces just before an analysis in order to evade it. In Section 5.9.3, we describe an optional hypervisor extension that enables per-VMPL permissions for shared pages using NPTs. However, note that the channel is TLS-protected and 00SEVen detects misbehaving hypercalls, limiting attacks against shared pages. Furthermore, in-VM attackers cannot trap page accesses by VMPL0 or the hypervisor but rather need to continuously scan all shared pages for changes to time an attack—increasing their risk of detection. Finally, as soon as VMPL1 has been paused for an analysis, VMPL1 can no longer interfere with any buffers (*R2*).

5.7.3 Beyond 00SEVen: Collusion Attacks

00SEVen's design excludes collusion attacks between the untrusted cloud platform (incl. hypervisor) and in-VM attackers (cf. Section 5.4.1). This assumption is in line with the threat model of SEV-SNP, which does not yet cover in-VM attackers at all [3]. Still, we now briefly discuss the theoretical impact and mitigation issues of collusion attacks.

The biggest risk of collusion attacks for 00SEVen are attempts to delay the VMI until all traces of an in-VM attacker have been erased. As the untrusted hypervisor is in control of the vCPU and VMPL scheduling, the hypervisor can delay scheduling of our agent and warn in-VM attackers of a pending VMI request message. That way, the in-VM attackers gain time to finish their attack and delete their attack traces to prevent detection or analysis. While 00SEVen's secure pausing locks the VMPL1 contexts, such that the hypervisor cannot resume their execution until the VMI has finished, there is an exploitable time window between the pausing request and the locking of the VMPL1 contexts (cf. Section 5.5.2.4). The root cause of this is SEV-SNP's reliance on the hypervisor to switch into VMPL0 and yield the VMPL1 contexts

Table 5.1: VMI policies and their targets, adopted from (147).

P1. process list	P6. process memory map
P2. escalated privileges	P7. keyboard sniffers
P3. Virtual File System hooks	P8. module list
P4. TTY keyloggers	P9. TCP4 "netstat-ops"
P5. syscall table hooks	P10. open files

before they can be locked. The hypervisor's platform control also makes it hard to prevent all communication between the hypervisor and VMPL1. Even if some direct channels could be blocked, e.g., shared pages via vTOM (cf. Section 5.9.3), there are several ways to create other (covert) channels not controllable by VMPL0, e.g., based on timed scheduling or VM exit events.

However, the cloud and in-VM attackers must be careful to not risk detection of collusion attacks. For instance, while small analysis delays might be hidden in the network jitter of the remote channel, too many or long delays could be detected by the client. Furthermore, hypervisor-VMPL1 interactions might leave new memory traces detectable via VMI. So even if collusion attacks are possible to DoS or delay an analysis, they increase the risk of detection. In addition, memory traps can still prevent malicious VMPL1 read/write accesses to critical regions, e.g., the syscall table. In Section 5.9.4, we will suggest SEV changes that further harden 00SEVen against collusion.

5.8 Evaluation

We now evaluate the analysis performance of our 00SEVen prototype, its effectiveness for detecting or preventing existing rootkits, and its VMPL0 memory and CPU overhead.

00SEVen's open-source prototype (cf. Section 5.11) [S4] consists of our in-VM agent that extends the SVSM, extensions to QEMU/KVM, and extended LibVMI library for analysis clients. It supports remote VMI operations, secure pausing, and page monitoring traps. As SVSM currently supports only Linux VMs, we focus on VMI of Linux—even though conceptually, support for other OSes is possible. The prototype does not yet support function traps and the optional shared buffer isolation (Section 5.9.3).

As the evaluation testbed, we use a Dell PowerEdge R6515 server as our cloud platform running Ubuntu 22.04 with our modified 5.14 kernel on a 2.85 GHz AMD 7443P CPU. The Dell server hosts our 00SEVen prototype including an SEV-SNP Ubuntu VM serving as the VMI target. As the LibVMI remote analysis client, we use a Debian 12 server with a 3.2 GHz AMD 74F3 CPU that shares a LAN with the VMI target. For comparison, we measure three additional setups: LibVMI with KVMi, and 00SEVen with a local (same-host) LibVMI—with TLS and without (TCP-only). Our baseline is LibVMI with the standard KVMi backend (v12) that we measure *locally* on the Dell server targeting a non-SEV VM—KVMi does not support remote analysis. By evaluating 00SEVen with local (TLS / TCP-only) and remote (TLS) clients, we can distinguish sheer network and TLS overhead.

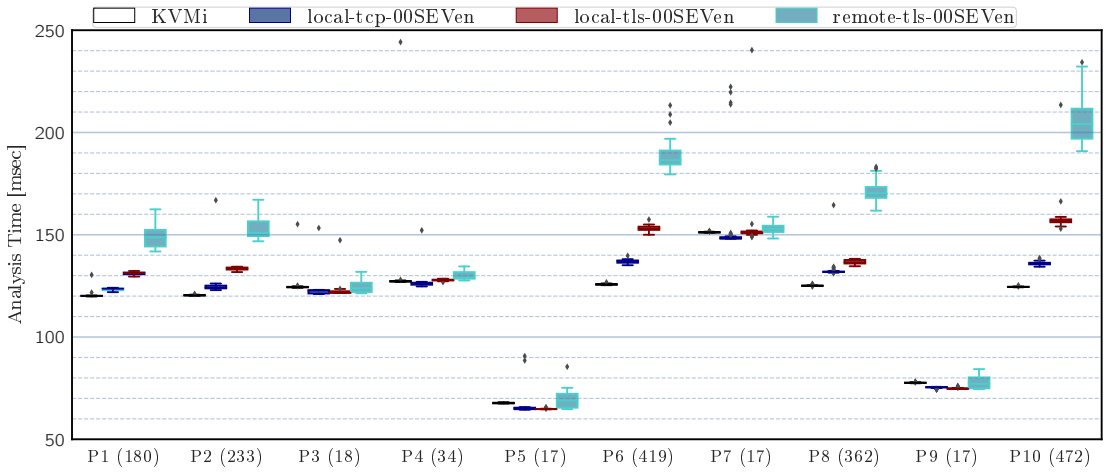

Figure 5.7: Analysis times for a local LibVMI-KVMi and three 00SEVen setups: with local LibVMI via TCP, local via TLS, and remote via TLS. (#VMI-queries is given in brackets)

5.8.1 (Remote) Analysis Performance

We now evaluate the VMI performance of our four setups. We adopted the LibVMI policies of the recent RDMI paper [147], which resemble practical queries for rootkit detection. Table 5.1 lists their analysis targets (cf. [147] for details) except for one, incompatible with our VM kernel version. We measured the initialization and analysis time of each setup for 50 iterations per policy, cleaning LibVMI's caches before every run. 00SEVen's local and remote setups showed negligible one-shot initialization times <2 s, while KVMi's default config takes ≤ 10 s. Figure 5.7 compares the LibVMI analysis times of the four setups. The analysis time captures all VMI queries to KVMi or 00SEVen's agent for the paused target VM. During the analysis, 00SEVen schedules only its agent, avoiding overhead by VMPL switches. The median analysis times are 68–151 ms for KVMi, 65–148 ms for 00SEVen with local TCP-only client, 65–157 ms with local, and 69–204 ms with remote TLS client. That is, compared to KVMi, 00SEVen faces reasonable average median overheads of +1.91 % (TCP-only), +6.85 % (local TLS), and +20.0 % (remote TLS).

00SEVen's relative overhead increases with the number of queries (cf. x-labels, Figure 5.7). Each VMI read inherently requires a page to be copied out of SEV-protected memory and sent via the VMI channel to the LibVMI client. The effect on the local setups is significantly smaller, showing that the biggest overhead can be attributed to the per-query network overhead. For small query numbers (≤ 34), 00SEVen was even slightly faster than KVMi. In addition, the local results show that our TLS support, currently missing CPU acceleration and zero-copy, adds noticeable extra overhead. Therefore, optimizing the TLS code and decreasing the number of remote messages by using more caching and offloading strategies (cf. Section 5.5.2.3 and 5.9.2) could further improve 00SEVen's performance. For instance, the P5 overhead is small because LibVMI's client-side page cache makes it require only 17 VMI queries for iterating the *hundreds* of co-located syscalls.

5.8.2 Rootkit Detection and Active Trapping

We now evaluate 00SEVen's capability of detecting or preventing rootkits. We adapted three open-source Linux rootkits (also used in [147]) to our VM's v5 kernel: Sutekh, Spy, and Diamorphine [211, 207, 57]. *Sutekh* hooks the `umask` and `execve` syscalls by overwriting their function pointers in the syscall table to enable user space processes to gain root privileges. *Spy* is a keylogger that registers itself as a keyboard listener to log all entered keystrokes. *Diamorphine* hooks the `kill`, `getdent`, and `getdent64` syscalls to enable processes to request service via signals, e.g., hiding of files or threads. For each rootkit, we implemented policies (inspired by Table 5.1) that detect an infection and evaluated their analysis performance. Our policies check if the syscall table entries point to valid (in-kernel) functions to detect their hooks (*Sutekh, Diamorphine*), check for privileged shells (*Sutekh*), and check for registered keyboard notifiers (*Spy*). Sutekh required a median analysis time of 256, 256, 265, and 282 ms (KVMi, tcp-only, local, remote), Spy of 219, 213, 217, and 217 ms, and Diamorphine of 136, 129, 131, and 132 ms. That is, the analysis times are in line with those of Figure 5.7: The median overheads for Spy and Diamorphine are slightly negative as they only require 21 and 18 VMI queries (compare P5), while Sutekh required 222 queries and showed overheads of 3.5 % for the local and 10.2 % for the remote TLS client.

Instead of detecting rootkits post-mortem, 00SEVen can also *actively prevent* the infection process using page access traps (Section 5.5.2.5). This is an advantage over approaches like RDMI [147] that have no trap support. We implemented event-based policies for all three rootkits that successfully leverage page write-traps that trigger when the rootkits are trying to tamper with the syscall table or keyboard notifier list. That way, the remote client can directly pause the VM and perform an analysis of the stopped exploitation chain. However, page traps add non-negligible overhead. They should be used preferably to monitor suspicious accesses to critical pages, e.g., write attempts to the syscall table. We measured the trap-and-resume overhead of a single write (`ADD`) showing median overheads of 13.0 µs for KVMi's traps (single stepping-based), 738.5 µs and 761.8 µs for 00SEVen's emulation-based traps in the local setups, and in the remote one ≈1 ms to 3 ms if the VM is not yielded, otherwise ≈45 ms (includes network overhead). 00SEVen's overhead is caused by the VMPL switches, TLS and network overhead, and missing single stepping. Suppressing VMPL switches accelerates the local setups to 85.7 µs and 93.3 µs, showing that 00SEVen benefits from reduced switching overhead, e.g., by adding a dedicated agent vCPU or improving AMD SEV (cf. Section 5.9.4). To further decrease 00SEVen's overhead, LibVMI's trap handlers could be partially offloaded into the agent's user space to avoid communication overhead, and the emulation could be optimized.

5.8.3 In-VM Requirements and Overhead

00SEVen is designed with a small idle load and TCB (*R7+8*) *without* a full-fledged OS kernel. Our in-VM TCB consists of only ≈13.3 kSLOC (≈12.5 k in Rust), including the SVSM (≈6.5), our VMI agent (≈5.1), and virtio drivers (≈1.7). By default, SVSM reserves only 256 MiB of a VM's RAM for VMPL0, i.e., 6.25 % for a 4 GiB VM. Our actual memory requirement is even smaller (in the order of 10 MiB) but depends on

the number of vCPUs and the virtio ring size. 00SEVen's agent does *not* impose performance overhead on VM workloads while no analysis is active (*R8*). By default, only the VM OS in VMPL1 is getting scheduled. VMPL0 is scheduled only if either VMPL1 calls into a SVSM memory service—which is typically done only during boot—or if the VM owner sends a remote request to 00SEVen's agent. In the "idle state", 00SEVen's VMPL0 components cause no overhead, in contrast to other recent (non-VMI) SEV designs, e.g., Hecate [83], which must actively virtualize scheduling and device I/O for VMPL1 (cf. Section 5.10).

5.9 Discussion and Outlooks

We now discuss the portability of 00SEVen to other confidential platforms and multiple directions for future software- and hardware-based extensions of 00SEVen. We explain how 00SEVen might be implemented on other confidential platforms, discuss additional software-based memory access optimizations, and propose hypervisor extensions and hardware-based SEV extensions to further improve secure VMI. Finally, we discuss how 00SEVen might leverage future hardware support for trusted device I/O to isolate and accelerate its VMI for data centers even further.

5.9.1 Other Confidential (VM) Platforms

00SEVen's current implementation is tailored to AMD SEV-SNP VMs. However, 00SEVen's concepts generalize to other confidential VM platforms, all of which are incompatible with existing VMI techniques by default (Section 5.4.3). In the following, we discuss how the concepts of 00SEVen can be implemented in Intel TDX [106] and Arm CCA [16].

Intel TDX Intel's confidential VMs are called trusted domains (TDs) and provide similar protection guarantees as SEV VMs using per-TD crypto keys. Intel has announced support for TD partitioning in future Intel TDX version 1.5, which enables up to four nested VM environments inside a single TD—comparable to VMPLs. TD partitioning provides the foundation for the isolation of 00SEVen's in-VM agent and the deprivileging of the untrusted VM OS. Intel's TDs also have a state-save area comparable to AMD's VMSAs, which are a key component for register introspection. In contrast to SEV, TDX features a so-called monitor, which acts as trusted intermediate between the untrusted hypervisor and the TDs. The monitor manages the TDs, has privileged access, e.g., to the state-save areas, and can provide trusted hypercalls to TDs. An implementation of 00SEVen for TDX might build on TD partitioning and offload some tasks to the monitor.

Arm CCA Arm CCA introduces realms that provide secure execution environments, e.g., for confidential VMs. In contrast to SEV and TDX, the isolation between realms is not based on VM-unique crypto keys but on nested PTs (stage 2 PTs). These NPTs are managed by the new trusted realm management monitor (RMM), which acts as intermediate between the realms and untrusted host hypervisor. The RMM executes

with hypervisor-like privileges (EL2), sharing some similarities with the TDX monitor. As the RMM manages NPTs for all realms, it should be possible to redesign existing NPT-based isolation concepts to protect an in-VM agent or create a co-located VMI VM [201, 242]. That way, a special per-realm domain for integrating concepts of 00SEVen's agent could be created. Some of 00SEVen's monitoring approaches could be adapted to benefit from RMM's trusted NPT management.

5.9.2 Agent-side Optimizations for Virtual Memory Access

In Section 5.5.2.3, we explained 00SEVen's memory access methods and its client-side caching strategies for faster consecutive page accesses and virtual memory accesses. While these optimizations achieve good performance (cf. Section 5.8), especially when inspecting co-located data, e.g., kernel structures or syscall table entries, high numbers of one-time accesses to different VM pages can still face noticeable access overheads. When accessing pages for the first time, the resulting client cache misses can potentially cause multiple network requests as part of the software-based address translations ($VA_{vmpl1} \rightarrow PA_{vm}$) with non-negligible network request latency. For future versions of 00SEVen, we therefore consider offloading the VMPL1 page table walks required to translate VA_{vmpl1} into the agent or even enabling the agent to directly access VA_{vmpl1} without additional translation steps in software.

By offloading the VMPL1 page table walk into the agent ($VA_{vmpl1} \rightarrow PA_{vm}$), we could decrease the communication overhead between agent and remote client by avoiding VMPL1 page table transfers to the client. Furthermore, we could then adjust the client's page-level data caching from $\{PA_{vm} \rightarrow$ page buffer$\}$ to $\{VA_{vmpl1} \rightarrow$ page buffer$\}$, such that the remote client need not lookup and send physical addresses (PA_{vm}) to the agent anymore.

Alternatively, instead of just offloading the translation, we could try to eliminate the translation steps from VA_{vmpl1} to VA_{vmpl0} entirely. By securely using parts of the untrusted VM OS (VMPL1) PTs directly from within our agent, we could omit the additional translation ($VA_{vmpl1} \rightarrow PA_{vm}$) and mapping ($PA_{vm} \rightarrow VA_{vmpl0}$) steps. That way, the client need not translate VA_{vmpl1} but can let the agent directly access the address with native speed, decreasing translation and access overhead on cache misses or renewals—particularly useful for live introspection support (cf. Section 5.5.2.6). However, existing VMI techniques following such an approach [242, 201] are not applicable to 00SEVen, as they rely on a *trusted* hypervisor managing nested (second-level) PT entries, permissions, and/or trapping PT changes of VMPL1—violating SEV-SNP's and our threat models. We leave the challenge of securely using parts of the untrusted VMPL1 page table entries from VMPL0 as future work.

5.9.3 Isolating Shared Buffers

As discussed in Section 5.7.2, 00SEVen requires a new hypervisor extension if its *shared* VM pages—virtio rings and GHCB buffers—should be isolated against in-VM VMPL1 attackers. Even though the feasibility and impact of attacks against these pages is limited, since they are hard to time, risk detection, are mainly DoS, and are shut down by VMPL1 pausing (Section 5.5.2.4), we now present a concept for blocking any access

by VMPL1 to 00SEVen's shared pages. Conceptually, we introduce per-VMPL nested page table (NPT) views with slightly different access permissions. The view for VMPL0 (NPT_{vmpl0}) is the unmodified default version as currently used for the VM. The view for VMPL1 (NPT_{vmpl1}) differs from it only in that the shared memory pages reserved for 00SEVen's GHCB buffers and remote channel (virtio rings, buffers) are marked as non-accessible. On VM boot, before VMPL1 has been initialized, 00SEVen's agent issues a new hypercall to inform KVM about the locations of these shared pages, such that KVM can create two versions of the respective NPT entries: permissive ones for NPT_{vmpl0} and restrictive ones for NPT_{vmpl1}. When KVM switches the VMPL contexts for a vCPU, in addition to the existing cache cleanup steps and GHCB swap, KVM swaps the NPT views. As a result, only VMPL0 and the hypervisor can access the virtio and GHCB buffers, denying any access attempts by VMPL1 attackers (*E2*).

Comparison to Hecate Hecate [83] introduced a concept to isolate shared memory pages inside SEV VMs without hypervisor support. Hecate instruments SEV's vTOM (virtual top of memory) feature to enforce any VMPL1 access to be treated as private page access. That way, any VMPL1 access to a shared page will result in a nested page fault [7]. However, in contrast to our concept, this approach cannot define individual permissions for each page, i.e., VM OS access to *all* shared VM pages is denied, including all device I/O buffers and GHCB buffers. Therefore, Hecate must virtualize several operations for VMPL1 inside VMPL0, including *all* device I/O and hypervisor communication. This design results in a more complex in-VM trusted computing base and additional runtime overhead—violating 00SEVen's design requirements *R7* and *R8* (cf. Section 5.4.2).

5.9.4 Improving AMD SEV for Secure VMI

00SEVen would benefit from new optimizations and features for AMD SEV regarding performance and security. 00SEVen's agent would benefit from VMPL0 support for directly yielding and locking lower-privileged VMPLs and intercepting VM exit events *without* relying on the hypervisor. That way, 00SEVen's secure pausing feature would not depend on support by the hypervisor, and 00SEVen could directly trap writes to control registers as required for consistent live VMI (Section 5.5.2.6) or use single-stepping for easier page access monitoring (Section 5.5.2.5). 00SEVen would also benefit from VMPL permissions for shared memory pages. That way, 00SEVen could prevent in-VM attackers from tampering with its virtio and hypercall buffers on a per-page granularity *without* relying on the hypervisor (cf. previous section and Section 5.7.2). Finally, as already observed by Ge et al. [83], VMPL switches through the hypervisor cause non-negligible overhead. Hardware support for directly switching VMPLs without hypervisor intervention would improve the performance of 00SEVen, especially its memory access and function traps.

5.9.5 Outlook: Trusted I/O Support

AMD and Intel have announced support for trusted device I/O (TIO, also: TEE-IO) for future versions of SEV and TDX VMs [6, 104]. This raises the question how 00SEVen

could benefit from this. Considering SEV as an example, SEV's TIO will enable secure device binding to a VMPL of an SEV VM, which grants the device access to the VM's *private* memory under the restrictions of the respective VMPL. That way, the VMPL and device can securely communicate without interference by less-privileged VMPLs or the cloud platform, including the untrusted hypervisor. While 00SEVen does not require TIO to be effective, 00SEVen could take advantage of it: First, 00SEVen could accelerate and further isolate the device I/O of the remote channel. By binding one I/O path of a SmartNIC to VMPL0, 00SEVen's agent could directly perform packet I/O from *private* memory, isolated from VMPL1 and the hypervisor—at the cost of additional network stacks and the SmartNIC becoming part of the in-VM TCB.

Another promising direction is to combine 00SEVen's in-VM agent with DMA-based remote memory aggregation solutions, as provided by PCILeech [78] or the recent RDMI [147]. These approaches enable fast remote memory aggregation using devices with direct memory access capability, e.g., SmartNICs or FPGAs. However, in contrast to 00SEVen, they are not compatible with confidential VMs (e.g., based on SEV) and do not have any control over the VM as required for instance for secure pausing or page monitoring techniques. In the future, 00SEVen's agent could be extended to securely bind the remote DMA feature of a SmartNIC to SEV's VMPL0 (or the analogous privileged domain of Intel TDX). 00SEVen's agent could then overcome the limitations of RDMI's memory aggregation by exposing VM control and trapping features to the remote analyst (secure pausing, register access, page and code monitoring), while the SmartNIC would enable fast memory access. That way, a combined design tailored to confidential VMs with TIO support could securely enable flexible remote VMI with high-speed memory access for data centers.

5.10 Related Work

Memory Forensic and VMI Most related to 00SEVen is work on non-confidential VMI and memory forensic, e.g., out-of-VM [242, 142] and in-VM [201] VMI. In Table 5.2, we provide a comparison of 00SEVen with existing VMI and forensic approaches. 00SEVen takes inspiration from these designs to fill the gap of enabling VMI techniques securely for SEV-SNP VMs. To the best of our knowledge, 00SEVen is the first solution enabling secure remote VMI for confidential VMs. LibVMI [142] is a common framework for non-confidential VMI providing memory and register access, address and symbol translation, as well as event-based traps. We designed 00SEVen to provide similar features and based our remote client on it to enable reuse of existing analysis scripts and tools (cf. Section 5.5.2.2). Zhao et. al [242] use the hypervisor to provide fast out-of-VM VMI using a special co-located VM (ImEE) that shares the untrusted page tables of the target VM securely using NPT permissions. SIM [201] provides an in-VM VMI agent for non-confidential VMs that is protected by the hypervisor and uses special call gates to switch into the agent without a VM exit. In contrast, 00SEVen focuses on remote VMI for SEV-SNP VMs, uses VMPLs to protect its in-VM agent, and designs VMI techniques *not* trusting the hypervisor. Bridging the semantic gap [115] is a fundamental issue of VMI, rendering all these techniques relevant to 00SEVen. Katana and LogicMem [77, 183] enable automatic symbol and data structure extraction, which

Table 5.2: Comparison of different types of existing VMI and memory forensic solutions with our 00SEVen system for confidential SEV-SNP VMs. (HV ≙ hypervisor, PTs ≙ page tables, SMM ≙ system management mode, ATS ≙ address translation service)

Name	Agent Location	Agent Isolation	TCB	Forensic Target
LibVMI	HV/outside VM	virtualization	HV	regular VMs
SIM [201]	inside VM	2nd level PTs	in-VM, HV	regular VMs
ImEE [242]	co-located VM	2nd level PTs	co-VM, HV	regular VMs
00SEVen	inside VM	SEV, VMPLs	VMPL0, SEV	SEV-SNP VMs
Smile [244]	in SMM+SGX	HW-isolated	SMM(, SGX)	SGX enclaves
RDMI [147]	NIC/P4 switch	dedicated HW	NIC, P4	bare-metal OS

is an orthogonal feature useful for 00SEVen. Similarly, Oliveri et. al [169] proposed OS-agnostic memory forensic, not requiring prior knowledge on the target, i.e., the VM OS. VMIfresh [51] improves LibVMI's caching (Section 5.5.2.3) using active monitoring of page table changes to prevent stale entries (cf. Section 5.5.2.6). 00SEVen could adopt this approach using its page-based memory access traps.

Other related work includes remote memory aggregation and forensics for trusted execution environments (TEEs). PCILeech [78] and RDMI [147] enable memory access via devices with (remote) direct memory access. While they provide fast access, they are vulnerable to redirection attacks by a malicious OS or hypervisor [116, 19] and are not tailored to VMI, lacking respective features, e.g, VM pausing and event traps. While there is no work on VMI for SEV VMs, Smile [244] provides secure live memory inspection for Intel SGX enclaves. Similar to 00SEVen, Smile had to securely overcome the hardware-based memory protection of enclaves. In contrast to 00SEVen, Smile relies on a semi-trusted out-of-enclave agent in the system management mode (SMM) and faces different design challenges, e.g., when accessing enclave memory. Furthermore, 00SEVen provides additional features, e.g., secure in-VM pausing and event traps. Guerra et. al [99] add VMI modules into Arm TrustZone to inspect the non-secure system but not the Arm TEE itself.

SEV Research There is several orthogonal research on confidential SEV VMs. Most related is Hecate [83], which supports legacy OSes inside SEV-SNP VMs by tailoring a single-VM-capable nested hypervisor to SEV-SNP. In addition, Hecate drafts support for in-VM kernel code integrity and network filter policy enforcement. Similar to Hecate, 00SEVen re-designs non-confidential hypervisor techniques securely for SEV-SNP VMs. However, 00SEVen's focus is on secure remote VMI and a small TCB with negligible runtime overhead while no analysis is active. Narayanan et. al [161] integrate a virtual TPM as a new orthogonal SVSM service into VMPL0, which could augment the attestation of 00SEVen's remote channel. Offensive work on SEV VMs explores their weaknesses (e.g., cipher side-channels) and proposes countermeasures orthogonal to 00SEVen [137, 226, 55, 139, 138, 34].

5.11 Artifacts

The prototypes of 00SEVen are available as open-source projects at `https://github.com/sev-vmi/00seven` [S4]. See page 9 for a list of all open-source prototypes covered by this dissertation.

5.12 Conclusion

00SEVen re-enables an essential security technique for confidential SEV VMs: secure remote VMI. 00SEVen introduces new concepts to redesign existing non-confidential VMI techniques for SEV-SNP VMs. By leveraging the recent virtual machine privilege levels of SEV-SNP, 00SEVen realizes an in-VM VMI agent that is hardware-isolated from out-of-VM and in-VM attackers. 00SEVen's agent provides the VM owner with secure remote inspection capabilities of the private VM memory and registers, as well as secure pausing and trapping mechanisms for consistent and event-based analysis—providing a positive answer to research question *RQ4* (see page 6) on the feasibility of remote forensics for confidential VMs (aka TEE VMs). Using 00SEVen, highly sensitive customers, e.g., of the finance and health sector, can securely offload their workloads to the cloud while retaining full introspection access for periodic security scans, incident response, or attack detection, prevention, and analysis.

This chapter concludes the second part of this dissertation which focused on meta research question *MQ2*, i.e., in how far we can securely enable attack detection or prevention techniques for TEE-based services. 00SEVen contributes to *MQ2* by enabling remote VM introspection for AMD's TEE VMs to detect, analyze, or even prevent runtime attacks, e.g., software exploits or rootkits. 00SEVen's VMI capabilities provide a valuable additional protection mechanism and add to the research line of re-enabling non-TEE VM security mechanisms for TEE VMs, like for instance Hecate [83]. In the next chapter, we will summarize the contributions of this dissertation and discuss ideas for future work before concluding.

6
Conclusion

6.1 Summary of Contributions

System-level compromises pose a serious threat to client, network, and cloud devices due to their strong disruption of existing software security mechanisms. Several TEEs with different protection boundaries have been proposed to isolate sensitive services from such attacks and enable new security schemes, e.g., secure cloud computing. However, security-critical network and web authentication services still missed out on the benefits of TEE-based designs due to their lack of TEE adoption. Furthermore, more coarse-grained TEEs like the recent VM-level TEEs, e.g., AMD SEV or Intel TDX, face a large attack surface as they incorporate a full OS with a high number of non-hardened user space services, resulting in a need for intra-TEE defenses.

In this dissertation, we therefore presented two lines of research that aimed at (1.) leveraging TEEs to assist critical network and web authentication services, and (2.) designing intra-TEE attack detection and prevention solutions, especially for VM-level TEEs. Specifically, in the first part of this dissertation, we addressed our first meta research question *(MQ1) How can security-critical network and web authentication services benefit from TEEs?* In the second part, we contributed to our second question *(MQ2) In how far can we securely enable attack detection or prevention techniques for TEE-based services?*

TEE-based Service Designs (MQ1)

Using TEEs to redesign security-critical network and web authentication services not only enables protection against system-level attacks but allows users to remotely guarantee that their security and privacy policies are diligently enforced. This significantly increases trust in services that process sensitive information (e.g., personal user data) or form important security anchors (e.g., network gateway firewalls). To address our first meta question *MQ1*, we answered our three research questions *RQ1* to *RQ3* (see below) by proposing three TEE-based service designs that demonstrate benefits of TEEs when applied to network firewalls, gateway routers, and token-based web authentication.

First, in Chapter 2, we looked into the question *(RQ1) Can we use TEEs to enable secure per-application traffic attribution and firewall policy enforcement?* We suggested SENG, a gateway firewall extension, that leverages client-side TEEs to enable secure traffic-to-application attribution for per-application policy enforcement. SENG demonstrates how TEEs can help to finally turn a long-demanded firewall feature into a secure solution that is robust against client-side compromises and rooted in trusted hardware. The remote attestation of Intel SGX enables to precisely identify the applications and vouch for their protection, in contrast to existing spoofable identifiers. Furthermore, SENG's traffic attribution allows for additional security measures on top, e.g., per-application traffic analysis models (see A17, Section 2.9).

Second, in Chapter 3, we contributed to *MQ1* by specifically focusing on the hardware-based isolation of TEEs by answering the question *(RQ2) Can we use TEEs to protect the network path and policy enforcement of routers against compromised user and kernel space services?* We presented TrustedGateway, our trusted router architecture, which, in contrast to SENG, leverages a system-level TEE that provides isolation at the kernel space, user space, and even peripheral level rather than merely for user

space processes. TrustedGateway showcases the benefits of system-level TEEs like Arm TrustZone for enabling the protection and guaranteed enforcement of an entire processing pipeline for standalone devices—in this case, the full network I/O path of gateway routers, including the NIC driver as well as routing and firewall services. Furthermore, TrustedGateway demonstrates that it is possible to achieve such benefits via a design based on widely-available off-the-shelf TEE hardware extensions and directly compatible with existing OSes and user space services.

Third, in Chapter 4, we showed how TEEs can tackle open challenges of existing security schemes by addressing the question *(RQ3) Can we use TEEs to overcome the deployment and account recovery challenges of FIDO2 web authentication based on user eIDs while preserving user data privacy?* We suggested FeIDo, a virtual FIDO2 token, that leverages TEE-based cloud services to securely derive web credentials based on sensitive, eID-extracted user information. FeIDo contributes to *MQ1* by demonstrating multiple benefits of TEEs for web authentication services. FeIDo shows how the protection and remote attestation of TEEs can guarantee that no sensitive user data is leaked to untrusted third parties—in this case, the cloud and web service vendors. That way, TEEs enable FeIDo's innovative design that overcomes the open deployment and recovery challenges of FIDO2 hardware tokens. In addition, FeIDo shows that TEEs can enable new features for FIDO2 by adding anonymous user credentials as part of the secure web authentication process, e.g., pseudonymized age information.

Attack Detection or Prevention for TEE Services (MQ2)

With an increasing adoption of TEEs, solutions for detecting and preventing attacks inside TEEs become more important. In particular, the upcoming VM-level TEEs, which host an entire OS with several user services, face a large attack surface prone to vulnerabilities and cannot rely on service-specific protection designs. Therefore, recent security extensions of VM-level TEEs, such as AMD's VM privilege levels (VMPLs), are an important step to foster research on defenses that mitigate the risks of intra-TEE attacks. In Chapter 5, we contributed to our second meta question *MQ2* on the feasibility of in-TEE defenses by tackling the research question *(RQ4) Can we enable forensic remote introspection of (potentially) compromised TEE VMs without breaking their security guarantees?* We presented 00SEVen which securely provides TEE VM owners with full remote VMI capabilities for attack detection, analysis, and prevention, despite the strong memory protection of SEV-SNP VMs. We explained how SEV-SNP's VMPLs allowed us to redesign and extend introspection techniques such that they integrate into SEV-SNP VMs while fully preserving the VMs' hardware-based security guarantees. Thus, VM owners can monitor their VMs for attacks without leaking any information to third parties, e.g., cloud providers hosting the VMs. In addition, we identified properties of the current SEV-SNP revision that impose limitations on secure VMI designs for TEE VMs. Based on these insights, in Section 5.9.4, we suggested improvements for future SEV-SNP revisions that would benefit secure introspection of TEE VMs and potentially other in-TEE defenses.

6.2 Future Research Directions

In the following, we will discuss ideas for future work that are inspired by this dissertation. They include broader research directions as well as follow-up ideas that are closely related to our presented solutions. We also refer to the chapter-specific discussions which already explore some limitations and associated research perspectives, especially Section 2.13 and Section 5.9.

6.2.1 Further Exploration of TEEs and their Extensions

This dissertation has only scratched the surface of existing TEE technologies and their potential use cases. There are many opportunities to extend our proposed designs based on upcoming TEE features, port them to other TEE platforms, or explore new TEE-based service designs and intra-TEE protection schemes.

Transferring/Porting our Concepts While Chapters 2 to 5 presented designs tailored to specific TEEs and settings, we regard the transfer of their concepts to other TEE platforms and use cases as interesting research opportunity. That way, users can benefit from their security features across multiple platforms, and researchers can gather additional insights into the differences and shared properties of the different TEEs.

SENG's (Chapter 2) focus has been mainly on gateway firewalls and applications running on client workstations. However, the concept of secure traffic-to-application attribution is also interesting in other settings. For instance, we could explore how to transfer the concepts of SENG to mobile TEE platforms or cloud-based TEE VMs in order to enable traffic attribution for a wider class of devices and applications, e.g., cloud services. In order to address the unfortunate deprecation of Intel SGX on client-side CPUs, it would be interesting to explore alternative solutions to re-enable SENG for client-side devices (also cf. Section 6.2.3). In addition, the concept of packet filtering also exists in embedded or non-IP networks, e.g., consider the communication in or across vehicles. For instance, we could explore how to port SENG's concepts to embedded TEEs (e.g., Arm TrustZone-M) in cars to attribute traffic to microcontroller firmware in order to prevent control message spoofing.

TrustedGateway (Chapter 3) is currently tailored to Arm-based devices, especially standalone routers. However, there are many x86-based router architectures as well as virtual network switches and routers running as containers or VMs on servers. Therefore, it would be useful to explore how we could securely implement TrustedGateway's protection architecture on a x86 network or server device, e.g., based on new micro-hypervisors or Intel's TEE VMs (TDs). Considering the announcement of trusted I/O support for Intel TDX version 2.0 [104], an implementation for Intel TDX based on a trusted I/O channel to the network peripheral (e.g., SmartNIC) seems particularly promising for future x86-based network devices (also cf. next paragraph).

FeIDo (Chapter 4) has shown how TEEs can process user data in a privacy-preserving manner, e.g., for web authentication. In particular the concept of anonymous credentials seems to be a promising direction that needs further exploration. Furthermore, it would be interesting to explore custom firmware extensions for adding FeIDo

directly to eIDs (also cf. Section 6.2.3), or consider co-designs between TEEs and other external hardware devices.

We also regard ports of 00SEVen's (Chapter 5) VMI concepts to Intel TDX and Arm CCA as important steps to provide protection across all platforms (cf. Section 5.9.1).

Upcoming TEE Features TEE VMs are currently on the rise with the release of Intel TDX, new feature announcements for AMD SEV-SNP and Intel TDX, as well as the pending Arm CCA technology. As TEE VMs are easy to adopt from a user perspective, as applications do not require any changes to run inside TEE VMs, we expect an increasing adoption of TEE VMs, especially on cloud platforms. Therefore, we regard a further exploration of protection schemes based on TEE VMs as an important research direction, in particular intra-TEE defenses based on VMPLs (AMD SEV) or TD partitioning (Intel TDX) as well as designs based on trusted I/O (when released). Specifically: *What security designs can be enabled by upcoming TEE features, and how could they further improve our proposed designs?*

00SEVen shows the potential of VMPLs for enabling secure introspection for SEV-SNP VMs. However, 00SEVen only leverages two of four VMPLs in order to isolate the VMI agent and deprivilege the OS. Therefore, an interesting research question is what protection schemes could be achieved by using *all* supported VMPLs? Could we even bypass the limitation of four VMPLs via software-based techniques, e.g., by dynamically changing VMPL permissions to emulate additional isolation domains? In this context, one particular idea is if we could use VMPLs in order to split the VM OS into multiple protection zones with increasing access capabilities, e.g., the user space in the most restricted VMPL3, closely followed by the system call interface in VMPL2, core kernel components in VMPL1, and the security-enforcing, most critical kernel or hardware services in VMPL0. That way, a compromise of parts of the OS kernel, e.g., the user-facing system call interface, would still not affect the security policies. Another interesting concept to explore is that of dynamic VMPL switching. Each VMSA is registered by the VM and hypervisor, and is associated with a set of virtual CPU registers as well as a VMPL. VMPLs cannot be modified by the hypervisor at runtime, i.e., to switch a VMPL, the hypervisor must schedule a VMSA with a different VMPL (cf. Section 5.5.1). It would be interesting to explore in how far VMPL0 could modify VMPLs of existing VMSAs, i.e., execution contexts, on demand, e.g., to dynamically elevate or lower the privileges of a service call. This might be an interesting primitive for enabling cross-privilege calls in a multi-VMPL OS design, as proposed just before, or alternatively, for dynamically quarantining an execution context on anomalous behavior, e.g., to stop a potential intrusion attempt.

We should not limit our focus on SEV-SNP VM but also explore Intel TDX and Arm CCA VMs. In particular, Intel TDX has announced TD partitioning for version 1.5 [105], which has similarities to VMPLs. Therefore, future research could investigate in how far the concepts of 00SEVen and the above ideas for VMPLs could also be applied to TD partitions, and what additional solutions TD partitions enable. Furthermore, future research should explore if similar features as those provided by VMPLs and TD partitions can be implemented for VMs/Realms based on Arm CCA.

Another interesting feature is trusted I/O (TIO) support as announced for fu-

ture versions of AMD SEV [6] and Intel TDX [104]. We regard the exploration of software-hardware co-designs where TEE VMs cooperate with trusted peripherals based on trusted I/O channels as promising perspective, e.g., see ideas to accelerate 00SEVen's VMI using SmartNICs (Section 5.9.5) or implement TruGW's I/O path on TDX (cf. above).

6.2.2 Enabling Cross-TEE Compatibility

While the exploration of TEE-specific designs is important, a major challenge of the current TEE landscape are incompatibilities across TEE implementations. It is non-trivial to port concepts and thus protection designs across TEEs due to different feature sets, protection boundaries, and development toolchains. For instance, a design like TrustedGateway requires trusted I/O support as provided by system-level TEEs like Arm TrustZone, however, user space TEEs like Intel SGX lack this capability. But even with similar protection boundaries, the concepts of some features might still be different which makes porting more difficult, e.g., AMD SEV's VMPLs vs. Intel TDX's TD partitioning, or the memory encryption of SEV and TDX vs. the NPT-based access control model of Arm CCA Realms (cf. Section 5.9.1). The incompatibility challenges and TEE dependencies of designs not only hinder the availability of security solutions across platforms but might even render them unfeasible, e.g., as happened on the cancellation of Intel SGX support for client CPUs. Therefore, we deem the following research question as important: *How can we make TEE-based security designs more portable across different CPU architectures and TEE implementations?*

One way to approach this question is by deriving an abstract TEE API model for designing protection schemes, which hides the platform-specific details. For instance, OpenEnclave [171] has been an early project that tried to abstract the APIs of enclaves and provides a backend for Intel SGX and Arm TrustZone. However, features like the new intra-TEE isolation or trusted I/O primitives go beyond this model. The important question is if we can design an abstract API model that is general enough to cover most use cases while preserving compatibility across different TEE platforms. In this context, it is also interesting to explore in how far we can emulate certain TEE features on different platforms, for instance, similar to how vSGX [241] virtualizes Intel SGX enclaves using AMD SEV. We might also draw inspiration from FIDO2 which defines different security levels based on the used backend technologies, e.g., software-based vs. TEE-based. For instance, we could explore different function/security levels based on the covered protection domain of a TEE, e.g., low for process-level TEEs like SGX, medium for VM-level TEEs like SEV, and high for system-level TEEs with trusted I/O like Arm TrustZone or future versions of SEV and TDX. A generalized TEE API model should then allow to dynamically detect the (un)availability of certain TEE features, e.g., trusted I/O, and adjust the offered functionality of the TEE-based service accordingly, optionally emulating missing TEE features if possible (cf. vSGX).

Another way to approach the question could be by trying to develop new TEEs that cover all the use cases and development models of previous TEE technologies. Flexible TEE architectures such as CURE [20] provide a first step into that direction. In the next section, we will focus on more ideas regarding custom hardware extensions.

6.2.3 Design of Custom Hardware Extensions

This dissertation has build on existing commodity TEEs, which has the benefit of making our provided designs widely deployable. However, during our projects, we have observed several limitations of existing TEEs, e.g., the missing trusted I/O support of Intel SGX, or missing support of AMD SEV for additional VMI features (cf. Section 5.9.4). Therefore, a promising future research direction is to develop own hardware extensions for TEEs or even entirely new types of TEEs. In particular the open RISC-V architecture enables fast prototyping of CPU extensions on FPGA boards, making it feasible to test them in realistic setups. Projects like CURE [20], ELASTICLAVE [237], or Graviton [223] are examples of research projects providing flexible CPU TEEs, user space enclaves (TEEs) with shared memory support, and GPU TEEs. With the increasing support for FPGAs in server and data center machines, custom hardware extensions also become much more feasible for practical deployment than before.

One particular type of TEEs that we would like to highlight are client-side TEEs. We regard the deprecation of the Intel SGX TEE on client-side devices as an unfortunate decision. Many research projects, including SENG (Chapter 2), have shown the potential for client-side TEEs. Therefore, we regard it as important to explore if it is possible to design a more versatile, robust client-side TEE based on the insights gained from the challenges faced by Intel SGX enclaves. While there are still TEEs available for client devices, e.g., Arm TrustZone on Android phones and Apple's Secure Enclave on iOS devices, these TEEs operate in closed environments, lacking SGX's flexible support for direct deployment of third-party code.

Another research direction includes firmware and hardware extensions to support specific TEE-assisted use cases. For instance, similar to how simTPM [38] has added TPM support to sim cards, we envision adding FeIDo support directly into eIDs, e.g., by integrating an embedded TEE inside eIDs that receives the shared KDF secret and locally derives the FIDO2 credentials based on the user information (cf. Section 4.6.3.2). Similarly, we regard the OpenTitan [172] hardware root of trust as an interesting project to explore new extensions that could assist TEE-based designs.

6.2.4 Protection Designs based on Non-TEE CPU Extensions

While the focus of this dissertation and the previous ideas is on TEEs, we want to emphasize the potential of the many non-TEE CPU extensions, such as memory protection keys, Intel Process Trace (PT), Intel Control-Flow Enforcement Technology (CET), Arm Pointer Authentication (PA), Arm Memory Tagging Extension (MTE), and many more. In this context, projects like GRIFFIN [82], ERIM [220] or Capacity [59] have shown how these CPU extensions can enable powerful OS-based attack detection or access control mechanisms. Therefore, we regard the exploration of new OS or hypervisor-level protection designs based on non-TEE CPU extensions as a promising research direction, with a focus on a slightly weaker threat model than TEE-based designs. Furthermore, in the long term, we envision many interesting research opportunities considering the question: *Can we design use case-specific protection designs based on lightweight (non-TEE) CPU extensions that provide similarly strong protection guarantees as TEEs?* This might include the development of new CPU extensions.

6.3 Concluding Thoughts

We see a high potential in TEEs to protect system and user services against strong system-level attacks and to remotely vouch for the enforcement of security and privacy policies. Therefore, throughout Chapters 2 to 4 of this dissertation, we showed several benefits of TEE-based service designs to foster wider adoption of TEEs. There has been an increasing trend towards hardware extensions (including TEEs) over the last years, especially with the advent of cloud computing and micro-architectural side-channel attacks, which we expect to continue. We expect that CPU vendors will develop additional TEE features worth exploring (cf. Section 6.2.1) as well as new flexible types of TEEs, similar to research projects like CURE [20] (cf. Section 6.2.3). In addition, TEE-like extensions are starting to enhance towards peripherals, e.g., as shown by TEE solutions for GPUs [223], or new proposals for PCIe device attestation. In this context, we hope that vendors and researchers tackle the challenge of cross-TEE incompatibilities (cf. Section 6.2.2) and avoid introducing new platform- or even peripheral-specific dependencies, restricting wide deployment of innovative protection designs. Furthermore, in particular for coarse-grained TEEs, we regard intra-TEE protections like 00SEVen (Chapter 5) as important in order to strengthen the users' trust in such technologies and thus foster their acceptance. While we did not focus on hardware and micro-architectural side-channel attacks, we want to highlight the need for additional hardware-software co-designs for protecting against such attacks. Vendors like AMD have already started providing new side-channel protections for SEV-SNP VMs, but there are still inherent architectural security-performance tradeoffs that render a full protection hard to achieve in practice, e.g., regarding CPU optimizations or the choice of the memory encryption algorithm of TEEs. In the long term, we envision a future with multiple types of TEEs: (1.) flexible general-purpose TEEs that are easy to adopt by users and provide a protection for the vast majority of users against a broad set of direct system-level and hardware attacks, as well as (2.) high-security TEEs for a limited set of critical components, which might face additional overhead and deployment costs but provide strong protection, potentially even achieving (full) side-channel resistance. That way, users can widely benefit from the additional security guarantees of TEEs without restricting high-end protection designs.

Bibliography

Author's Papers for this Dissertation

[P1] **Schwarz**, **F.** and Rossow, C. SENG, the SGX-Enforcing Network Gateway: Authorizing Communication from Shielded Clients. In: *29th USENIX Security Symposium*. 2020.

[P2] **Schwarz**, **F.** TrustedGateway: TEE-Assisted Routing and Firewall Enforcement Using ARM TrustZone. In: *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*. ACM, 2022.

[P3] **Schwarz**, **F.**, Do, K., Heide, G., Hanzlik, L., and Rossow, C. FeIDo: Recoverable FIDO2 Tokens Using Electronic IDs. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2022.

[P4] **Schwarz**, **F.** and Rossow, C. 00SEVen – Re-enabling Virtual Machine Forensics: Introspecting Confidential VMs using privileged in-VM Agents. In: *major revision (conditional accept) at USENIX Security 2024*.

Author's Technical Reports

[T1] **Schwarz**, **F.**, Do, K., Heide, G., Hanzlik, L., and Rossow, C. *FeIDo: Recoverable FIDO2 Tokens Using Electronic IDs (Extended Version)*. Technical Report. `https://publications.cispa.saarland/3894/`. 2023.

Open-Source Prototypes of this Dissertation

[S1] *SENG Prototypes*. URL: `https://github.com/sengsgx`.

[S2] *TrustedGateway Prototypes*. URL: `https://github.com/trugw`.

[S3] *FeIDo Prototypes*. URL: `https://github.com/feido-token`.

[S4] *00SEVen Prototypes*. URL: `https://github.com/sev-vmi/00seven`.

Other references

[1] Abdalla, M., Fouque, P.-A., and Pointcheval, D. Password-Based Authenticated Key Exchange in the Three-Party Setting. In: *Public Key Cryptography - PKC*. Springer, 2005.

[2] Abera, T., Asokan, N., Davi, L., Ekberg, J.-E., Nyman, T., Paverd, A., Sadeghi, A.-R., and Tsudik, G. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2016.

[3] Advanced Micro Devices, Inc. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. Tech. rep. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf. Jan. 2020.

[4] Advanced Micro Devices, Inc. *Secure VM Service Module for SEV-SNP Guests*. Tech. rep. Revision: 0.50, https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/58019.pdf. Aug. 2022.

[5] Advanced Micro Devices, Inc. *SEV Secure Nested Paging Firmware ABI Specification*. Tech. rep. https://www.amd.com/system/files/TechDocs/56860.pdf. Nov. 2022.

[6] Advanced Micro Devices, Inc. *AMD SEV-TIO: Trusted I/O for Secure Encrypted Virtualization*. Tech. rep. https://www.amd.com/system/files/documents/sev-tio-whitepaper.pdf. Mar. 2023.

[7] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Tech. rep. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf. 2023.

[8] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*. Tech. rep. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24594.pdf. 2023.

[9] Advanced Micro Devices, Inc. *SEV-ES Guest-Hypervisor Communication Block Standardization*. Tech. rep. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56421.pdf. Jan. 2023.

[10] Ahmad, A., Kim, K., Sarfaraz, M. I., and Lee, B. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. In: *25th Annual Network and Distributed System Security Symposium*. The Internet Society, 2018.

[11] Akhawe, D., Weinberger, J., Braun, F., and Marier, F. *Subresource Integrity*. W3C Recommendation. https://www.w3.org/TR/2016/REC-SRI-20160623/. W3C, June 2016.

[12] Amazon Web Services. *Amazon EC2 now supports AMD SEV-SNP*. 2023. URL: https://aws.amazon.com/about-aws/whats-new/2023/04/amazon-ec2-amd-sev-snp/?nc1=h_ls.

[13] Amazon Web Services. *Nitro Enclaves*. 2023. URL: https://aws.amazon.com/ec2/nitro/nitro-enclaves/.

[14] Arm Limited. *Trusted Board Boot Requirements CLIENT (TBBR-CLIENT) Armv8-A*. https://developer.arm.com/documentation/den0006/latest. 2018.

[15] Arm Limited. *Arm Platform Security Architecture Trusted Boot and Firmware Update 1.0*. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/PSA/DEN0072-PSA_TBFU_1-0-REL.pdf. 2019.

[16] Arm Limited. *Arm Confidential Compute Architecture*. 2023. URL: https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture.

[17] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., and Fetzer, C. SCONE: Secure Linux Containers with Intel SGX. In: *12th USENIX Symposium on Operating Systems Design and Implementation*. 2016.

[18] Asoni, D. E., Sasaki, T., and Perrig, A. Alcatraz: Data Exfiltration-Resilient Corporate Network Architecture. In: *IEEE 4th International Conference on Collaboration and Internet Computing*. 2018.

[19] Atamli, A., Petracca, G., and Crowcroft, J. IO-Trust: An out-of-Band Trusted Memory Acquisition for Intrusion Detection and Forensics Investigations in Cloud IOMMU Based Systems. In: *14th International Conference on Availability, Reliability and Security*. ACM, 2019.

[20] Bahmani, R., Brasser, F., Dessouky, G., Jauernig, P., Klimmek, M., Sadeghi, A.-R., and Stapf, E. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In: *30th USENIX Security Symposium*. 2021.

[21] Barabosch, T., Bergmann, N., Dombeck, A., and Padilla, E. Quincy: Detecting Host-Based Code Injection Attacks in Memory Dumps. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017.

[22] Barabosch, T. and Gerhards-Padilla, E. Host-based code injection attacks: A popular technique used by malware. *Proceedings of IEEE International Conference on Malicious and Unwanted Software (MALCON)* (2014).

[23] Barth, A., Jackson, C., and Mitchell, J. C. Robust Defenses for Cross-Site Request Forgery. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2008.

[24] Baumann, A., Peinado, M., and Hunt, G. Shielding Applications from an Untrusted Cloud with Haven. In: *11th USENIX Symposium on Operating Systems Design and Implementation*. 2014.

[25] Beesley, P., Garg, S., and Bailleux, S. *Trusted Board Boot*. 2020. URL: https://github.com/ARM-software/arm-trusted-firmware/blob/master/docs/design/trusted-board-boot.rst.

[26] Bellare, M. New proofs for NMAC and HMAC: Security without collision resistance. *Journal of Cryptology* 28, 4 (2015).

[27] Bender, J., Fischlin, M., and Kügler, D. Security Analysis of the PACE Key-Agreement Protocol. In: *Information Security*. Springer, 2009.

[28] Biometrics Research Group, Inc. *Apple launches web authentication using FIDO standard with Touch ID or Face ID biometrics in Safari*. 2020. URL: https://www.biometricupdate.com/202006/apple-launches-web-authentication-using-fido-standard-with-touch-id-or-face-id-biometrics-in-safari.

[29] Blue Bite LLC. *Android NFC Compatibility*. 2021. URL: https://www.bluebite.com/nfc/android-nfc-compatibility.

[30] Boundary Devices. *i.MX6 Embedded Single Board Computer (Nitrogen6X)*. 2022. URL: https://boundarydevices.com/product/nitrogen6x/.

[31] Brasser, F., Gens, D., Jauernig, P., Sadeghi, A.-R., and Stapf, E. SANCTUARY: ARMing TrustZone with User-space Enclaves. In: *26th Annual Network and Distributed System Security Symposium*. The Internet Society, 2019.

[32] Brenner, S., Wulf, C., Goltzsche, D., Weichbrodt, N., Lorenz, M., Fetzer, C., Pietzuch, P., and Kapitza, R. SecureKeeper: Confidential ZooKeeper Using Intel SGX. In: *Middleware Conference*. 2016.

[33] Brown, R. *Welcome to the OpenWrt Project*. 2022. URL: https://openwrt.org/.

[34] Buhren, R., Jacob, H.-N., Krachenfels, T., and Seifert, J.-P. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2021.

[35] Cabuk, S., Brodley, C. E., and Shields, C. IP Covert Timing Channels: Design and Detection. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2004.

[36] Carlini, M. Secure Boot on Arm systems. In: Linaro Connect San Francisco (SFO17). https://www.slideshare.net/linaroorg/secure-boot-on-arm-systems-building-a-complete-chain-of-trust-upon-existing-industry-standards-using-opensource-firmware-sfo17201. 2017.

[37] Chakraborty, D. and Bugiel, S. SimFIDO: FIDO2 User Authentication with simTPM. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019.

[38] Chakraborty, D., Hanzlik, L., and Bugiel, S. simTPM: User-centric TPM for Mobile Devices. In: *28th USENIX Security Symposium*. 2019.

[39] Checkoway, S. and Shacham, H. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2013.

[40] Chen, Y., Li, J., Xu, G., Zhou, Y., Wang, Z., Wang, C., and Ren, K. SGXLock: Towards Efficiently Establishing Mutual Distrust Between Host Application and Enclave for SGX. In: *31st USENIX Security Symposium*. 2022.

[41] Cheng, Y. and Ding, X. Guardian: Hypervisor as Security Foothold for Personal Computers. In: *Trust and Trustworthy Computing - 6th International Conference*. Springer, 2013.

[42] Cisco Systems, Inc. *Cisco IOS XR Software Release 6.0 Operational Enhancements Data Sheet*. Nov. 2015. URL: https://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-xr-software/datasheet-c78-736154.html.

[43] Cisco Systems, Inc. *KVM App Hosting on a Cisco Router*. Mar. 2020. URL: https://www.cisco.com/c/en/us/products/collateral/routers/4000-series-integrated-services-routers-isr/at-a-glance-c45-737753.html.

[44] Cisco Systems, Inc. *Troubleshoot High CPU Usage in Catalyst Switch Platforms Running IOS-XE 16.x*. Jan. 2021. URL: https://www.cisco.com/c/en/us/support/docs/ios-nx-os-software/ios-xe-16/213549-troubleshoot-high-cpu-usage-in-catalyst.html.

[45] Cisco Systems, Inc. *NVM*. URL: https://www.cisco.com/c/dam/global/en_au/assets/pdf/anyconnect-network-visibility.pdf.

[46] *The CLIP OS Project*. 2020. URL: https://clip-os.org/en/.

[47] Cloosters, T., Rodler, M., and Davi, L. TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves. In: *29th USENIX Security Symposium*. 2020.

[48] Corbet, J., Rubini, A., and Kroah-Hartman, G. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005. ISBN: 0596005903.

[49] Costan, V. and Devadas, S. *Intel SGX Explained*. Cryptology ePrint Archive, Paper 2016/086. https://eprint.iacr.org/2016/086. 2016.

[50] Dagdelen, Ö. and Fischlin, M. Security Analysis of the Extended Access Control Protocol for Machine Readable Travel Documents. In: *Information Security*. Springer, 2011.

[51] Dangl, T., Sentanoe, S., and Reiser, H. P. VMIFresh: Efficient and Fresh Caches for Virtual Machine Introspection. In: *17th International Conference on Availability, Reliability and Security*. ACM, 2022.

[52] Danial, A. *cloc: Count Lines of Code*. 2022. URL: https://github.com/AlDanial/cloc.

[53] Danisevskis, J. *Android Protected Confirmation: Taking transaction security to the next level*. 2018. URL: https://android-developers.googleblog.com/2018/10/android-protected-confirmation.html.

[54] Dell'Amico, M., Michiardi, P., and Roudier, Y. Password strength: An empirical analysis. In: *Proceedings IEEE INFOCOM*. 2010.

[55] Deng, S., Li, M., Tang, Y., Wang, S., Yan, S., and Zhang, Y. CipherH: Automated Detection of Ciphertext Side-channel Vulnerabilities in Cryptographic Implementations. In: *32nd USENIX Security Symposium*. 2023.

[56] Dessouky, G., Frassetto, T., and Sadeghi, A.-R. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In: *29th USENIX Security Symposium*. 2020.

[57] *Diamorphine*. URL: https://github.com/m0nad/Diamorphine.

[58] Dingledine, R., Mathewson, N., and Syverson, P. Tor: The Second-Generation Onion Router. In: *13th USENIX Security Symposium*. 2004.

[59] Dinh Duy, K., Cho, K., Noh, T., and Lee, H. Capacity: Cryptographically-Enforced In-Process Capabilities for Modern ARM Architectures. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2023.

[60] *Docker networking*. URL: https://docs.docker.com/network/.

[61] Donenfeld, J. A. WireGuard: Next Generation Kernel Network Tunnel. In: *24th Annual Network and Distributed System Security Symposium*. The Internet Society, 2017.

[62] Duan, H., Wang, C., Yuan, X., Zhou, Y., Wang, Q., and Ren, K. LightBox: Full-Stack Protected Stateful Middlebox at Lightning Speed. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2019.

[63] Duo Labs. *WebAuthn.io (Github)*. 2020. URL: https://github.com/duo-labs/webauthn.io.

[64] Duo Labs. *WebAuthn.io: A demo of the WebAuthn specification*. 2021. URL: https://webauthn.io/.

[65] Eiband, M., Khamis, M., Von Zezschwitz, E., Hussmann, H., and Alt, F. Understanding shoulder surfing in the wild: Stories from users and observers. In: *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2017.

[66] embeDD GmbH. *DD-WRT*. 2022. URL: https://dd-wrt.com/.

[67] Emscripten Contributors. *Emscripten documentation*. 2022. URL: https://emscripten.org.

[68] Eskandarian, S., Cogan, J., Birnbaum, S., Brandon, P. C. W., Franke, D., Fraser, F., Garcia, G., Gong, E., Nguyen, H. T., Sethi, T. K., Subbiah, V., Backes, M., Pellegrino, G., and Boneh, D. Fidelius: Protecting User Secrets from Compromised Browsers. In: *IEEE Symposium on Security and Privacy*. 2019.

[69] Kuvaiskii, D. *Add exitless system calls (PR 405)*. URL: https://github.com/oscarlab/graphene/pull/405.

[70] f1yyy. *The Great Escape of ESXi: Breaking Out of a Sandboxed Virtual Machine*. 2019. URL: https://media.ccc.de/v/36c3-10505-the_great_escape_of_esxi.

[71] FIDO Alliance. *Using FIDO with eIDAS Services*. 2020. URL: https://fidoalliance.org/wp-content/uploads/2020/04/FIDO-deploying-FIDO2-eIDAS-QTSPs-eID-schemes-white-paper.pdf.

[72] FIDO Alliance. *Choosing FIDO Authenticators for Enterprise Use Cases*. 2021. URL: https://fidoalliance.org/wp-content/uploads/2021/09/FIDO-White-Paper-Choosing-FIDO-Authenticators-for-Enterprise-Use-Cases.pdf.

[73] FIDO Alliance. *FIDO Security Reference*. 2021. URL: https://fidoalliance.org/specs/common-specs/fido-security-ref-v2.1-rd-20210525.html.

[74] FIDO Alliance. *How FIDO Addresses a Full Range of Use Cases*. 2022. URL: https://fidoalliance.org/wp-content/uploads/2022/03/How-FIDO-Addresses-a-Full-Range-of-Use-Cases-March24.pdf.

[75] FireEye. *M-Trends 2019*. URL: https://web.archive.org/web/20230329034215/https://content.fireeye.com/m-trends/rpt-m-trends-2019.

[76] *FireMon's State of the Firewall*. 2019. URL: https://web.archive.org/web/20220330075718/www.firemon.com/2019-state-of-the-firewall-report/.

[77] Franzen, F., Holl, T., Andreas, M., Kirsch, J., and Grossklags, J. Katana: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots. In: *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*. ACM, 2022.

[78] Frisk, U. *PCILeech*. URL: https://github.com/ufrisk/pcileech.

[79] Frisk, U. *The LeechCore Physical Memory Acquisition Library*. URL: https://github.com/ufrisk/LeechCore.

[80] Gallenmüller, S., Schöffmann, D., Scholz, D., Geyer, F., and Carle, G. *DTLS Performance - How Expensive is Security?* 2019. arXiv: 1904.11423 [cs.NI].

[81] Garfinkel, T. and Rosenblum, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: *Annual Network and Distributed System Security Symposium*. The Internet Society, 2003.

[82] Ge, X., Cui, W., and Jaeger, T. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In: *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017.

[83] Ge, X., Kuo, H.-C., and Cui, W. Hecate: Lifting and Shifting On-Premises Workloads to an Untrusted Cloud. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2022.

[84] Ghorbani Lyastani, S., Schilling, M., Neumayr, M., Backes, M., and Bugiel, S. Is FIDO2 the Kingslayer of User Authentication? A Comparative Usability Study of FIDO2 Passwordless Authentication. In: *IEEE Symposium on Security and Privacy*. 2020.

[85] Gkantsidis, C., Karagiannis, T., Naylor, D., Li, R., and Steenkiste, P. *And Then There Were More: Secure Communication for More Than Two Parties*. Tech. rep. MSR-TR-2017-24. https://www.microsoft.com/en-us/research/publication/therewere-secure-communication-two-parties/. 2017.

[86] Global Platform Inc. *TEE Client API Specification v1.0*. https://globalplatform.org/specs-library/tee-client-api-specification/. 2010.

[87] Goltzsche, D., Rüsch, S., Nieke, M., Vaucher, S., Weichbrodt, N., Schiavoni, V., Aublin, P., Cosa, P., Fetzer, C., Felber, P., Pietzuch, P., and Kapitza, R. EndBox: Scalable Middlebox Functions Using Client-Side Trusted Execution. In: *IEEE/IFIP Conference on Dependable Systems and Networks*. 2018.

[88] Goltzsche, D., Wulf, C., Muthukumaran, D., Rieck, K., Pietzuch, P. R., and Kapitza, R. TrustJS: Trusted Client-side Execution of JavaScript. In: *Proceedings of the 10th European Workshop on Systems Security*. ACM, 2017.

[89] Gonçalves, S., Tomasi, A., Bisegna, A., Pellizzari, G., and Ranise, S. Verifiable Contracting: A Use Case for Onboarding and Contract Offering in Financial Services with eIDAS and Verifiable Credentials. In: *Computer Security*. Springer, 2020.

[90] Google LLC. *Trusty TEE | Android Open Source Project*. 2020. URL: https://source.android.com/security/trusty.

[91] Google LLC. *chrome.enterprise.platformKeys – Chrome Developers*. 2022. URL: https://developer.chrome.com/docs/extensions/reference/enterprise_platformKeys/.

[92] Google LLC. *Hardware-backed Keystore | Android Open Source Project*. 2022. URL: https://source.android.com/security/keystore.

[93] Google LLC. *Confidential Computing concepts*. 2023. URL: https://cloud.google.com/compute/confidential-vm/docs/about-cvm.

[94] Google LLC. *GRR Rapid Reponse*. URL: https://github.com/google/grr.

[95] Governikus GmbH & Co. KG. *AusweisApp2: Passende Smartphones & Tablets für die Online-Ausweisfunktion*. 2022. URL: https://www.ausweisapp.bund.de/mobile-geraete.

[96] Gowda, A., Withrow, M., and Bontha, H. *Kata confidential containers with Azure Kubernetes Service*. 2023. URL: https://techcommunity.microsoft.com/t5/azure-confidential-computing/aligning-with-kata-confidential-containers-to-achieve-zero-trust/ba-p/3797876.

[97] *Gramine*. URL: https://gramineproject.io/.

[98] Gu, Z., Deng, Z., Xu, D., and Jiang, X. Process Implanting: A New Active Introspection Framework for Virtualization. In: *Symposium on Reliable Distributed Systems*. IEEE Computer Society, 2011.

[99] Guerra, M., Taubmann, B., Reiser, H. P., Yalew, S., and Correia, M. Introspection for ARM TrustZone with the ITZ Library. In: *IEEE International Conference on Software Quality, Reliability and Security*. 2018.

[100] gVisor Authors. *What is gVisor?* 2024. URL: https://gvisor.dev/docs/.

[101] Hanzlik, L., Loss, J., and Wagner, B. *Token meets Wallet: Formalizing Privacy and Revocation for FIDO2*. 2022. URL: https://ia.cr/2022/084.

[102] Houmansadr, A., Brubaker, C., and Shmatikov, V. The Parrot Is Dead: Observing Unobservable Network Communications. In: *IEEE Symposium on Security and Privacy*. 2013.

[103] Intel Corporation. *Intel SGX Data Center Attestation Primitives*. 2022. URL: https://download.01.org/intel-sgx/sgx-dcap/1.14/linux/docs/DCAP_ECDSA_Orientation.pdf.

[104] Intel Corporation. *Intel TDX Connect TEE-IO Device Guide*. Tech. rep. https://cdrdv2.intel.com/v1/dl/getContent/772642. Feb. 2023.

[105] Intel Corporation. *Intel TDX Module v1.5 TD Partitioning Architecture Specification*. 2023. URL: https://www.intel.com/content/www/us/en/content-details/773039/intel-tdx-module-v1-5-td-partitioning-architecture-specification.html.

[106] Intel Corporation. *Intel Trust Domain Extensions (Intel TDX)*. 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[107] Intel Corporation. *Intel SGX SDK for Linux OS*. URL: https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/linux-overview.html.

[108] International Civil Avaiation Organization. *Machine Readable Travel Documents Part 11: Security Mechanisms for MRTDs*. Tech. rep. https://www.icao.int/publications/documents/9303_p11_cons_en.pdf. 2021.

[109] International Civil Avaiation Organization. *Machine Readable Travel Documents Part 3: Specifications Common to all MRTDs*. Tech. rep. https://www.icao.int/publications/Documents/9303_p3_cons_en.pdf. 2021.

[110] Interpol. *I-Checkit - FAQs brochure - Private Sector Partners*. 2022. URL: https://www.interpol.int/content/download/12470/file/I-Checkit_FAQs_brochure_private%20sector_EN_LR_02.pdf?inLanguage=eng-GB.

[111] *iPerf - The TCP, UDP and SCTP network bandwidth measurement tool*. 2022. URL: https://iperf.fr/.

[112] *iptables Application level firewalling*. 2005. URL: https://debian-administration.org/article/120/Application_level_firewalling.

[113] Islam, M. S., Zamani, M., Kim, C. H., Khan, L., and Hamlen, K. W. Confidential Execution of Deep Learning Inference at the Untrusted Edge with ARM TrustZone. In: *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy*. 2023.

[114] Jaeger, T., King, D. H., Butler, K. R., Hallyn, S., Latten, J., and Zhang, X. Leveraging IPsec for Mandatory Per-Packet Access Control. In: *Securecomm and Workshops*. IEEE, 2006.

[115] Jain, B., Baig, M. B., Zhang, D., Porter, D. E., and Sion, R. SoK: Introspections on Trust and the Semantic Gap. In: *IEEE Symposium on Security and Privacy*. 2014.

[116] Jang, D., Lee, H., Kim, M., Kim, D., Kim, D., and Kang, B. B. ATRA: Address Translation Redirection Attack against Hardware-Based External Monitors. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2014.

[117] Jang, Y. Building trust in the user I/O in computer systems. https://repository.gatech.edu/entities/publication/d079e6ad-5e2f-4e46-bb7a-67d456d0a9ca. PhD thesis. 2017.

[118] Juniper Networks, Inc. *Junos OS Evolved Overview*. May 2022. URL: https://www.juniper.net/documentation/us/en/software/junos/overview-evo/topics/concept/evo-overview.html.

[119] Juniper Networks, Inc. *Junos OS Overview*. May 2022. URL: https://www.juniper.net/documentation/us/en/software/junos/junos-install-upgrade/topics/topic-map/junos-os-overview.html.

[120] Juniper Networks, Inc. *VM Host Overview (Junos OS)*. May 2022. URL: https://www.juniper.net/documentation/us/en/software/junos/junos-install-upgrade/topics/topic-map/vm-host-overview.html.

[121] Kang, Q., Xue, L., Morrison, A., Tang, Y., Chen, A., and Luo, X. Programmable In-Network Security for Context-aware BYOD Policies. In: *29th USENIX Security Symposium*. 2020.

[122] Karande, V., Bauman, E., Lin, Z., and Khan, L. SGX-Log: Securing System Logs With SGX. In: *Proceedings of the ACM on Asia Conference on Computer and Communications Security*. 2017.

[123] Kerrisk, M. *ip-route(8) - Linux manual page*. 2012. URL: https://man7.org/linux/man-pages/man8/ip-route.8.html.

[124] *Keystone Enclave*. 2019. URL: https://keystone-enclave.org/.

[125] Kim, S. W., Lee, C., Jeon, M., Kwon, H. Y., Lee, H. W., and Yoo, C. Secure device access for automotive software. In: *International Conference on Connected Vehicles and Expo*. IEEE, 2013.

[126] Kim, S., Han, J., Ha, J., Kim, T., and Han, D. Enhancing Security and Privacy of Tor's Ecosystem by Using Trusted Execution Environments. In: *14th USENIX Symposium on Networked Systems Design and Implementation*. 2017.

[127] Kim, T. and Zeldovich, N. Practical and Effective Sandboxing for Non-root Users. In: *USENIX Annual Technical Conference*. 2013.

[128] Knauth, T., Steiner, M., Chakrabarti, S., Lei, L., Xing, C., and Vij, M. Integrating Remote Attestation with Transport Layer Security. *CoRR* abs/1801.05863 (2018). arXiv: 1801.05863.

[129] Krawczyk, H. Cryptographic extraction and key derivation: The HKDF scheme. In: *Annual Cryptology Conference*. Springer. 2010.

[130] Kunke, J., Wiefling, S., Ullmann, M., and Lo Iacono, L. Evaluation of Account Recovery Strategies with FIDO2-based Passwordless Authentication. In: *Open Identity Summit*. Gesellschaft für Informatik e.V., 2021.

[131] Lal, R. and Pappachan, P. M. An architecture methodology for secure video conferencing. In: *IEEE International Conference on Technologies for Homeland Security*. 2013.

[132] Le Pochat, V., Van Goethem, T., Tajalizadehkhoob, S., Korczyński, M., and Joosen, W. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In: *26th Annual Network and Distributed System Security Symposium*. The Internet Society, 2019.

[133] Lee, S., Shih, M.-W., Gera, P., Kim, T., Kim, H., and Peinado, M. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In: *26th USENIX Security Symposium*. 2017.

[134] Lei, Z., Nan, Y., Fratantonio, Y., and Bianchi, A. On the Insecurity of SMS One-Time Password Messages against Local Attackers in Modern Mobile Devices. In: *28th Annual Network and Distributed System Security Symposium*. The Internet Society, 2021.

[135] Lentz, M., Sen, R., Druschel, P., and Bhattacharjee, B. SeCloak: ARM Trustzone-Based Mobile Peripheral Control. In: *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2018.

[136] Leonard, T. *QubesOS Mirage Firewall*. 2022. URL: https://github.com/mirage/qubes-mirage-firewall/.

[137] Li, M., Wilke, L., Wichelmann, J., Eisenbarth, T., Teodorescu, R., and Zhang, Y. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In: *IEEE Symposium on Security and Privacy*. 2022.

[138] Li, M., Zhang, Y., and Lin, Z. CrossLine: Breaking "Security-by-Crash" Based Memory Isolation in AMD SEV. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2021.

[139] Li, M., Zhang, Y., Lin, Z., and Solihin, Y. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In: *28th USENIX Security Symposium*. 2019.

[140] Li, W., Ma, M., Han, J., Xia, Y., Zang, B., Chu, C.-K., and Li, T. Building Trusted Path on Untrusted Device Drivers for Mobile Devices. In: *Proceedings of 5th Asia-Pacific Workshop on Systems*. ACM, 2014.

[141] *libuv*. URL: https://libuv.org/.

[142] LibVMI Project. *LibVMI: Simplified Virtual Machine Introspection*. URL: https://github.com/libvmi/libvmi.

[143] Ligh, M. H., Case, A., Levy, J., and Walters, A. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. 1st. Wiley Publishing, 2014. ISBN: 1118825098.

[144] Linaro Limited. *DeviceTree*. 2022. URL: https://www.devicetree.org.

[145] Linaro Limited. *Open Portable Trusted Execution Environment - OP-TEE*. 2022. URL: https://www.op-tee.org/.

[146] Lind, J., Priebe, C., Muthukumaran, D., O'Keeffe, D., Aublin, P.-L., Kelbert, F., Reiher, T., Goltzsche, D., Eyers, D., Kapitza, R., Fetzer, C., and Pietzuch, P. Glamdring: Automatic Application Partitioning for Intel SGX. In: *USENIX Annual Technical Conference*. 2017.

[147] Liu, H., Xing, J., Huang, Y., Zhuo, D., Devadas, S., and Chen, A. Remote Direct Memory Introspection. In: *32nd USENIX Security Symposium*. 2023.

[148] Liu, R. and Srivastava, M. PROTC: PROTeCting Drone's Peripherals through ARM TrustZone. In: *Proceedings of the 3rd Workshop on Micro Aerial Vehicle Networks, Systems, and Applications*. ACM, 2017.

[149] Luber, S. and Donner, A. *Was ist eine Fritzbox?* Nov. 2019. URL: https://www.ip-insider.de/was-ist-eine-fritzbox-a-883753/.

[150] Lundberg, E., Jones, M., Jones, J., Kumar, A., and Hodges, J. *Web Authentication: An API for accessing Public Key Credentials - Level 2*. W3C Recommendation. https://www.w3.org/TR/2021/REC-webauthn-2-20210408/. W3C, 2021.

[151] *lwIP*. URL: https://savannah.nongnu.org/projects/lwip/.

[152] Matetic, S., Schneider, M., Miller, A., Juels, A., and Capkun, S. DelegaTEE: Brokered Delegation Using Trusted Execution Environments. In: *27th USENIX Security Symposium*. 2018.

[153] McCormack, M., Vasudevan, A., Liu, G., Echeverría, S., O'Meara, K., Lewis, G., and Sekar, V. Towards an Architecture for Trusted Edge IoT Security Gateways. In: *3rd USENIX Workshop on Hot Topics in Edge Computing*. 2020.

[154] Ménétrey, J., Göttel, C., Pasin, M., Felber, P., and Schiavoni, V. An Exploratory Study of Attestation Mechanisms for Trusted Execution Environments. In: *5th Workshop on System Software for Trusted Execution*. 2022.

[155] Morbitzer, M., Proskurin, S., Radev, M., Dorfhuber, M., and Salas, E. SEVerity: Code Injection Attacks against Encrypted Virtual Machines. In: *Security and Privacy Workshops*. IEEE Computer Society, 2021.

[156] Morbitzer, M., Huber, M., and Horsch, J. Extracting Secrets from Encrypted Virtual Machines. In: *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*. 2019.

[157] Morbitzer, M., Huber, M., Horsch, J., and Wessel, S. SEVered: Subverting AMD's Virtual Machine Encryption. In: *Proceedings of the 11th European Workshop on Systems Security*. ACM, 2018.

[158] Morgner, F., Bastian, P., and Fischlin, M. Securing Transactions with the eIDAS Protocols. In: *Information Security Theory and Practice*. Springer, 2016.

[159] Morgner, F. and Oepen, D. *OpenPACE*. URL: https://frankmorgner.github.io/openpace/.

[160] Nadim, M., Lee, W., and Akopian, D. *Kernel-level Rootkit Detection, Prevention and Behavior Profiling: A Taxonomy and Survey*. 2023. arXiv: 2304.00473 [cs.CR].

[161] Narayanan, V., Carvalho, C., Ruocco, A., Almási, G., Bottomley, J., Ye, M., Feldman-Fitzthum, T., Buono, D., Franke, H., and Burtsev, A. *Remote attestation of SEV-SNP confidential VMs using e-vTPMs*. 2023. arXiv: 2303.16463 [cs.CR].

[162] *National ID cards: 2016-2021 facts and trends*. 2021. URL: https://www.thalesgroup.com/en/markets/digital-identity-and-security/government/identity/2016-national-id-card-trends.

[163] *netfilter*. 2019. URL: https://www.netfilter.org/.

[164] NXP Semiconductors. *i.MX 6Dual/6Quad Applications Processors for Consumer Products*. https://www.nxp.com/docs/en/data-sheet/IMX6DQCEC.pdf. 2018.

[165] OASIS Open. Virtual I/O Device (VIRTIO) Version 1.1. In: ed. by Tsirkin, M. S. and Huck, C. https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html. OASIS Committee, 2019.

[166] Oest, A., Zhang, P., Wardman, B., Nunes, E., Burgis, J., Zand, A., Thomas, K., Doupé, A., and Ahn, G.-J. Sunrise to sunset: Analyzing the end-to-end life cycle and effectiveness of phishing attacks at scale. In: *29th USENIX Security Symposium*. 2020.

[167] Oku, K., Matsuno, T., Murase, D., and Mitsunari, S. *PicoHTTPParser*. 2021. URL: https://github.com/h2o/picohttpparser.

[168] Oleksenko, O., Trach, B., Krahn, R., Silberstein, M., and Fetzer, C. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In: *USENIX Annual Technical Conference*. 2018.

[169] Oliveri, A., Dell'Amico, M., and Balzarotti, D. An OS-agnostic Approach to Memory Forensics. In: *30th Annual Network and Distributed System Security Symposium*. The Internet Society, 2023.

[170] Oostdijk, M. *JMRTD: An Open Source Java Implementation of Machine Readable Travel Documents*. URL: https://jmrtd.org/.

[171] *Open Enclave SDK*. URL: https://openenclave.io/sdk/.

[172] *Open source silicon root of trust (RoT) | OpenTitan*. URL: https://opentitan.org/.

[173] Orenbach, M., Lifshits, P., Minkin, M., and Silberstein, M. Eleos: ExitLess OS Services for SGX Enclaves. In: *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017.

[174] Organización Internacional de Normalización. *ISO IEC 7816-4: Identification cards–Integrated circuit cards. Organization, security and commands for interchange*. ISO, 2020.

[175] Park, H., Zhai, S., Lu, L., and Lin, F. X. StreamBox-TZ: Secure Stream Analytics at the Edge with TrustZone. In: *USENIX Annual Technical Conference*. 2019.

[176] Parno, B., Zhou, Z., and Perrig, A. Using Trustworthy Host-based Information in the Network. In: *Workshop on Scalable Trusted Computing (STC)*. ACM, 2012.

[177] Payne, B. D. An Introduction to Virtual Machine Introspection Using LibVMI (slides). In: *Malware Memory Forensics Workshop (MMF)*. MMF '14. 2014.

[178] Pinto, S. and Santos, N. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv.* 51, 6 (2019).

[179] Poddar, R., Lan, C., Popa, R. A., and Ratnasamy, S. SafeBricks: Shielding Network Functions in the Cloud. In: *15th USENIX Symposium on Networked Systems Design and Implementation*. 2018.

[180] Pontarelli, S., Bifulco, R., Bonola, M., Cascone, C., Spaziani, M., Bruschi, V., Sanvito, D., Siracusano, G., Capone, A., Honda, M., Huici, F., and Siracusano, G. FlowBlaze: Stateful Packet Processing in Hardware. In: *16th USENIX Symposium on Networked Systems Design and Implementation*. 2019.

[181] *Popular Baby Names (US)*. 2021. URL: https://www.ssa.gov/oact/babynames/limits.html.

[182] Priebe, C., Vaswani, K., and Costa, M. EnclaveDB: A Secure Database Using SGX. In: *IEEE Symposium on Security and Privacy*. 2018.

[183] Qi, Z., Qu, Y., and Yin, H. LogicMEM: Automatic Profile Generation for Binary-Only Memory Forensics via Logic Inference. In: *29th Annual Network and Distributed System Security Symposium*. The Internet Society, 2022.

[184] Qian, C., Hu, H., Alharthi, M., Chung, P. H., Kim, T., and Lee, W. RAZOR: A Framework for Post-deployment Software Debloating. In: *28th USENIX Security Symposium*. 2019.

[185] Quach, A., Prakash, A., and Yan, L. Debloating Software through Piece-Wise Compilation and Loading. In: *27th USENIX Security Symposium*. 2018.

[186] *The Qubes OS Project*. 2020. URL: https://www.qubes-os.org/.

[187] Rasiukevicius, M. *NPF-Router: a demo NPF+DPDK application*. 2021. URL: https://github.com/rmind/npf/tree/master/app.

[188] Rasiukevicius, M. *NPF: stateful packet filter supporting NAT, IP sets, etc*. 2021. URL: https://github.com/rmind/npf.

[189] *Regulation (EU) 2019/1157 of the European Parliament and of the Council*. 2019. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32019R1157.

[190] Rostedt, S. *ftrace*. 2008. URL: https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[191] Saleem, H. and Naveed, M. SoK: Anatomy of Data Breaches. *Proc. Priv. Enhancing Technol.* 2020, 4 (2020).

[192] Sasy, S., Gorbunov, S., and Fletcher, C. W. ZeroTrace : Oblivious Memory Primitives from Intel SGX. In: *25th Annual Network and Distributed System Security Symposium*. The Internet Society, 2018.

[193] SatoshiLabs s.r.o. *Trezor Hardware Wallet*. 2022. URL: https://trezor.io/.

[194] Schumilo, S., Aschermann, C., Abbasi, A., Wörner, S., and Holz, T. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In: *30th USENIX Security Symposium*. 2021.

[195] Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., and Holz, T. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In: *26th USENIX Security Symposium*. 2017.

[196] SEG / DrayTek UK. *O/S versions on Vigor 2760 Series Routers (DrayOS/Linux)*. Dec. 2015. URL: https://www.draytek.co.uk/support/guides/os-versions-on-vigor-2760-series-routers.

[197] *SELinux*. 2019. URL: https://selinuxproject.org/page/NB_LSM.

[198] Seshadri, A., Luk, M., Qu, N., and Perrig, A. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. *SIGOPS Oper. Syst. Rev.* 41 (2007).

[199] Scarlata, V., Johnson, S., Beaney, J., and Zmijewski, P. *Supporting Third Party Attestation for Intel SGX with Intel Data Center Attestation Primitives*. https://cdrdv2-public.intel.com/671314/intel-sgx-support-for-third-party-attestation.pdf. 2018.

[200] *shadowsocks*. URL: https://shadowsocks.org/.

[201] Sharif, M. I., Lee, W., Cui, W., and Lanzi, A. Secure In-VM Monitoring Using Hardware Virtualization. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2009.

[202] Shen, Y., Tian, H., Chen, Y., Chen, K., Wang, R., Xu, Y., and Xia, Y. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2020.

[203] Shih, M.-W., Lee, S., Kim, T., and Peinado, M. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In: *Network and Distributed System Security Symposium*. 2017.

[204] Shinde, S., Tien, D. L., Tople, S., and Saxena, P. Panoply: Low-TCB Linux Applications With SGX Enclaves. In: *24th Annual Network and Distributed System Security Symposium*. The Internet Society, 2017.

[205] SJB Research Ltd. *Confirmed: iOS 13 to include support for NFC passport reading - NFCW*. 2019. URL: https://www.nfcw.com/2019/06/07/362943/confirmed-ios-13-to-include-support-for-nfc-passport-reading/.

[206] *Smack (LSM)*. 2019. URL: https://schaufler-ca.com/.

[207] *Spy*. 2021. URL: https://github.com/jarun/spy.

[208] Srivastava, A. and Giffin, J. Tamper-Resistant, Application-Aware Blocking of Malicious Network Connections. In: *Recent Advances in Intrusion Detection, 11th International Symposium*. Springer, 2008.

[209] *Standardized Digital Identity on National Identity Cards*. 2020. URL: https://www.calctopia.com/2020/02/14/standardized-digital-identity-on-national-identity-cards/.

[210] Suciu, D., McLaughlin, S., Simon, L., and Sion, R. Horizontal Privilege Escalation in Trusted Applications. In: *29th USENIX Security Symposium*. 2020.

[211] *Sutekh*. URL: https://github.com/PinkP4nther/Sutekh.

[212] Sweeney, L. Simple demographics often identify people uniquely. *Health (San Francisco)* 671, 2000 (2000).

[213] Team libtom. *LibTomCrypt*. 2022. URL: https://github.com/libtom/libtomcrypt.

[214] *The electronic passport in 2021 and beyond*. 2021. URL: https://www.thalesgroup.com/en/markets/digital-identity-and-security/government/passport/electronic-passport-trends.

[215] The kernel development community. *Interrupts – The Linux Kernel documentation*. 2021. URL: https://linux-kernel-labs.github.io/refs/heads/master/lectures/interrupts.html#interrupt-context.

[216] The Linux Foundation. *Xen Project*. 2022. URL: https://xenproject.org/.

[217] The Qubes OS Project and others. *Firewall | Qubes OS*. 2022. URL: https://www.qubes-os.org/doc/firewall/.

[218] Trach, B., Krohmer, A., Gregor, F., Arnautov, S., Bhatotia, P., and Fetzer, C. ShieldBox: Secure Middleboxes Using Shielded Execution. In: *Symposium on SDN Research*. ACM.

[219] Tsai, C.-c., Porter, D. E., and Vij, M. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In: *USENIX Annual Technical Conference*. 2017.

[220] Vahldiek-Oberwagner, A., Elnikety, E., Duarte, N. O., Sammler, M., Druschel, P., and Garg, D. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In: *28th USENIX Security Symposium*. 2019.

[221] *VirtIO-drivers-rs*. URL: https://github.com/rcore-os/virtio-drivers.

[222] Volatility Foundation. *The Volatility Foundation - Open Source Memory Forensics*. URL: https://www.volatilityfoundation.org/.

[223] Volos, S., Vaswani, K., and Bruno, R. Graviton: Trusted Execution Environments on GPUs. In: *13th USENIX Symposium on Operating Systems Design and Implementation*. 2018.

[224] Vykhodtsev, A. *page-load-time*. 2021. URL: https://github.com/alex-vv/page-load-time.

[225] Wang, W., Chen, G., Pan, X., Zhang, Y., Wang, X., Bindschaedler, V., Tang, H., and Gunter, C. A. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2017.

[226] Wichelmann, J., Pätschke, A., Wilke, L., and Eisenbarth, T. Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software. In: *32nd USENIX Security Symposium*. 2023.

[227] Wilhelm, F. Tracing Privileged Memory Accesses to Discover Software Vulnerabilities. Master Thesis. Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, 2015.

[228] Wilke, L., Wichelmann, J., Morbitzer, M., and Eisenbarth, T. SEVurity: No Security Without Integrity : Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In: *IEEE Symposium on Security and Privacy*. 2020.

[229] Willems, C., Hund, R., and Holz, T. CXPInspector : Hypervisor-Based , Hardware-Assisted System Monitoring. In: Ruhr-Universitat Bochum, 2012.

[230] Wilson, A. *Website Load Time Statistics*. Feb. 2020. URL: https://www.top10-websitehosting.co.uk/website-load-time-statistics/.

[231] Wójcik, B. *Windows Hot Patching Mechanism Explained*. 2020. URL: https://dev.to/bartosz/windows-hot-patching-mechanism-explained-2m1f.

[232] Wu, W., Chen, Y., Xing, X., and Zou, W. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In: *28th USENIX Security Symposium*. 2019.

[233] Xiao, Y., Li, M., Chen, S., and Zhang, Y. STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2017.

[234] Xing, J., Kang, Q., and Chen, A. NetWarden: Mitigating Network Covert Channels while Preserving Performance. In: *29th USENIX Security Symposium*. 2020.

[235] Yee, B., Sehr, D., Dardyk, G., Chen, B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In: *IEEE Symposium on Security and Privacy*. 2009.

[236] Ying, K., Ahlawat, A., Alsharifi, B., Jiang, Y., Thavai, P., and Du, W. TruZ-Droid: Integrating TrustZone with Mobile Operating System. In: *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2018.

[237] Yu, J. Z., Shinde, S., Carlson, T. E., and Saxena, P. Elasticlave: An Efficient Memory Model for Enclaves. In: *31st USENIX Security Symposium*. 2022.

[238] Yubico. *Losing Your YubiKey - Yubico*. 2021. URL: https://support.yubico.com/hc/en-us/articles/360013647620-Losing-Your-YubiKey.

[239] Yubico. *Spare YubiKeys*. 2022. URL: https://www.yubico.com/spare/.

[240] Yubico. *WebAuthn - Account Recovery*. 2022. URL: https://developers.yubico.com/WebAuthn/WebAuthn_Developer_Guide/Account_Recovery.html.

[241] Zhao, S., Li, M., Zhangyz, Y., and Lin, Z. vSGX: Virtualizing SGX Enclaves on AMD SEV. In: *IEEE Symposium on Security and Privacy*. 2022.

[242] Zhao, S., Ding, X., Xu, W., and Gu, D. Seeing Through The Same Lens: Introspecting Guest Address Space At Native Speed. In: *26th USENIX Security Symposium*. 2017.

[243] Zhao, W., Lu, K., Qi, Y., and Qi, S. MPTEE: bringing flexible and efficient memory protection to Intel SGX. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. ACM, 2020.

[244] Zhou, L., Ding, X., and Zhang, F. Smile: Secure Memory Introspection for Live Enclave. In: *IEEE Symposium on Security and Privacy*. 2022.

[245] Zhou, Z., Gligor, V. D., Newsome, J., and McCune, J. M. Building Verifiable Trusted Path on Commodity x86 Computers. In: *IEEE Symposium on Security and Privacy*. 2012.

[246] Zhou, Z., Yu, M., and Gligor, V. D. Dancing with Giants: Wimpy Kernels for On-Demand Isolated I/O. In: *IEEE Symposium on Security and Privacy*. 2014.

[247] Zhuo, D., Zhang, K., Zhu, Y., Liu, H. H., Rockett, M., Krishnamurthy, A., and Anderson, T. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In: *16th USENIX Symposium on Networked Systems Design and Implementation*. 2019.

[248] Zou, Y.-H., Bai, J.-J., Zhou, J., Tan, J., Qin, C., and Hu, S.-M. TCP-Fuzz: Detecting Memory and Semantic Bugs in TCP Stacks with Fuzzing. In: *USENIX Annual Technical Conference*. 2021.