DISS. ETH No. 25674

Security with Intel SGX: Enhancements, Applications and Privacy

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

SINIŠA MATETIĆ

MSc in Security and Mobile Computing,
Aalto University, Finland and NTNU, Norway
MSc in Information and Communication Technology,
University of Zagreb, Croatia

born on 22.06.1990

citizen of Croatia

accepted on the recommendation of

Prof. Dr. Srdjan Čapkun, examiner
Prof. Dr. David Basin, co-examiner
Prof. Dr. Andrew Miller, co-examiner
Dr. Marko Vukolić, co-examiner

2018

Neka ti uvijek bude na umu da samo
stvaralački rad stvara fizionomiju ličnosti.
— A. B.

dedicated to *baka Dragica...*
the single person with the greatest impact in my life...

Abstract

With the ever growing development of Internet, security has become a crucial topic for both the application developers and end users. A popular approach in achieving security is through isolated code execution by using virtualization techniques. Virtualization helps in partitioning the resources of the underlying platform to multiple software components, ensuring that these components are unaware of each other. However, existing virtualization approaches suffer from intrinsic limitations, such as large trusted computing bases, making them vulnerable to a plethora of attacks.

To overcome some of these limitations, the concept has been adapted resulting in the development of hardware-assisted isolated execution environments. This brought the idea of Trusted Execution Environments (TEEs). A TEE represents a secure and integrity-protected environment that provides processing, memory and storage capabilities. Numerous TEEs have emerged over time, and a recent one, called Intel Software Guard Extensions (SGX), has caught increasing attention. Intel SGX is an extension to the x86 instruction set and provides a minimal trusted computing base including only the CPU package. SGX allows creation of protected containers that run security-critical and sensitive code in isolation from the rest of the untrusted residing platform.

In this thesis, we focus on the security with Intel SGX and make the following contributions. First, we address one of the major drawbacks of SGX, susceptibility to state rollback, by proposing a new distributed system. Second, we explore how to extend the usability and security of existing services by presenting a new concept, called brokered delegation. Third, we show how SGX could enhance privacy in one of the most popular technology areas today, blockchains, by solving another major shortcoming of SGX for this specific use-case, susceptibility to side-channel attacks. We show through the example of Intel SGX that such TEEs could be used in various new application scenarios, increase performance over existing solutions while maintaining security, and make security more easily available.

Zusammenfassung

Mit der wachsenden Digitalisierung ist Informationssicherheit zu einem wichtigen Thema für Applikationsentwickler und Endnutzer geworden. Isolation von Programmen durch Virtualisierungstechniken ist ein beliebter Ansatz um Sicherheit zu gewährleisten. Virtualisierung hilft dabei, die Resourcen der zugrundeliegenden Plattform auf mehrere Softwarekomponenten aufzuteilen und zu garantieren, dass diese Komponenten voneinander isoliert sind. Die Sicherheit bestehender Virtualisierungslösungen ist schwer zu garantieren da geläufige Implementationen eine grosse, so genannte Trusted Computing Base haben und dadurch eine grosse Angriffsfläche präsentieren.

Um einen Teil dieser Probleme zu beheben, wurden konzeptionell andere hardwareunterstützte isolierte Laufzeitumgebungen entwickelt, auch Trusted Execution Environments (TEEs) genannt. TEEs sind sichere Umgebungen, die Datenverarbeitung, Arbeitsspeicher und Langzeitspeicher zur Verfügung stellen und deren Integrität geschützt ist. Im Laufe der Zeit wurden mehrere TEEs entwickelt, allerdings hat vorallem eine kürzlich entwickelte Technologie mit dem Namen Intel Software Guard Extensions (SGX) grösseres Interesse geweckt. Intel SGX erweitert das x86 Instruction Set und stellt eine minimale Trusted Computing Base zur Verfügung, die nur CPU und minimale Software umschliesst. SGX erlaubt das Erstellen von geschützten Containern, in denen sicherheitsrelevanter Code isoliert vom Rest der nicht vertrauenswürdigen Plattform ausgeführt wird.,

In dieser Doktorarbeit konzentrieren wir uns auf Informationssicherheit mit Intel SGX und machen die folgenden Forschungsbeiträge. Erstens addressieren wir eines der grössten Mankos von SGX, nämlich die Anfälligkeit auf Rollback-Attacken, indem wir ein neues verteiltes System vorschlagen. Zweitens untersuchen wir, wie die Benutzerfreundlichkeit und Sicherheit von existierenden Dienstleistungen erweitert werden können und stellen dafür ein neues Konzept, genannt Brokered Delegation, vor. Drittens zeigen wir, wie SGX genutzt werden kann, um den Datenschutz in einer der beliebte-

sten Technologien der heutigen Zeit, nämlich Blockchains, zu verbessern, indem wir die Anfälligkeit von SGX gegenüber Side-Channel Attacken für diesen spezifischen Anwendungsfall lösen. Wir zeigen am Beispiel von SGX, dass Trusted Execution eine vielversprechende Technologie ist, die in vielen Anwendungsfällen benutzt werden kann, die Effizienz im Vergleich zu kryptographischen Lösungen unter Beibehaltung der Sicherheit verbessert und Sicherheit praktikabler macht.

Acknowledgments

It is done. I thought that writing this last piece of this dissertation would be the easiest thing at the end of the journey. However, it comes hard. Flashbacks are swarming and as much as waiting for this moment was eagerly anticipated, it is difficult to let go and reconcile that the journey has come to an end. During the course of my Ph.D., I have for certain grown professionally, but also personally. This opportunity has given me a chance to learn numerous life lessons and gain skills that will follow me throughout my life in the future. I would like to thank all the people who were involved in my journey, offered understanding, support, encouragement and above all, long lasting friendships.

I have said so many times that I would have never finished my Ph.D. without my advisor, Prof. Dr. Srdjan Čapkun. And I truly believe in that. Srdjan offered his unprecedented support over the years and gave trust that allowed me to grow. His positive energy, words of encouragement and the attitude towards mentoring, teaching and helping people grow is unique. I still do not know why he took me as his student, but there are no words which I can use to explain my gratitude for having him as my advisor, and further on as my friend. Numerous discussions and countless chats have had a major impact on aligning my life views and goals. I can say without hesitation that he has had the biggest influence in development and change of my adult life, both professionally and personally.

I would like to thank Prof. Dr. David Basin, Prof. Dr. Andrew Miller, and Dr. Marko Vukolić for accepting the invitation to be the members of my thesis committee and for taking time to review my dissertation.

A would like to express special gratitude towards Dr. Kari Kostiainen. I thank him for the infinite amount of patience that he had with me, guiding me from my very first project to the last one. Let me just say, it was not easy for him, for sure. Kari offered me invaluable support and contributed greatly to my development in the academic space and it will never be forgotten. Thank you!

During my stay at ETH and in the System Security Group I had a great honor to meet and become friends with some awesome people. I thank Dr. Luka Mališa for all the conversations over numerous coffees, for his support and friendly guidance in difficult times. A special thank you goes to Moritz Schneider, who started as my Master student and ended up being a part of our group. Our collaboration and his great, irreplaceable work have had a high impact on the outcomes of this thesis which I will always have in my mind. I thank Karl Wüst on the awesome collaborations that we have and Aritra Dhar for being a good friend and one of the few people that could strip my nervousness instantly. I thank my colleagues Prof. Dr. Aanjhan Ranganathan, Ivan Puddu, David Sommer, Mridula Singh, Patrick Lei, Daniele Lain, Mansoor Ahmed, Dr. Claudio Marforio, Dr. Nikolaos Karapanos, Hildur Olafsdottir, Dr. Hubert Ritzdorf, Stephanos Matsumoto and Thilo Weghorn for making my journey truly memorable. I also thank Mrs. Barbara Pfändner for all the support and help with administrative matters during the years.

I express my gratitude to external collaborators Prof. Dr. Ari Juels, Prof. Dr. Andrew Miller, Dr. Ghassan Karame, and Dr. Ian Miers for their cooperation and support.

Last but not least, I would like to extend gratitude to my family and friends for their immense support. Special thanks to my girlfriend Ana Marija Lancic for all the unconditional love, support and understanding, for standing alongside me during both good and bad times, and for tolerating my craziness that basically boils down to an everyday basis. You are my shining star. I thank Prof. Dr. Slavomir Stankov for having a great influence on the choice of my studies that eventually affected my complete professional path.

Finally, I would like to dedicate this doctoral dissertation to my grandma, Dragica Bulić. She has been the person with the single greatest impact on my life. If it weren't for her, I would have never been the person that I am today. Let all of this work be in your honor, as you have certainly deserved it.

In memoriam of my grandpa Prof. Dr. Ante Bulić, whose honorable work and harsh life path have always fueled me to strive for the best. You would have been proud.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last three decades we have witnessed an ever growing advent of digital technologies. The fast-paced development and adoption of various devices made them available to the global population, making today's life almost unimaginable without these *smart* devices and digital services.

Security has always been an important topic, however due to the ever growing adoption of complete digitalization, it has become crucial. Device manufacturers, application developers and protocol designers are under tremendous pressure since achieving security while maintaining high-usability is a challenge [44]. Protocols and systems are broken on an everyday basis, and hackers are constantly lurking in attempt to steal data and gain monetary advantages. Due to the natural shift to the digital world of businesses and operations in almost all service sectors, except the ones tied to *brick and mortar* solutions, security has also become one of the primary concerns for users as well. However, continuous reports show that regardless of the fact that users are aware and conscious about the possible threats, they tend to ignore security, often preferring new functionality and cheap price instead [15, 61, 111].

In the last decade, information privacy has also emerged as one of the most important topics. Today, the competition between device manufacturers and service providers in offering new products and services is astonishingly extensive, and the lack of security and privacy guarantees often serves as hindrance in adopting new technologies. Providing stronger security and privacy guarantees should be a must and serve more as a commodity nowadays, rather than a special, extra feature, offered to the end users.

To achieve security, one always goes back to the basic principles of information security: Confidentiality, Integrity and Availability. Historically and

nowadays, the most fundamental approach in achieving this security is through cryptography. Modern cryptography is based on mathematical theory and almost all other approaches to security rely or use some sort of cryptography primitives, to guarantee the basic principles of security. Another common approach in achieving security is isolated code execution. A well known and prevalent way to create isolated code execution is through virtualization technology [207]. Virtualization allows creating an illusion to system software that it is the only one operating on a specific hardware platform, while there might exist multiple software components sharing the underlying resource [30]. However, existing virtualization based approaches suffer from some intrinsic limitation, such as dependence on hypervisors that usually have a large Trusted Computing Base (TCB) reaching more than 500k lines of code (LoC) [92], the inability to tackle firmware rootkits (limiting security) and large performance overhead (limiting usability).

To overcome the challenges of software isolated code execution, researchers have proposed hardware-assisted isolated execution environments. The approach itself is simple. Use tamper-resistant hardware that provides more security with isolated code execution to achieve crucial protection of computer systems. This concept brings us to the idea of Trusted Execution Environments (TEEs). A TEE represents a secure and integrity-protected environment that provides processing, memory and storage capabilities [62]. It is isolated from the rest of the general processing environment. The general processing environment includes the hypervisor, the operating system and running applications. These intrinsic characteristics of TEEs make their adoption appealing for different use-cases. TEEs have been available in numerous different device types for more than a decade, however their usage has not been exploited enough.

Different types of hardware-assisted trusted execution environments have emerged over time. System Management Mode (SMM) provides a hardware-isolated environment that implements platform-specific control functions by directly triggering CPU interrupts [59]. In x86-based architectures, the Trusted Computing Group (TCG) [13] introduced Dynamic Root of Trust for Measurement (DRTM). Coupled with a Trusted Platform Module (TPM) [190], a tamper-resistant secure co-processor with a small footprint, a processor is able to invoke small pieces of trusted software isolated from the OS to verify integrity [62]. Moreover, Intel developed Trusted eXecution Technology (TXT) [108] to implement DRTM that allows a trusted way of loading and executing system software. In almost all recent Intel processors we can find the Intel Management Engine (ME) [90], an autonomous micro-computer embedded into the processor chipset, that allows executing security-critical applica-

tions, such as privacy identification, identity protection, and boot guard [164]. AMD's version of Intel ME is the Platform Security Processor (PSP) [207] that enables secure boot. It can also work with the ARM TrustZone technology to enable running external trusted applications in an isolated environment. ARM TrustZone [18] is a hardware-based architecture that allow secure execution by introducing two worlds or environments on a system. Namely, the regular processing and execution (often called, Rich Execution Environment (REE)) is done in the *normal* world, while the trusted execution environment for secure processing is done in the *secure* world. The complete isolation between the worlds is done through TrustZone's security extensions for system hardware including peripherals, memory and the CPU.

One of the recent hardware-based TEEs was introduced by Intel and is called Intel Software Guard Extensions (SGX) [55]. Intel SGX is an extension to the x86 instruction set architecture. Alike other mentioned TEEs, Intel SGX has a minimal trusted computing base that only includes the processor package and the applications that can run in the isolated environments provided by SGX [19, 89, 146]. The instruction set allows users to instantiate protected containers, called enclaves. The architecture is designed in an easy-to-use way, allowing potential users to run arbitrary code inside the enclaves while still enjoying the promised protections, just by using the provided SDK (and, of course, following secure coding practices). Namely, Intel SGX offers isolated enclave code execution, remote attestation of the enclaves (alike its predecessors, TXT [72] coupled with a TPM [190]), protected and encrypted memory and long-term locally sealed storage. Even more, SGX provides confidentiality and integrity regardless of the trust model in the BIOS, firmware, hypervisors or operating systems. Several different studies support the claims of the SGX potential to not only support traditional TEE applications [32, 132], but also enhance security and privacy of existing and new, other applications.

However, like other TEEs, Intel SGX does not offer perfect security and privacy. The design itself is susceptible to side-channel leakage, potential physical attacks, and it also suffers from some shortcomings, like rollback.

The focus of this thesis is on the security with Intel SGX solving three explicit problems. First, we address one of the major drawbacks of SGX, susceptibility to state rollback attacks, that could hinder its wide adoption and propose enhancements in form of a distributed system. Second, we explore the possible application space that could extend the usability and security of currently operated platforms and services by presenting a new concept called *brokered delegation*. Third and last, we provide proof on direct privacy implications in one of the most popular technology areas today, blockchains. There we show how SGX can be used to enhance privacy, and in order to

deploy it we solve another major shortcoming of SGX, susceptibility to side-channel attacks, for this specific use-case. We argue and show, through the example of Intel SGX, that such TEEs could be used in various application scenarios, increase performance over existing solutions while maintaining security, and make security more easily available and attractive. SGX has the potential to change on how we think about applied system security, and, in long-term, change how we design, create and develop systems and services in the future, however, a lot of research is still pending to make that possible.

1.1 Contributions

Below, we summarize and describe the contributions made in this dissertation towards showing the potential and applicability of Intel SGX for improving security through isolated execution in modern computing platforms running security critical services.

Rollback Protection for Trusted Execution Environments: Security architectures such as Intel SGX suffer from lacking the capabilities to verify freshness of long-term stored and encrypted states. Namely, across reboots and restarts the TEEs lose the notion of their previous state. While they are able to verify the authenticity and data integrity of loaded states by successful decryption (serving as data authentication), they need protection against rollback attacks. In a rollback attack the adversary violates the freshness integrity of a protected application state by replaying old persistently stored data or by starting multiple application instances. Successful rollback attacks have serious consequences on applications such as financial services. For example, an attacker with a an account stored in the state can first spend money, then force restart the environment before a successful state update, and subsequently load the old state back, allowing double-spending.

We propose a new approach for rollback protection on SGX. The intuition behind our approach is simple. A single platform cannot efficiently prevent rollback, but in many practical scenarios, multiple processors can be enrolled to assist each other. We design and implement a rollback protection system called ROTE that realizes integrity protection as a distributed system. Each participating node keeps the latest state version of other members in the protection group (where multiple processors agreed to mutual assistance). A restarted enclave is able to reload and apply its state only if the protection group can prove to the enclave itself that the loaded version is actually the latest updated one.

We construct a model that captures strong adversarial ability to schedule enclave execution and show that our solution achieves a strong security property called *all-or-nothing* rollback: the only way to violate integrity is to reset all participating platforms to their initial state, and during the system's operation no rollback can ever occur (the enclave will never reload an older state version, while it might be reset to zero, which we do not consider to be rollback *per se*). We implement ROTE and demonstrate that distributed rollback protection can provide significantly better performance than previously known solutions based on local non-volatile memory.

Brokered Delegation using Trusted Execution Environments We introduce a new concept called *brokered delegation*. Brokered delegation allows users to safely, flexibly and selectively delegate credentials and rights for a range of service providers to other users and third parties. We explore how brokered delegation can be implemented using trusted execution environments, namely Intel SGX. We introduce a system called DELEGATEE that enables users (Delegatees) to log into different online services using the credentials of other users (Owners). Credentials in DELEGATEE are never revealed to Delegatees and Owners can restrict access to their accounts using a range of rich, contextually dependent delegation policies. We explore DELEGATEE design space using two system architectures: a purely decentralized P2P system, and what we call a Centrally Brokered system, in which a third party runs the secure enclaves. Our system also allows Owners and Delegatees to interact in two distinct models, an identity-based and pseudonymous model.

DELEGATEE fundamentally shifts existing, password-based, access control models for centralized online services. It does so by using Intel SGX's isolated execution and protection mechanism to permit access delegation *at the user's discretion*. DELEGATEE thus effectively reduces mandatory access control (MAC) in this context to discretionary access control (DAC). The system demonstrates the significant potential for TEEs to create new forms of resource sharing around online services without the direct service support.

We present a full implementation of DELEGATEE using Intel SGX and demonstrate its use in four real-world applications: email access (SMTP/IMAP), restricted website access using a HTTPS proxy, e-banking/credit card, and a third-party payment system (PayPal). Our analysis shows negligible overhead imposed by the system, thus opening the space for further adoption of the concept into many other services and systems. We finalize our findings by providing a discussion regarding open problems, potential deployment difficulties and controversy related to the concept itself, as in many cases our system breaks service's terms of use and can also be used to fuel illegal activities.

Privacy Preservation for Lightweight Cryptocurrency Clients Decentralized blockchains in the form of cryptocurrencies offer attractive advantages over traditional payments such as the ability to operate without a trusted authority and increased user privacy. Cryptocurrencies record transactions between parties in a blockchain maintained by a peer-to-peer network. The verification of blockchain payments requires the user to download and process the entire chain which can be infeasible for resource-constrained devices, such as mobile phones. To address such concerns, most major blockchain systems support lightweight clients that outsource most of the computational and storage burden to full blockchain nodes. However, such payment verification methods leak considerable information about the underlying clients, thus defeating user privacy that is considered one of the main goals of decentralized cryptocurrencies. On the other hand, in most cryptocurrencies and regardless of the light clients, transactions explicitly identify the previous transaction providing the funds they are spending, revealing the amount and sender/recipient pseudonyms. This itself is a considerable privacy issue.

The explicit contribution is twofold: First, we propose a new approach to protect the privacy of existing lightweight clients in blockchain systems like Bitcoin. Our main idea is to leverage commonly available trusted execution capabilities, such as SGX enclaves. We design and implement a system called BITE where enclaves on full nodes serve privacy-preserving requests from lightweight clients. As we will show, naive serving of client requests from within SGX enclaves alone is not sufficient to protect client's privacy, as enclave execution can leak information in numerous ways (specifically through various side-channel attacks). BITE therefore integrates several privacy preservation and protective measures from the fields of private information retrieval and side-channel protection to our solution, namely Oblivious RAM and hiding of the process's execution flow and branching.

Second, we investigate Zcash, a cryptocurrency that effectively resolves the privacy concern of most other decentralized currencies. In particular, Zcash offers privacy by using zero-knowledge proofs to hide both the source, destination and amount of the transacted funds. To receive payments in Zcash, however, the recipient must fully scan the blockchain, testing if each transaction is destined for them. Naturally, as with Bitcoin, this is not practical for mobile and other bandwidth constrained devices. Due to outlined specificity no lightweight clients even exist for Zcash. Based on the approach created for Bitcoin lightweight clients, we carefully modify the design for the Zcash specification that enables private retrieval of transactional data. Additionally, we design the necessary bandwidth efficient mechanism for the client to keep an up-to-date version of the witness needed in order to spend the funds they

previously received. Thus, we are the first to present a viable solution for the emergence of light client for Zcash.

We show that the resulting solutions provide strong privacy protection and at the same time allow high performance.

1.2 Thesis Organization

This doctoral dissertation is organized as follows:

We begin the thesis with an overview of existing trusted execution environments, comparing their specifications and characteristics in Chapter 2.

Chapter 3 describes the design and implementation of the ROTE system that provides rollback protection for trusted execution environments, specifically Intel SGX. First, we give the background and motivate the problem, followed by the protocol design and security analysis. We further present the implementation and evaluate the performance. Chapter ends with the discussion of open questions and the comparison with related work.

In Chapter 4, we present a novel concept of brokered delegation. First, we start by motivating the problem and evaluating potential solutions. Second, we present the DELEGATEE system along with the general protocol design. We further provide the security analysis where different implications of using TEEs are discussed. Lastly, we show our concept in form of four implemented use-case scenarios along with evaluation and usability discussion.

In Chapter 5, we motivate the example of using Intel SGX for privacy protection in lightweight clients of two cryptocurrencies, Bitcoin and Zcash. We further outline that naive application of SGX does not suffice in order to provide privacy due to the susceptibility to side-channel attacks. Thus, we continue by carefully crafting a solution using components from various different research areas in order to build a side-channel resilient privacy solution for the named light clients. This chapter also presents the protocol, implementation, security analysis and evaluation details of our two systems.

We conclude the thesis in Chapter 6 with a summary of our findings, possible future work directions and final remarks.

After the closing remarks, the readers can find Appendices A, B, and C related to Chapters 3, 4, and 5, respectively. The appendices include additional details, analysis, prototype demos and proofs that can serve as additional information relevant to the original chapters.

1.3 Publications

The chapters in this thesis correspond to the above outlined contributions from the following publications:

- Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, Srdjan Capkun, **ROTE: Rollback Protection for Trusted Execution**, *In Proceeding of the 26th USENIX Security Symposium, (USENIX Security)*, 2017.

- Sinisa Matetic, Moritz Schneider, Andrew Miller, Ari Juels, Srdjan Capkun, **DelegaTEE: Brokered Delegation using Trusted Execution Environments**, *In Proceeding of the 27th USENIX Security Symposium, (USENIX Security)*, 2018.

- Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiainen, Ghassan Karame, Srdjan Capkun, **BITE: Bitcoin Lightweight Client Privacy using Trusted Execution**, *IACR Cryptology ePrint Archive, Report 2018/803*, 2018. *(in conference submission)*

- Karl Wüst[1], Sinisa Matetic[1], Moritz Schneider, Ian Miers, Kari Kostiainen, Srdjan Capkun, **ZLiTE: Lightweight Client for Shielded Transactions using Trusted Execution**, *In International Conference on Financial Cryptography and Data Security*, 2018.

[1] Equally contributing authors.

Chapter 2

Background

2.1 Introduction

With the ever growing global digitalization we are experiencing trends of designing complex systems on an everyday basis. With digitalization and the complexity that follows it, new challenges arise making the task of satisfying security requirements increasingly demanding. Traditional security technologies get outdated very fast and can not always meet newly required security guarantees [165]. Trusted computing as a general concept was introduced in order to help systems in achieving secure computation and data protection. The original idea of trusted computing is used to describe a set of technologies that allow the creation of trust in local or remote platforms by using trustworthy components, usually as separate hardware modules on the platform, which perform security-critical tasks and provide good, functional interfaces to achieve security [26].

One of the common approaches in protecting sensitive code and applications is isolated execution. The approach considers the design of systems based on well-known operating systems and hardware components. One of the most representative examples is virtualization that isolates execution of a particular piece of software (albeit an application or a complete operating system) from other software running on the same platform. Virtualization techniques, however, usually involve applications that have hundreds of thousand lines of code [27], providing weak assurances that the code itself is not vulnerable. These techniques also come with a lot of trust assumption that cannot satisfy security against strong adversaries that have platform access.

To overcome the shortcomings, and provide systems that guarantee higher levels of isolation with more secure execution of applications and protection of data, hardware-based trusted execution environments were proposed. The main idea is to actively reduce the trusted computing base of such security-enabling systems (both in terms of software and necessary hardware), and at the same time provide strong security guarantees on executing sensitive tasks.

In this chapter, we provide a brief overview of what trusted execution environments are. Further on, we present the background on Intel SGX by explaining the architecture, mechanisms and offered protections needed to understand the work done in this thesis. This is followed by an overview of the state-of-the-art similar and related hardware-supported TEEs that have a relation to SGX. Finally, we discuss the differences between the presented technologies and provide a summary table. The chapters that follow after this one target specific research areas in conjunction with SGX and their backgrounds are conceptually positioned during problem introduction in the chapters themselves. Combining these into this background chapter would negatively affect the reading flow.

2.2 Trusted Execution Environments

Academic research on hardware-based trusted computing dates back to the 70s [26, 198], although the exact terms identifying them have been changing. Even today, the term TEE is used extensively by device and chip manufacturers to advertise enhanced security capabilities of these systems. No common, and widely adopted, standardized framework exists to describe the exact nature of TEEs. For example, in the literature, one can find many different definitions on what a Trusted Execution Environment [165]. We cite some of these below:

- "*TEE is a dedicated closed virtual machine that is isolated from the rest of the platform. Through hardware memory protection and cryptographic protection of storage, its contents are protected from observation and tampering by unauthorized parties.*" [66]
- "*The TEE is an execution environment that runs alongside but isolated from the device main operating system. It protects its assets against general software attacks. It can be implemented using multiple technologies, and its level of security varies accordingly.*" [160]
- "*The set of features intended to enable trusted execution are the following: isolated execution, secure storage, remote attestation, secure provisioning and trusted path.*" [194]

We do not have the intention of providing a new TEE definition that captures all of its essential properties, since these change based on the context in which the TEE is used. In general, we envision a TEE based on a simple definition given by [62]: "*A trusted execution environment (TEE) is a secure, integrity-protected environment, consisting of processing, memory, and storage capabilities.*" Thus, TEEs are isolated from the general processing environment, often called Rich Execution Environment (REE), that hosts an untrusted operating system and all other applications. These normal processing environments are usually *richer* in terms of extended functionalities, but, however, also suffer from an increased vulnerability surface. When combined with TEEs, applications from the REE can be provided with improved security, enabling execution of sensitive operations and storage of sensitive data away from a potential adversary.

Recent research has also focused on investigating on alternative mechanisms to enable trusted computing, developing new trust anchors in form of physically unclonable functions (PUFs) [141, 185], but also enabling the existence of TEEs on more resource-constrained devices [63,183]. The interest in trusted computing and TEEs from both standardization bodies and industry has also sparked numerous implementation proposals. Trusted Computing Group (TCG) [13] is putting considerable efforts in the standardization of the trusted computing space, while Global Platform (GP) [160] has been aiming at creating a TEE specification intended for mobile devices. Despite the increased focus on TEEs and their deployment in many application areas, the application developers still suffer from the lack of widely available means for exploiting TEE functionalities, since they only remain as a part of research or proprietary work [26]. However, with the emergence of new standardization efforts, we can expect to see standardized interfaces for accessing and using TEEs, added across different platforms, in the near future. Wide availability of TEEs would fuel the spark resulting in novel ideas and new architectures.

In Chapter 1 we outlined the existence of different hardware-assisted trusted execution environments that have emerged over time, such as System Management Mode (SMM), Dynamic Root of Trust Measurement (DRTM), Trusted Platform Modules (TPMs), Intel's Trusted eXecution Technology (TXT), Intel Management Engine (ME), AMD Platform Security Processor (PSP), ARM TrustZone, etc. We investigate the different properties of some of these systems and architectures along with the TEE that is in the core focus of this thesis, Intel SGX, further on in this chapter.

2.3 Intel SGX Background

Intel Software Guard Extensions (Intel SGX) [100] is the latest Intel's iteration in the design of trusted computing solutions that has the main goal of solving secure remote computation problem by using trusted hardware, that is, the problem of performing execution on an untrusted remote device while keeping confidentiality and integrity guarantees. Additionally, SGX aims to address application security by allowing security-critical software to run in isolation from the rest of the system, thus effectively preventing attacks from the code running in higher privilege rings of the platform.

The traditional model of processor's ring model includes four privilege levels [117]. Ring 0, as the highest privileged space, is usually reserved for the kernel of the operating system, Ring 1 supports a part of the device drivers, Ring 2 serves the rest of device drivers and potentially a guest operating system, while the most interesting for us is the Ring 3, the application space where all other unprivileged code resides. The protection model works well in terms of protecting higher privilege levels from malicious code running on lower level privileges. However, this model does not allow any protection of the application space if, for example, the operating system is malicious. Thus, the problem of secure remote computation remains unsolved.

In a nutshell, Intel introduced SGX in the 6th generation of its CPUs as an instruction set extensions to the original Intel processor architecture and as a set of new protective mechanisms that aim to resolve the above mentioned problems. The trusted hardware creates a secure container, called *a secure enclave*, that executes in an isolated environment. This isolation protects the integrity and confidentiality of the container's security-critical computation from any software running on the residing platform where the software stack, including BIOS [118], OS and hypervisor [48], other software application [200], or even malicious peripherals [83] such as compromised network cards, are potentially malicious. With the security guarantees offered by SGX, the attacks surface and the TCB are effectively reduced to the CPU package and the application running inside the secure enclave. Additionally, SGX support performing remote software attestation, which helps to prove to the user or the remote verifier that a specific piece of software is running inside this secure container (application initialized correctly), that the offered security properties are guaranteed by trusted hardware (the platform supports Intel SGX) which hosts the container, and that the application has not been tampered with (the code measurement of the enclaved application code). In order to guarantee confidentiality and integrity the processor is also equipped with the new processor-specific cryptographic keys that are fused-in to the

processor die. This key is used in the key derivation process which is invoked for numerous enclave operations that encompass cryptographic operations.

SGX is also designed to impose minimal performance overhead for computations done inside the secure environment. In Section 2.3.7 we provide the performance analysis and comparisons for executing several different computations in order to back-up these claims.

In the following, we explain the main protective mechanisms of Intel SGX that are relevant for understanding the thesis and its contributions. The background is compiled based on the exhaustive Intel SGX analysis [55], the original papers that introduced the scheme [19, 89, 146], Intel's SGX website [107], manuals encompassing the development guide [101], SGX tutorials [97, 106], Intel's blog posts [93–95, 102], SGX Software Development Kit (SDK) [105], and research papers looking more closely to specific mechanism of SGX, such as the environment's lifecycle [1], sealing of enclave's data [17] and remote software attestation [112]. A complete, in-depth, elaboration of the architecture, covering as well all the necessary background from computer architecture literature and primitives used in SGX development are available in [55]. We assume reader familiarity with some basics of the x86 architecture.

2.3.1 Intel SGX Instruction Set Extension

In order to accommodate changes that allow for the creation and management of secure enclaves, Intel introduced a set of instruction extensions that we present and briefly describe in Table 2.1. The extension is composed of 18 new instructions and during the course of this section we will use some of the terms to better explain the logic and flow of SGX operation.

2.3.2 Runtime Isolation

The SGX security architecture guarantees that enclaves are *isolated* from all software running outside of the enclave, including the OS, other enclaves, and peripherals. By isolation we mean that the control-flow integrity of the enclave is preserved and other software cannot observe its state. The isolation is achieved via protection mechanisms that are enforced by the processor. The content of the enclaves, including code, data and associated data structures is stored in a protected memory area called Enclave Page Cache (EPC). SGX is intrinsically a multi-process environment, indicating that multiple secure enclaves can coexist at runtime. In order to accommodate this, the SGX EPC is split into equal-size pages where each page is assigned to a specific enclave. EPC resides in Processor Reserved Memory (PRM), which is a continuous range of memory forming a subset of DRAM that cannot be accessed by the

Instructions	Description
(Supervisor instr.)	
EADD	Add a page
EBLOCK	Block an EPC page when eviction is going to occur
ECREATE	Create an enclave by converting a free EPC page to SECS
EDBGRD	Read data by the debugger enclave
EDBGWR	Write data by the debugger enclave
EEXTEND	Extend (update) the EPC page (specifying an enclave) measurement
EINIT	Initialize an enclave after it has been built
ELDB	Load an evicted EPC page into the memory as blocked
ELDU	Load an evicted EPC page from the memory as unblocked
EPA	Add a version array
EREMOVE	Remove a page from the EPC
ETRACK	Activate EBLOCK (check TLB flushing in logical processors)
EWB	Write back/invalidate a protected EPC page by evicting it
AEX	Asynchronous enclave exit (invoked with exception or interrupts)
(User inst.)	
EENTER	Enter an enclave
EEXIT	Exit an enclave
EGETKEY	Create a cryptographic key (unique symmetric key)
EREPORT	Create a cryptographic report (enclave report used for attestation)
ERESUME	Re-enter the enclave and resume

Table 2.1: *Intel SGX's instruction set extensions.*

OS, applications or direct memory accesses. The PRM protection is based on a series of memory access checks in the processor. Non-enclave software is only allowed to access memory regions outside the PRM range, while enclave code can access both the non-PRM memory and the EPC pages owned by the enclave.

The EPC is managed by the underlying software managing the platform, e.g., a hypervisor or OS, which also includes complete management of the platform's physical memory. This means that the system software actually assigns unused EPC pages to enclaves, and at the same time frees previously allocated pages by evicting them into the untrusted DRAM so they can be loaded back at a later stage. While the evicted EPC pages are stored in the untrusted memory, SGX assures their confidentiality, integrity and freshness via cryptographic protections. The architecture includes the Memory Encryption Engine (MEE) which is a part of the processor uncore (microprocessor function close to but not integrated into the core) and sits next to the processor's memory controller. The MEE encrypts and authenticates (HMACs) the enclave data that is evicted to the non-protected memory, and ensures enclave data freshness at runtime using counters and a Merkle-tree structure (based on the design of Aegis secure processor explained in the next section). The root of the tree structure is stored on the processor die. Additionally, the MEE is connected to the Memory

Controller and is used to protect SGX's Enclave Page Cache against physical attacks, e.g., bus taping, by performing encryption of the memory contents leaving the processor package.

Lastly, the runtime isolation is supported by another minor hardware modification, namely, the Page Miss Handler (PMH). The PMH resolves Translation Lookaside Buffer (TLB) misses using a fast path that relies on a page walker and only resorts to microcode as a fallback for handling edge cases. To implement the access control enforcement that SGX requires, PMH is set to trigger the microcode assist for all address translations when a logical processor operates with enclaves.

2.3.3 Enclaves - Creation and Lifecycle

The SGX presents a new model for creating applications that want to use SGX features. Each application consists of the untrusted and trusted part. It is up to the developers to carefully design their applications, such that all the security-critical and sensitive operations are contained in the trusted part. The trusted part is actually represented by the secure enclave. The untrusted part of the applications serves as a running process, while the enclave itself cannot independently run. As mentioned, the enclave implements sensitive computation in form of function calls initiated from the untrusted part. The interface between these application parts is implemented using `ocalls` and `ecalls`, calls from the trusted to the untrusted part, and vice-versa, respectively. During an `ocall`/`ecall` all arguments are copied to trusted/untrusted memory and then executed in order to maintain a clear partition of trusted and untrusted parts. These interfaces are defined and implemented by Intel in the Intel SGX SDK. Additionally, enclaves cannot execute system calls and do not have access to secure peripherals. Only the unprotected application component that runs in normal user space has the access, and is thus handling communication with the OS, i.e., operations concerning networking and file accesses. This does not mean that the enclave cannot have a secure communication channel with an outside entity. Either a proprietary encrypted channel can be created directly from the enclave, or the enclave can be used to implement a TLS endpoint (as is the case in our work presented in subsequent chapters).

An enclave is created by the system software. During enclave creation, the system software specifies the enclave code. Security mechanisms in the processor create a data structure called SGX Enclave Control Structure (SECS) that is stored in a protected memory area and represents the metadata for the created enclave. Because enclaves are created by the system software running on the OS, their code cannot contain sensitive data. In general, the enclave file residing on the local storage can be disassembled indicating that enclave code

is not a secret. Also, there are no secrets inside the enclave before the enclave is created, and all sensitive information must be delivered from another source afterwards. The start of the enclave is recorded by the processor, reflecting the content of the enclave code as well as the loading procedure (sequence of instructions). The recording of an enclave start is called *measurement* (see below) and it can be used for later attestation (see below). Once an enclave is no longer needed, the OS can terminate it and thus erase its memory structure from the protected memory.

We briefly describe the typical lifecycle of an enclave along with the specified instructions needed to manage the enclave cycle:

(**1**) **Neutral:** The normal state of the system platform where no enclaves currently exists. This state can be envisioned as the state before any SGX instruction has been invoked, or the state that is formed after the enclave is terminated, done by invoking the EREMOTE instruction that removes a page from EPC corresponding to an enclave.

(**2**) **Creation:** The creation of a new enclave starts by invoking the ECREATE instruction which first takes a free page in the EPC and turns it into the SECS. The new SECS is initialized with the required fields and ECREATE also performs the validation of that information. Creation of the enclave does not entail its running state, and no execution can be done until the created enclave is initialized using the EINIT and EENTER instructions.

(**3**) **Loading:** After the SECS is created, the system software can invoke the EADD instruction to load the initial code and data to the enclave. EADD also validates its inputs before adding a page to the EPC page. This is important since invoking EADD for an enclave that is already initialized (e.g., in an attempt to change the code) and running will result in an exception. The loading also includes calling of the EEXTEND instruction that updates enclave's measurements used subsequently (sealing and attestation).

(**4**) **Initialization:** Enclave initialization is done by the system software invoking the *Launch Enclave (LE)* used to obtain the EINIT Token Structure. The LE is a special, privileged enclave that is set-up, provided and cryptographically signed with a special hardcoded SGX key by Intel. LE has to be used to initialize usage of enclaves not created by Intel itself on a particular system. The given token is provided to the EINIT instruction that marks the SECS of the enclave as initialized. After EINIT completes successfully, the enclave's INIT attribute is set to true, opening the possibility for enclave execution.

(**5**) **Running:** In order to start the enclave execution the system software needs to invoke the EENTER instruction, which effectively switches the

context to the secure enclave. A return out of the enclave can be initiated by using the EEXIT instruction. A re-entry to the enclave on which a return was performed can be done using the ERESUME instruction.

(**6**) **Termination:** Once the enclave is not needed anymore, e.g., it has performed all the requested computations, the system software executes the EREMOVE instruction which de-allocates the EPC pages used by the enclave. EREMOVE also marks the emptied EPC page as available. Note that the page itself is not re-written or deleted, however the information is unusable since the ECREATE and EADD instruction will zero out the page before using it. The last and final part of enclave termination done by EREMOVE involves freeing the SECS page, which can only be executed if no other page has references to it (thus indicating that all other enclave information has to be destroyed before SECS removal).

2.3.4 Measurements

Enclave measurements are needed in order to have a secure way of creating unforgeable enclave identities. These identities are unique and are subsequently used in the process of attestation and sealing that we describe in the following paragraphs. In order to create the measurements, the SGX architecture offers two registers in the enclave SECS, MRENCLAVE and MRSIGNER. MRENCLAVE represents the enclave's identity and corresponds to the value of a SHA-256 hash of all the record logs that are created during the enclave initialization (ECREATE, EADD, EEXTEND, EINIT). Since MRENCLAVE is a part of SECS, no updates can occur on the register after the enclave has been initialized. The content of the register effectively identifies all of the important page contents related to a specific enclave, including the code, data and enclave stack. Additionally, it records relative positions of the pages within the enclave along with their security properties. In summary, the MRENCLAVE holds all the important, security-sensitive variables that are needed to identify a particular enclave.

On the other hand, the enclaves can have a second identity, usually referred to as *sealing identity* which is used for data protection and verification on the local storage of the enclave's residing platform. This identity is stored in the MRSIGNER register after the EINIT instruction is invoked and completes successfully. MRSIGNER as the sealing identity is composed out of the product ID, a version number and a so-called *sealing authority*. The sealing authority represents the entity that owns the enclave, usually a content provider or an enclave developer, and subsequently signs it prior to distribution. The sealing authority is only stored in the MRSIGNER register if the SGX architecture

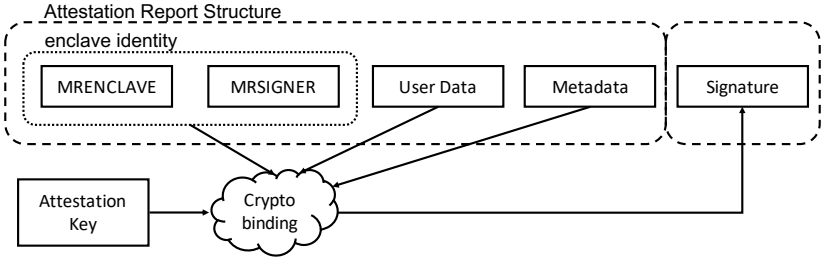

Figure 2.1: *Attestation Report Structure.*

successfully verifies the measurements. Namely, the sealing authority needs to present to the SGX architecture an expected value of the MRENCLAVE and a public key in form of a signed certificate of an enclave, known as SIGSTRUCT. SGX performs a comparison between the expected MRENCLAVE value and the actual performed measurement, verifies the certificate signature with the appended public key, and only if successful, stores the public key of the sealing authority to the MRSIGNER. As a side note, with this process, multiple different enclaves can have the same sealing identity if they have the same sealing authority. This entails the possibility of the enclaves created by the same authority to share the sealed resources in their execution. However, a single provider or developer can have multiple sealing authorities, since not all of the developer's enclave need to be able to access each others' sealed data.

2.3.5 Attestation

Attestation refers to the process of proving that a piece of software has been instantiated and initialized properly on the residing platform. Attestation is particularly important for systems that use trusted hardware. Trusted hardware usually instantiates an isolated container running a piece of software. That software can ask the underlying hardware to produce the measurement of the isolated container, create a small structure of attestation data and further on sign it with its secret key. This attestation signature can further on convince a potential verifier that a particular software is running in an isolated environment backed up by trusted hardware.

Intel SGX also implements an attestation scheme which follows the same principles outlined above. Namely, an SGX-enabled CPU first computes a measurement of each enclave, then starts a process that results in the attestation signature that includes both the enclave measurement and an optional message. This mechanism is important to satisfy one of the main goals of Intel SGX, secure remote computation. To create the end attestation signature, SGX first

creates the so-called attestation report structure which can be seen in Figure 2.1. The report structure includes both the enclave identity and the sealing identity associated to the enclave, user data which the environment associates with itself (can be used by enclave developers arbitrarily) and enclave metadata. A signature is formed with the attestation key and represents a cryptographic binding of the previously mentioned enclave data and the platform creating the attestation report. SGX supports two different attestation types which influence how the attestation key is used. First of these is the *local attestation* that is used to create authenticated attestation reports between two enclaves running on the same platform. The second is *remote attestation* that is used to provide proof of security guarantees to a remote verifier residing outside the platform, e.g., the service provider in the remote computation scenario.

Local Attestation With the local attestation an enclave can prove its identity to another target enclave, and that it runs inside a trusted platform. This is done by calling the EREPORT instruction which produces a specific attestation report (referred to as REPORT). This report is a signed structure similar to the previously mentioned attestation report (Figure 2.1) which cryptographically binds the enclave's measurement with a supplied message. Namely, the EREPORT reads the current enclave's information from its SECS, copies the MRENCLAVE, MRSIGNER and attribute fields. The cryptographical binding involves computing a MAC using a symmetric key, called the *report key*, that is shared between the target enclave and the residing platform. In this case, the report key serves as the attestation key that we saw in the attestation report structure. An enclave, in order to verify the attestation, can request the report key from the platform with the EGETKEY instruction, which actually derives the key from the secret key embedded in the processor and the target enclave's measurement. Note that when the EREPORT instruction is invoked, the same procedure with the EGETKEY is performed to obtain the report key in order to compute the MAC over the report.

A successful completion of local attestation indicates two-sided verification of enclave identities and checks that the enclaves run in SGX. The local attestation protocol works as follows and is showed in Figure 2.2:

(**1**) The $enclave_i$ receives the target enclave's identity in form of a MRENCLAVEj.

(**2**) The $enclave_i$ invokes the EREPORT instruction with the MRENCLAVEj which results in the creation of a $\text{REPORT}_{(i,j)}$ and send it to the $enclave_j$.

(**3**) The $enclave_j$ invokes the EGETKEY to retrieve the report key, and uses that key to recalculate the MAC over the received $\text{REPORT}_{(i,j)}$. A successful

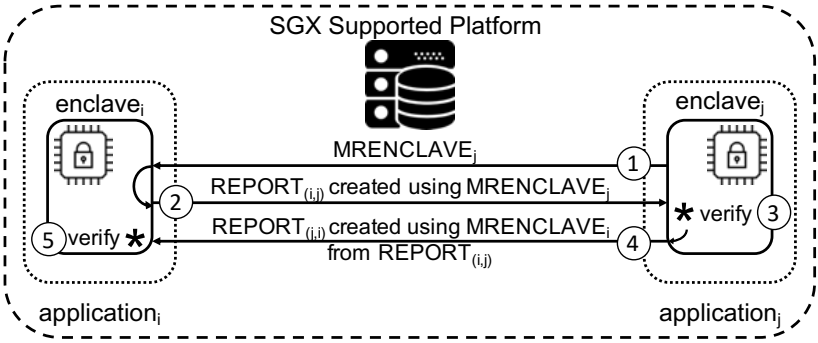

Figure 2.2: *Local Attestation Protocol..*

comparison of the MAC values guarantees to the $enclave_j$ that the $enclave_i$ is genuinely running inside SGX.

(**4**) The $enclave_j$ now uses the MRENCLAVE$_i$ in order to generate a REPORT$_{(j,i)}$ and sends it back to the $enclave_i$.

(**5**) The $enclave_i$ performs the same steps in order to perform verification of the $enclave_j$'s identity and that it is indeed running in SGX.

Remote Attestation. Performing local attestation is relatively easy since the enclaves share the same residing platform and secure hardware. The secret keying material embedded in the processor along with the enclave's identities is enough to derive keys and to provide verification for those enclaves. However, when a remote service provider wants to verify the correctness and guarantees of an enclave residing in a remote platform, such process is not sufficient, and previously described attestation keys cannot be used. For the remote attestation, SGX uses a different attestation signature. The facilitation of this process is too complex to implement in hardware, thus SGX has a special privileged enclave, called the *Quoting Enclave (QE)*, that performs the signing in remote attestation procedures. Since the signing functionality is in the QE, and the attestation has to be done for another target enclave on the same platform, the first step is to perform a local attestation between the target enclave and the QE. This enables the creation of a secure communication channel between these enclaves, and also the possibility for the QE to verify the target enclave's identity. Instead of a symmetric key, QE uses the *SGX attestation key* for remote attestation, which is a platform specific private key. The attestation report structure and the report itself are in this case called QUOTE instead of REPORT as in the local attestation.

Secret Key(s) Management. The remote attestation scheme depends on the Intel's key generation and provisioning services. During the manufacturing, an SGX-supported CPU communicates with the key generation facility to obtain two keys (provisioning and seal secret) that are burned into the e-fuses. E-fuses are simple one-time programmable storage spaces included in the CPU die. The provisioning secret is generated in the key facility and Intel is aware of its value, thus making it a shared secret between the SGX platform and Intel, while the seal secret is finally derived in the processor, without Intel knowing its end value. The SGX architecture also has another privileged enclave, called the *Provisioning Enclave (PE)*, which is used to connect to the Intel's provisioning service in order to authenticate the SGX platform it resides on, and the fused in secret keying material that was provisioned during manufacturing. Once the Intel's provisioning service verifies the target SGX platform, it generates the attestation key that is sent to the PE. The PE encrypts this key with the Provisioning Seal Key, and hands it off to system software for long-term storage. The remote attestation procedure using QE can only be invoked after the initial provisioning has been done for the first time.

QE's remote attestation and the attestation key use Intel's *Enhanced Privacy ID* (EPID)cryptosystem. EPID is a group signature scheme that has the main intention of providing anonymity to the signers (e.g., if the same attestation key would be used without any group signature scheme, one could easily identify an SGX platform throughout its lifetime and also track it). Intel's provisioning service is the issuer of the EPID scheme and after the provisioning enclave authenticates itself to the service, an EPID Member Private key is generated to serve as the attestation key. EPID Member Private key is connected to a Group key known by Intel, but performing attestation with the member key in a blinded fashion effectively disables Intel to tie specific attestation statements with specific attestation keys and individual chips that hold them. QE is the only one that can use the EPID key.

A successful completion of remote attestation indicates that a service provider can verify the target enclave's identities and check that the enclaves run in remote platform that supports SGX. The remote attestation protocol works as follows and is showed in Figure 2.3:

(**1**) The Service Provider creates a challenge with a nonce and sends it to the untrusted part of the SGX application. The challenge binds the remote attestation scenario to a specific instance (from the Service Provider's point of view) and the nonce prevents replay attacks.

(**2**) The untrusted application part forwards the challenge along with the identity of the Quoting Enclave to the target enclave (TE) on the platform.

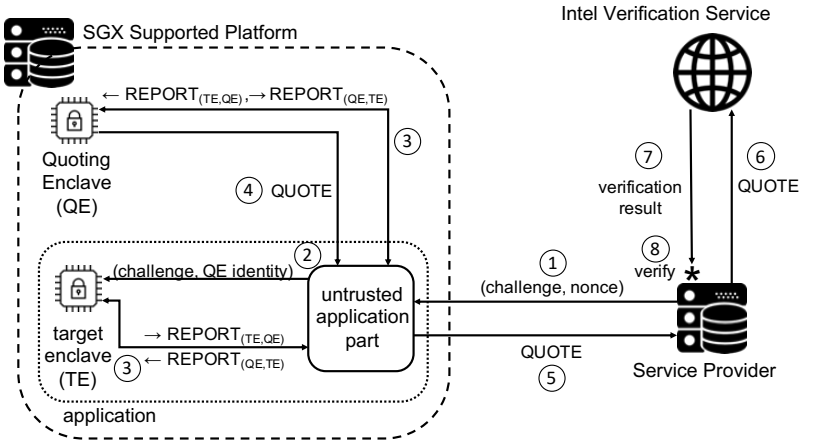

Figure 2.3: *Remote Attestation Protocol..*

(**3**) The target enclave performs local attestation with the Quoting Enclave as described above. The process is done through the untrusted application part that facilitates communication between the enclaves.

(**4**) If the local attestation is successful, the QE enclave creates a QUOTE, an attestation report that is returned back to the application.

(**5**) QUOTE is further on sent to the service provider along with a possible additional data/message.

(**6**) The service provider forwards the QUOTE to the attestation verification service provided by Intel in order to verify the signature of the QUOTE report.

(**7**) The verification service checks the QUOTE signature, returns the result to the service provider.

(**8**) In the end step, the service provider performs verification if the challenge, that was initially sent, is composed in the response. If yes, the remote attestation is complete and the service provider can be sure that the correct software is running inside an SGX enclave (note that the QUOTE contains the enclave measurement as the REPORT in local attestation does, allowing the service provider to compare code and data that are supposed to run in an enclave).

In general, Intel is the only entity currently providing the verification service for attestation. Technically, this verification service could be provided by some other service operator, however, then the service operator would need to obtain the Group Key from Intel to check if the EPID scheme provided

a valid signature. Thus, Intel currently has to be involved in every genuine enclave usage which poses some serious usability and also security concerns.

2.3.6 Sealing

The above outlined details regarding runtime isolation only provide protections in respect to confidentiality and integrity for live enclaves. In the end of the enclave lifecycle, when the enclave is completely terminated, all of the runtime data in the memory is lost with it. Naturally, we want a mechanism that allows re-using of some data across multiple executions of the same enclave. This, however, involves storing data outside of the SGX TCB, or more precisely, outside the CPU package and into some permanent local storage, such as a hard drive on the residing platform that is controlled by the untrusted OS. SGX supports such a mechanism and is referred to as *sealing*. Sealing is the process of encrypting and authenticating enclave data for persistent storage.

To perform encryption and authentication of sealed data we need to use some key material. SGX offers to perform sealing based on two different policies that identify the so-called *seal keys*, and based on the policy the keys for sealing are either based on the enclave identity or the sealing identity. In order to create a seal key, the EGETKEY instruction is invoked by the user with the additional attributes indicating which policy is to be used. For example, if the enclave choses to seal data based on its enclave identity, the EGETKEY instruction will derive a symmetric cryptographic key based on the value of the enclave's MRENCLAVE register. Respectively, if we chose to seal data based on the sealing identity, the EGETKEY will use the information from the MRSIGNER register in the key derivation process.

These two approaches have some trade-offs. If enclave identity is used to create the seal key, the sealed data can be accessed only by this particular enclave on the platform that the enclave was created (since that also affects the value of the enclave identity). This provides increased security since sealed data and the enclave cannot just be ported to another platform, nor can another enclave, even with the same developer and sealing identity, unlock the data. However, this mechanism also affects usability. Since the key is so tightly tied to the enclave identity, the enclave code cannot be updated with another version by the authorized developer and the original data unsealed subsequently to continue using it. Intel's recommendation includes using this method only in situation where destroying the old permanently data is a viable option in case of some vulnerabilities or attacks on the enclave (for which it might be difficult to find a use case).

On the other hand, if the sealing identity is used to create the seal keys, offline migration of sealed data is possible. Additionally, enclaves can be

updated and their new versions are able to access the data as well. The sealing identity has the authority to sign multiple different enclave certificates with the same private key, thus resulting in the same MRSIGNER value in the SGX architecture. To enable the sealing authority to also arbitrarily disallow access to specific, potentially old, sealing data, the previously mentioned version number is included in the MRSIGNER register. In this case, if the sealing authority creates another version of the enclave with different functionalities, it just has to increase the version and thus invalidate all previously sealed old data when the new enclave is running. For the other way around, we also want to enable new enclaves to sometimes access old data, SGX supports specifying a range of version numbers that are accepted as valid for the unsealing process. Lastly, this mechanisms effectively disables old enclave versions to access data sealed by newer enclaves, rounding up a complete set of procedures for seamless upgrade and versioning of secure enclave applications.

2.3.7 Performance Measurements

In order to verify claims about SGX performance we have conducted some in-enclave tests. Table 2.2 provides measurements of cryptographic operations on SGX. We report average time over 1M repetitions. All enclave operations are reported switching time excluded. The test platform was running Windows 10 on Intel i7-6500U, 8GB RAM and 256GB SSD. As a summary, we observe that the performance inside the enclave is in the same order of magnitude as the performance on the outside. We encounter around 10% overhead across all observed operations. One of the main reasons for the overhead is the additional steps taken within the MEE when the EPC pages are swapped out to main memory. In order to increase performance, frequent switching to and from the enclave context should be avoided.

Operation	**Time**
enclave switching time	2.6 (± 0.0) μs
SHA256	2.4 (± 0.0) μs
Opening and closing ECC context	2.4 (± 0.0) μs
ECDSA signing (0.5KB)	457.5 (± 0.3) μs
ECDSA verfication (0.5KB)	843.6 (± 0.9) μs
Sealing (1KB)	9.6 (± 0.1) μs
Unsealing (1KB)	4.5 (± 0.1) μs
AES-CTR$_{128}$ encryption (0.5KB)	0.63 (± 0.0) μs
AES-CTR$_{128}$ decryption (0.5KB)	0.62 (± 0.0) μs
AES-GCM encryption + MAC (1KB)	1.05 (± 0.0) μs
AES-GCM decryption + verification (1KB)	1.07 (± 0.0) μs

Table 2.2: *Cryptographic operations on SGX.*

2.3.8 Discussion

Intel SGX's threat model includes the lack of trust from privileged platform software. Security guarantees are based on the microcode implementation and are kept within the CPU package. Additionally, memory protection is guaranteed through encryption, authentication and freshness checks that can withstand even the physical attacks on DRAM. However, SGX is vulnerable, and as clearly stated by Intel, against side-channel attacks. Side-channel leakage is one of the main topics in Chapter 5 where we tackle a specific privacy problem for lightweight clients in popular cryptocurrencies. Additionally, recent attacks on SGX, such as SGXPECTRE [49] and FORESHADOW [191] show that SGX in its current state can be completely subverted by a strong attacker (the one that can observe memory accesses and control flow of the process at low level granularity, executing side-channel attacks). This, however, does not reduce the value of the architecture, as also stated by the authors of these papers, but merely gives a greater picture to the research and industry community that further development is needed.

Another problem in respect to SGX security is the freshness of its long-term states that are stored through sealing. Even though SGX offers mechanisms for state freshness, we investigate them in Chapter 3 and reason about their security and usability. We propose a much efficient solution with hard security guarantees that enables usage of SGX in a much broader application space where frequent and continuous state updates to local storage are needed.

Our goal is not to reason about all the security characteristics of SGX but to show the potential of the architecture and pick several important scenarios to perform an in-depth investigation. A major concern regarding SGX is, that since its inception, there exists a serious lack of transparency and publicly available information about the implementation details that still remain proprietary and secret information. The research over the last several years has allowed us to gain more insights into the specifics of the SGX implementation, but the complete picture is still yet to be achieved.

2.4 Overview of Trusted Hardware Architectures

In this section we provide a brief overview of a larger set of architectures on trusted hardware projects that are related to Intel SGX and outline the differences in the protections they offer. A much broader comparison can be found in the *Intel SGX Explained* paper [55], that in more than a hundred pages explain Intel SGX along with all of its implications, design, architecture and related work in great detail.

2.4.1 Secure Hardware Components

IBM 4765 Secure Co-processor. The general idea of secure co-processors is to encapsulate a whole computer system withing a tamper-resistant environment [205]. A whole computer system includes the CPU package, DRAM, peripheral controllers and other relevant components depending on the system. The tamper-resistant environment is protected by an array of different sensor that can detect physical attacks, and if such an attack is detected, the co-processor destroys all the stored secrets. While the offered protections against physical attacks are good, the relative ratio between the price of the computer system and the tamper-resistant enclosure is not proportional, making these co-processors very expensive.

One of the most representative co-processors is the IBM's 4765 [25]. The security is based heavily on physical isolation, where the system software is running on the service processor, while the other applications have their own separate application processor. The two are connected with a special bus controller that prevents the application space from reaching any privileged resources tied to the system software. IBM 4765 also supports software attestation for the applications loaded in the co-processor. The keys are stored in battery supported memory only available to the service processor. IBM 4765 offers strong security protections, but the price of such physical tamper-resistance is high and coupled with the low-usability programming environments that are not often compatible with a wide range of services makes their adoption applicable for certain, and specific, niche use-cases [178].

Trusted Platform Module (TPM). The Trusted Platform Module (TPM) [75, 76, 190] is another example of a trusted co-processor that represent an auxiliary tamper-resistant chip running one isolation container. TPM itself does not require any hardware modifications to the CPU, and hence it is widely deployed on the commodity desktop and server platforms. The module is used only to store an attestation key and perform software attestation. Additionally, TPM can store some other keys and perform limited operations with them, but TPMs, in general, are far from being general computing processors.

The software attestation and the resulting attesation signature performed by a TPM include the entire OS kernel, kernel module and device drivers. The attestation procedure relies on the CPU to report its own measurement from each of the booting phases. Namely, the firmware on most computer platforms implements the Unified Extensible Firmware Interface (UEFI) specification where each platform initialization phase is responsible for measuring the firmware related to the next phase. TPM has Platform Configuration Registers

(PCRs) that are used to store the received measurements. Each time a specific boot stage during the initialization procedure sends a hash, the TPM updates the PCRs and incorporates it. All received hashes up to a certain point are tied to the previous hashes to prevent the software at a certain stage from removing itself from the measurement and changing the behavior.

TPM's security guarantees rely on the fact that the first hash sent to the TPM has to reflect the exact software running in the first stage. Even though, the TPM threat model assumes that the first stage boot loading is securely embedded on the motherboard of the device, and protected from unwanted modification, the realistic situation is somewhat different. In almost all modern computers the firmware is stored in flash memory of a re-programmable chip on the motherboard, thus making it easy for an attacker with physical presence to subvert the whole system, regardless of its tamper-resistant properties.

Intel Trusted Execution Technology (TXT). Intel's Trusted Execution Technology (TXT) [73, 108] is a scheme that combines the usage of TPM's software attestation and secure container together with their CPU's hardware virtualization features. Unlike the TPM model where the whole system software is covered by the container, the TXT aims to provide isolation between virtual machines (including the guest OS and applications). TXT isolates the software residing inside the secure container from all other software by ensuring that only that container has exclusive control over the entire system when it is running. For its deployment, the TXT requires the existence of a TPM with an extended register set, thus containing a static route of trust measurement for the platform itself and a dynamic route of trust measurement for the virtual machines (when it updates the registers depending on the running software).

However, TXT does not implement any DRAM protection, such as encryption and integrity verification, making it vulnerable to physical attacks. Early version implementations were vulnerable to attacks where a malicious OS could reprogram devices, executing direct memory access transfers into DRAM regions used by the TXT containers. Newer versions were used in conjunction with the System Management Model (SMM), a hardware-assisted isolated execution environment controlling platform specific system functions, however, SMM has proven to be vulnerable as well.

2.4.2 CPU-based Hardware Security

ARM TrustZone. ARM TrustZone [18, 23, 199] is a security architecture developed for the ARM platform and it represents a collection of hardware modules that can be used to partition the residing platform's resources between

a so-called *secure world*, hosting a secure isolated container, and the *normal world*, hosting an untrusted software stack. The architecture is based on the extended address lines of the AMBA AXI system bus, where a signal is set in order to indicate if a particular access belongs to the normal or the secure world. Not all of the processors support TrustZone (similar as in the SGX case, e.g., Xeon with a high core count do not yet support it), and the ones that do, have to include the TrustZone's security extensions. Each processor core can switch the execution between the normal and secure world, and the address in each bus access will indicate in which world each core operates.

A TrustZone supported system is initiated by first putting the processor in a secure mode and starting the first stage boot process that is stored in the on-chip ROM memory. Hence, the first part that is initialized is the secure world, upon which the boot loader for the normal world is initialized. A secure container has an implemented monitor that performs checks on context switching between the secure and normal world of a specific processor core and handles hardware exceptions and interrupts. By design, TrustZone's monitor from the secure world has *root* access to the normal world, and of course the secure world where it is running. Unlike SGX, where the trusted code is only the enclave, TrustZone's TCB includes the whole secure world composed out of the firmware, OS and other arbitrary applications, thus being much larger. A direct implication of this feature is that there exists no isolation between the applications running in the secure world, and if there is potentially any malicious code in the whole secure world, it can effectively dismantle any operation performed in the normal world, due to the unrestricted access.

Since TrustZone only provides the architecture and the extension set, it is not always entirely clear how these are implemented by processor manufacturers. A security analysis of the architecture is orthogonal to the understanding how the architecture performs on an exact chip. Device manufacturers often omit low-level details, that in case of TrustZone applicability, are very important to reason about the end security. One example is that an undocumented process on cache handling by the processor cores might results in a simple software side-channel attack, subverting the protections of the secure world by running attacks from the untrusted normal world.

TrustZone components were not designed to have protections against physical attacks. The general threat model includes the trust boundary to be the CPU package, while the AXI bus is either way designed to connect components in a system-on-chip design, making it very hard to perform tapping. TrustZone also recommends that the whole secure world software package is stored in the on-chip SRAM, which is protected. However, this is somewhat not practical and limits the usability of the architecture. SRAM

sizes are usually very small and the price ratio compared to a more general DRAM is several orders of magnitude higher.

XOM Architecture. XOM stands for *eXecute-Only Memory*. The architecture was introduced by Lie et al. in 2000 [133], and includes an approach of having secure isolated containers managed within the untrusted host system that would execute sensitive code and data. XOM is based on several different properties. First, the processor's memory controller protects the memory of a single container by integrating automatic encryption and HMAC functionality. Second, in the processor package each cache line is tagged with an identifier that specifies the container using it. Additionally, before serving an interrupt the register states are transferred to the protected memory area. Third, hardware exceptions, such as page faults, are not coexistent with the architecture and are not supported. Fourth, XOM offers a so called software distribution scheme where the secret container content is encrypted with a symmetric key, serving as the container identifier as well. These identifiers are encrypted with the trusted CPU's public key. A form of software attestation can be achieved if the container owner embeds a secret into the container data that is subsequently used to authenticate the container.

Encrypted memory with HMAC performed by the CPU's memory controller was designed to protect XOM from physical attacks on dynamic random-access memory (DRAM) of the residing platform. However, XOM does not have a mechanism to guarantee memory freshness, making it vulnerable to physical memory replay attacks. Additionally, the memory access patterns are not protected, leaving side-channel leakage as a serious threat. Namely, using cache timing attacks against container software leaks data. In the processor itself, access to cache lines is only allowed to containers that have a matching identifier to the cache line tag. Thus, the OS and other untrusted software (which usually also have a tag identifier with null value) are unable to gain access. Lastly, even though XOM offers some container authentication with its simple attestation scheme, it does not enable checking of the container's software environment by the container author (no outside measurements of the platform are done).

The Aegis Secure Processor. In 2003, Suh et al. [184] proposed Aegis, a single-chip architecture for Tamper-Evident and Tamper Resistant processing that would protect against both physical and software attacks. Aegis's trusted computing base consist of the processor chip and a part of the operating system. This part of the operating system forms a trusted core part called *security*

kernel. The security kernel represents a smaller subset of the normal OS kernel which operates at a higher protection (by exploiting processor features to isolate itself) level in order to prevent possible attacks from untrusted parts of the operating system. In this subset, tasks such as virtual memory management, processes and hardware exceptions are handled. To prove its guarantees, Aegis can perform a software attestation procedure which includes the kernel's cryptographic hash in the measurement. In order to satisfy the mechanism of software attestation, Aegis also needs to have a protected private key for signing. In the follow-up work [186], Suh presents the design and implementation of the Aegis secure processor and argues that instead of using e-fuses for hardcoding the key, a set of Physically Unclonable Functions (PUFs) can be used to provide a better resilience to physical attacks.

The architecture also allows the existence of secure containers that are isolated by the security kernel. To enable memory protection, Aegis incorporates two mechanisms: integrity-verification that protects the integrity of data residing outside of the chip, and symmetric encryption that serves to protect confidentiality (privacy) of the data. Integrity verification is done using Merkle trees where chunks of memory represent tree leaves. The parents are the hash of the concatenated children and each of them represents one L2 cache block. With this design, the processor is able to verify freshness of the data, thus being one of the first solutions that protects against physical replay attacks.

Isolation of secure containers is achieved by configuring page tables for address translation, thereby disallowing untrusted software to access the content of secure isolated containers. Even though the untrusted OS can evict pages from container memory, the security kernel verifies the correctness of these operations. Kernel additionally uses the same techniques as the memory controller to guarantee integrity, confidentiality and freshness of container related pages that are swapped in and out of the memory. However, due to the possibility of the untrusted OS to handle pages, Aegis leaks memory access patterns and is also vulnerable to cache-timing attacks, alike the XOM.

The Bastion Architecture. Bastion was introduced as a new architecture that combines hardware and software level protections to ensure security of critical software modules running under an untrusted software stack [47]. The architecture is composed of an enhanced trusted hypervisor that works along a microprocessor. Namely, the trusted hypervisor provides secure containers for software running inside an untrusted operating system, by ensuring that the OS cannot interfere with the execution of the secure containers. To enforce protection by the hypervisor, each trusted software module has a *security*

segment that lists the virtual addresses and permissions of all the container's pages. These segments are kept in the *module state table* (MST) that associates each physical memory page to a specific container and the virtual address space. At runtime, the protections are enforced in the processor, where every memory lookup is checked against the rules specified in the Translation Lookaside Buffer (TLB). If a miss occurs, the trusted hypervisor is invoked before the TLB can be updated with the address translation results. The hypervisor then checks if the virtual addresses to be used in translation match the expected values associated with a specific security segment of a container in the MST.

Alike the characteristics of XOM and Aegis, Bastion also allow the untrusted OS to manage pages of the secure containers. Even though the pages are protected using HMAC (authenticity), encryption (confidentiality) and covered by the Merkle tree to protect freshness against physical replay attacks (similar to XOM), the OS's ability to randomly evict pages leaks memory access patterns on a page level. Additionally, Bastion also does not protect against cache-timing side-channel attacks. Since Bastion's threat model includes an untrusted platform and firmware, a cryptographic hash of the hypervisor is calculated after the boot procedure is executed, and that hash is later added to the measurement of the software attestation process.

Bastion was the first architecture to provide direct hardware protection of the hypervisor from both software and physical attacks. The hardware protections are executed before allowing the hypervisor to provide protection to application modules. Bastion effectively demonstrates the feasibility of bypassing an untrusted OS to provide application security with better performance than the industry standard, TPM.

Sanctum. Costan et al. [56], authors of the *Intel SGX Explained* paper, present Sanctum, a software isolation scheme that is a co-design combining minimal and minimally invasive hardware modifications to provide strong provable isolation of secure containers running in parallel and sharing resources. Sanctum's scheme is implemented in the trusted software security monitor and no cryptographic operations using keys are performed. The security monitor acts as the first firmware that is executed upon start and has similar properties and characteristics as the Aegis's security kernel. Basically, Sanctum offers the same promises as Intel SGX, but also adds protection against a completely different class of software attacks that target the memory access patterns to infer secret information (namely, side-channels), against which, as stated by the design itself, Intel does not offer any protections.

To achieve its additional guarantees, Sanctum uses a cache partitioning scheme in which the platform's available DRAM is split into continuous DRAM regions of the same size. Each region also corresponds to a distinct set of the last-level cache (LLC), shared between the processor cores. Subsequently, when operating secure containers, each container is assigned to exactly one of the described DRAM regions. This makes containers isolated in both DRAM and LLC. However, lower level caches still present a problem. In order to create isolation there, all other caches are flushed when the context is switched, e.g., an entry-exit is performed on the secure container. The cache partitioning scheme is not only used for secure containers but also for the untrusted software (such as the hypervisor, OS, and other applications), thus effectively isolating them from other trusted code.

On each start the security monitor is measured and the cryptographic hash of the measurement is added to the software attestation. After that, the monitor is able to verify resource allocation decisions of the operating system in order to verify if each container has only single access to its region, and no region is shared. Since Sanctum containers manage its own page mappings for DRAM regions, and thus, handle those page faults for its regions, the OS is unable to infer any information about the virtual address translations.

Another valuable benefit of Sanctum is its open implementation (prototype on a RISC-V core) that allows the research community to investigate the design, adapt and modify it, and reason about Sanctum's security properties. Unlike Intel SGX, that hides its protections behind opaque, proprietary microcode and publicly undisclosed details regarding the design, Sanctum's proposed extension can be adapted to other existing processors, since no changes to the processor building blocks are made, by just adding hardware on the interfaces between these generic building blocks. These hardware modification essentially check that the container's pages only reference the memory region assigned to that container.

Sanctum, however, focuses purely on software-based attacks, thus lacking the protection from physical attacks. The contributions presented in Sanctum, and its proposed hardware modification, could be combined with the protections guaranteed by, e.g., Aegis, resulting in increased overall resilience against physical attacks.

2.5 Architecture Comparison in Relation to SGX

Intel Software Guard Extensions is an architecture that implements secure containers, called enclaves, that execute application in isolation from the rest of the untrusted platform, including the OS, hypervisor and the firmware.The

trusted computing base of SGX includes the CPU chip package, the proprietary microcode modifications and a few privileged enclaves that enable SGX functionalities, such as the Launch Enclave (for creating application enclaves), Provisioning Enclave (for registration, key provisioning and derivation), and the Quoting Enclave (for remote attestation).

SGX bases its protections by forming a complete context switch in the processor's cache when an enclave is being executed allowing isolated execution in a specific core from an untrusted software stack. A core's TLB is always flushed when the context switch occurs. Similar to Bastion's architecture, SGX allows the OS to manage memory allocations, and thus, perform page management, evicting and loading pages depending on the currently executing process. However, SGX implements additional protections to ensure successful context switching and offloading of protected memory content. More specifically, the confidentiality, authenticity and freshness is guaranteed for enclave's pages by hardware encryption and by following Aegis's approach of using Merkle trees for versioning. DRAM protections also include confidentiality and authenticity, following the same properties XOM, Aegis and Bastion.

The architecture itself, unfortunately, is not designed to protect leakage from memory access patterns, as, for example, Sanctum does. The vulnerability to side-channel attacks allows a potential adversary to learn memory access patterns of a secure enclave on page level granularity, thus sharing the same vulnerabilities as Aegis and Bastion. Cache-timing represents a viable threat, since enclaves cannot perform data accesses and operations based on secrets. Moreover, not only the enclaves from external developer are vulnerable, but also the privileged SGX enclaves that form the TCB of the architecture and provide security features on which everything is based on. Recently, many research groups have proven that side-channel are indeed possible [40, 71, 151, 170], and research has also demonstrated that platform vulnerabilities like Spectre [120], Meltdown [136], and an SGX-focused Foreshadow [191] can be leveraged to extract attestation keys from SGX processors [49]. Even though fixes were implemented immediately to reduce the attack surface, we cannot disregard these vulnerabilities.

In Table 2.3 we show a comparison of the most important architecture characteristics between SGX and other system reviewed in this background (where ✓ indicates that the threat is addressed, and ✗ the opposite, respectively). This can give a reader an intuition of the space where we do our research, and what specific properties we tackle, use or implement in order to forgo the investigation of SGX as a modern security enabler.

Threat	SGX	IBM4765	TPM	TXT+TPM	TrustZone	XOM	Aegis	Bastion	Sanctum
Malicious OS									
direct probing	✓[a]	N/A[c]	N/A[c]	✓[e]	✓[a]	✓	✓[c]	✓[b;i]	✓[a]
page faults	✗	N/A[c]	N/A[c]	✓[e]	✓	N/A	✗	✗	✓
cache timing	✗	N/A[c]	N/A[c]	✓[e]	✗	✗	✗	✗	✓
Malicious Containers									
direct probing	✓[a]	✓	N/A[d]	N/A[d]	N/A[f]	✓	✓[c]	✓[a]	✓[a]
cache timing	✗	✓	N/A[d]	N/A[d]	N/A[f]	✗	✗	✗	✓
Malicious Hypervisor									
direct probing	✓[a]	N/A[c]	N/A[c]	✓[e]	✓[a]	N/A[h]	N/A[h]	N/A[h]	✓[a]
Malicious Firmware									
direct attack	✓[a]	N/A[c]	✓	✓	N/A[f]	N/A[h]	N/A[h]	✓	✓
Physical DRAM attack									
read	✓[b]	✓	✗	✗	✓[g]	✓[b]	✓[b]	✓[b]	✗
write	✓[b]	✓	✗	✗	✓[g]	✓[b]	✓[i]	✓[i]	✗
rollback	✗	✓	✗	✗	✓[g]	✗	✓[i]	✓[i]	✗
address read	✗	✓	✗	✗	✓[g]	✗	✗	✗	✗
Direct Memory Access									
malicious peripherals	✓[b]	✓	✗	✓	✓	✓[b]	✓[b]	✓[b]	✓
HW TCB size	CPU package	Chip package	Mother-board	Mother-board	CPU package	CPU package	CPU package	CPU package	CPU package
SW TCB size	Containers	FW; OS	All SW	OS; APP	FW; OS; APP	APP + HYP	APP + kernel	APP + HYP	APP + monitor

[a] Access check on TLB deployed.
[b] DRAM is encrypted and protected.
[c] Hypervisor and OS are measured and trusted.
[d] Concurrent containers not supported.
[e] Hypervisor and OS preempted at late launch.
[f] Secure world is trusted.
[g] Secure world in on-chip SRAM.
[h] No hypervisor or firmware support.
[i] Merkle tree over DRAM, HMAC.

Table 2.3: *Overview and comparison of Hardware-based TEEs in relation to SGX. Closely resembles [55].*

Chapter 3

ROTE: Rollback Protection for Trusted Execution

3.1 Introduction

As desribed in Chapter 2, Intel SGX enables execution of security-critical application code, called *enclaves*, in isolation from the untrusted system software. To protect enclave data across executions, SGX provides a security mechanism called *sealing* that allows each enclave to encrypt and authenticate data for persistent storage.

The architecture has also its limitations. While sealing prevents a malicious OS from reading or arbitrarily modifying persistently stored enclave data, *rollback attacks* [55, 157, 180, 182] remain a threat. In a rollback attack a malicious OS replaces the latest sealed data with an older encrypted and authenticated version. Enclaves cannot easily detect this replay, because the processor is unable to maintain persistent state across enclave executions that may include platform reboots. Another way to violate state integrity is to create two instances of the same enclave and route update requests to one instance and read requests to the other. To remote clients that perform attestation, the instances are indistinguishable.

Data integrity violation through rollback attacks can have severe implications. Consider, for example, a financial application implemented as an enclave. The enclave repeatedly processes incoming transactions at high speed and maintains an account balance for each user or a history of all transactions in the system. If the adversary manages to revert the enclave to its previous

state, the maintained account balance or the queried transaction history does not match the executed transactions.

To address rollback attacks, two basic approaches are known. The first is to store the persistent state of enclaves in a non-volatile memory element on the same platform. The SGX architecture was recently updated to support monotonic counters that leverage non-volatile memory [103]. However, the security guarantees and the performance limits of this mechanism are not precisely documented. Our experiments show that writes of counter values to this memory are slow (80-250 ms), which limits its use in high-throughput applications. More importantly, this memory allows only a limited number of write operations. We show that this limit is reached within just few days of continuous system use after which the memory becomes unusable. Similar limitations also apply to rollback protection techniques that leverage Trusted Platform Modules (TPMs) [157, 180, 182].

The second common approach is to maintain integrity information for protected applications in a separate trusted server [116, 121, 192]. The drawback of such solutions is that the server becomes an obvious target for attacks. Server replication using standard Byzantine consensus protocols [45] avoids a single point of failure, but requires high communication overhead and multiple replicas for each faulty node.

In this chapter we propose a new approach to protect SGX enclaves from rollback attacks. The intuition behind our solution is simple. A single SGX platform cannot prevent rollback attacks efficiently, but in many practical scenarios the owner or the owners of processors can assign multiple processors to assist each other. Our approach realizes rollback protection as a distributed system. When an enclave updates its state, it stores a counter to a set of enclaves running on assisting processors. Later, when the enclave needs to recover its state, it obtains counter values from assisting enclaves to verify that the recovered state data is of the latest version.

We consider a powerful adversary that controls the OS on the target platform and on *any* of the assisting platforms. Additionally, we even assume that the adversary can break SGX protections on some of the assisting processors and control all network communication between the platforms. Our adversary model combines commonly considered network control based on the standard Dolev-Yao model [58] and Byzantine faults [126, 159], but additionally captures the ability of the adversary to restart trusted processes from a previously saved state and to run multiple instances of the same trusted process. Such adversarial capabilities are crucial for the security analysis of our system, and we believe that the model is of general interest. In fact, using our model we found potential vulnerabilities in recent SGX systems [169, 171, 180].

Secure and practical realization of distributed rollback protection under such a strong adversarial model involves several challenges. One of the main challenges is that when an assisting enclave receives a counter, its own state changes, which implies a set of new state updates that would in turn propagate. To prevent endless update propagation, the counter value must be stored in the volatile runtime memory of enclaves. However, the assisting enclaves may be restarted at any time. Moreover, the adversary can also create multiple instances of the same enclave on all assisting platforms and route counter writes and reads to separate instances.

We design and implement a rollback protection system called ROTE (Rollback Protection for Trusted Execution). The main components of our solution are a state update mechanism, a recovery mechanism that obtains lost counters from the rest of the protection group upon enclave restart, and a session key update mechanism to address attacks based on multiple enclave instances. More specifically, the general high-level model of our rollback protection system can be seen as the Byzantine Fault Tolerant (BFT) write/read storage in the asynchronous model [78, 80, 123, 124]. However, as we will point out further on, the specific properties of SGX enclaves and our system/adversary model require substantial modification in order to achieve safety and liveness. The distributed characteristics of the state update mechanism can also be seen as an optimized version of the consistent broadcast protocols [43, 163].

Our solution achieves a strong security property that we call *all-or-nothing rollback*. Although the attacker can restart enclaves freely, and thus implement subtle attacks where enclave state updates and recovery are interleaved, the adversary cannot roll back any single enclave to its previous state. The only way to violate data integrity is to reset the entire group to its initial state. If desired, similar to [157, 182], our approach can also provide crash resilience, assuming deterministic enclaves and a slightly weaker notion of rollback prevention (the latest input can be executed twice).

We implemented ROTE on SGX and evaluated its performance on four SGX machines. We tested larger groups of up to 20 platforms using a simulated implementation over a local network and geographically distributed enclaves. Our evaluation shows that state updates in ROTE can be very fast (1-2 ms). The number of counter increments is unlimited. This is in contrast to solutions based on SGX counters and TPMs, where state updates are approximately 100 times slower and limited. Compared to BFT write/read storage protocols, our approach instantiates a robust atomic storage with fast read and write access under appropriate assumptions, and thus achieves a minimal bound of $f + 2u + 1$ needed replicas, where u identifies replicas that can fail, e.g., be non-responsive, and f identifies malicious replicas, e.g., deviating arbitrarily

from the protocol. Enclave developers can use our system through a simple API. The ROTE TCB increment is moderate (1100 LoC).

Contributions. In summary, we make the following contributions:

- *New security model.* We introduce a new security model for reasoning about the integrity and freshness of SGX applications. Using the model we identified potential security weaknesses in existing SGX systems.
- *SGX counter experiments.* We show that SGX counters have severe performance limitations.
- *Novel approach.* We propose a novel way to protect SGX enclaves from rollback of their persistent sealed state. Our main idea is to realize rollback protection by storing enclave-specific counters in a distributed system of collaborative enclaves on distinct nodes.
- ROTE. We propose and implement a system called ROTE that effectively protects against rollback attacks. ROTE ensures integrity and freshness of application data in a powerful adversarial model.
- *Experimental evaluation.* We demonstrate that distributed rollback protection incurs only a small performance overhead. When deployed over a low-latency network, the state update overhead is only 1-2 ms.

The rest of this chapter is organized as follows. Section 3.2 explains models and rollbacks attacks. Section 3.3 describes our approach. Section 3.4 describes the ROTE system and Section 3.5 provides security analysis. Section 3.6 provides performance evaluation and Section 3.7 further discussion. We review related work in Section 3.8. Section 3.9 concludes the chapter.

3.2 Problem Statement

In this section we define models for the SGX architecture and the adversary. After that, we explain rollback attacks, limitations of known solutions, and our requirements.

3.2.1 SGX Model

Figure 3.1 illustrates our SGX model. We model enclaves and the operating system, their main functionality, and the operations through which they interact. Our model captures the main SGX functionalities that are available on all SGX platforms as previously described in Chapter 2.

Scheduling operations. Enclave execution is scheduled by the OS.

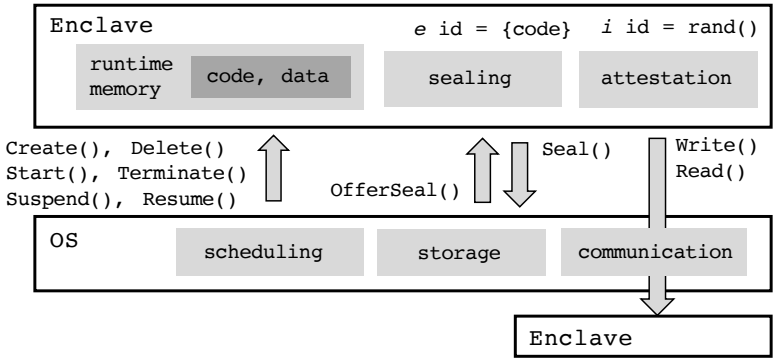

Figure 3.1: *Modeled SGX operations.*

• $e \leftarrow \texttt{Create}(code)$. The system software running on the OS can create an enclave by providing its code. The SGX architecture creates a unique enclave identifier e that is defined by the code measurement.

• $i \leftarrow \texttt{Start}(e)$. The system software can start a created enclave using its enclave identifier e. The enclave generates a random and unique instance identifier i for the enclave instance that executes the code that was assigned to it during creation. While an enclave instance is running, the OS and other enclaves are isolated from its runtime memory. Each enclave instance has its own program counter and runtime memory.

• $\texttt{Suspend}(i)$ and $\texttt{Resume}(i)$. The OS can suspend the execution of an enclave. When an enclave is suspended, its program counter and runtime memory retain their values. The OS can resume suspended enclave execution.

• $\texttt{Terminate}(i)$. The OS can terminate the enclave execution. At termination, the enclave runtime memory is erased by the SGX architecture and the enclave instance i is rendered unusable.

Storage operations. The second set of operations is related to sealing data for local persistent storage.

• $s \leftarrow \texttt{Seal}(data)$. An enclave can save data for local persistent storage. This operation creates an encrypted, authenticated data structure s that is passed to the OS.

• $\texttt{OfferSeal}(i,s)$. The OS can offer sealed data s. The enclave can verify that it previously created the seal, but the enclave cannot distinguish which seal is the latest. Every enclave instance i can unseal data previously sealed by an instance of the same enclave identity e.

Communication operations. Due to attestation, a client can write data such that only a particular enclave can read it. The client can read data from an enclave and verify which enclave wrote it. We model these primitives as single operations that can be called from the same or remote platform, although attestation is an interactive protocol between the enclave and client.

• `Write`(m_e, i). The OS can write message m_e to an enclave instance i. Only an enclave with enclave identity e can read the written message m_e.

• $m_e \leftarrow$ `Read`(i). The OS can read message m_e from an enclave instance i. The read message m_e identifies the enclave identity e that wrote the data.

Note that remote attestation identifies the enclave identity, but not the platform identity, because the attestation protocol is either anonymous or returns client-specific pseudonyms. In local attestation the platform is implicitly known.

We do not model platform reboots, as those have the same effect as enclave restarts. Our model assumes that the runtime memory of each enclave instance is perfectly isolated from the untrusted OS and other enclaves. We consider information leakage from side-channel attacks a realistic threat [40, 170, 204], but an orthogonal problem to rollback attacks, and thus outside of our model.

3.2.2 Local Adversary Model

We consider a powerful adversary who, after an initial trusted setup phase, controls all system software on the target platform, including the OS. Based on the SGX model, the adversary can schedule enclaves and start multiple instances of the same enclave, offer the latest and previous versions of sealed data, and block, delay, read and modify all messages sent by the enclaves.

The adversary cannot read or modify the enclave runtime memory or learn any information about the secrets held in enclave data. The adversary has no access to processor-specific keys, such as the sealing key or the attestation key, and the adversary cannot break cryptographic primitives provided by the SGX architecture. The enclaves may also implement additional cryptographic operations that the adversary cannot break.

The adversarial capabilities that we identified as part of the model can be critical for many SGX systems. The ability to schedule, restart and create multiple enclave instances, enables subtle attacks that we address in this paper. We analyzed SGX systems using this model and found vulnerabilities that can be addressed through the techniques developed in this paper. These findings are reported in Appendix A.2.

3.2.3 Rollback Attacks

The goal of the adversary is to violate the integrity of the enclave's state. This is possible with a simple rollback attack. After an enclave has sealed at least two data elements $s_1 \leftarrow \texttt{Seal}(d_1)$ and $s_2 \leftarrow \texttt{Seal}(d_2)$, the adversary performs `Terminate()` and `Start()` to erase the runtime memory of the enclave. When the enclave requests for the latest sealed data d_2, the adversary performs $\texttt{OfferSeal}(i, s_1)$ and the enclave accepts d_1 as d_2. When the sealed data captures the state of the enclave at the time of sealing, we say that the rollback attack reverts the enclave back to its previous state.

Another approach is a *forking attack*, where the adversary leverages two concurrently running enclave instances. The adversary starts two instances $i_1 \leftarrow \texttt{Start}(e)$ and $i_2 \leftarrow \texttt{Start}(e)$ of the same enclave e. The OS receives a request from a remote client to write data m_e to enclave e. The OS writes the data to the first enclave instance $\texttt{Write}(m_e, i_1)$ which causes a state change. Another remote client sends a request to read data from the enclave e. The OS reads data from the second instance $m_e \leftarrow \texttt{Read}(i_2)$ which has an outdated state and returns m_e to the client. The SGX architecture does not enable one enclave instance to check if another instance of the same enclave code is already running [109].

Such attacks can have severe implications, especially for applications that maintain financial data, such as account balances or transaction histories.

3.2.4 Limitations of Known Solutions

SGX counters. Intel has recently added support for monotonic counters [103] as an optional SGX feature that an enclave developer may use for rollback attack protection, when available. However, the security and performance properties of this mechanism are not precisely documented. We performed a detailed analysis of SGX counters and report our findings in Appendix A.1.

To summarize, we found out that counter updates take 80-250 ms and counter reads take 60-140 ms. The non-volatile memory used to implement the counter wears out after approximately one million writes, making the counter functionality unusable after a couple of days of continuous use. Thus, SGX counters are unsuitable for frequent and continuous state update systems. Additionally, since the non-volatile memory used to store the counters resides outside the processor package, the mechanism is likely vulnerable to bus tapping and flash mirroring attacks [177] (see Appendix A.1 for details).

Integrity servers. Another approach is to leverage a trusted server to maintain state for protected applications [116, 121, 192]. The drawback of this approach is that the centralized integrity server becomes an obvious target

for attacks. To eliminate a single point of failure, the integrity server could be replicated using a Byzantine consensus mechanism. However, standard consensus protocols, such as PBFT [45], require several rounds of communication, have high message complexity, and require at least three replicas for each faulty node.

TPM solutions. TPMs provide monotonic counters and NVRAM that can be used to prevent rollback attacks [157, 180, 182]. The TPM counter interface is rate-limited (typically one increment every 5 seconds) to prevent memory wear out. Writing to NVRAM takes approximately 100 ms and the memory becomes unusable after 300K to 1.4M writes (few days of continuous use) [182]. Thus, also TPM based solutions are unsuitable for applications that require fast and continuous updates[1].

Architecture modifications. Finally, the SGX architecture could be modified such that the untrusted OS cannot erase the enclave runtime memory. However, this approach would prevent the OS from performing resource management and would not scale to many enclaves. Additionally, rollback attacks through forced reboots and multiple enclave instances would remain possible. Another approach would be to enhance the processor with a non-volatile memory element. Such changes are costly and current NVRAM technologies have the performance limitations we discussed above.

3.2.5 Rollback Protection Requirements

Our main goal is to design a rollback protection mechanism that overcomes the performance and security limitations of SGX counters and other known solutions. In particular, our solution should support unlimited and fast state updates, considering a strong adversary model without a single point of failure. When there is a trade-off between security and robustness, we favor security.

3.3 Our Approach

The intuition behind our approach is that a single SGX platform cannot efficiently prevent rollback attacks, but the owner or the owners of SGX platforms can enroll multiple processors to assist each other. Thus, our goal is to design rollback protection for SGX as a distributed system between multiple enclaves

[1]The TPM 2.0 specifications introduce *high-endurance non-volatile memory* that enables rapidly incremented counters [190]. The counter value is maintained in RAM and the value is flushed to non-volatile memory periodically (e.g., mod 100) and at controlled system shutdown. However, if the system is rebooted without calling TPM Shutdown, the counter value is lost and at start-up the TPM assumes the next periodic value. Therefore, such counters do not prevent attacks where the adversary reboots the system.

running on separate processors. Our distributed system is customized for the task of rollback protection to reduce the number of required replicas and communication.

To realize rollback protection, the distributed system should provide, for each participating platform, an abstraction of a *secure counter storage* that consists of two operations:

- `WriteCounter(`*value*`)`. An enclave can use this operation to write a counter value to the secure storage.[2]
- *value*/*empty* ← `ReadCounter()`. An enclave can use this operation to read a counter value from the secure storage. The operation returns the last written value or an empty value if no counter was previously written.

When an enclave performs a security-critical state update operation (e.g., modifies an account balance or extends a transaction history), it distributes a monotonically increasing counter value over the network to a set of enclaves running on assisting processors (`WriteCounter`), stores the counter value to its runtime memory and seals its state together with the counter value for local persistent storage. When the enclave is restarted, it can recover its latest state by unsealing the saved data, obtaining the counter values from enclaves on the assisting processors (`ReadCounter`) and verifying that the sealed state is of the latest version. The same technique allows potentially concurrently running instances of the same enclave identity to determine that they have the latest state. When an enclave needs to verify its state freshness (e.g., upon receiving a request to return the current account balance or transaction history to a remote client), it obtains the counter value from the network (`ReadCounter`) and compares it to the one in its runtime memory. By using enclaves on the assisting platforms, we reduce the required trust assumptions on these platforms.

3.3.1 Distributed Model

We use the term *target platform* to refer to the node which performs state updates that require rollback protection. We assume n SGX platforms that assist the target platform in rollback protection. The platforms can belong to a single administrative domain or they could be owned by private individuals who all benefit from collaborative rollback protection. We model each platform using the SGX model described in Section 3.2.1. The distributed system can be seen as a composition of $n+1$ SGX instances (target platform included) that

[2] We use counter *write* abstraction instead of counter *increment*, because our distributed secure storage implementation allows writing of any counter value to the storage. However, the ROTE system only performs monotonic counter increments using this functionality.

are connected over a network. We make no assumptions about the reliability of the communication network, messages may be delayed or lost completely. We assume that while participating in collaborative rollback protection, some platforms may be temporarily down or unreachable.

Distributed adversary model. On each platform, the adversary has the capabilities listed in Section 3.2.2. Additionally, we assume that the adversary can compromise the SGX protections on $f < n$ participating nodes, excluding the target platform. Such compromise is possible, e.g., through physical attacks. On the compromised SGX nodes the adversary can freely modify the runtime memory (code and data) of any enclave, and read all enclave secrets and the SGX processor keys.

This adversarial model combines a standard Dolev-Yao network adversary [58] with adversarial behaviour (Byzantine faults) on a subset of participating platforms [126, 159]. In addition, the adversary can schedule the execution of trusted processes, replay old versions of persistently stored data, and start multiple trusted process instances on the same platform. In Section 3.5 we explain subtle attacks enabled by such additional adversarial capabilities.

3.3.2 Challenges

Secure and practical realization of our approach under a strong adversarial model involves challenges.

Network partitioning. A simple solution would be to store a counter with all the assisting enclaves, and at the time of unsealing require that the counter value is obtained from all assisting enclaves. However, if one of the platforms is unreachable at the time of unsealing (e.g., due to network error, maintenance or reboot), the operation would fail. Our goal is to design a system that can proceed even if some of the participating enclaves are unreachable. In such a system, some of the assisting enclaves may have outdated counter values, and the system must ensure that only the latest counter value is ever recovered, assuming an adversary that can block messages, and partition the network by choosing which nodes are reachable at any given time.

Coordinated enclave restarts. When an enclave seals data, it sends a counter value to a set of enclaves running on assisting platforms and each enclave must store the received counter. However, sealing the received counter for persistent storage would cause a new state update that would propagate endlessly. Therefore, the enclaves must maintain the received counters in their runtime memory. The participating enclaves may be restarted at any time, which causes them to lose their runtime memory. Thus, the rollback protection system must provide a recovery mechanism that allows the assisting enclaves to restore the lost counters from the other assisting enclaves. Such a

recovery mechanism opens up a new attack vector. The adversary can launch coordinated attacks where he restarts assisting enclaves to trigger recovery while the target platform is distributing its current counter value.

Multiple enclave instances. Simple approaches that store a counter to a number of assisting enclaves and later read the counter from sufficiently many enclaves are vulnerable to attacks where the adversary creates multiple instances of the same enclave. Assume that a counter is saved to the runtime memory of all assisting enclaves. The adversary that controls the OS on all assisting platforms starts a second instance of the same enclave on all platforms. The target enclave updates its state and sends an incremented counter to these instances. Later, the target enclave obtains an old counter value from the first instances and recovers a previous state from the persistent storage.

3.3.3 Why the Problem is Different

Based on the problem described in the above sections, one could be easily convinced that the simplest solution would be to implement a BFT consensus protocol among the distributed group of protection nodes. However, the realistic setting is different. In BFT style consensus protocols, such as Practical BFT [45], the protocol presumes f faulty nodes that can be Byzantine (e.g., arbitrarily deviate from the protocol). In our model we presume a different adversary type (stronger adversary) that on the abstract level can affect the protocol execution in two different ways. First, our adversary controls the OS of *all* assisting nodes in the protection group and is, thus, able to block, delay and modify messages. Additionally, this gives the adversary a unique ability to terminate, restart and fork SGX enclaves. These capabilities cannot be captured by the BFT consensus protocols [60]. Second, we consider an adversary that can break the protections offered by SGX enclaves and fabricate authentic messages which is close to the model of byzantine nodes. As we will show through our system later on in this chapter, the combination of these adversarial actions requires a modified protocol architecture.

On the other hand, in Byzantine consensus protocols all honest nodes must *agree* on a value, whereas our goal is only that a target platform needs to be able to write an increasing monotonic counter value to a set of distributed nodes from the protection group for each state update and later on restore the latest counter representing the latest sealed state. In light of these requirements, a closer approach is to use the BFT write/read storage [78, 80, 123, 124]. In BFT storage protocols, there exists a target platform that writes or reads to set of processes running on distributed nodes, and the protocol can finish as long as a large enough quorum of available nodes is responsive (that depends both on the number of malicious and unavailable nodes). However, the models

under which these protocols operate are not able to capture the capability of our adversary to fork SGX enclaves. Namely, with forking the number of assisting nodes in the group would increase, while the model would still execute under minimal quorum bounds. This would effectively violate the safety property and enable the adversary to perform a rollback attack. To remedy this, we introduce an optimization in form of a second protocol round that eliminates the vulnerability. In the next section we explain the details of our protocol followed by the security analysis.

3.4 ROTE System

In this section we describe ROTE, a distributed system for state integrity and rollback protection on SGX. We explain the counter increment technique, our system architecture, group assignment and system initialization. After that, we describe the rollback protection protocols.

3.4.1 Counter Increment Technique

There exist two common techniques for counter-based rollback protection. The first technique is *inc-then-store*, where the enclave first increments the trusted counter and after that updates its internal state and stores the sealed state together with the counter value on disk. This approach provides a strong security property (no rollback to any previous state), but if the enclave crashes between the increment and store operations, the system cannot recover.

The second technique is *store-then-inc*, where the enclave first saves its state on the disk together with the latest input value, after that increments the trusted counter, and finally performs the state update [157, 182]. If the system crashes, it can recover from the previous state using the saved input. This technique requires a deterministic enclave and provides a slightly weaker security property: arbitrary rollback is not possible, but the last input may be executed twice on the same enclave state [182].

The stronger security guarantee is needed, for example, in enclaves that generate random numbers, communicate with external parties or create timestamps. Consider a financial enclave that receives a request message from an external party and for each request it should create only one signed response that is randomized or includes a timestamp (`sgx_get_trusted_time` [104]). If store-then-inc is used, the adversary can create multiple different signed responses for the same request.[3] This weaker security guarantee is

[3]While some enclaves that require random numbers can be made deterministic by using a stateful PRNG and including its state to the saved enclave state, this may be difficult for enclaves

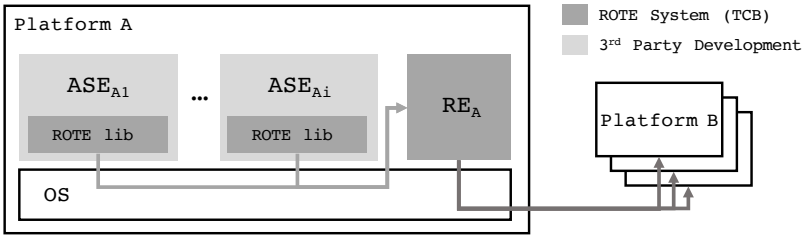

Figure 3.2: *The* ROTE *system architecture.*

sufficient in applications where the execution of the same input on the same state provides no advantage for the adversary.

We instantiate ROTE using inc-then-store because of its strong security guarantee for any enclave. Our goal is to build a generic platform service that can protect various applications. We emphasize that if crash tolerance is required, then store-then-inc should be used. A rollback protection system could even support both counter increment techniques and allow developers to choose the protection style based on their application.

3.4.2 System Architecture

Figure 3.2 shows our system architecture. Each platform may run multiple user applications that have a matching Application-Specific Enclave (ASE). The ROTE system consists of a system service that we call the Rollback Enclave (RE) and a ROTE library that ASEs can use for rollback protection.

When an ASE needs to update its state, it calls a counter increment function from the ROTE library. Once the RE returns a counter value, the ASE can safely update its state, save the counter value to its memory and seal any data together with the counter value. When an ASE needs to verify the freshness of its state, it can again call a function from the ROTE library to obtain the latest counter value to verify the freshness of unsealed seal data (or state in its runtime memory).

The RE maintains a Monotonic Counter (MC), increases it for every ASE update, distributes it to REs running on assisting platforms, and includes the counter value to its own sealed data. When the RE needs to verify the freshness of its own state, it obtains the latest counter value from the assisting nodes. The RE realizes the secure counter storage functionality (`WriteCounter` and `ReadCounter`) described in Section 3.3.

that reuse code from existing libraries not designed for this. Similarly, some replay issues can be addressed on the protocol level, but enclave developers do not always have the freedom to change (standardized) protocols.

The design choice of introducing a dedicated system service (RE) hides the distributed counter maintenance from the applications. Having a separate RE increases the TCB of our system slightly, but we consider easier application development more important.

The ROTE system has three configurable parameters:

- n is the number of assisting platforms,
- f is the maximum number of potentially compromised processors, and
- u is the maximum number of assisting platforms that can be unreachable or non-responsive at the time of state update or read for the system to proceed. Platform restarts are typically less frequent events and during them we require all the assisting platforms to be responsive.

These parameters have a dependency $n = f + 2u + 1$ (see Section 3.5). As an example, a system administrator can select the desired level of security f and robustness u which together determine the required number of assisting platforms n. Alternatively, given n assisting platforms, the administrator can pick f and u. Distinguishing and separating the bounds for liveness (u) and the number of malicious nodes (f) has been investigated in many works over the past 15 years [123]. In fact, our mentioned dependency can be encountered in [124], and as we will show in Section 3.5, it has already been proven as the minimal lower bound for asynchronous BFT storage protocols which we implement in a modified version for our setting.

To avoid *shared-fate* scenarios due to power outages or communication blockades, the participating platforms would ideally have independent or redundant power supply, battery backup, networking and OS maintenance.

3.4.3 System Initialization

Our system is agnostic to the way the n assisting SGX platforms are chosen. Here we explain an example approach based on a trusted offline authority. Such group assignment is practical when all assisting platforms belong to a single administrative domain (e.g., multiple servers in the same data center). We call the trusted authority, that selects the assisting nodes, the *group owner*. The group owner can be a fully offline entity to reduce its attack surface. To establish a *protection group*, the group owner selects n platforms.

In this section, we assume that the operating systems on these platforms are trusted at the time of system initialization (e.g., freshly installed OS). Note that although SGX supports remote attestation, this assumption is required, if the group needs to be established among *pre-defined* platforms. The SGX attestation is anonymous (or pseudonymous) and therefore it does not identify the attested platform. If the application scenario allows that the protection

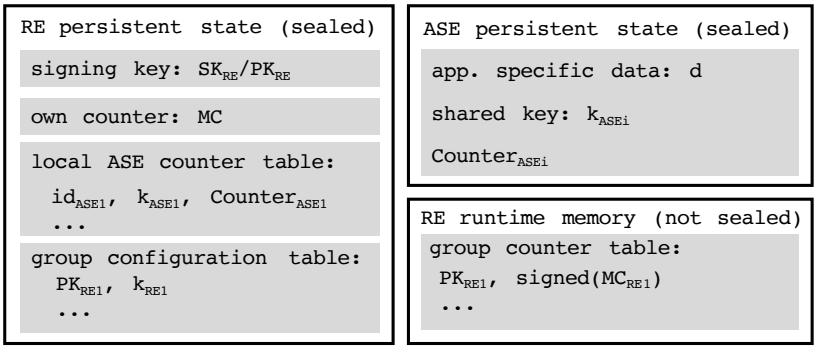

Figure 3.3: *The* ROTE *system state structures.*

group can be established among *any* SGX platforms, then system initialization is possible without initially trusted operating systems using remote attestation. We discuss such group setup alternatives in Section 3.4.7.

During its first execution, the RE on each platform generates an asymmetric key pair SK_{RE}/PK_{RE}, and exports the public key. The public keys are delivered to the group owner securely, and the owner issues a certificate by signing all group member keys. The group certificate can be verified by the RE on each selected platform by hard-coding the public key of the group owner to the RE.

The RE is started a second time with the certified list of public keys and a secret *initialization key* as input parameters. The purpose of this secret key for initialization is to indicate a legitimate group establishment operation and to prevent a later, parallel group creation by compromised operating systems on the same certified platforms (see Section 3.5). The initialization key is hard coded to the RE implementation in hashed format and the RE verifies the correctness of the provided key by hashing it. Without the correct key, the RE aborts initialization. The RE saves the list of certified public keys PK_{REi} to a *group configuration table* and runs an authenticated key agreement protocol to establish pair-wise session keys k_{REi} with all REs, and adds them to the group configuration table. Finally, the RE creates a monotonic counter (MC), sets it to zero, and seals its state.

When an ASE wants to use the ROTE system for the first time, it performs *local* attestation on the RE. The code measurement of the RE can be hard-coded to the ASE implementation or provisioned by the ASE developer. The ASE runs an authenticated key establishment protocol with the RE. The RE adds the established shared key k_{ASEi} to a *local ASE counter table* together with a locally unique enclave identifier id_{ASEi} and adds the same key to its own state. The used state structures are shown in Figure 3.3.

3.4.4 ASE State Update Protocol

When an ASE is ready to update its state (e.g., a financial application has received a new transaction and is ready to process it and update the maintained account balances), it starts the state update protocol shown in Figure 3.4. The first part of this protocol, i.e., first communication round, can be seen as a customized version of the asynchronous Single Writer Multiple Reader (SWMR) BFT write/read storage protocol [78, 80] (of the *write* part). We additionally implement ideas from the Echo broadcast [163], as discussed in Section 3.8. We respect the lower bounds for quorums defined by asynchronous BFT storage. The second part of this protocol, i.e., second communication round, is an added optimization which enables ROTE to operate under the defined SGX (see Section 3.2.1) and distributed (see Section 3.3.1) model, and prevents forking and intra-protocol restarts during execution.

The communication between the enclaves is encrypted and authenticated using the shared session keys in all of our protocols. We add nonces and end point identifiers to each message to prevent message replay. The protocol proceeds as follows:

(**1**) The ASE triggers a counter increment using the RE.

(**2**) The RE increments a counter for the ASE, increases its own MC, and signs the MC using SK_{RE}. The counter is signed to preserve its integrity in the case of compromised assisting REs.

(**3**) The RE sends the signed counter to all REs in the protection group.

(**4**) Upon receiving the signed MC, each RE updates its group counter table. The table is kept in the runtime memory, and not sealed after every update, to avoid endless propagation.

(**5**) The REs that received the counter send an *echo* message that contains the received signed MC. The REs also save the *echo* in runtime memory for later comparison.

(**6**) After receiving a quorum $q = u + f + 1 = \frac{n+f+1}{2}$ *echos*, the RE returns the *echos* to their senders.[4] The second round of communication is needed to prevent attacks based on RE restarts during the update protocol.

(**7**) Upon receiving back the *echo*, each RE finds the self-sent *echo* in its memory and checks if the MC value from it matches the one in the group

[4]It might seem that waiting for more than q responses, and therefore allowing more than q nodes to complete the protocol, would increase system robustness. However, the quorum is designed such that writing the latest counter to more than q nodes does not help the system to proceed in case of node unavailability or restarts (see Section 3.5).

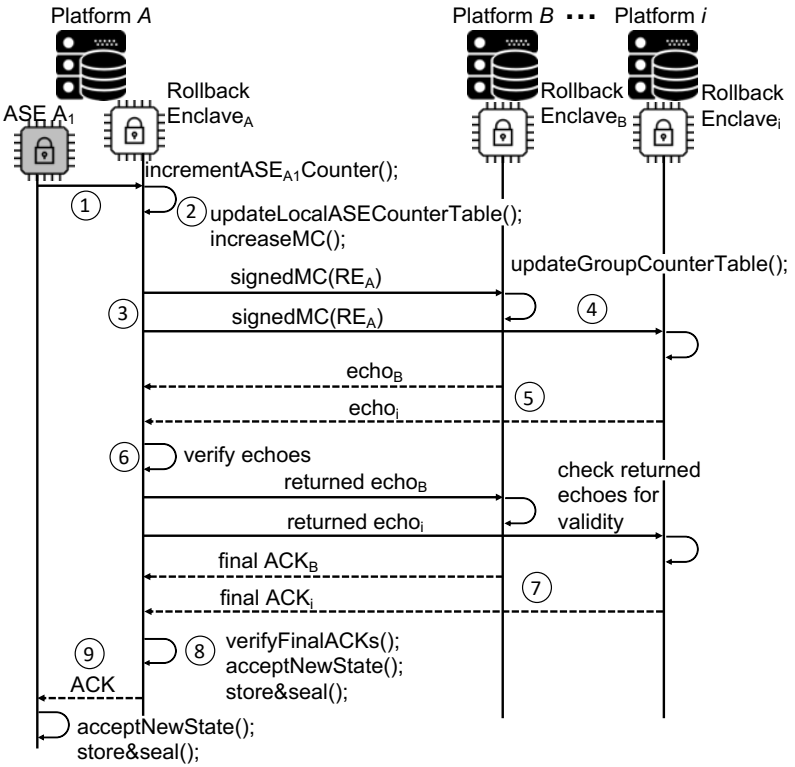

Figure 3.4: *The ASE state update protocol.*

counter table and the one received from the target RE. If this is the case, the RE replies with a final ACK message.

(**8**) After receiving q final ACKs, the RE seals its own state together with the MC value to the disk.

(**9**) The RE returns the incremented ASE counter value. The ASE can now safely perform the state update (e.g., update account balance), save the counter value to its runtime memory for later comparison, and seal its state with the counter.

3.4.5 RE Restart Protocol

Figure 3.5 shows the protocol that the RE runs after a restart. The goal of the protocol is to allow the RE to join the existing protection group, retrieve its

counter value and the MC values of the other nodes. Again, we can see this protocol as a modified asynchronous SWMR BFT write/read protocol (of the *read* part), where we presume optimal conditions (availability of all nodes during session key re-establishment, safety and liveness properties satisfied under the defined quorum) and have a single communication round in order to retrieve the counter value(s).

At restart the RE loses all previously established session keys and has to establish new session keys. In order to preserve our security guarantees, the target RE waits until it establishes new session keys with all other REs residing in the protection group. All assisting REs update their group configuration tables accordingly. The session key refresh mechanism prevents nodes from communicating with multiple RE instances on one platform (see Section 3.5). Another condition for successfully joining the protection group is that sufficiently many nodes return non-zero counter values (step 6 below). This check prevents simultaneously restarted REs from establishing a second, parallel protection group. This guarantee can be maintained when at most u nodes restart simultaneously. The protocol proceeds as follows:

(**1**) Session key establishment with other nodes and update of the group configuration table.

(**2**) The RE queries the OS for the sealed state.

(**3**) The RE unseals the state (if received) and extracts the MC.

(**4**) The RE sends a request to all other REs in the protection group to retrieve its MC.

(**5**) The assisting REs check their group counter table. If the MC is found, the enclaves reply with the signed MC. Additionally, the complete table of other signed MCs that the responding node has is sent to the target RE.

(**6**) When the RE receives q responses from the group (recall that $q = u+f+1$ and $q \geq n/2$), it selects the maximum value and verifies the signature. We select the maximum value because some REs might have an old counter value or they may have purposefully sent one. The target RE verifies signatures and compares all the group counter table entries with received values for other nodes. For each assisting RE, the target RE picks the highest MC and updates its own group counter table with the value. The RE also verifies that at least $f + 1$ of the received counter values are not zero to prevent creation of the parallel network. If the obtained counter value matches the one in the unsealed data, the unsealed state can be accepted.

(**7**) The RE stores and seals the updated state. The RE will also save the counter value to its runtime memory.

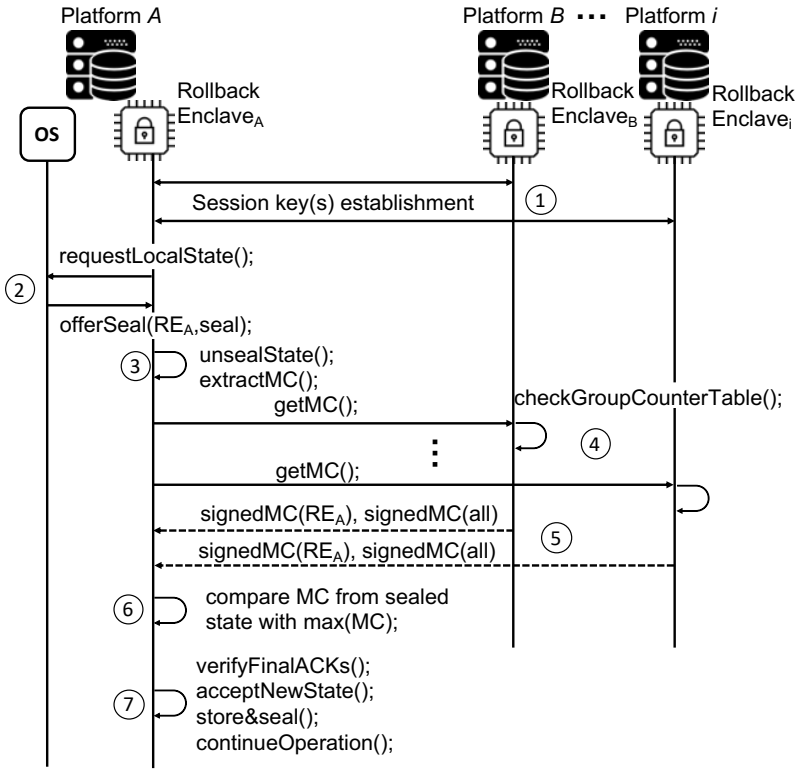

Figure 3.5: *The RE restart protocol.*

The Rollback Enclave now has an updated group counter table that reflects the latest counters for each node in the group.

3.4.6 ASE Start/Read Protocol

When an ASE needs to verify the freshness of its state, it performs the protocol shown in Figure 3.6. This is needed to verify the freshness of unsealed state after an ASE restart or when an ASE replies to a client request asking its current state (e.g., account balance). The ASE must verify that another ASE instance does not have a newer state. The protocol proceeds as follows:

(**1**) The ASE queries the OS for the sealed data representing its state.

(**2**) The ASE unseals the data (if it has received it from the OS) and obtains the counter value from it.

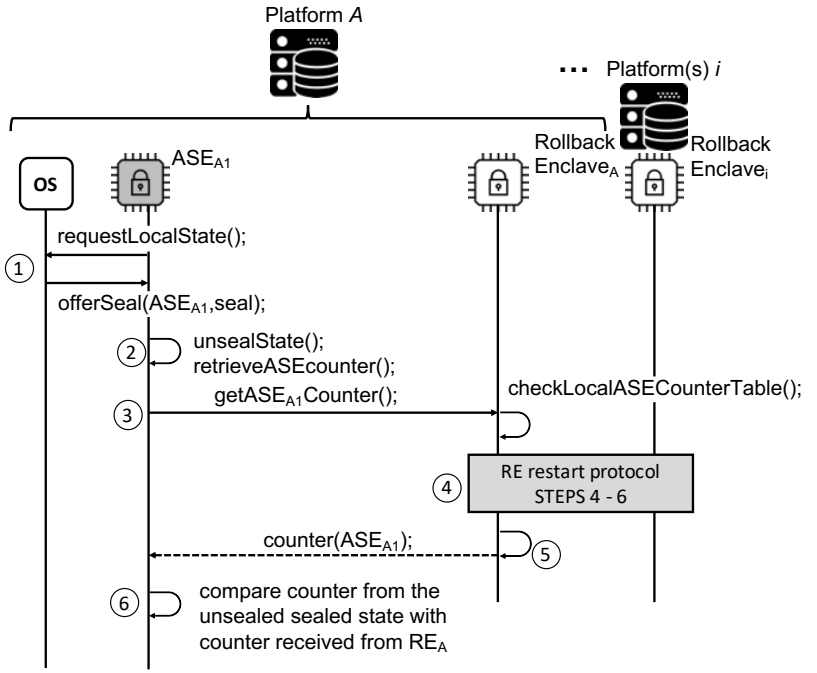

Figure 3.6: *The ASE start/read protocol.*

(**3**) The ASE requests the local RE to retrieve its latest ASE counter value.

(**4**) To verify the freshness of its runtime state, the RE performs the steps 4-6 from the RE Restart protocol, to obtain the latest MC from the network. This is needed to prevent forking attacks with multiple RE instances. If the obtained MC does not match the MC residing in the memory, the state of the RE is not the latest, so, the RE must abort and be restarted. This is an indication that another instance of the same RE was running and updated the state in the meantime. If the values match, the current data is fresh and the RE can continue normal operation.

(**5**) If all verification checks are successful, the RE returns a value from the local ASE counter table.

(**6**) The ASE compares the received counter value to the one obtained from the sealed data.

If the counters match, the Application-Specific Enclave loads the previously sealed state and completes a security-critical client request.

3.4.7 Group Management

The group owner issues a signed list of public parts of the public-private key pairs generated by each Rollback Enclave that define the protection group. Assume that later one or more processors in the group are found compromised or need replacement. The group owner should be able to update the previously established group (i.e., exclude or add new nodes) without interrupting the system operation.

Group updates. During system initialization, the RE verifies the signed list of group member keys and seals the group configuration. When a group update is needed, the group owner issues an updated list that will be processed and sealed by the RE. This approach does not require the entry of the secret initialization key such as in first group establishment. However, the adversary should not be able to revert the group to its previous configuration (e.g., one including compromised nodes) by re-playing the previous group configuration. Since group updates are typically infrequent, they can be protected using SGX counter service or TPM counters.

At system initialization, the RE creates a monotonic counter using SGX counter service or on a local TPM. If this is done using TPM, establishing a shared secret with the TPM (see session authorization in [190]) is necessary. The group owner includes a version number to every issued group configuration. When the RE processes the signed list, it increments the SGX or TPM counter to match the group version, and includes the version number in the sealed data. For every group update, the RE increments either of these counters. When the RE is restarted, it verifies that the version number in the unsealed group configuration matches the counter. The NVRAM memory in TPMs is expected to support approximately 100K write cycles, while with SGX counters support approximately 1M cycles, sufficient for most group management needs. For example, if group updates are issued once a week, the NVRAM would last 2000 years using TPMs and 20000 years using SGX counters.

Group setup with attestation. In Section 3.4.3 we described group setup for pre-defined platforms. The drawback of this approach is that it requires trusted operating systems at initialization. If the application scenario allows group establishment among *any* SGX platforms, similar trust assumption is not needed. The group owner can attest $n+1$ group members using the attestation mode that returns a pseudonym for each attested platform, establish secure channels to all group members, and distribute keys that group members use to authenticate each other. Because each platform reports a different pseudonym,

this process guarantees that the protection group consists of $n + 1$ separate platforms in contrast to multiple instances on one compromised CPU.

3.5 Security Analysis

Our system is designed to provide the following security property: an ASE cannot be rolled back to a previous state. In Section 3.5.1 we first show that given a secure storage functionality, as defined in Section 3.3, an RE can verify that its state is the latest. After that, in Section 3.5.2, we show that the participating REs realize the secure counter storage as a distributed system. Finally, by putting these two together, we show that ASEs cannot be rolled back if the RE cannot be rolled back.

Our system achieves a security guarantee that we call *all-or-nothing rollback*. The only way to violate enclave data integrity is to reset all nodes and bring the entire group to its initial state. In many applications such integrity violation is easily detectable, and we do not consider it as an attack on ROTE.

In the event of crashes, restarts or node unavailability, the system may fail to proceed temporarily or permanently. We distinguish three such cases: *Halt-1* where the system may be able to proceed automatically by simply trying again later (e.g., temporary network issue); *Halt-2* where manual intervention from the system administrator is needed (e.g., faulty node that needs to be fixed); and *Halt-X* where the complete system has to be re-initialized and the latest state of enclaves will be lost (e.g., simultaneous crash of all nodes). Recall that as the adversary controls the OS on all nodes, denial of service is always possible.

3.5.1 Protection with Secure Storage

Given the secure counter storage functionality (see Section 3.3) rollback can be prevented using the inc-then-store technique. In Figure 3.7 we illustrate a state transition diagram that represents RE states during sealing, unsealing and memory reading using the secure storage functionality. The notion of state in this section is an *execution state*, in contrast to enclave *data states* created and stored using sealing. We show that any combination of adversary operations, in any of the enclave execution states, cannot force the RE to accept a previous version of sealed data. We also show that in spite of multiple local RE instances, the read enclave state is always the latest. Note that this state transition diagram does not capture system initialization.

First start. The RE execution begins from State 1, after creating and starting the enclave using $e \leftarrow$ `Create(code)` and $i \leftarrow$ `Start(e)`. The MC is set to zero in the runtime memory and RE proceeds to State 2. The RE

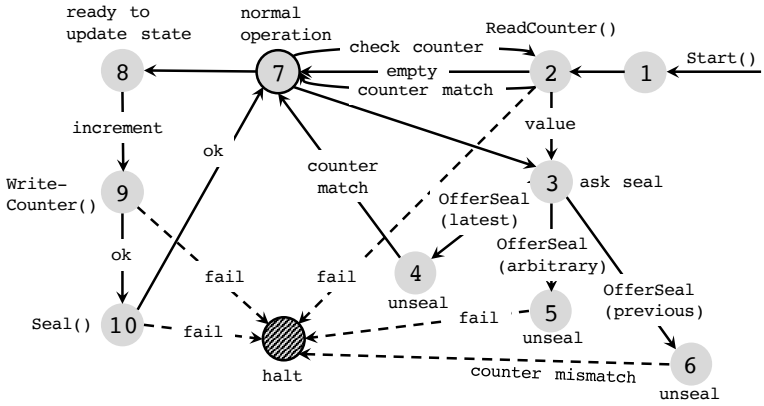

Figure 3.7: *Transition diagram showing enclave* execution states *using an ideal secure counter storage functionality.*

reads the counter value from the secure storage using `ReadCounter()`. If the `ReadCounter()` operation fails, the RE halts (Halt-1). On the first execution the operation returns *empty* and the RE continues to State 7 to continue normal operation. From State 7 the execution moves to State 2 for verifying freshness if a `Read()` request is received, while the `Write()` request moves execution to State 8.

Sealing. When the RE needs to seal data for local persistent storage, it proceeds to State 8. The RE increments the MC, and performs the write operation (`WriteCounter()`) to the secure storage in State 9. The RE continues to State 10 if the operation succeeds, otherwise it halts (Halt-1). In State 10, the RE seals data ($s \leftarrow$ `Seal(data)`) of its current state along with the counter value. OS confirmation moves the enclave to normal operation in State 7. If sealing fails, the node can try again (Halt-1). If that does not help, the node loses its latest state and becomes unavailable, and a group update is needed (Halt-2).

Unsealing. When the RE needs to unseal data (recover its state), the RE proceeds from State 7 to State 3. The adversary can offer the correct sealed data (`OfferSeal` ($latest \equiv s$)) which moves the execution to State 4. Unsealing is successful and the counter value in the seal matches the MC value in the runtime memory, bringing the RE back to State 7. The adversary can offer a previously sealed state (`OfferSeal` (*previous*)) which moves the execution to State 6. Unsealing is successful, but counter values do not

match and the RE halts (Halt-1 or Halt-2).[5] Finally, the adversary can offer any other data (`OfferSeal(arbitrary)`) which moves the RE to State 5 where unsealing fails and RE halts (Halt-1 or Halt-2).

Forking. If a new instance of the RE is started, the execution for it moves to State 1 following *First start*. Other instances remain in their original states. If for every `Write()` and `Read()` operation a counter is incremented or respectively retrieved from the secure counter storage to verify freshness, no rollback is possible. When the RE needs to read its runtime state (e.g., to complete a client request), the RE proceeds from State 7 to State 2. The RE reads the MC from the secure counter storage (if this fails, Halt-1) and compares the value to the one residing in its memory. This check is needed to guarantee that another instance of the same enclave does not have a newer state. If comparison succeeds, RE has the latest internal memory state and proceeds back to State 7. If the comparison fails (retrieved MC is higher), the RE moves to State 3 to obtain the latest seal (see above).

Restart. After an RE restart, the execution proceeds to State 2. If the `ReadCounter()` operation returns a non-empty value, the RE proceeds to State 3, otherwise to State 7, from where we follow the same steps as above. If the counter read operation fails, RE enters Halt-1.

If in any of these states the RE is terminated or restarted, its execution continues from State 1. Deleting and creating the same enclave again has the same effect. `Suspend()` and `Resume()` have no effect, i.e., the enclave remains in the same execution state. We conclude that, assuming the secure storage functionality, the adversary cannot rollback the state of the RE.

3.5.2 Distributed Secure Storage Realization

Next, we show how ROTE realizes the secure counter storage functionality as a distributed system. When obtaining a counter from the distributed protection group (`ReadCounter`), RE receives the latest value that was sent to the protection group (`WriteCounter`). We divide the analysis into six parts: atomicity and persistence, lower bound, quorum size, platform resets, two-phase counter writing, and forking attacks.

Persistence and Atomicity. Lamport [123] explains the nature of asynchronous communication by the fact that all communication ultimately involves a medium whose state is changed by a set of writers and observed by a set of readers. Additionally, there are two kinds of communication acts, namely, transient and persistent. In the transient one, the communication

[5] If the OS provides an incorrect sealed data, most likely it is faulty an needs to be fixed. From some OS errors it may be possible to recover by simply trying again.

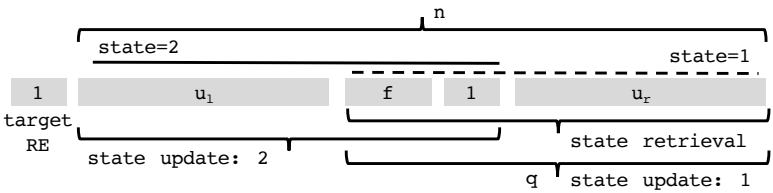

Figure 3.8: *Network partitioning example where the adversary intentionally blocks a part of the nodes.*

medium is changed only during the writing period and the observer can only notice it during the operation while after execution the state is reverted to the *normal* one. In the setting of asynchronous message exchange, as is the case in ROTE, persistent communication is required, where the writer changes the state of the medium and the reader is able to sense the change at a later point in time. Our setting can be seen as set of value registers available at every node of the protection group that only have a single writer (the target platform for the nodes are storing the monotonic counter). However, since that value is read both by the target platform and other nodes upon restart, it can be seen as a shared register.

For the single writer case we must observe the cases where writes and reads can overlap concurrently to determine the properties of our register. As Lamport continues to explain, there exist three distinct possibilities [123]. First, in a *safe* register, only a read that is not concurrent with a write can obtain the correct value (the latest one) and no assumptions are made on the observed value in case of concurrency. Second, in a *regular* register, which has the property of being *safe*, the observed value of the register when a read occurs concurrent to a write can be either the old or the new value in respect to the write operation. Finally, an *atomic* register is *safe* and all reads and writes behave as if they would occur in some definite order.

ROTE implements a message buffer, where the operations are ordered based on the arrival time and are executed sequentially, and not in parallel (we do not support any multi-threading). If we would observe a single target platform with the protection group where other nodes are only used to store the counter of the target platform and otherwise do not engage in the rollback protection of their own state, our system could be seen as SWMR BFT with Atomic storage. Since all the nodes from the protection group execute ROTE and enjoy rollback protection, the system can be observed as SWMR BFT with Regular semantics.

Lower bound. Due to the specific properties of our SGX and adversary model, our system distinguishes unresponsive and malicious nodes. The defined dependency of our system, $n = f + 2u + 1$, has been also encountered as the lower bound for asynchronous consensus [124]. Guerraoui et al. [78] reports that it is also well known how this dependency is a lower bound for resilience in any safe BFT storage protocol instantiated as the asynchronous system [143]. Even though the optimal resilience lower bound is proven for the special case where $f = u$, it is not difficult to extend this result for $f \neq u$ using the same techniques. Moreover, the optimality of these bounds under the SWMR storage has been researched and proven in several different works [42, 60, 79]. As Guerraoui et al. [80] presents with its refined quorums systems in respect to both storage and consensus, bounds for both the BFT write/read storage and BFT consensus tend to be basically the same. In this Chapter we do not go into detailed elaboration of these proofs, but use the results since they effectively apply to the design of our protocol.

Quorum size. The ROTE system has three parameters: the number of assisting nodes n, compromised nodes f, and unresponsive nodes u. The required quorum for responses at the time of counter writing and reading is $q = f + u + 1 = \frac{n+f+1}{2}$.

Figure 3.8 illustrates the optimal quorum size for our setting in a simplified example. We consider an example where the adversary performs network partitioning by blocking messages during writing and reading. On the first write, the attacker allows the counter value 1 to reach the right side of the group by blocking the messages sent to the left side. On the second write, the adversary allows the counter value 2 to reach the left side of the group by blocking the right side. Finally, on counter read, the adversary blocks the left side again. If the counter is successfully written to $q = f + u + 1$ nodes, there always exists at least $u + 1$ honest platforms in the group that have the latest counter value in the memory. Because counter reading requires the same number of responses, at least one correct counter value is obtained upon reading. The maximum number of tolerated compromised platforms is $f = n - 1$, if $u = 0$ and $q = n$. If the quorum cannot be satisfied in either the state update protocol or any counter retrieval, the system enters Halt-1 and can try to perform the same operation again. Recall that our protocol implements a modified version of the SWMR asynchronous storage system to provide safety and liveness. Our assumptions are lowered when it comes to long-term liveness, since we expect that all protocol runs will eventually be executed.

Platform restarts. If an assisting RE is restarted, it needs to first establish session keys and then recover the lost MC values from the protection group.

Session key establishment procedure is explained below under *Forking attacks*; the main take-away is that up to u nodes may restart simultaneously and after the nodes are online again the RE needs to establish session keys with *every* node in the group before proceeding with MC recovery.[6] Once the keys are established, some assisting nodes can be inactive or restarted. Three distinct cases are possible. First, the number of inactive/restarted REs is at most u. Since the number of running nodes is $u+f+1 = q$ there are sufficient available platforms with the correct MC for the counter retrieval. Second, more than u platforms, but not the entire protection group, are restarted. The number of remaining platforms is insufficient for RE recovery and the distributed system no longer provides successful MC access, but no rollback is possible (Halt-X, since there is no guarantee that the non-restarted nodes have the latest counter, thereby risking a rollback. However, before re-initializing the system, the latest states from the non-restarted nodes can be manually saved.) Third, all $n+1$ nodes are restarted at the same time, in which case a new system configuration has to be deployed again by the group owner to re-initialize the system (Halt-X).

Two-round counter writing. Additionally, it remains to be shown how our update protocol successfully writes the counter to q nodes, despite possible RE restarts *during* the protocol. We illustrate the challenges of counter writing through an example attack on a single-round variant of the update protocol that completes after the RE has received q echoes. During state update the adversary blocks all communication and performs sequential message passing. First, the attacker allows message delivery to only one node that saves the counter and returns an echo. After that, the attacker restarts the RE on that node, which initiates the recovery procedure from the rest of the protection group. The adversary blocks the communication to the target platform, and the restarted RE recovers the previous counter value, because other reachable REs have not yet received the new value. The adversary repeats the same process for all platforms. As a result, the target node has received q echos and accepts the state update, but all the assisting nodes have the previous counter value. Rollback is possible.

The second communication round in our protocol prevents such attacks. No combination of RE restarts during the state update protocol allows the target RE to complete it, unless the counter was written to q nodes. There are four distinct cases to consider. Below, we assume that the adversary restarts

[6] Consider an example, where two nodes are restarted at the same time. The first node wakes up and attempts to establish new session keys with all assisting nodes. This node has to wait, until the second restarted node wakes up and can communicate. After this point, both of the restarted nodes can establish session keys (with all nodes) and proceed with the RE Restart protocol.

at most u platforms simultaneously. If more are restarted, recovery is not possible (Halt-X).

• *Case 1: Echo blocking.* If the attacker blocks communication or restarts assisting REs so that q nodes cannot send the echo, the protocol does not complete (Halt-1).

• *Case 2: No echo blocking.* If the attacker allows at least q echoes to pass, RE starts returning them and we have two cases to observe:

• *Case 2a: No restarts during first round.* If none of the assisting REs were restarted during the first protocol round, then at least $u + 1$ nodes have the updated MC. If the adversary restarts assisting REs before they sent the final ACK and after they received the self-sent *echo* back from the target RE, the protocol will not complete (Halt-1), because fewer than q final ACKs will be received. The protocol run may be repeated again. The adversary can also restart assisting REs after they have sent the final ACK which will result in successful state update, and successful state recovery of the restarted REs since a sufficient number of assisting nodes already has the updated counter.

• *Case 2b: Restarts during first round.* If the adversary restarts assisting REs during the first round, the update protocol will either successfully complete (q final ACKs received) or halt execution (Halt-1) depending on the number of simultaneously restarted nodes. Sequential node restarts, as discussed in the example attack above, are detected. Upon receiving q echoes, the RE sends each of the received echoes to the original sender. Because of sequential RE restarts, all assisting nodes have the previous MC value in their runtime memory, and thus the protocol will fail upon comparison of the echoes and the MC values. None of the assisting REs will deliver the final ACK, and the protocol will not complete (Halt-1).

We conclude that the successful completion of the two-phase state update protocol guarantees that at least q nodes received and at least $u + 1$ honest nodes have (i.e., correctly stored) the correct MC.

Forking attacks. Our system prevents attacks based on multiple enclave instances by requiring that the ASE start/read and RE restart protocols contact the assisting nodes and verify the latest counter from the protection group. If the latest counter is correct, RE can be certain that it made the last update. If the session's keys are outdated, communication with other nodes is disabled and RE knows another instance has run in parallel.

The session key refresh mechanism allows us to uniquely identify the latest running instance and prevents parallel communication with two instances running on one platform. After every RE start, keys have to be established *with*

all nodes from the protection group to prevent the attacker from instantiating new REs on different platforms in a one-by-one manner while keeping some of the nodes disconnected. Other nodes delete the old session key that they shared with the previous instance residing on the same platform, rendering its communication unusable. The protection group only allows keys for one running instance on each platform. Also, by forcing state retrieval and freshness verification after each instantiation and for all ASE requests, the running instance on each platform will always have the latest state and highest MC, thus preventing rollback.

Our system also ensures that the adversary cannot establish a parallel protection group on the same platforms and re-direct ASEs to the rogue system causing a rollback. If no initialization key is provided and the RE receives all zero MC values from others in the group during setup, it will abort execution. A new network may only be created under the supervision of the group owner with the correct initialization key.

Summary. If the target RE has the latest MC that it sent, it is able to distinguish its latest sealed state, and if the latest sealed state is loaded, all the ASEs state counters kept within are fresh. Upon retrieval, the ASE always receives the latest counter, and thus each ASEs can verify that it has the latest state data. If the target RE is not able to recover the latest MC, the system end ups in either Halt-1, Halt-2 or Halt-X.

3.6 Performance Analysis

In this section we describe our performance evaluation. First, we describe our implementation that consists of the following components. We implemented the RE (950 LoC), an accompanying *rollback relay* application (1600 LoC), ROTE library (150 LoC), a simple test ASE (100 LoC), and a matching *test relay* application (100 LoC). The purpose of the relays is to mediate enclave-to-enclave communication. We implemented all components in C++, the relays were implemented for the Windows platform. The local communication between the relay applications was implemented using Windows named pipes. The total TCB accounts for 1100 LoC.

The enclaves use asymmetric cryptography for signing (ECDSA) and encryption (256-bit ECC). Our implementation establishes shared keys using authenticated Diffie-Hellman key exchange. For symmetric message encryption and authentication we use 128-bit AES-GCM in encrypt-then-MAC mode. Standard Intel SGX libraries provide all of the used crypto primitives.

3.6.1 State Update and Read Delay

The main performance metrics that we measure are the ASE state update and state read delays that include the counter writing to and reading from the protection group. The delay depends on the network characteristics and the size of the protection group $(n + 1)$. The RE restart operation is typically performed once per platform boot, and thus the operation is not similarly time-critical so we do not measure it. In all test cases we set $u = f = 0$, as their values do not affect state update and read delays.[7]

Experimental setup. Our first experimental setup consisted of four SGX laptops and our second experimental setup consisted of 20 identical desktop computers, both connected via local network (1Gbps, ping $\leq 1ms$). Our third experimental setup was a geographically distributed (in order, US (West), Europe, Asia, S. America, Australia, US (East)) protection group of sizes from two to six nodes that we tested on Amazon AWS EC2. For the first setup we used the real ROTE implementation while the latter two we used a simulated implementation (the same protocol, but no enclaves).

Results. The state update delay consists of two components: networking and processing overhead. Context switching to enclave execution is fast (few microseconds). Symmetric encryption used in the protocol is also fast (less than a microsecond). The only computationally expensive operation that we use is asymmetric signatures (0.46 ms per signing operation). We provided more SGX performance benchmarks in Section 2.3.7.

The ASE state update protocol has one signature creation which is verified later in the RE and ASE start/read protocols. The required processing time of the state update protocol is less than 0.6 ms, where the creation of the first protocol message takes 0.51 ms (signing). The state read protocol requires one round trip, while the state update protocol needs two. All messages passed between the nodes are 224 bytes (200 payload + 24 header).

Figures 3.9, 3.10, 3.11 show the results from our three experimental setups. The first figure shows ROTE performance for protection groups that are connected over a local network, the second figure shows the simulated performance for a larger group also over a local network, while the third figure is for geographically distributed protection groups. Figure 3.9 illustrates that the state update delay was approximately 2 ms, while the state read delay was approximately 1.3 ms for group sizes from two to four nodes using the ROTE implementation. Figure 3.10 illustrates an increase in the delay as

[7] The state update protocol proceeds immediately after receiving q responses, and therefore node unavailability does not affect update delay. Similarly, up to f compromised nodes can discard counter values or return fake values, but that does not affect the protocol delay.

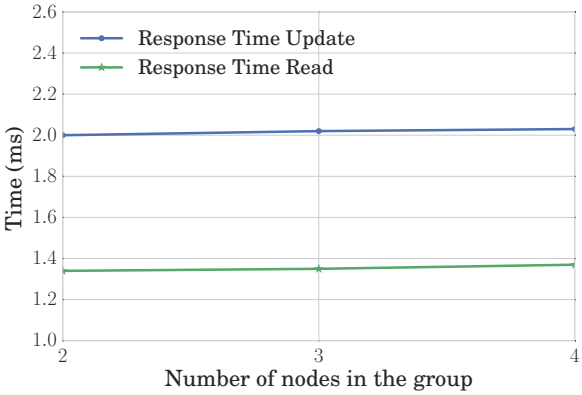

Figure 3.9: *1st exp. setup (*ROTE *implementation)*

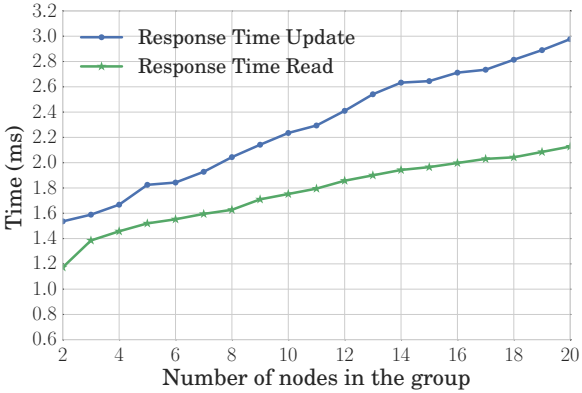

Figure 3.10: *2nd exp. setup (local group, simulated)*

the group size grows. This is as expected, since the target platform needs to communicate with more platforms. For group size of 20 nodes, the delay is 2.98 ms and 2.13 ms, respectively. Lastly, Figure 3.11 illustrates a less systematic increase in delay, due to the dependency on network connections between various geographic locations in the protection group. The update time between two locations takes 654 ms while between five the update time is 1.37 seconds. The read delay is respectively 342 ms and 810 ms.

We draw two conclusions from these experiments. First, the performance overhead imposed by ROTE is defined largely by the network connections

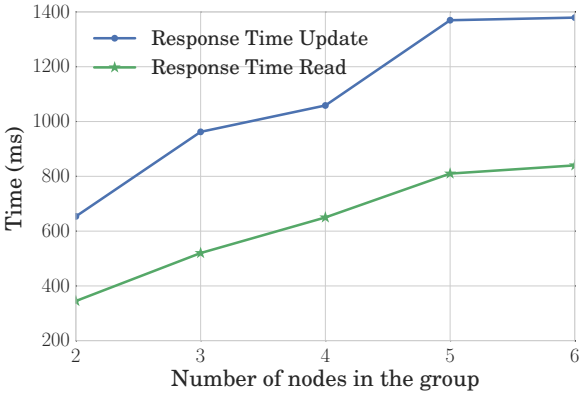

Figure 3.11: *3rd exp. setup (global group, simulated)*

Request type	**State size** (KB)	**no rollback protection** (ms)	ROTE **system** (ms)	**SGX counter protection** (ms)
Write state	1	3.85 (± 0.06)	5.17 (± 0.03)	160.7 (± 0.7)
	10	4.65 (± 0.05)	6.03 (± 0.03)	162.7 (± 1.6)
	100	6.49 (± 0.04)	7.83 (± 0.05)	169.1 (± 2.1)
Read state	1	0.06 (± 0.00)	1.41 (± 0.02)	61.04 (± 3.1)
	10	0.19 (± 0.00)	1.53 (± 0.01)	61.17 (± 3.1)
	100	1.76 (± 0.05)	3.1 (± 0.02)	62.74 (± 3.2)

Table 3.1: *Example application throughput without rollback protection, using* ROTE *and using SGX counters.*

between the nodes. Second, if the nodes are connected over a low-delay network, ROTE supports applications requiring very fast state updates (1-2 ms). For applications tolerating larger delays (e.g., more than 600 ms per state update), ROTE can be run on geographically distant groups.

3.6.2 Example Application Throughput

Additionally, we measured the throughput of an example financial enclave that processes incoming transactions repeatedly (the transaction buffer is never empty). We tested the enclave using (a) no rollback protection, (b) the ROTE implementation, and (c) SGX counter based rollback protection. The experimental setup was a protection group of four nodes. For every update

transaction, the enclave updates its state, creates a new seal, and writes it to the disk, while the read transaction includes reading from the disk, unsealing and retrieving the counter for comparison. In case of ROTE and SGX counter variants, the enclave also performs a counter increment. We tested three different enclave state sizes (1 KB, 10 KB, 100 KB) since the state size for transactions can differ based on the exact use case.

Results. Table 3.1 shows our results. In all three cases the ROTE system provides significantly better state update performance than using SGX counters (e.g., 190 over 6 tx/s for 1KB) while suffering a 20-25% performance drop in comparison to systems which have no rollback protection (e.g., 260 over 190 tx/s for 1KB). We conclude that our system provides significantly faster rollback protection than methods based on local non-volatile memory. Compared to systems with no rollback protection, our solution imposes a moderate overhead.

3.7 Discussion

Data migration. Although sealing binds encrypted enclave data to a specific processor, our solution enables data migration within the protection group. Migration is especially useful before planned hardware replacements and group updates (e.g., node removal). In a migration operation, an ASE first unseals its persistent data and passes it to the RE. The RE sends the enclave data to another Rollback Enclave within the same protection group together with the measurement of the ASE. The communication channel between the REs is encrypted and authenticated. On the receiving processor, the RE passes the enclave data to an instance of the same ASE (based on attestation using the received measurement) which can seal it. Note that the RE is agnostic to the internal state of ASEs and just re-encrypts data it receives from an ASE without the need to understand its semantics. Combined with group updates (Section 3.4.7), such enclave data migration enables flexible management of available computing resources. Similar data migration is discussed in [181].

Information leakage. Our model excludes execution side-channels. Here we briefly discuss additional information leakage that our solution may add. Each enclave state update and read causes network communication. An adversary that can observe the network, but does not have access to the local persistent storage, can use the information leakage to determine the timing of sealing and unsealing events. Also the reboot of the target platform causes an observable network pattern. We consider such information leakage a practical concern but developing countermeasures is outside the scope of this paper.

Performance. The main performance characteristic of our solution, the state update delay, is dominated by the networking and the asymmetric signature operation required for the first message of the state update protocol. In case of a local, 1Gpbs, network and an average laptop, the networking takes approximately 1 ms and the signature operation 0.5 ms. A possible optimization is to pre-compute the asymmetric signatures. Since the signed data is predictable MC values, we can pre-compute and store them. This pre-computation may be done at times when the expected load is low or at system initialization depending on the specific scenario.

For communication between the enclaves we use symmetric keys derived from the key agreement protocol for performance reasons, since it is computationally much less expensive. However, depending on the application scenario we could use asymmetric keys which would enable, for example, post-incident forensics. This design choice is dependent on the use case and performance requirements. ROTE can accommodate both approaches.

Consensus applications. In the specific case where all participating enclaves implement a distributed application with the purpose to maintain a consensus (e.g., permissioned blockchain), our rollback protection can be optimized further. In such an application, all participating enclaves have a shared, global state and the state update protocol can be replaced with a suitable Byzantine agreement protocol. When an enclave is restarted (or determines its latest state), it queries its latest state from the participating enclaves similar to our RE restart protocol. We leave a detailed design as future work.

Forking prevention. The current SGX architecture does not provide the ability for one enclave instance to check if another instance of the same enclave is already running. The implementation of this feature would simplify rollback protection significantly.

Forking prevention could be implemented using a TPM. After system boot, the RE instance could extend a PCR that has a known value at boot. If a second RE instance is started, it can check if the PCR value differs from its known initial value [182]. The drawback of this approach is the increase of the system security perimeter outside of the processor.

Periodic check-pointing. For increased robustness, our rollback protection can be complemented with periodic check-pointing. An example approach would be to increment a counter on local NVRAM on selected updates (e.g., mod 100). If all nodes crash at the same time, the administrator has an option to recover from the latest saved checkpoint with the risk of possible rollback.

3.8 Related work

SGX-counter and TPM solutions. Ariadne [182] uses TPM NVRAM or SGX counters for enclave rollback protection. The counter is incremented using store-then-inc that provides crash resilience, but allows two executions of the latest input. Ariadne minimizes the TPM NVRAM wear using counter increments that flip only a single bit. Compared to our solution, SGX counters are an optional feature, increments are slow and make the non-volatile memory unusable after few days of continuous use. Similar performance limitations apply to TPM NVRAM. SGX counters are also likely vulnerable to bus tapping and flash mirroring attacks [177], while in our solution the trust perimeter is the processor package.

Memoir [157] also leverages TPM NVRAM for rollback protection, and therefore has similar performance limitations. An optimized variant of Memoir assumes the availability of an Uninterrupted Power Supply (UPS). This variant stores the state updates to volatile Platform Configuration Registers (PCRs) and at system shutdown writes the recorded update history to the NVRAM.

ICE [180] enhances the CPU with protected volatile memory, a power supply and a capacitor that at system shutdown the flushes the latest state to non-volatile memory. Both the optimized Memoir and ICE require hardware changes. Additionally, reliably flushing data upon a crash or power outage can be challenging in practice.

Client-side detection. Brandenburger [38] proposes client-side rollback detection for SGX in the context of cloud computing. The main difference to our work is that this approach does not prevent a rollback directly on the server. Instead, it allows mutually trusting clients to remain synchronized, and given that certain connectivity requirements are met, detect consistency and integrity violations (including rollback) after the incident.

Integrity servers. Verena [116] maintains authenticated data structures for web applications and stores integrity information on a separate, trusted server. Another use case is to prevent the usage of disabled credentials on mobile devices by storing counters on an integrity-protected server [121]. In such solutions the integrity server becomes a single point of failure.

Byzantine broadcast and agreement. Our state update protocol follows the approach of Echo broadcast [163] with an additional confirmation message in the end. Like other byzantine broadcast primitives, our state update protocol requires $O(n)$ messages. Byzantine agreement typically require $O(n^2)$ messages. Byzantine broadcast and agreement protocol operate on arbitrary values and assume a potentially malicious sender. Recall that such protocols require $3f + 1$ replicas, and offer different properties and assumptions not in

line with our requirements. Regardless of that, they represent an important concept in the distributed systems research and it is fair to compare them in relation to ROTE.

Byzantine storage. The research area of shared storage with unreliable execution has been studied through multiple approaches over the years. For example, in the beginning, tolerating server crash failures was the main focus [28], while later on studying the behavior of such storage considered arbitrary server failures [110, 125]. In relation to our setting, Martin et al. [143] first proved the lower bound for any safe storage in case where the number of malicious and non-responsive nodes are not distinguished, $f = u$. As pointed out by Guerraoui et al. [78], the proof is easily extended to the case where $f \neq u$. Additionally, [78, 79] describe a fast (lucky) SWMR wait-free atomic register that is the closest to our work in this chapter. They describe an optimistic implementation that is able to complete both the writes and reads in one protocol round under some synchrony assumptions during the protocol execution. In [42], Cachin et al. introduces a BFT storage protocol that distinguishes metadata from the actual data payload and is able to reduce the regular number of replicas to only $2f + 1$ for the actual payload. Our case only involves storing of the monotonic counter as the payload, thus separating it from metadata is meaningless. Other relevant referent work includes Phalanx [142], a secure and scalable replication system that builds a persistent survivable data repository for shared data, Efficient BFT erasure-coded storage [70], an implementation of a wait-free Multiple Writer Multiple Reader (MWMR) atomic storage, and Minimal Byzantine Storage [143], a proposal for the MWMR optimally resilient atomic storage.

Secure audit logs. Secure audit log systems [57, 138, 168, 174] provide accountability and in particular prevent manipulation of previous log entries *after* the target platform becomes compromised. Most such audit log systems assume a trusted but infrequently accessible storage. Our goal is to design a system that has no single point of failure, and therefore in ROTE the trusted storage is realized as a distributed system amongst a set of assisting nodes (some of which can be compromised).

Accountability for distributed systems. PeerReview [82] provides accountability for distributed systems and in particular detect nodes that violate from expected behaviour. Instead of fault detection, our goal is to realize distributed secure storage, customized for rollback protection, in the presence of faulty nodes.

Adversary models. Agreement has been considered under models where the faulty nodes have some trusted functionality (e.g., an unmodifiable hardware primitive). Such approaches reduce the number of required replicas to

$2f + 1$ [52, 54, 130, 137] or $f + 1$ [114]. We have no trust assumptions on the compromised nodes. Byzantine agreement has also been considered with dual failure models [65, 148, 176] where the adversary can fully control the faulty processes and can read the secrets of other processes. In our case, the adversary cannot read secrets from trusted enclaves, but it can extract keys from f compromised nodes, and additionally schedule enclaves' on all nodes.

Several recently proposed SGX systems [41,81,158,171,175,189,195,206] consider an adversary model with an untrusted OS. To the best of our knowledge, our work is the first to define a model with explicit adversarial capabilities that cover enclave restarts and multiple instances. These capabilities are critical for the security of our system and also other SGX systems (see Appendix A.2).

3.9 Conclusion

In this chapter we have proposed a new approach for rollback protection on Intel SGX. Our main idea is to implement integrity protection as a distributed system across collaborative enclaves running on separate processors. We consider a powerful adversary that controls the OS on all participating platforms and has even compromised a subset of the assisting processors. We show that our system provides a strong security guarantee that we call *all-or-nothing rollback*. Our experiments demonstrate that distributed rollback protection provides significantly better performance compared to solutions based on local non-volatile memory.

Chapter 4

DelegaTEE: Brokered Delegation using Trusted Execution

4.1 Introduction

Delegation, the ability to share a portion of one's authority with another, is a well-studied concept in access control. However, delegation remains mostly unsupported in today's online services. Email provides no delegation support at all, for example, while other services, such as Facebook, support delegation in a limited and coarse-grained way. Facebook allows a user to delegate to a third-party application the authority to post to the user's wall, but not to impose, e.g., a limit of three posts per day. In any case, the expression and enforcement of delegation policies lies entirely at the discretion of the services.

The ability to delegate access to existing online accounts and services, *safely and selectively*, could give rise to new forms of cooperation among users. Delegation may be useful for sharing digital content, such as access to streaming services like Netflix. Users may wish to delegate online tasks to remote workers, for example to reply to emails involving a particular topic or group. Delegation of access to financial services, such as Paypal, could enable broader access to banking.

Today, when delegation is needed in a way unsupported by the service, users must resort to credential sharing. This results in the *Delegatees* gaining

full access to the *Owners'* accounts. Such delegation mostly works only in closed circles with high levels of mutual trust.

In this chapter, we argue that the emergence of Trusted Execution Environments (TEEs), such as Intel Software Guard Extensions (SGX), has enabled an alternative way to achieve fine-grained delegation without trust between the Owner and Delegatee. We refer this new type of delegation — specifically with delegation restricted under a policy enforced by a TEE enclave holding the credential — as *brokered delegation*. Brokered delegation is a new and powerful tool that allows users to flexibly share and delegate access, without requiring the explicit support (or even knowledge) of the service providers.

To demonstrate the potential of brokered delegation, we design DELEGATEE, a system that provides brokered delegation for many existing web services according to complex contextual access-control policies. DELEGATEE also preserves the confidentiality of the managed credentials. We develop several application prototypes to demonstrate how brokered delegation can support new forms of resource sharing and give rise to new markets: secure outsourcing of personal and commercial microtasks, tokenization (i.e., creation of fungible, tradeable units), resale of resources and services, and new payment methods - all without changes to the legacy infrastructure. One of the key features of DELEGATEE is that it *requires no changes to the service managing the resource or to users' accounts*.

We present two design variations for DELEGATEE. The first design encompasses a purely decentralized peer-to-peer (P2P) system in which a Delegatee that wants to use brokered credentials executes the secure enclave on her machine. The Owner of the credentials connects to the enclave and delivers the credentials along with the access control policy under which the Delegatee can access a specific service. The second design is based on a centralized broker service operated by a third party. In this architecture, an Owner can register credentials and an accompanying policy, authorizing use by a specific population of Delegatees. Both system designs provide a comprehensive solution for brokered delegation and can be used based on users' preferences.

DELEGATEE also demonstrates a broader insight about the security consequences of trusted hardware: TEEs can fundamentally subvert access-control policy enforcement in existing online services. Depending on the application, DELEGATEE can either enrich a target service or undermine its security policies (or both). For example, reselling limited access to a paid subscription service in regions where the service is unavailable undermines the service's security policy, while delegating access to office tools such as mail, calendar, etc. to administrative assistants can enrich the capabilities

and usability of the service itself. Brokered delegation can also facilitate violations of web services' terms of use. Users may thereby circumvent *mandatory access control* (MAC) policies, reducing them to *discretionary access control* (DAC). The effect is similar to allowing use of `setuid` [131] in Unix irrespective of MAC policies [166, 187].

The fine-grained delegation offered by DELEGATEE can support new forms of meaningful cooperation among users, which existing online services do not provide. In this way DELEGATEE may be related to new technology-fueled resource-sharing models such as Airbnb and Uber, which have challenged legal and regulatory frameworks while creating and delivering appealing new services. We thus view DELEGATEE as a catalyst for such new contributions to the sharing economy.

Contributions. In summary, we make the following contributions:

• *Brokered delegation.* We advance a new model for user-specified safe delegation of resources and services governed by fine-grained access control. Our approach involves credential outsourcing to trusted hardware.

• DELEGATEE. We present DELEGATEE, a system that realizes brokered delegation via Intel SGX. We present two implemented versions: One based on a hardened third party acting as a credential broker (Centrally Brokered) and the other as a peer-to-peer system where users directly store, manage, delegate, and use credentials.

• *Security analysis.* We show that both DELEGATEE versions provide security in a strong adversarial model, protecting against some compromised SGX platforms as well as the full software stack of victims' machines.

• *Prototype implementations.* We describe and implement four applications on top of DELEGATEE: email, PayPal, credit card/e-banking, and full website access through an HTTPS proxy. We run these with commercial services such as Gmail and PayPal using real user credentials. We document minimal performance overhead and the ability to support many concurrent users.

• *Impact on access control.* We show that TEEs can be used to circumvent MAC policies in online services and allow discretionary access control, enabling users to delegate rights and access at their discretion.

The rest of this chapter is organized as follows. Section 4.2 describes the motivation. Section 4.3 presents DELEGATEE. Section 4.4 provides security analysis. Section 4.5 describes the prototype implementations. Section 4.6 provides the performance evaluation and Section 4.7 further discussion. We review related work in Section 4.8. Section 4.9 concludes the chapter.

4.2 Motivation and Problem Statement

4.2.1 Motivation

There are two major motivations for our work: To demonstrate the many settings in which brokered delegation gives rise to new functionality, and to demonstrate how (for good or bad) hardware TEEs can transform practically any mandatory access control policy in an online service into a discretionary one. Our four different application scenarios illustrate both motivations.

Mail/Office. Full or restricted delegation of a personal mailbox or other office tasks can be appealing for many reasons. These include a desire to delegate work to administrative assistants (e.g., read-only access, send mail only to a specific domain) or to allow limited access to law-enforcement authorities (e.g., read emails from a certain time window relevant to a court case). The first is especially valuable for virtual-assistant services, which outsource office tasks off-site [127]. Today, these services require users to completely share their credentials, a dangerous practice that discourages many potential users.
Payments. Virtually all payments, cash and cryptocurrencies excepted, happen through intermediaries. Users may naturally desire a richer array of choices of these intermediaries. Consider, for example, a payment system where the users pay using each others' bank accounts, credit cards, or third-party providers (e.g., PayPal). This can have large benefits in terms of cost-saving, business operations, and anonymity guarantees.

Imagine that a company wants to allow its employees to execute online purchases with the company credit card or PayPal, but restricted to a certain limit per expenditure and specific merchants. Currently, this cannot be done since access to the card details or PayPal credentials allows users to execute arbitrary payments. Companies therefore typically provide such information only to a few employees who then execute payments for the rest, resulting in a highly inefficient process.

Delegation of payment credentials can also enable direct cost-savings for the end user. An example online system based on this premise is Sofort [119]. Sofort works as an internet payment middleman, with lower transaction fees than for credit cards. Sofort pays merchants for clients' online purchases and is repaid by clients via bank transfer. To guarantee repayment, Sofort requires users to share their e-banking credentials with the service, a practice that clearly raises security and privacy risks.

Finally, payment delegation can benefit "underbanked" populations with limited access to online payment systems, by enabling them to leverage social ties (e.g., via brokered delegation to the accounts of friends, family, and peers).

Full Website Access. The most versatile form of delegation is delegation for arbitrary existing web services, which typically authenticate user accounts through password challenges and then cookies over HTTPS. This model includes access to users' social networking sites, video services, online media such as news and music, and general website content available only to registered users. One appealing example from the academia is *Sci-Hub*. "The site bypasses publishers' paywalls using a collection of credentials (user IDs and passwords) belonging to educational institutions which have purchased access to the journals." Many anonymous academics from around the world donate their credentials voluntarily [36]. Some services, such as Netflix and various news sites, already offer users the ability to log in from different devices. Users can thus share their subscriptions by sharing credentials, but only in a dangerous all-or-nothing manner. More fine-grained, service-specific, and secure delegation could facilitate much broader sharing (for good and bad).
Sharing Economy. The examples above involve an Owner delegating credentials to known Delegatees, e.g., friends or colleagues. However, Owners can also offer access to their services on an open market to a wide range of potentially pseudonymous or anonymous Delegatees. This would result in a shared economy in which Owners sell time-limited and restricted access to their accounts in return for other services or financial compensation. For example, users could sell access to Netflix accounts on an open market. They could also sell space in their social networking accounts to advertisers; e.g., a user could sell the ability to post in her name, enabling an advertiser to target her social network. The right to post could be restricted to a certain volume and type of content to prevent abuse by advertisers.

4.2.2 Problem Statement

If service providers regularly offered richly featured native delegation options, there would be no need for **brokered delegation**. Most do not, however, usually for business or regulatory reasons. Our work aims to change this situation fundamentally — DELEGATEE empowers users to delegate their authority, making use of any existing internet service, such that:

- The Owner's credentials remain confidential.
- The Owner can restrict access to her account, e.g., in terms of time, duration of access, number of reads/writes etc.
- The system logs the actions of Owners and Delegatees so that post-hoc attribution of their behaviors is possible (as means of resolving disputes).

• The system minimizes the ability of a service to distinguish between access by the Delegatee and that of the legitimate Owner, thus, preventing delegation. (As we shall discuss, this is not achievable for all services.)

4.2.3 Why the Problem is Hard

DELEGATEE leverages SGX to implement functionality that without SGX or equivalent mechanisms would be infeasible or impossible to achieve. Consider our delegated payment scenarios involving PayPal, credit card or e-banking. Such delegation would be easy to support on the back end; e.g., PayPal could offer a delegation API.

Without back end support, however, there are only two possible implementation strategies. The first is that the Owner remains online and mediates requests, which forecloses on the possibility of private transactions or her inability to provide continuous service availability.

The second is that the Owner provides the Delegatee with a digital resource for unmediated access to the target resource. This, however, would require black-box obfuscation to construct a functionality that establishes a TLS connection, authenticates a user with a concealed password, and supports a series of policy-constrained transactions. General virtual black-box (VBB) obfuscation is known to be impossible [29]. It is unclear whether indistinguishability obfuscation ($i\mathcal{O}$), whose realization remains an open problem [53], could achieve this functionality. $i\mathcal{O}$, would in any case require circuit complexity well beyond the bounds of feasible deployment. It would also be subject to replay attacks unless the functionality could somehow change or revoke the credential atomically with permissible operations. In summary, SGX is required to solve our problem as stated, and even with SGX, as we now explain, solution remains challenging.

4.3 DelegaTEE System

The main idea behind the DELEGATEE system is to send the Owner's credentials (passwords, etc.) to a Trusted Execution Environment that implements the delegation policy. The Delegatee communicates with the resource (web service) indirectly, using the TEE as a proxy. In this section, we present the DELEGATEE system design, usage models and policy implications.

4.3.1 System Design

We explore the DELEGATEE design space through two system architectures: a purely decentralized *P2P system*, and what we call a *Centrally Brokered* system, in which a third party runs the enclaves. Both architectures involve three

distinct classes of parties: credential *Owner(s)* A, *Delegatee(s)* B, and *service(s)* G. Additionally, the system distinguishes two data types: *credential(s)* C and *access control policy(ies)* P. Owners and Delegatees are generically referred to as *users*.

The system supports a potentially large population of credential Owners $A_1 ... A_n$ (henceforth referred to as Owners) and Delegatees $B_1 ... B_n$. In general, the Owner A_i has access to a service G_k. The Delegatee B_j does not have access to the service, but she can get access by using credentials C_x of the Owner A_i. However, the Owner A_i does not want to reveal the credentials to the Delegatee B_j. The Owner A_i wants her credentials to remain confidential and used only by an authorized Delegatee. Additionally, the Owner wants to restrict access to the services that she enjoys (i.e., G_k) according to an access control policy P_{ijxk} specific to this delegation relationship. P_{ijxk} defines an policy involving Owner $A_{\mathbf{i}}$, Delegatee $B_{\mathbf{j}}$, credentials $C_{\mathbf{x}}$, and service $G_{\mathbf{k}}$. The type and structure of the access control policy depends on the service that the Owner delegates. Definition and enforcement of the policies are described in Section 4.3.3.

P2P system architecture. In our peer-to-peer system, there is no need for a central management entity to mediate between the Owners and the Delegatees. A Delegatee can directly coordinate with the Owner to gain access to a specific service from group G. In order to execute this setup, a Delegatee from party B has to have a Intel SGX supported machine. The steps to execute secure credential delegation, also given in Figure 4.1, are:

(**1**) The Owner A_i agrees directly with the Delegatee B_j for which specific service (G_k) access will be granted using her credentials (C_x). The agreement is done at the users discretion and through any available channel such as online messaging, email, phone call, etc. Additionally, users need to establish a method for authentication upon enclave start (e.g., a pre-shared key, certificates). This step can be executed in an any informal communication channel that the users consider appropriate. However, the emphasis should be on the confidentiality of the channel (e.g., chat over a coffee).

(**2**) (optional[1]) After that, A_i prepares the enclave.

(**3**) (optional[1]) Owner A_i sends the executable to B_j.

[1]Enclaves used for the credential delegation can also be downloaded from a trusted source. Each different service requires implementation of specific enclaves due to access complexity. The Owner and the Delegatee can verify the enclave trustworthiness with attestation.

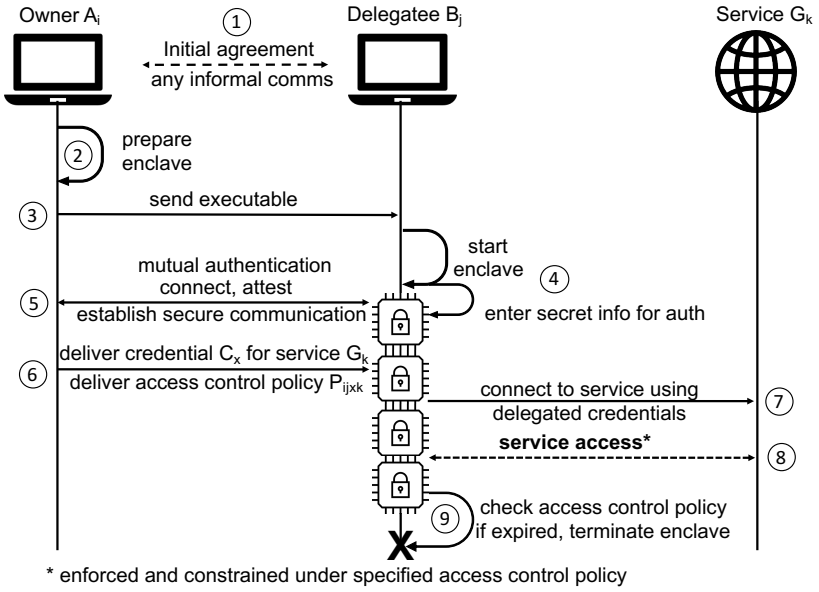

Figure 4.1: DELEGATEE*'s P2P system architecture.*

(**4**) The Delegatee B_j starts the enclave and enters the secret information (shared secret exchanged during the initial agreement) to the enclave needed for mutual authentication and secure connection establishment.

(**5**) After the Delegatee B_j starts the enclave, the Owner A_i connects to the enclave, attests it to verify that it is the correct code with respect to the requested service delegation, and subsequently uses the secret information to authenticate and create a secure communication channel.

(**6**) The A_i sends credentials C_x for the service G_k with the access control policy P_{ijxk} using the secure channel.

(**7**) The Delegatee B_j now uses the enclave as a proxy to connect to the service G_k using the delegated credentials.

(**8**) The scope of usage is strictly limited by the defined policy and therefore Delegatee B_j cannot use the parts of the service not allowed by the Owner A_i.

(**9**) If the access control policy has a time limit, the Delegatee B_j's access to the service is terminated after the time has passed, unless the Owner A_i extends the policy. The enclave restarts do not change this fact, requiring the connection from the Owner A_i to the enclave to deliver the information again.

The enclave is stateless, meaning that any interruption, restart or termination after the initial start and the delivery of confidential information is going to result in service abortion.

Authentication mechanisms. The agreement between the users and their mutual identification and authentication is of utmost importance. The Owner needs to be certain that the enclave used to access a specific service with her credentials is running on the machine of the intended Delegatee. Attestation only gives us proof that the enclave is executing the presumed code, but without any information under whose control the machine is. To allow mutual authentication between the Owner and the Delegatee, a separate authentication method is needed.

Several authentication mechanisms are possible. First, the parties could use an out-of-band confidential and authenticated channel to exchange a shared secret key. After the enclave start, the Delegatee enters this pre-shared key into the enclave. The Owner uses the same key to establish a TLS (PSK mode) session with the enclave. If an attacker attempts to establish an impostor or man-in-the-middle session with the Owner, the keys will mismatch. As an alternative, we could use a trusted PKI so that the Owner obtains Delegatee's public key certificate, later used to establish a TLS session. This requires the Delegatee to provide her private and public keys to the enclave. Our design is agnostic to the authentication method while the prototype uses the TLS-PSK.

Centrally Brokered system architecture. Alternatively to the P2P configuration, the Centrally Brokered system consists of a central server that mediates all transactions and communication between the involved parties and also serves as a management entity. The server has a trusted execution environment (SGX enclaves) that performs security-critical operations. Thus, the system can be attested to verify the running code and authenticated to verify the service provider. In this case, the Owners and the Delegatees do not need to have SGX. The steps to execute secure credential delegation, also given in Figure 4.2, are:

(**1**) Both the Owners (A_1...A_n) and the Delegatees (B_1...B_n) need to register with the system to acquire unique login information (username and password) for access. After registration, both Owners and Delegatees can execute credential delegation for service access.

(**2**) The Owners A_1...A_n now establish a secure channel to the system (using the ordinary web PKI) and start storing the credentials C_1...C_n for specific services G_1...G_n. The variety of credentials that can be stored depends on the supported services (see Section 4.5 for details).

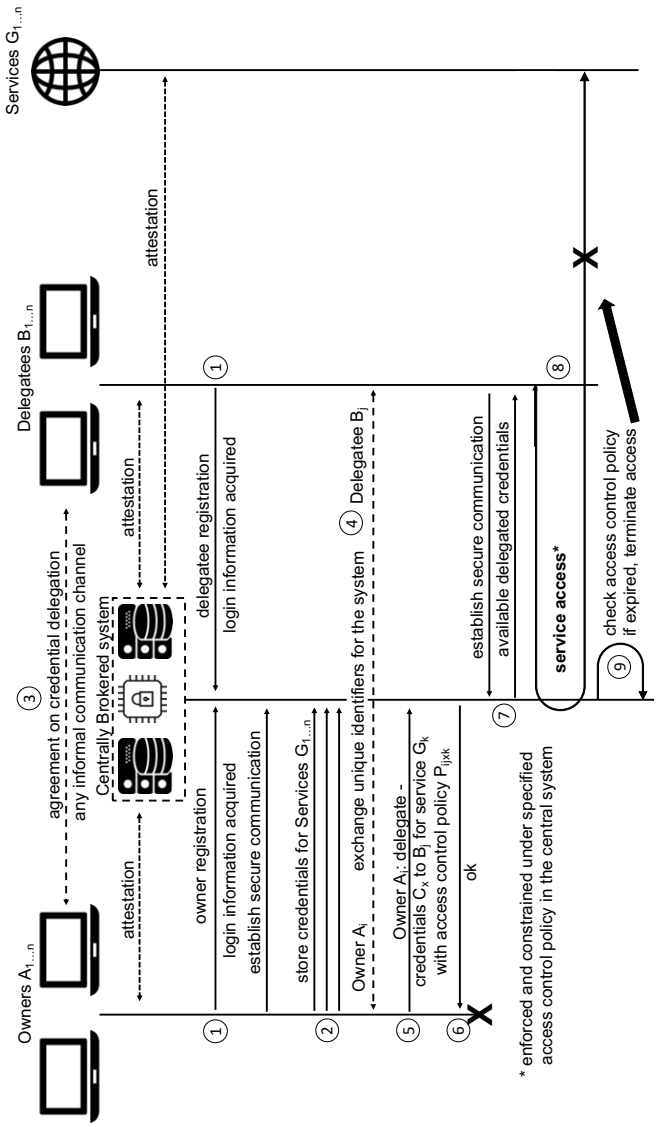

Figure 4.2: DELEGATEE's *Centrally Brokered system architecture.*

(**3**) The Owners $A_1 ... A_n$ may agree directly with the Delegatees $B_1 ... B_n$ for which specific service (G_k) the Owner will grant access using her credentials (C_x). The agreement is done at the users discretion through any available out-of-band channel and is limited by the implemented technical capabilities of the system (i.e., for supported use cases implemented by DELEGATEE).

(**4**) During the agreement, users exchange their unique identifiers (i.e., system username) so that the Owner from party A knows whom to authorize from party B.

(**5**) The Owner A_i establishes a secure channel to the system, specifies for which credentials (C_x) she wants to perform the delegation, for which service (G_k) and to whom (username of B_j), while she additionally specifies the access control policy P_{ijxk} that restricts usage.

(**6**) After receiving the confirmation, A_i disconnects.

(**7**) The Delegatee B_j now establishes a secure channel to the system and can immediately see that she has been delegated credentials for a certain service. The credentials are hidden for the Delegatee B_j. If the Delegatee wants to access the service G_k, she may proceed.

(**8**) The access to the service is always proxied through the central broker with no direct communication between the Delegatee and the service. Any attempt to circumvent this results in protocol termination (e.g., if the user clicks an external link outside the proxied service).

(**9**) After the defined access control policy expires (e.g., if it is time limited) the Delegatee B_j loses access and the credentials are no longer delegated.

Interoperability. In Section 4.5 we describe the implementation of DELEGATEE. Our prototype implementation is based on the Centrally Brokered architecture, since this is the most plausible deployment scenario, although we discuss how the P2P model applies to each supported application. The implemented enclaves have two operation modes that can be chosen and set prior to the execution. In case of the Centrally Brokered system, the enclave retrieves important data regarding services, credentials, and access control from the management enclave, while in the P2P system, the enclave awaits the connection from its issuer to receive all information.

4.3.2 Usage With and Without Anonymity

DELEGATEE supports both identity-based (non-anonymous) and anonymous use models, as follows.

Identity-based model. An identity-based model follows directly from the model and examples given above. Here, the users know each other in some

way, have a communication channel and can mutually identify each other. The Owner directly delegates her credentials to a specific Delegatee. Common use case examples include family sharing, delegation among friends and colleagues, etc.

Anonymous model. As DELEGATEE conceals an Owner's credentials, it naturally preserves her anonymity, even in the P2P model where the Delegatee operates the enclave executing DELEGATEE. However, the agreement is necessary in order to specify details for the delegation relationship. An Owner and Delegatee may negotiate and perform credential delegation without direct interaction. For example, a bulletin board (available on the Centrally Brokered system) might allow Owners to publicly list services they are willing to delegate, specifying accompanying access control policies and costs (or offer that is free). Owners may identify themselves with pseudonyms, e.g., onion addresses. In the P2P model, the bulletin board can be hosted on a third-party website, while the protocol runs through Tor Hidden Services, thereby ensuring privacy protection for both the Owner and Delegatee.

4.3.3 Policy Creation and Enforcement

Securely enforcing defined policies presents a challenge on its own. We aim to prevent all attackers from modifying the policies or circumventing the enforcement by applying a combination of allowed actions in order to reach a desirable state. While the security analysis (Section 4.4) ensures that the owner-provided access control policy is respected, the burden remains on the Owner to choose an appropriate access control policy in the first place. An Owner who wants to delegate restricted access for a specific service needs to be able to define all allowed actions through a rich access control policy, denoted as P_{ijxk}. For increased security, we prefer the white-listing of operations based on the least-privileges in order to prevent unwanted access and usage of the delegated account. Unfortunately, a general model for a wide variety of different services cannot be used. For every specific service category, and sometimes even for every specific service provider in the same category, a new policy must be created that resembles the exact capabilities and actions which a fully allowed user may invoke. We discuss these limitations in Section 4.7.

Policies in DelegaTEE. We designed and implemented policies for all scenarios defined in Section 4.2.1, i.e., for mail, payments, and website access.

In mail, DELEGATEE relies on the IMAP and SMTP protocols which are standardized and well defined. Inside the enclave we parse all incoming and outgoing request (to and from the Delegatee) and compare them against the defined access policy. Consider a concrete scenario: the organizer of a

conference wishes to delegate her email account to an assistant for responding to logistical questions from attendees. The Delegatee should be granted read access to only *subset* of the organizer's email (e.g., defined by a regular expression query like `(*#Usenix18*)`). The organizer might also wish to enforce restrictions on message sending. Rather than sending to any possible email address, the assistant may only be allowed to *reply* to emails and deleting emails should be prevented. In general, for the inbox requests the Delegatee can be limited based on criteria such as date, time, sender, subject or content of the email. In outgoing requests, the limitation is set on the subject or content, and the intended recipient(s). Additionally, the Owner can rate-limit sending within a time interval, applying a spam and abuse filter for outgoing messages.

In payments, the main restriction is on limiting the allowed amount per transaction or the total amount using the delegated credential for either a credit card or any other third party payment service. Additionally, the DELEGATEE can enforce restrictions on the source, limiting the Delegatee to perform payments only on specific sites or identified merchants/services, and white-listed geographical locations based on the IP address.

In the full website access, DELEGATEE implements limiting the use of login credentials to specific sites (e.g., the Owner can have the same credentials for two different services. However, full access is only achieved to the site allowed by the policy). As work in progress, the policies are expanded to restrict specific actions on sites after the login, including, clicks on various links, loading of specific site content or access to the account settings.

Our prototype implements delegation policies targeted at particular services, directly in `C++`. These policies rely on the mechanisms explained above; the Owner only needs to configure the value of the policy attributes (e.g., time limit, max amount, regular expression, etc.). In principle, the credential Owners could describe their own delegation policy in a general programming language. In Section 4.8 we mention existing and generic ways to extend our general functionality regarding access control. However, specifying the policies is difficult to do correctly. We envision that a likely deployment scenario is a curated "app store", to which entrepreneurs or power users submit useful policies they develop. These policies are then evaluated by experts and users. Web services are constantly updated, and the interfaces change over time, requiring delegation scenarios to be continuously maintained as well. In Section 4.7 we discuss further challenges if the services seek to actively prevent these delegation relationships.

4.4 Security Analysis

Brokered delegation provides a new usage pattern for potentially any existing online service. It, therefore, provides new security challenges as well, arising especially because each new service requires a customized delegation mechanism. In this section we describe the main security properties that DELEGATEE is designed to ensure across all applications:

(**a**) *First and foremost, the Owner's access credentials remain confidential.*

(**b**) *The use of the delegated credentials is defined by the access control policy which will not be violated.*

(**c**) *Use of the credentials should only be granted to the intended Delegatee, as authorized by the Owner.*

The DELEGATEE system is designed to provide these security guarantees even against a strong attacker model. We assume that an attacker neither corrupts the full software stack of the Owner's and Delegatee's machines (unless the Delegatee is the attacker), nor the online service, as existing web authentication mechanisms rely on them anyway. However, we consider an attacker that controls everything else (i.e., including the standard Dolev-Yao adversary [58] that can read and manipulate network traffic between parties). The two architectures we develop, P2P and Centrally Brokered, differ mainly in where the enclave is hosted (respectively, on the Delegatee's own device or at an independent third-party). Although we rely on a TEE, our system is designed to tolerate vulnerabilities in the SGX enclaves as long as the software stack on the machine running the enclave is also not compromised. Below we discuss several attacker configurations and the design decisions made to mitigate them. It is of utmost importance to note that our system is designed in a way that breaking the SGX protection mechanism on an arbitrary enclave will not subvert our system. The attacker would need to break the exact enclave running DELEGATEE, bypass the authentication mechanism, and compromise the full software stack on the same machine to violate the security properties. Side-channel attacks are considered out of scope in this chapter.

4.4.1 Security through Trusted Enclaves

We first describe how these properties are ensured assuming the TEE enclaves are secure, even if the software stack of the hosting system is compromised.

In the Centrally Brokered architecture, the TEE guarantees security properties (a) and (b) even if the central broker and the Delegatee are otherwise corrupted. The Owner only transmits her credential after validating the attestation that the enclave is running the correct code and if the authentication is

successful. The mechanism for authenticating the Delegatee to the broker also lies inside the enclave, in the broker's API enclave. This means that property (c) is guaranteed even if the broker's full software stack is compromised since all security-critical operations are performed inside the enclave.

In the P2P architecture, even if the Delegatee's software stack is corrupted, the Owner's credentials are kept confidential. In Step (5), the Owner receives a TEE attestation before communicating further over the TLS channel, and validates it against the DELEGATEE enclave executable. Since the Owner only sends her credentials along this channel directly to the enclave, it is never exposed to the Delegatee's host machine, thereby ensuring property (a). The only way the Delegatee can make use of the credentials is by providing commands as input to the enclave (all access is proxied through the enclave), where they are processed according to the access control policy P_{ijxk}, ensuring the enforcement of (b). Since the TEE is hosted locally by the Delegatee (that also has to authenticate to the Owner using the agreed shared secret), then property (c) is ensured against an external attacker if he cannot steal the shared secret; against a rogue Delegatee, this property is not meaningful anyway.

We note that in either architecture, the code running in the enclave must use the credentials in application-specific ways. We stress that in our proposed system, the owner-provided access control policy P_{ijxk} for service G_k is a configuration parameter given as input to one of the supported application specific enclaves. Hence the proof burden is on us to show that properties (b) and (c) hold for any policy P_{ijxk}. We refer the reader to Section 4.3.3 and Section 4.5 for a detailed explanation. To summarize, each application makes use of the credentials only to authenticate with the corresponding service.

Finally, we note that Denial-of-Service attacks are out of scope since an external (network) adversary can always drop messages.

4.4.2 Robustness to Compromised Enclaves

Our system relies on the TEE to provide security against a compromised Delegatee or the broker service. However, DELEGATEE is also designed to provide defense in depth where possible, such that even a partial compromise of the TEE does not impact security (as long as the host machine is also not compromised). In particular, we consider an attacker that can recover the internal keys (e.g., sealing, memory encryption, etc.) of the Intel SGX. This strong attacker would be able to decrypt any sealed persistent storage or encrypted memory pages and create false attestations. Recent works [49, 136, 191] have shown that this is indeed possible.

We address these concerns by designing our protocol so that all communication channels are authenticated end-to-end, even when communicating

with an attested SGX enclave. To illustrate, first consider the P2P architecture. Our authentication mechanism defends against such an attacker by requiring authentication input from the Delegatee before establishing the TLS endpoint in the enclave. Notice that by Step (5), the Owner A_i opens the TLS endpoint to the Delegatee's enclave, over a potentially insecure channel. At this point, if the attacker can forge an enclave attestation, then the TLS channel may actually be an impostor channel. However, by authenticating the TLS channel against the pre-shared secret initially established with the Delegatee, the Owner would detect and invalidate such an impostor channel. This authentication occurs before the Owner ever transmits the credentials, ensuring desired property (a). Furthermore, the enclave software is guaranteed to be the correct DELEGATEE executable transmitted by the Owner, as long as the Delegatee's host OS is uncorrupted. This ensures that properties (b) and (c) hold as well. However if a rogue Delegatee colludes with an attacker that can forge TEE attestations, or if the Delegatee's software stack is fully compromised, then the credentials would be forfeit.

Our Centrally Brokered architecture is also designed with end-to-end authentication to mitigate against a potentially compromised TEE. The Delegatee and the Owner each establish authenticated TLS channels to the central broker (authenticating the broker's enclaves using the typical certificate PKI), and only communicate to the broker over this channel. Hence all three security properties are ensured as long as the service's own software stack is not accessible to the attacker, regardless of any forged TEE attestations the attacker may produce or if the attacker can guess the SGX keys used by the enclave.

We also avoid the use of persistent encrypted storage in the P2P model, thus, preventing potential rollback attacks described in Chapter 3, which may otherwise occur if the enclave's sealing keys can be derived by the attacker. Our DELEGATEE enclaves, therefore, do not provide any means to resume a previously-established delegation session if the processor is power cycled. Instead, their state is restarted from scratch. In the Centrally Brokered system, we do presume that the attacker has no presence on the full software stack, thus, for continuous operation of the system we make use of the persistent encrypted storage. If the attacker model would be expanded to allow attacker presence on the software stack, methods and techniques such as the ROTE system described in Chapter 3 could be applied to prevent rollback.

For ease of exposition, we have only discussed the highlights of our security design. A systematic security analysis can be found in Appendix B.1.

4.4.3 Other Security Properties

Mandatory Logging. A well-chosen policy should ideally prevent any misuse from occurring. To be prudent, we would also like to ensure support for forensic investigation in the case that an incorrect policy is abused. We propose that all the requests and responses exchanged between the service provider and the Delegatee are securely logged using a timestamped statement signed by the enclave, for a possible later review. For example, in the payment scenario, if the Delegatee uses the Owner's credit card, the following events are registered: time and date, the website and the amount of the executed payment. As another example, in the mail scenario, if the Delegatee manages to evade the abuse filter and send offensive emails, these messages should be logged. We imagine such logs may be used later on to prove that the Delegatee herself performed some action and indemnify the credential Owner. This discourages the Delegatees to perform any actions that could harm the Owner. To detect suppression of log entries, we could make use of a hardware monotonic counter. The enclave could additionally require a "receipt" from an independent backup service that replicates the log entry, before continuing.

Delegatee protection. So far our security analysis has only focused on protecting the Owner. Security for the Delegatees may be important too. For example, if a Delegatee wishes to use the Owner's payment account to purchase a sensitive item, they may not wish for the transaction details to be disclosed to the Owner. A delegation policy supporting the Delegatee could, in this case, offer a way to automatically delete payment transaction logs (if this is possible at all using the payment service).

4.5 Prototype Implementation

In this section we describe our prototype implementation for the selected use cases mentioned throughout the chapter. All enclaves rely on the OS to handle incoming and outgoing TCP connections while the SSL endpoints reside in the trusted enclaves. We use the `mbedtls` library developed by ARM [134], which also comprises the bulk of our trusted-computing-base (TCB). The interface between the OS and the enclaves consists of one `ecall` and ten `ocalls`, all of which are needed by the SSL library to use the OS's capability to handle the TCP connections. The small number of calls and the small TCB, as shown in Table 4.1, facilitate code verification and reduce the surface area that may be affected by vulnerabilities.

To demonstrate our use cases, we implemented four service specific enclaves for delegated use of mail, PayPal, credit card/e-banking, and full website

Enclave type	Core	mbedtls	Total
API	4.0 (7.3%)	51.0 (92.7%)	55.0
Mail	1.9 (3.6%)	51.0 (96.4%)	52.9
Paypal	2.6 (4.9%)	51.0 (95.1%)	53.6
CreditCard	2.5 (4.7%)	51.0 (95.3%)	53.5
HTTPS Proxy	2.7 (5.0%)	51.0 (95.0%)	53.7

Table 4.1: *TCB of* DELEGATEE *in LoC (thousands).*

access through an HTTPS proxy. Additionally, the fifth management enclave is used to authenticate the users and store credentials, implemented as a RESTful API, further referred as the API. The API enclave is not used in the P2P system since it is not needed. Only service specific enclaves are deployed on the Delegatee's machine. Additionally, we implemented a browser extension that communicates directly with the Centrally Brokered system and allows ease-of-use for the delegated credentials by the Delegatee (page parsing, detection of forms, choosing delegated credentials, etc.). All communication between the users, the enclaves and the browser extension is done using TLS with replay protection. We refer the reader to Appendix B.2 for prototype screenshots of chosen examples. In these implementation details we presume that the Owner A_i and Delegatee B_j already registered to the system and that the Owner authorized the Delegatee by storing the credentials C_x and defining the access policy P_{ijxk} for a specific service. Thus, the Owner A_i is not shown in the figures.

Multithreading in Intel SGX. Intel SGX does not support traditional multithreading within an enclave. Additional threads cannot be started by an enclave, instead multiple threads of the untrusted app can simultaneously perform an `ecall`, resulting in parallel enclave execution. The amount of concurrency is specified during compilation of the enclave and is limited by the number of logical cores in the processor.

Additional Authentication. In Section 4.7, we discuss limitations concerning the modern authentication challenges and DELEGATEE. Our implementation supports one advanced authentication method involving *CAPTCHA*. In case of website login or PayPal, a captcha may be required as an additional authentication step. We successfully overcome this issue by extracting the secret image, presenting it to the Delegatee through browser extension generated pop-up, allowing her to solve it and continue with executing the desired operation. We refer the reader to Appendix B.2 for prototype screenshots.

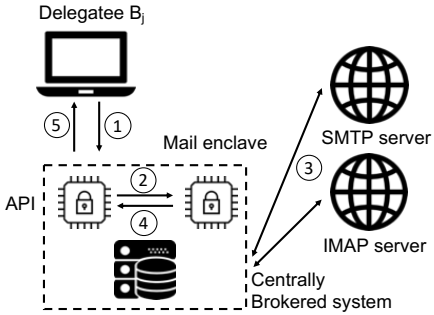

Figure 4.3: *Architecture overview: Mail model.*

4.5.1 Mail/Office

Delegation of email accounts under a specific access policy, one of the DELEGATEE motivated applications, is implemented in the mail enclave. IMAP and SMTP clients are implemented to allow a Delegatee B_j to read and send emails using the delegated credentials C_x. Below we describe the architecture depicted in Figure 4.3:

(**1**) The Delegatee B_j wants to use some credentials C_x that have been delegated by A_i. B_j connects securely to the centralized API using her username and password (for P2P model the communication is established as described in Section 4.3.1, with both methods supported). She then requests to perform some action using C_x.

(**2**) The API verifies that the Delegatee has access to C_x and then forwards the request, C_x and the corresponding policy P_{ijxk} to the mail enclave.

(**3**) The mail enclave connects to either the SMTP server (for sending mail) or the IMAP server (for receiving mail) and executes the requested operation.

(**4**) P_{ijxk} gets applied to the response from the external servers (IMAP) or to the outgoing requests (SMTP) and the resulting response gets forwarded to the API.

(**5**) The API delivers the final response to B_j.

4.5.2 Payments

PayPal. PayPal does not want to endorse giving away your credentials or automating the payments as this could compromise their security. Thus it is non-trivial to automate a PayPal payment and there is no public API. We must emulate a browser inside our enclave that accurately simulates a real user. Normally the payment process relies on a javascript library but running a

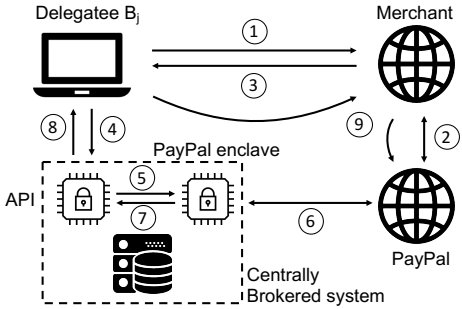

Figure 4.4: *Architecture overview: PayPal transaction model.*

javascript interpreter in Intel SGX would bloat the TCB, and create potential vulnerabilities associated with running an unmeasured, externally provided script inside an enclave. We instead use the no javascript fallback mechanism from PayPal. Our implemented emulated browser follows redirects, fills known forms, and handles cookies until the final confirmation page is reached. The enclave then returns a confirmation `id` to the issuer that is used by the merchant to finalize the payment. Our implementation was tested using PayPal's sandbox and real-world environment, executing a real payment. Our browser extension simplifies the use of delegated PayPal credentials by adding a DELEGATEE checkout button next to the original PayPal checkout button if the Delegatee is logged in to our system and has some delegated credentials. Upon clicking on the DELEGATEE checkout the Delegatee can choose one of the available PayPal credentials delegated to her and then the automated payment process starts (please see Appendix B.2 for screenshots). After that, no further user interaction is needed and the Delegatee will be forwarded to the confirmation page of the merchant if the payment succeeds. Below we describe the architecture depicted in Figure 4.4:

(**1**) The Delegatee B_j wants to buy something from a merchant using credentials C_x delegated by A_i. B_j connects to the merchant and asks for a PayPal payment.

(**2**) The merchant uses PayPal API to create a payment.

(**3**) The payment is then forwarded to B_j.

(**4**) B_j connects securely to the centralized API enclave using her username and password (for P2P model the communication methods are described in Section 4.3.1). She then requests to pay with PayPal using C_x.

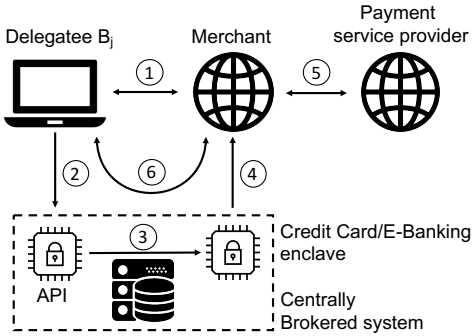

Figure 4.5: *Architecture overview: Credit card / e-banking model.*

(**5**) The API enclave verifies that the user can access to C_x and then forwards the request, C_x and the corresponding policy P_{ijxk} to the PayPal enclave.

(**6**) The PayPal enclave connects to PayPal and pays with C_x if it is allowed by the policy P_{ijxk}. The PayPal service returns a confirmation number.

(**7**) The confirmation number is forwarded to the API.

(**8**) The API delivers the confirmation number to B_j.

(**9**) B_j forwards the confirmation number to the merchant and then the payment is finalized by the PayPal API using the received confirmation number.

Credit card/e-banking. Credit card payments are similar to PayPal: upon checkout on the merchant's website, the browser extension is triggered if the payment form is available. The Delegatee chooses any delegated credentials she is authorized to use. The enclave fills the form with the credentials received either from the centralized API (or directly from A_i in the P2P model). Our implementation was tested without a service provider that would finalize the transaction. Figure 4.5 shows the detailed architecture with these steps:

(**1**) The Delegatee B_j wants to buy something from a merchant using some credentials C_x containing credit card or e-banking information that have been delegated by A_i. B_j connects to the website and the browser extension renders a second button beside the normal credentials submit form button.

(**2**) Upon clicking the injected button, the browser extension requests a payment with C_x from the API.

(**3**) The API verifies user's allowed access to C_x and then forwards the request, C_x and the corresponding policy P_{ijxk} to the credit card/e-banking enclave.

(**4**) The enclave fills C_x into the request while taking the policy P_{ijxk} into account and forwards it to the merchant.

(**5**) Finalization is done by the payment service provider.

(**6**) Response is routed through the enclaves to B_j.

4.5.3 Full Website Access

HTTPS Proxy. For secure browsing we implemented a HTTPS proxy enclave. We want to proxy selected websites and if a user leaves the website, he also leaves the proxy. We implemented this by using cookies to set the correct host name. The user sends any request to the proxy and he sets a cookie with the host name he wants to visit through the proxy. The enclave then parses the request, replaces the host name and sends it on to the real website. The response is also modified by the enclave so that the host name points to the proxy again. All links in the response are left unmodified so all relative links point to the proxy but all absolute links direct to a different website. The website certificates are checked against the statically compiled root certificate list in the enclave.

Login. To log into a service using delegated credentials we leverage similar technologies as in the HTTPS proxy and we thus only extended the proxy enclave to support delegated authentication for websites. Analogous to the HTTPS proxy we use cookies to specify the Delegatee's *session token* and which credentials C_x she wants to use. The enclave then asks the API whether the Delegatee with the specified *session token* is allowed to use C_x. If everything checks out, the API responds with the details of C_x and P_{ijxk} and the proxy enclave fills the login form before forwarding it to the website. As websites *session tokens* are usually stored in cookies, we encrypt all cookies forwarded to and from the website in order to prevent session stealing by an adversarial Delegatee. We use the browser extension in the same way as in the PayPal example: a button is rendered next to the original login. Figure 4.6 depicts the architecture and the detailed steps:

(**1**) The Delegatee B_j wants to log into a website using some credentials C_x that have been delegated by A_i. B_j connects to the website and the browser extension renders a second button beside the normal login button.

(**2**) Upon clicking this button, the extension changes the URL pointing to the proxy and appends cookies, specifying the credentials B_j wants to use.

(**3**) The proxy asks the API for C_x. The API checks if B_j has the rights to use C_x and then forwards C_x.

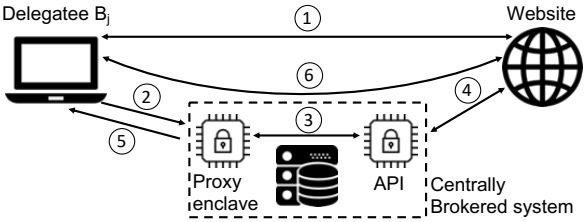

Figure 4.6: *Architecture overview: Login model.*

(**4**) The proxy enclave fills in the username and password into the login request and proceeds to send it to the website and receives the response.

(**5**) The proxy rewrites the header of the response to encrypt cookies and then forwards it to B_j.

(**6**) All subsequent connections have to go through the proxy where the policy P_{ijxk} can be enforced.

4.6 Performance Analysis

In this section we show that the overhead imposed by our solution stays within reasonable bounds. The performance testing was done using two i7-7700 machines with 16 GB RAM, connected via the internet and local network. Our results show that we can serve around 100 users concurrently even running on consumer grade hardware.

Table 4.2-a) shows an overhead of around 50ms for a full SSL handshake using `mbedtls` inside an enclave. The handshake involves three exchanged messages, thus at least three `ocalls`/`ecalls`, all of which have to copy buffers. In our measurements we recorded 19 `ocalls` during a request to the enclave. Overhead for `ocalls` and `ecalls` is measured and analyzed in [196] and is significant for copying buffers from the untrusted memory.

The mail enclave incurs minimal overhead (Table 4.2-b) with the extra handshake to the IMAP server (P2P system). In our test we retrieve all emails from the account inbox. In the Centrally Brokered system an additional handshake with the API is leading to a higher delay.

The PayPal example does not seem to suffer from any delay added by our implementation (Table 4.2-c). Note that we performed tests using the sandbox environment, provided by PayPal itself for testing integration with their services. This environment is feature-complete but slow as it is only functionality-oriented. Most time falls in waiting for the PayPal servers. As the enclave uses the fallback mechanism to execute PayPal transactions

Type	Test case	Mean (± std)
a) SSL handshake	`openssl`	52.12ms (± 3.62)
	`mbedtls`	57.14ms (± 3.37)
	`mbedtls` in SGX	105.22ms (± 4.23)
b) Mail	direct	1.12s (± 0.27)
	mail enclave	1.19s (± 0.22)
	API/mail enclave	1.45s (± 0.25)
c) PayPal	direct	25.92s (± 6.83)
	direct, no js	29.96s (± 8.51)
	PayPal enclave	27.00s (± 4.35)

Table 4.2: *Latency for a) SSL handshakes, b) receiving e-mails in inbox, and c) executing PayPal transactions. Sample: 1000.*

Target (site)	Test case	Mean (± std)
small (2.6KB)	direct	5.0ms (± 2.7)
	proxy enclave	64.3ms (± 2.5)
medium (411KB)	direct	12.2ms (± 1.2)
	proxy enclave	76.8ms (± 3.3)
big (15.7MB)	direct	202.6ms (±19.9)
	proxy enclave	432.2ms (±16.0)

Table 4.3: *HTTPS proxy latency with various page sizes.*

without JavaScript, we measured both variants: one allowing JavaScript and one blocking it. We also conducted tests in the real PayPal environment using the Centrally Brokered system, executing a real payment and buying an item online with a merchant supporting PayPal. However, due to the *CAPTCHA* protective mechanism involving the Delegatees' actions, it is not feasible to measure performance, since it depends on the user input.

Table 4.3 shows that the proxy adds the biggest latency overhead compared to normal browsing but still well below 0.1 seconds for small to medium websites, a response time limit for seamless user interaction [155]. A part of the high delay stems from the enclave waiting for the whole web server response before forwarding it to the Delegatee. Message parsing, the additional handshake, and the fact that all communication has to cross the `ocall`/`ecall` interface twice also adds to the overhead. A full HTTPS proxy enclave is in the works to reduce the waiting time and support all client connections.

We have also tested video streaming through our proxy, supporting DELEGATEE's streaming service examples (i.e., Netflix). We modeled streaming

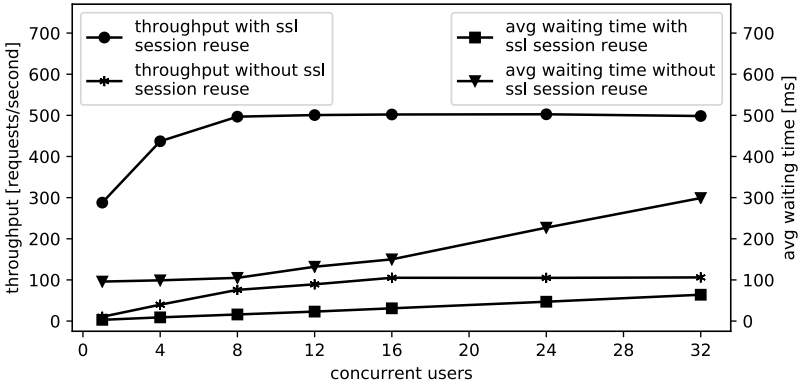

Figure 4.7: *Concurrency shown in throughput and average waiting time for client requests. Sample: 100.*

as a client that requests some video from a webserver. Therefore the performance of video-streaming through DELEGATEE is analogous to the ordinary HTTPS proxy use case. There was no additional overhead compared to normal streaming for a single user, e.g., as in the P2P model (the standard deviation is larger than the initial waiting time, for both the normal streaming and the proxied one). The streaming service was tested on the Centrally Brokered system where the delegatee connects to the proxy from the internet. This test was only done for a single user streaming at one point in time due to hardware and bandwidth limitations. As in the previous test, the overhead is negligible once the streaming starts while the initiation depends on the current latency.

Our multithreaded implementation was tested using 8 threads. Incoming connections are kept in a queue and served by the enclave threads, thus reaching maximum throughput with 8 concurrent users sending requests, as shown in Figure 4.7. The average waiting time stays constant until the same 8 user threshold, increasing linearly as new requests get queued. Our implementation supports SSL session reuse which significantly improves the throughput and lowers the waiting time. Without session reuse we can accommodate maximum 100 req/sec for 32 concurrent users, while with session reuse this grows to 500. Numbers could vary depending on the chosen cipher suite (which in our case is ECDHE-RSA-AES256-GCM-SHA384).

4.7 Discussion

In this section we explore limitations of DELEGATEE, mainly focusing on how brokered delegation faces technical challenges in the authentication process, as well as business and regulatory challenges arising from users controlling their own resources in a more flexible and fine-grained way than service providers intend.

Authentication challenges. Authentication in modern web services is complex. It can involve not just passwords but additional factors such as personal questions, email challenges, phone challenges, and "two-step authentication" apps such as Authy and Google Authenticator. Some of these can be supported with DELEGATEE, such as, email challenges or 2FA apps that could run inside the enclave as well, while for some, e.g., phone challenges, DELEGATEE cannot overcome the challenge.

Contextual factors often additionally come into play, such as the IP address, time of day, and nature of service requests. Financial services, e.g., PayPal, have particularly sophisticated fraud detection regimes; e.g., ordering unusual products with Paypal may trigger a fraud alert. Consequently, a single credential in the form of a password may not suffice to delegate a resource or service via DELEGATEE. In Section 4.5 we outline a solution for additional authentication in form of a CAPTCHA, required by some online services.

To illustrate, consider a scenario in the P2P model where an Owner Alice (an inhabitant of the U.S.) delegates a password to DELEGATEE and allows her PayPal account to be rented. Suppose then that Delegatee Bob, in Nigeria, rents Alice's PayPal account in a prescribed way and attempts to execute a transaction. Paypal will see an unusual request coming from an IP address in a country with a different risk profile than the U.S., and potentially one that Alice has never visited. Bob's transaction request is likely to be suspicious. PayPal may deny the transaction or request additional confirmation, e.g., via e-mail, to proceed. If Alice is unavailable or denies the transaction - which she may fail to recognize as originating from delegation - the transaction will fail.

For future production deployment of DELEGATEE, we will address these complications in several ways:

• *Application-specific delegation:* Authentication systems vary considerably across applications and service providers. Each DELEGATEE application will include configuration not just for the APIs of a given target Service, but also its authentication policies.

• *Delegation of multiple credentials:* For services that require multiple credentials, DELEGATEE may require more than a password from an Owner. For example, two-step authentication apps can be executed within the enclaved DELEGATEE application and set up by an Owner as an additional authentication factor. Similarly, an Owner may delegate her email to the enclave to respond to email-based authentication challenges. The SGX platform performing the delegation may be situated in the same country or region as the Owner. Finally, an Owner can perform a set of legitimate transactions through DELEGATEE in order to confirm that required credentials are present and to white-list the platform with the authentication system of the target service.

• *Failure modes:* Periodic delegation failures are inevitable, just as legitimate users' transactions fail sporadically due to false positives in the fraud-detection systems. As DELEGATEE is not intended for mission-critical uses, it could include graceful failure modes.

Authentication collisions. Attempts at simultaneous use of a resource may fail, as many web services do not support multiple concurrent sessions for a given account. For example, if Alice has delegated use of her bank account to Bob, then she may be unable to use it herself while Bob (or DELEGATEE, to be precise) is logged in. Such collisions can be treated by invoking failure modes like those for basic authentication failures. Other policies are possible, however. For example, Owner Alice may set a policy that only delegates her resource at times when she is unlikely to use it. A small enhancement can also enable Alice to preempt the session of a Delegatee if desired.

Usability, Deployment and Service Prevention. Throughout the chapter we have presented multiple use-cases and implemented prototypes that support delegation of different services. The usability of these services by potential Delegatees is as if they were using the original service as its Owner. However, the usability of the DELEGATEE in general depends on the supported use-cases. A limitation of our system is that for each and every use-case a specific module (that matches the capabilities and technical challenges) has to be implemented. Until now, we have not found a way in order to develop a generic module that could support a wide variety of services. For example, interpreted languages, such as Javascript, remain an open problem since by executing unmeasured code in an enclave running the interpreter we cannot guarantee the security properties of DELEGATEE. In addition to that, almost all services (even the ones from the same category) have different user mechanisms, UI and control. Thus, a specific policy needs to be created that matches these

controls in order to allow Owners to specify how their service could be used by potential Delegatees. Due to the complexity, for now, the policies have to be created beforehand along with the implemented delegation scenario, while the end-user involvement is limited to configuring parameters, out of a set of given policy characteristics.

If all service operators would share a unique set of API calls that could cover the full functionality of their services, then the deployment of DELEGATEE would be feasible for almost all service categories. This would also allow for the creation of more general and richer access control policies that could be created by the end-users of the service as well, possibly overcoming the initially discussed complexity of complete policies that require serious engineering and evaluation of each specific use-case scenario.

However, it is hard to imagine that the service operators would view the above even as a viable option. In many cases, DELEGATEE allows the creation of secondary markets (see the last paragraph of the section) and poses a threat to the revenue stream of the original service operator. Additionally, DELEGATEE reduces the operators' ability to control and track their users since, virtually, the number of users could grow but they would be seen only through the increased activity of users registered to the original service. Thus, most service operators would try to deny service access if executed through this form of delegation. As already mentioned, IP geofencing, pattern matching of actions and service usage, 2FA, along with the already existing fraud-detection mechanisms may endanger the functionality of our system. We have addressed several of them, however, future work involves investigation into further improvements that could make the distinction between the Owner and any Delegatee less possible.

Scalability. Scalability for all other supported services except video streaming is generally not a constraint. It comes down to running a proxy which can be adapted in terms of processing power (adding more enclaves horizontally) like any other service provider, while the bandwidth requirements remain moderate. However, in the case of video streaming in the centralized approach, the limitation is in the number of running connections since all video material is re-routed through the proxy. Namely, the proxy would need to have extremely high bandwidth, processing power and be scalable almost as the video service provider itself. We did not perform scalability tests to see how many users in parallel we could support for the video streaming example. This would require server grade hardware which we do not possess.However, for the P2P model, since the enclave resides on the Delegatees themselves, a single Owner can support multiple delegation of his, e.g., Netflix account (at least based

on the limit of Netflix itself – 2 or 4 devices based on the subscription). The streaming is done directly to the Delegatee, and the access will be valid until the policy expires.

Secondary markets. Brokered delegation could give rise to offerings that compete directly with those of the very platforms hosting the resources.

Facebook users could sell opportunities for "sponsored post" - unsolicited advertisements sent to their networks of friends or shown on their walls, as discussed above. Facebook users would then compete with Facebook itself in selling ads. Similarly, users could rent use of their Netflix account. Account sharing is already common within families and close friend circles. Brokered delegation could enable broad reselling and foster competition with direct sales of the subscription service.

Such secondary markets would in many cases violate providers' existing terms of service and might resemble markets for underground sales of virtual goods [129, 203]. Those underground markets have met with two responses, sometimes used in tandem: (1) providers aim to detect facilitators of secondary markets and penalize or ban them, and, (2) providers themselves seek to capture the revenue streams generated by secondary markets; e.g., online role-playing game providers have offered virtual goods for sale through their own shops [140]. DelegaTEE could provoke similar responses.

Peer-to-peer cryptocurrency-for-fiat exchanges is another setting that can benefit from DelegaTEE. Today, websites like LocalBitcoins.com receive Bitcoin deposits and hold them in escrow. Then they match-make and allow a buyer and a seller to negotiate a e-banking transfer. When the receiver gets the bank transfer, they instruct the LocalBitcoins service to complete the payment from the escrowed funds. If the receiver raises a dispute, then the service must investigate and ultimately determine whether to release the funds. However, such services naturally have limited investigative ability. They may call the user's bank, or ask both parties for evidence (i.e., screenshots). Neither option is satisfactory; the latter is prone to forgery, while the former may inadvertently draw suspicion to the user's bank account. Credential delegation provides an alternative, simplifying this business model and implementing a secure intermediary that guarantees execution and fair exchange.

4.8 Related Work

TEEs are widely used today. ARM TrustZone, for example, is commonly used to protect data on mobile devices, e.g., biometric templates and encryption keys in iOS devices [22]. Intel SGX has been proposed for a number of

applications, including confidential map-reduce tasks [169], trustworthy data feeds for blockchain oracles [206] and retrofitting of legacy applications [32], secure payment channels [135], etc. With DELEGATEE we extend this line of work with a new class of applications based on credential delegation.

Delegation of authority has been an important focus in access control security. Two mechanisms are commonly used. First, the credential Owner can interact with an authentication service to mint new credentials or tokens (representations of capabilities) for the Delegatee (e.g., Active Directory, Kerberos, and Oauth [84, 85]). Second, using chains of cryptographic assertions or certificates (as in X.509 or SPKI/SDSI), which can be digitally signed and communicated without interacting with a central server [31, 34, 37, 67, 154]. In either case, the delegation mechanism must be supported by the resource (or a reference monitor guarding the resource). Our system is different in that we use a trusted enclave-based proxy that stores the user's credentials and is transparent to the resource. It is, therefore, used to retrofit delegation for existing web services, without requiring additional effort (or even explicit support) from the provider.

Many web services like Facebook, and Twitter, support delegation for third-party applications typically using OAuth or OpenID (e.g., a user may delegate to a Facebook app the authority to read her friends-list but not to post new messages on her behalf). However, this delegation is not very expressive. The authority to post on a user's Facebook wall is all-or-nothing, for example; we cannot express restrictions such as "no more than 1 post per day." Much of the research literature has focused on flexible languages for specifying and reasoning about delegation policies [31, 37, 91, 173]. Our approach is complementary, as our enclave-based proxy can be used to apply more expressive policies to existing services.

Without support for fine-grained delegation, users sometimes resort to sharing passwords with each other or with third parties [173]. For example, to use the financial dashboard service Mint.com, users often need to share their bank account passwords with the service [197].

Delegation based on TEEs promises a more secure alternative to this status quo. Credential delegation using SGX was first explored in [206] to support "oracle" queries. Use of SGX for credential management was also proposed in [113]; there the goal was validation and resale of credentials for criminal purposes. More recent work involves the delegation of private keys for cryptocurrencies in order to secure a payment channel [135]. DELEGATEE is much more general than these prior works, as it supports delegation of credentials for any desired goal.

4.9 Conclusion

In this chapter we propose a new concept called brokered delegation, using TEEs to enable flexible delegation of credentials and access rights to internet services. We explored two design spaces, the decentralized P2P mode as well as a more pragmatic Centrally Brokered mode. Our implementation and experiments show that Delegatee in either mode can be applied to several real-world applications with minimal overhead, while preserving security against a strong attacker. Delegatee therefore has potential to enable delegation for any existing services, even without support from the service itself. This raises significant questions for future work: Can we enable robust delegation even against services that act to prevent it? Or can services defend against unwanted delegation? Lastly, given secure delegation, how would the economy of online services change?

Chapter 5

Privacy Preservation for Lightweight Cryptocurrency Clients

5.1 Introduction

Since its inception in 2008, Bitcoin has fueled considerable interest in decentralized currencies and other blockchain applications. The main goals of blockchains include a distributed trust model and increased user privacy. Several other blockchain platforms, such as Ethereum [5], leverage the same open or permissionless model as Bitcoin, while platforms like Hyperledger Fabric [20], Ripple [11] and R3 [10], enable closed or permissioned blockchains. Most blockchains implement a decentralized time-stamping mechanism that ensures eventual consistency of data, such as *transactions*, by collecting them from the underlying peer-to-peer (P2P) network, verifying their correctness, and including them in connected blocks.

In general, decentralized cryptocurrencies offer the potential to revolutionize payments. By providing transparent means to audit transactions, they reduce the need to rely on trusted incumbents and allow new innovation on financial applications. But this same transparency renders nearly all cryptocurrencies completely unsuited for wide-scale adoption: all transaction are broadcast publicly in a manner that can be readily linked to real world identities [21, 147], raising issues with government surveillance, harassment and the viability of business competition when competitors can see all cash flow.

A variety of protocols have been proposed, such as Solidus [14, 46], Cryptonote [193], Zerocoin [149] and Zerocash [33], that, with varying effectiveness [115, 122, 152], alleviate these issues. For example, in Cryptonote the destination address is always a newly generated one-time public key derived from the receiver's public key and some randomness from the sender. Zerocoin functions as an overlay on Bitcoin, where users mint a zerocoin and issue a transaction to transfer the funds to its commitment. The coin can be further spent by using zero-knowledge proofs. The most promising of these protocols, Zerocash, removes all information, such as sender/recipient identity, value, and linkability through the use of a zero-knowledge proof that there exists some past transaction which gave the user the funds they are spending. Zerocash is deployed in the cryptocurrency Zcash.

Payment Notification. Unlike in traditional means of payment, like credit cards and cash payments, in nearly all cryptocurrencies, including Bitcoin, Ethereum, and Zcash, it is possible to send money to a recipient's address without direct interaction or communication with the recipient. The recipient is paid, but only learns this next time when she is online. This raises the problem of *payment notification*, that is, the recipient must find out they were paid. Some cryptocurrencies, like Ethereum, use an account model where there is a single, well-defined location for payments. As a result, the recipient and, more significantly, anyone else, can see when and for how much someone is paid. Other cryptocurrencies, including Bitcoin and Zerocash, eschew this approach for improved privacy, storing payments individually as unspent transaction outputs (UTXOs) in unpredictable locations. In such systems, there must be some mechanism for users to discover a UTXO belongs to them. The simplest way to do this is to scan the blockchain and check each transaction.

This underlying process fueling blockchains imposes heavy requirements on bandwidth, computing, and storage resources of blockchain nodes that need to fetch all transactions and blocks issued in the blockchain, locally index them, and verify their correctness against all prior transactions. For instance, a typical Bitcoin installation requires more than 160 GB of storage today, and the sizes of popular blockchains are growing fast (e.g., Bitcoin's blockchain grew over 60% in the last 14 months) [3, 6]. This makes usage of blockchains infeasible on resource-constrained clients like mobile devices.

Payment notification is a particular problem for privacy-preserving systems like Zcash. Transactions in Zcash, consist of an opaque commitment, a ciphertext, a serial number to prevent double spending, and a zero-knowledge proof of the transaction's correctness and the existence of funds to spend. In particular, there is no metadata to identify the sender or recipient. The

only way for a client to identify if a payment is directed to them is by trial decryption of the ciphertext associated with a transaction: each transaction contains a ciphertext under the recipient's public key. To monitor for payments, clients must, therefore, conduct a trial decryption for *every* transaction on the blockchain. While this is completely feasible for well-resourced clients, running on platforms like standard desktop PCs, it is not desirable, nor often feasible, for resource-constrained clients like mobile devices where both power and bandwidth are major constraints. In this chapter, we focus on such resource-constrained clients.

Light Client Model. To address such heavy resource requirements, most open blockchain platforms support *lightweight clients*, targeted for devices like smartphones, that only download and verify a small part of the chain. E.g., the Bitcoin community provides the BitcoinJ [2], PicoCoin [9] and Electrum [4] clients implementing the Simple Payment Verification (SPV) mode [153], where the clients entrust and connect to a full node that has access to the complete blockchain and can assist the client by responding to queries about payments to a given address, thus performing transaction confirmation for them. The SPV protocol reveals to the server which (pseudonymous) *addresses* belong to a client and thus links multiple addresses together and potentially to real world identities, reducing user privacy.

To improve user privacy, several clients support *filters* (e.g., Bitcoin's BIP37 [87] and Ethereum's LES [7]). The goal of filters is to allow the client to define an anonymity set in an attempt to hide its real addresses from the full node. For instance, Bitcoin's BIP37 supports Bloom filters [35] that allow the client to define a set of transactions, with false positives, that are requested from the full node. Essentially, this approach presents a trade-off between communication efficiency and privacy: a Bloom filter that returns many false positives provides a larger anonymity set but requires more communication. Although such filters can be configured to be efficient, recent studies have shown that in practice they offer almost no privacy [68]. Consequently, none of the current Bitcoin lightweight clients provides adequate privacy protection with practical performance overhead.

Even more, directly applying the same model to Zcash is not possible without revealing the client's decryption key to the server so that it can perform the trial decryption for transactions, and thus completely breaking the privacy properties of Zcash, that intrinsically, unlike other cryptocurrencies, offers full privacy for users. Another challenge for resource-constrained clients is that simply notifying users that they received funds is not sufficient to use them for new Zcash payments. To spend funds sent in a previous transaction tx

in block n, users must prove that there exists a path (called *witness*) w from the root of a Merkle tree (called *note commitment tree*) to tx. Moreover, this information is not static and needs to be updated as new transactions are added.

Our solution. The main goal of this chapter is twofold. First, we want to *improve the privacy* of Bitcoin lightweight clients *without compromising their performance*, and second, we want to create a scheme that enables efficient *privacy-preserving light clients for Zcash*.

To reach this goal, we combine techniques from several separate fields, including trusted computing, private information retrieval and side-channel protection making the processing of client queries oblivious towards a powerful adversary controlling the supporting server. We stress here that a naive composition would result in a poor trade-off between privacy and performance. The primary problem that we solve in this chapter is how to combine known and new techniques such that the resulting solution provides strong privacy, good performance and easy adoption at the same time.

Our approach follows the common "light client and server" model, thus minimizing the client bandwidth and computation requirements by offloading processing to the server. To tackle the privacy problem of client queries, we leverage commonly available trusted computing capabilities, specifically Intel SGX enclaves, on full nodes. As described earlier SGX enables the execution of enclaves in isolation from any untrusted software such as the OS and protects the integrity of enclave execution and the confidentiality of enclave data. Thus, a potentially untrusted entity can run a full node with an SGX enclave that privately serves clients' transaction verification requests.

Although this approach is conceptually simple, realizing it securely requires overcoming technical challenges. First, *external* reads and writes from the SGX enclave to the blocks stored on the server or to response buffers can leak which transactions belong to the client. Second, SGX enclaves are susceptible to side-channel attacks, where malicious software on the same platform infers secret-dependent data access patterns or control flow in the enclave by monitoring usage of shared hardware-resources, such as caches [39, 40, 71, 151, 170, 204] that can leak their *internal* memory access patterns. Third, our system also needs to ensure that the residing platform cannot mount a combination of eclipse attacks [88, 202] on the blockchain and replay client messages to identify queried transactions.

We propose two new systems. We call the first one, focusing on Bitcoin, BITE (for ***BI****tcoin lightweight client privacy using* ***T****rusted* ***E****xecution*). The second one, focusing on Zcash, we call ZLITE (for ***Z****cash* ***Li****ghtweight clients for shielded transactions using* ***T****rusted* ***E****xecution*).

With such enclave leakage in mind, we design two BITE variants. Our first variant is called *Scanning Window* and its operation is similar to the current SPV clients that verify transactions using block headers and Merkle paths received from the full node. To prevent leakage through external data access patterns, we design a customized chain access mechanism that hides the client's transactions and the relationship between the size of the response and the number of read blocks. Our second variant is called *Oblivious Database* and it allows the client to verify the amount of coins associated with its addresses by querying a specially-crafted version of the unspent transaction output (UTXO) database. To prevent leakage from database accesses, we leverage a well-known Oblivious RAM (ORAM) algorithm [179] to hide access patterns to an encrypted storage. This second variant allows even *lighter* clients that no longer need to download and verify Merkle paths.

ZLITE follows similar principles of the Scanning Window variant of BITE. However, due to the specific privacy properties of Zcash, direct application of BITE is not feasible. Zcash transactions are encrypted, and thus, for the full node to even be able to assist, oblivious decryption and checks with the appropriate viewing key of a particular client is needed. Additionally, we need to efficiently provide the client with up-to-date Merkle tree witnesses needed to spend funds from a given transaction without leaking any private information. ZLITE further on presents a new commitment tree update mechanism that allows the client to obtain efficiently from the server all the needed information to spend the received funds.

To prevent software-based side-channels, we adopt protections from recent SGX research. The basic building block for our control-flow hiding is the `cmov` instruction [8] that enables building oblivious execution of branches. To prevent leakage from data accesses we adopt additional defenses, such as iterating over the entire data structure when an element is accessed based on the protected client address. In our use case, full nodes need to process large blockchain databases to serve client request, and thus straightforward usage of known SGX side-channel protection systems, such as [39,77,162,167], would result in either excessive performance overhead or imperfect side-channel protection. Instead of using such systems directly, we carefully pick low-level primitives and apply them at critical points in our system.

We argue that BITE emerges as the *first* practical solution that provides strong privacy protection for lightweight Bitcoin clients like mobile devices. On top of that, ZLITE represents the first and, currently, only solution that enables light clients in Zcash with respect to the privacy requirements of the cryptocurrency itself. Our solutions can be integrated into existing full nodes, and with minor modifications, to the existing software.

Contributions. In summary, we make the following contributions:

• *Novel approach.* We propose leveraging commonly available trusted execution capabilities of SGX enclaves for improved lightweight Bitcoin client privacy and the emergence of viable Zcash light clients.

• *New systems.* We design and implement two systems called BITE and ZLITE that carefully combine a number of known and new private information retrieval along with side-channel protection techniques to prevent information leakage.

• *Evaluation.* We show that our solution significantly improves both privacy and performance of current Bitcoin clients, and enable Zcash light clients while keeping the intrinsic properties of the cryptocurrency. We argue that BITE and ZLITE are the first practical ways to provide strong privacy for lightweight clients.

The rest of this chapter is organized as follows. Section 5.2 provides background information. Section 5.3 describes our problem and Section 5.4 outlines our approach. Sections 5.5, 5.6 and 5.7 explain the details, security and performance analysis of our system BITE, while Sections 5.8, 5.9 and 5.10 respectively cover ZLITE. We discuss directions for future work in Section 5.11, review related work in Section 5.12, and conclude in Section 5.13.

5.2 Background

In this section we provide a brief background on cryptocurrency transactions, their operational modes, full nodes, lightweight clients, and the oblivious RAM technique.

5.2.1 Transactions in Cryptocurrencies.

Bitcoin. Bitcoin [153] is the first and still most popular cryptocurrency based on blockchain technology. It enables users to perform payments by issuing transactions. While Bitcoin enables execution of a simple scripting language, regular Bitcoin transactions generally transfer Bitcoins (BTC) from one or more transaction inputs to one or more outputs. Each of the outputs is bound to an *address* that is derived from a user's public key. A user that knows the corresponding private key will then be able to spend the Bitcoin contained in the transaction output.

When a user wants to perform a payment, she creates a transaction that contains inputs, outputs, and the signatures that allow her to spend the inputs. Subsequently, the transaction is propagated to all nodes using a peer-to-peer network created by the system's participants. Miners, a special type of nodes,

collect valid transactions into blocks and solve a hash-based Proof-of-Work puzzle to make the contained transactions hard to revert. A miner that successfully finds a valid Proof-of-Work for a candidate block, broadcasts the block to all other nodes, who then verify its correctness and include it in their copy of the blockchain if valid.

In general, Bitcoin and many other cryptocurrencies operate in the so called Unspent Transaction Output (UTXO) model. In this model, a transaction consists of a set of outputs, each with a numerical amount of money and an *address*, and a set of inputs each of which references the output of a previous transaction. For a transaction to be valid, the following conditions have to be met: (1) referenced outputs must exist, (2) inputs must be signed by the key specified in the referenced output address, (3) the $\sum$(output amounts) must be $\leq$ to the $\sum$(input amounts), and (4) referenced outputs must not be spent by a previous transaction.

In Bitcoin, this is accomplished by directly identifying the referenced outputs, checking that they are not referenced by any other transactions, and then checking the sum inputs and outputs. If a transaction validates, then the outputs it references are removed from the UTXO set and the outputs it generates added. Transactions in most cryptocurrencies are validated via a peer-to-peer network and assembled into blocks (e.g., every 10 minutes), that are broadcast to the network.

Zcash. In Zcash there are two types of transactions: transparent and shielded transactions. The transparent kind is directly derived from Bitcoin. When mentioning Zcash in the rest of this chapter we solely focus on the shielded transactions due to their specificity while the approach related to the unshielded transactions is the same as we are discussing Bitcoin transactions.

Shielded transactions also take some inputs and create new outputs, but the similarities end there. Outputs, also called notes, are created by so called *joinsplits* and are a commitment to an amount and the address it belongs to. A *joinsplit* takes a transparent input and up to two notes as input and creates one transparent output and up to two notes as output. However this information is encrypted and can only be inspected by the receiver. Additionally a Merkle tree is constructed over all notes on the blockchain forming the *note commitment tree*. A zero-knowledge proof forms the second part of the transaction and shows that above mentioned conditions (1)-(3) for transaction validity hold with respect to that Merkle tree root. Because the "outputs" that a shielded transaction spends are not revealed, they cannot be removed from the UTXO set. Instead, a unique serial number, sometimes called a *nullifier*, is produced by the transaction that ensures the referenced outputs cannot be used again. This prevents double spending.

To perform operations in Zcash, each user has two keys associated to his shielded address. First, the *spending key* that is used during the creation of a zero-knowledge proof allowing the users to prove ownership of the received funds. Second, the *viewing key* that is used to decrypt the shielded transaction in the blocks and verify if each transaction belongs to the user.

5.2.2 Full Nodes and Light Clients.

To interact with a cryptocurrency, one must have a client. In both Bitcoin and Zcash, the default client is a *full node*, which receives and validates every block, and contains the full state of the blockchain. Full nodes do not need to trust other entities, provided the system functions as assumed, e.g., for Proof-of-Work systems the majority of the network's computational power is honest and messages disseminate without problems. While full nodes offer the best security and privacy, they entail considerable resource usage. The computation and network resources necessary to maintain a full node are a major impediment and in some cases, e.g. mobile devices, simply prohibitive.

In order to verify transactions, Bitcoin users, or clients, need to store the full history of all Bitcoin transactions. This approach puts a heavy load on client implementations in terms of network and storage, and as a consequence, makes transaction confirmation on mobile clients infeasible. To address this concern, the original Bitcoin paper proposed a solution called *Simplified Payment Verification* (SPV) [153]. In SPV, a new type of clients, called lightweight clients, are introduced. These clients store only block headers that are sufficient to check the Proof-of-Work on each block and verify the presence of transactions by checking their inclusion in the Merkle tree whose root is contained in the block header, thus having a smaller resource footprint.

The reduced resource usage of SPV clients comes at a major cost: privacy. As the light client must request individual transactions from a full node, it reveals which transactions and addresses belong to her. In Bitcoin, this allows multiple addresses to be linked together. In Zcash, this effect is far more pronounced since the client completely loses privacy: without such queries, no shielded transactions can be linked together (adversary learns nothing).

Improvement proposal BIP 37 [87] introduced Bloom filters [35] that allow a lightweight client to request a subset of all transactions to preserve some privacy without needing to download all transactions for each block. A Bloom filter [35] is a probabilistic data structure that consists of a set of hash functions and a bit array where each bit is set to one if one of the hash functions hashes one of its inputs to the index of the bit in the array. This data structure allows checking if a value is contained in the filter by hashing the value with each of the hash functions and checking whether the corresponding bit is set. If this is

not the case, the value was not an input. If it is the case, however, the value *might* have been an input or it could be a false positive. The false positive rate can be set by the creator of the filter.

In Bitcoin lightweight clients, Bloom filters are used to encode transactions or addresses, and allow a full node to determine which transactions to send to a lightweight client without letting the full node know the exact addresses. A lightweight client prepares a Bloom filter to which she adds all of her addresses and sends it to the full node. The full node then checks for incoming (or past, if requested) transactions whether they match the Bloom filter. If they match, she sends them to the client together with the Merkle path needed for verification. The client can adjust the false positive rate to increase her privacy. If the false positive rate is higher, the client will receive more irrelevant transactions, in an attempt to hide her true addresses with a larger anonymity set. The usage of Bloom filters is not possible with Zcash since all shielded transactions are encrypted, thus obfuscated from the full node.

5.2.3 Oblivious RAM

Encryption provides data confidentiality but access patterns can leak information possibly leading to reconstruction of the content itself. Oblivious RAM (ORAM) [69], is a well-known technique that hides access patterns to an encrypted storage medium. A typical ORAM model is one where a trusted client wants to store sensitive information on an untrusted server. Encrypting each data record before storing it on the server provides confidentiality, but access patterns to stored encrypted records can leak information, such as correlation of multiple accesses to the same record. The intuition behind the security definition of ORAM is to prevent the adversary from learning anything about the access pattern. In ORAM, the adversary does not learn any information about which data is being accessed and when, whether the same data is being repeatedly accessed (i.e., unlinkability), the pattern of the access itself, and lastly the purpose, type of the access (i.e., write or read). However, one should note that ORAM techniques cannot hide access timing.

In this chapter, we use a popular and simple algorithm called Path ORAM [179] that provides a good trade-off between client side storage and bandwidth. The storage is organized as a binary tree with buckets containing Z chunks each. The position of each chunk is stored in a *position map* that maps a database entry to a leaf in the tree, and for every access the leaf of the accessed entry is re-randomized. A small amount of entries is stored in a local (i.e., memory) structure – *stash*.

Every access involves reading all buckets of a path from the root to a leaf into the *stash* and then writing back new or old re-randomized data from

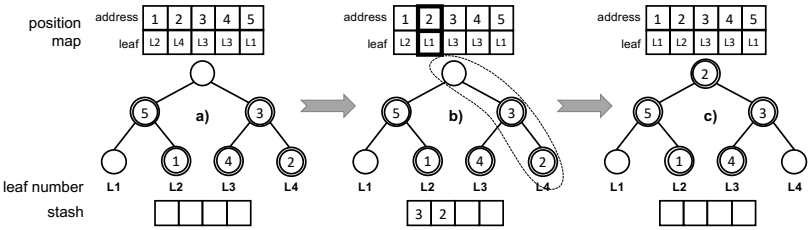

Figure 5.1: *a) The client wants to access the chunk 2 that is stored in Path ORAM. b) The position map specifies that the chunk 2 is on the path to leaf 4. Therefore, the server reads all entries on the path into the stash and re-randomizes the position map entry of the requested chunk. c) The server writes back as many chunks as possible on the previously read path.*

the *stash* to the same path resulting in an overhead of $O(\log N)$ read/write operations. If the requested chunk is already in the stash, an entire path still gets read and written. The summary of ORAM operations is:

(**1**) get leaf from *position map*

(**2**) generate new random leaf for the database entry and insert it into the *position map*, then read all buckets along the path to the leaf and put them into the *stash*

(**3**) if access is a write, replace the specified chunk in the stash with the new chunk

(**4**) write back some chunks from the *stash* to the path. Chunks can only be put into the path if their leaf from the *position map* allows it. Chunks are pushed down as far as possible into the tree to minimize *stash* capacity.

(**5**) return requested chunk

Figure 5.1 shows an example of data access in Path ORAM.

5.3 Problem Statement

In Bitcoin, the use of Bloom filters to receive transactions from an assisting full node inherently creates a trade off between performance and privacy. If a client increases the false positive rate she receives more transactions which provides increased privacy, as any of the matching addresses could be her real addresses, but it also means that she needs the network capacity to download all of these transactions. In the extreme cases, the filter matches everything, i.e., the client downloads the full blocks, or the filter only matches the client's addresses, i.e., she has no privacy towards the full nodes.

In addition, Gervais et al. [68] have shown that the use of Bloom filters in Bitcoin lightweight clients leaks more information than was previously thought. In particular, they showed that if the Bloom filter only contains a moderate number of addresses, the attacker is able to guess addresses correctly with high probability. For example, with 10 addresses the probability for a correct guess is 0.99. They also show that, even with a larger number of addresses, the attacker is able to correctly identify a client's addresses with high probability if she is in possession of two distinct Bloom filters from the same client (e.g., due to a client restart). Hearn [86] later expanded on why solving these issues is hard (e.g., need for resizing). Furthermore, it is likely that an attacker using additional de-anonymization heuristics, such as the ones described in [21, 147], could further increase the probability to guess correctly.

Finally, a lightweight client cannot be sure that she receives *all* transactions that fit her filter from a full node. While the full node cannot include faulty transactions in the response, as this would be detected by the client when recomputing the Merkle root, the client cannot detect whether she has received all requested transactions. This problem can be solved by requesting transactions from multiple nodes, which again imposes more network load.

In contrast, while there is no effective way to completely preserve privacy for Bitcoin lightweight clients, a scheme for Zcash lightweight clients does not exist at all. The only way that a Zcash client can operate in a light way is to offload its viewing (decryption) key to a full node, in order for the full node to check the blockchain and deliver any matching transaction. However, in this case the main goal and idea of Zcash are forfeit.

5.4 Our Approach

5.4.1 Requirements

The main goal of this chapter is to design a solution that enables privacy-preserving light clients assisted by full-node servers for both Bitcoin and Zcash without compromising performance. More precisely, we specify the following requirements for our solution:

R1 Privacy. Bitcoin light clients should be able to verify that their transactions are confirmed on the blockchain or check the amount of coins associated with their addresses without revealing their addresses to the potentially untrusted entity that controls the assisting full node. Zcash light clients should be able to privately retrieve all transaction related data without revealing sensitive information (e.g., viewing key, transaction count, blocks containing transactions) to the server.

R2 Integrity. The server that is assisting the light client should not be able to steal funds or make a client falsely accept a payment.

R3 Completeness. The retrieval of transactions should guarantee that the light client receives *all* data necessary for updating the balance and spending the funds they received.

R4 Performance. The solution should have minimal bandwidth and processing requirement for the client (for Zcash) that is comparable or better then the current solution (for Bitcoin). The server's processing should be in the same order of magnitude as the normal full node operation.

5.4.2 Main Idea

Our main idea is to leverage commonly-available TEEs and apply them to full nodes (servers) to enable privacy-preserving light clients for both Bitcoin and Zcash. In particular, we use Intel's SGX as explained in Chapter 2.

Similar to SPV in Bitcoin, we assume deployments where the light clients may be assisted by *any number of* servers (full nodes) that support TEEs. Some of the servers could be run by well-known companies as commercial services where light clients may have to pay a small fee for the service. Other servers could be run by private individuals, like members of the cryptocurrency community, as a free service. As in SPV, the light clients are free to choose which servers to use, if any. In this regard, our solution retains the decentralized nature of the Bitcoin and Zcash cryptocurrencies.

5.4.3 Controversy of TEEs

The use of TEEs is often controversial. TEEs rely on a trusted authority to design a secure processor and issue some form of certification for it. TEE attestations can be forged either via exploiting design flaws or by corrupting the provider and falsely claiming that an attestation came from a genuine piece of hardware. The hardware and software are frequently closed source and the manufacturers' opaque. These kind of trust assumptions are frequently an anathema, especially for cryptocurrencies. Moreover, usage of TEEs often seems like lazy systems design choice, since, if one assumes fully trusted TEEs (e.g., enclaves cannot be compromised, no side-channel leakage, full resilience to physical attacks), solving many problems becomes relatively easy.

However, current TEEs including SGX enclaves have noteworthy limitations such as side-channel attacks that leak information and no resilience to physical attacks. We argue that the real research challenge is to leverage TEEs such that one can enable improved performance and privacy, but at the

same time address the limitations of TEEs such as side-channel leakage. In the (unlikely) case that TEEs are fully broken (e.g., a new severe processor vulnerability is discovered), the system should fail gracefully. In our example, we achieve graceful failure so that the affected clients' privacy may be reduced, but integrity is preserved, i.e., in a cryptocurrency, no money is lost or stolen.

5.4.4 Adversary Model and Challenges

We consider the standard SGX adversary model where the attacker controls the OS and all other system software in the supporting server running the full node. In practice, the adversary could be a malicious administrator in a company that provides the full node service, an external attacker that has compromised the OS on the full node server, or a malicious individual operating a free server.

Since the adversary controls the OS, she can schedule and restart enclaves, start multiple instances, and block, delay, read, or modify all messages sent by enclaves, either to the OS itself or to other entities over the network. Additionally, the adversary is able to perform digital side-channel attacks [39, 40, 71]. We assume that she is able to *perfectly* observe the enclave's control flow with instruction-level granularity and its data accesses with byte-level granularity (best known attacks are cache-line granularity). We overestimate the attacker capabilities, as all current side-channel attacks suffer from significant noise and cannot extract perfect traces in pratcice. By assuming such an adversary, we design our solution for future attacks that may be able to mount more precise side-channels. Additionally, the adversary has full control over the communication and can thus read, modify, block or delay all enclave messages.

The adversary *cannot* break the hardware security enforcements of Intel SGX along with cryptographic primitives such as encryption schemes and signatures. That is, the adversary cannot access processor-specific keys (e.g., attestation or sealing key) and she cannot access enclave's runtime memory that is encrypted and integrity-protected by the CPU. (Although we consider SGX trusted, in Section 5.6 and 5.9 we discuss enclave compromise and show that both of our solutions for Bitcoin and Zcash can handle it gracefully.)

In summary, secure and practical realization of our approach under the defined attacker model involves several technical challenges:

Leakage through external accesses. Since the adversary controls the OS, she can observe access patterns to any *external* resources, such as files or databases stored on the disk. Although externally stored data can be sealed (encrypted by the CPU such that only the same enclave can decrypt), the OS may be able to infer information about the accessed element by observing access patterns to individual records, such as files or database entries.

Similarly, enclaves rely on the OS to perform communication operations which allows it to infer information about the communication patterns of the enclave. Even if messages are encrypted by the enclave, the message sizes, frequency and destination can leak information to the OS.

Leakage through side channels. The SGX architecture can also be susceptible to *internal* leakage. While Intel acknowledges the possibility of side-channel attacks on enclaves [101], they consider it out of scope for the SGX adversary model. However, recent research shows that such attacks are practical and need to be taken into account. For example, by monitoring usage of shared hardware resources, such as CPU caches, the OS may be able to mount software-based side-channels and infer secret-dependent data and code accesses inside the enclave's memory [40, 71, 151, 170]. In SGX, the memory management is left to the untrusted OS, and therefore the OS may also be able to infer enclave's secrets by monitoring the memory pages that the enclave requests from the OS [204]. Researchers have also demonstrated side-channel attacks using the CPU's branch prediction functionality [128] and speculative execution (the Spectre attack) [49].

Known side-channel attacks can be classified with respect to which memory content is targeted. Code monitoring can identify secret-dependent execution paths, that is, control flow. Data access monitoring can identify secret-dependent data object usage. Branch prediction attacks [128] target execution paths, while most demonstrated cache attacks target data accesses [40, 71, 151, 170], although cache attacks can target control-flow as well.

5.4.5 Strawman Solutions

In case of Bitcoin, a simple way to leverage SGX would be a solution, where the lightweight client sends its wallet private key to an enclave on the assisting full node. Using the wallet key, the enclave can perform any operation on behalf of the user, including transaction verification. However, such simple solution has a critical drawback. If the used enclave is compromised, the adversary can steal all user's coins. Such approach might give the owners of full nodes an undesirable economic incentive to break their own SGX processors, e.g., using physical attacks.

To avoid such incentives, we choose a different approach. In our solution, when a client needs to verify a transaction or check the amount of coins associated with the user's addresses, the client connects to one of the full nodes that supports our service. The client performs remote attestation and establishes a secure channel to the enclave. Then, the lightweight client sends the addresses that the user is interested in to the enclave. The enclave obtains all the required

verification information from the locally stored blockchain or custom unspent transactions database (UTXO) and sends back a response to the client that can verify it. Importantly, the client's private key is never shared with the enclave which enables safe adoption of our solution.

In case of Zcash, if client privacy relies on TEEs, it becomes natural to ask if one needs a complicated solution like Zcash and if anonymous payments can be realized through a much simpler solution using TEEs. To answer this question, we consider the limitations of a few strawman solutions.

Our first strawman solution is that clients send all transactions in an encrypted format to a set of authorized TEEs that process them privately. Such a solution would protect user privacy, but in case the enclaves get broken, the adversary can perform unlimited double spending on all users. Additionally, such a solution would not be decentralized.

Our second strawman solution is to use pseudonymous transactions that are published to a permissionless ledger, similar to Bitcoin, and mix them in one or more TEEs for improved privacy. Such a solution would prevent double spending, ensuring security for all users, even in the event that TEEs are broken. However, such a solution does not provide the same strong privacy protection, namely *unlinkability*, as Zcash, since the anonymity set for a transaction output only consists of the inputs of the mixed transaction. An adversary controlling the OS on the mixing service can further reduce anonymity by blocking incoming transactions or injecting his own.

Our third strawman solution follows the outlined Bitcoin strawman solution, and includes using the Zcash system, due to its strong privacy properties, but allowing light clients to offload their complete wallets to TEEs that perform new payments and notification of received payments for them. The main drawback of this approach is that if the TEE would be compromised, it would incur direct monetary loss for a high number of clients.

Goal. Our goal is to design a solution that offers full privacy for Bitcoin light clients and enables light clients for Zcash (still benefiting from Zcash's sophisticated privacy protections), but avoids the above discussed limitations of simple TEE-based solutions.

5.4.6 Solution Overview

In our solution, when a light clients wants to be notified about received funds or make new payments, she connects to one of the TEE-enabled full node servers, performs remote attestation of the server's SGX enclave, and establishes a secure channel using TLS to it.

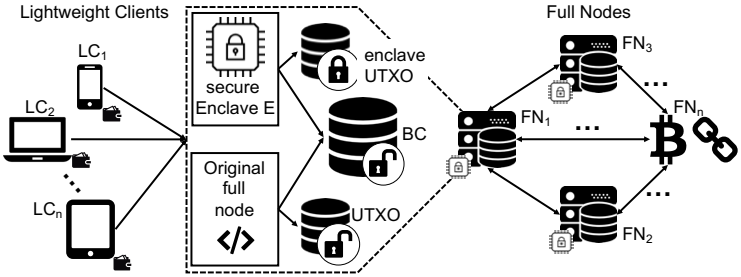

Figure 5.2: ***System model.*** *Lightweight clients request transaction verification service from enclaves hosted on full nodes.*

To enable payment notification, the client sends its addresses (in case of Bitcoin) or *viewing keys* for the addresses (in case of Zcash) that she owns to the enclave and indicates from which point on (e.g., the latest known block to the client) she wishes to update the light client's state. The enclave obtains the data and information from the locally stored blockchain and processes it in a *side-channel oblivious* manner based on the client request and sends back the response to the client. Additionally, to enable new Zcash payments by the client, the server also prepares a witness for each new transaction of the client, as well as the note commitment tree update and sends them to the client. Given this information, the client can efficiently create new transactions, and the associated zero-knowledge proofs, using the received funds, without revealing his *spending key* to the enclave.

In the next sections we first describe the design and implementation of the BITE system, along with the security and the performance analysis, followed by ZLITE system details in the same respective fashion.

5.5 BITE: System

In this section we first present a system model and then a system called BITE that realizes the above approach securely for Bitcoin Lightweight Clients and addresses the aforementioned challenges.

5.5.1 System Model

Figure 5.2 shows our system model that consists of full nodes $FN_1 ... FN_m$ and lightweight clients $LC_1 ... LC_n$. When a lightweight client LC_i wants to acquire information about its transactions or addresses, it can connect to any full node FN_j that supports our service and hosts an enclave E_j. Full nodes download and store the entire blockchain (BC) locally and based on that maintain a

database that contains all unspent transaction outputs (UTXO). Our system additionally maintains a specially-crafted version of the UTXO, called *enclave UTXO*, in an encrypted (sealed) form.

In SGX, enclave memory is limited to 128MB. Although swapping memory pages is supported (swapping requires expensive encryption and integrity verification [24]), the complete blockchain (BC) and the database of unspent transaction outputs (UTXO) are significantly larger than the enclave's memory limits (160GB or more). Therefore, these databases are stored on local persistent storage such as disk outside enclave's memory.

5.5.2 Operation Overview

Our system BITE can be presented through two variants of the system that serve slightly different purposes.

Our first variant is called *Scanning Window* and it can be seen as an extension to the current SPV verification mode, but without reliance on bloom filters. Based on the client request, an enclave on the full node *scans* the blockchain and replies with a set of Merkle paths that the client can use to verify its transactions using downloaded block headers. This variant allows the client to check that each of its transactions are confirmed on the blockchain. As Bitcoin provides only eventual consensus, the client may want to additionally verify that the blocks where its transactions are placed have been extended with a sufficient number of valid blocks (e.g., six).

Our second variant is called *Oblivious Database* and it can be seen as a completely new verification mode for lightweight clients. In this variant, the enclave on full node maintains a specially-crafted version of the unspent transaction outputs (UTXO) database and when a client sends a verification request, it checks for the presence of client's outputs in this database using oblivious database access (ORAM [179]) and responds accordingly. Such verification allows the client to check how many coins is currently associated to its addresses, with significant performance improvements over SPV.

In both variants, the client performs remote attestation on the used enclave and establishes a TLS connection to it. We note that current lightweight clients communicate with the full nodes without encryption. Existing full node functionality, such as participation in the P2P network and mining, remain unaffected. Therefore, our system can be seen as a simple add-on to existing full nodes. For clients, payment execution remains unchanged. Payment verification requires minor additions (attestation and TLS) when Scanning Window variant is used or slightly bigger changes when Oblivious Database variant is used.

5.5.3 Scanning Window Variant

In our first variant, we want to improve the privacy of the current SPV verification mode. When a client needs to verify transactions, it constructs a request that specifies the addresses of interest and the last block that it has in its internal state and sends that to the secure enclave residing on the full node. The enclave reads the locally stored blockchain database using a custom scanning technique that normalizes the relationship between response sizes and actually accessed data to hide the data/block access patterns and ensure client privacy. Figure 5.3 shows the operation of this variant, and we describe the details as follows:

Initialization and continuous operation.

(**a**) On initialization the Full Node FN_j connects to the P2P Bitcoin network (**a-1**) and downloads the full blockchain (**a-2**). Similarly, the locally stored blockchain database is updated for each new blocks that is appended to the chain (i.e., as new blocks are received over the P2P network).

(**b**) The lightweight client installation package includes a checkpoint block header from a recent date. When the client is started for the first time, it downloads all newer block headers from the peer-to-peer network and verifies that (i) they all have correct Proof of Work and (ii) the hash chain of the downloaded headers leads to the checkpoint. Once the client's internal state is synchronized with the peer-to-peer network, it stores a small number of the newest headers (e.g., six blocks from the head of the chain to handle shallow forks). The client can update its internal state by downloading newest block headers periodically or before each transaction verification request. The network and storage requirements of this process are minor and easily met even by clients with severe resource constraints.[1]

Synchronization of Transactions. Clients perform transaction verification as follows:

(**1**) The Lightweight Client LC_i performs attestation with the secure Enclave E_j residing on the full node FN_j.

(**2**) If the attestation was successful, the Lightweight Client LC_i establishes a secure communication channel to the Enclave E_j using TLS.

(**3**) The Lightweight Client LC_i sends a request containing the addresses of interest and a block number that specifies how deep in the chain transactions

[1] For example, obtaining block headers for a checkpoint that is one month old, would require 300 kB of downloaded data (one-time operation) and updating the block headers once per day would require 10 kB of communication per day. Storing latest six headers takes less than 1 kB of storage.

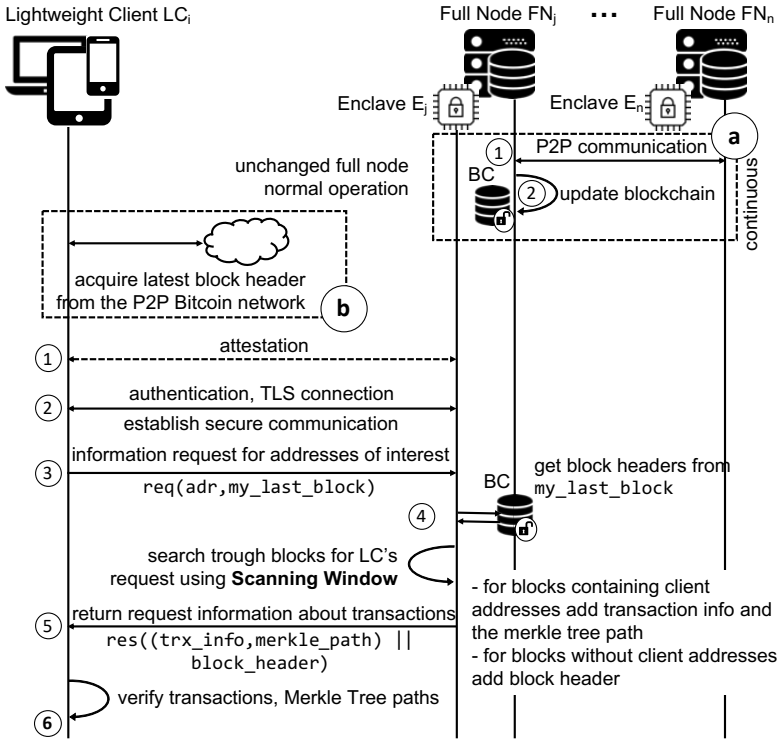

Figure 5.3: ***Scanning Window operation.*** *Light client establishes a secure connection to an enclave on full node and sends a request that contains its address and last known block. The enclave scans a number of blocks from the locally stored complete chain and prepares a response.*

should be searched for verification. Typically, this number would be saved from the previous interaction with a full node or in the case of the first transaction verification the number could roughly match the date when the client started using Bitcoin.

(**4**) The Enclave E_j starts *scanning* its locally stored copy of the blockchain (BC) for the requested address and range of blocks using a scanning technique described in detail below.

(**5**) In preparation of the response, the Enclave E_j does the following: for blocks containing client addresses it adds the full transaction information and

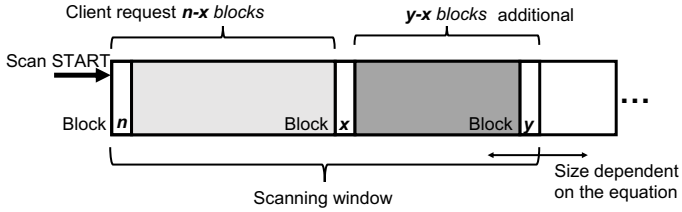

Figure 5.4: ***Block reading in Scanning Window****. Depending on the number of requested blocks (up to x) and the number of matching transaction in them, our scanning technique reads potentially extraneous blocks (up to y) to keep the ratio between the read blocks and the response message size constant.*

the underlying Merkle tree path to the response, while for blocks without client addresses it only adds the block header.

(**6**) The Lightweight Client LC_i verifies that (i) the received block headers match its internal state and (ii) the received transactions and Merkle Tree paths match to the block headers. The client considers such received transactions as confirmed (assuming that they are sufficiently deep in the chain). The client updates its internal state regarding the latest verified block number and closes the connection to the enclave.

Block scanning details. As explained in Section 5.4.3, enclave execution can leak information in various ways. For example, if our solution would simply return each matching transaction (and the corresponding Merkle Tree) in the specified range of blocks, based on the size of the response the adversary could deduce how much information of interest for the client was contained within the scanned blocks. Over a period of time, by tracking requests and response sizes, the adversary could gain significant information about the client's addresses and transactions.

We address such leakage by using a tailor-made block scanning scheme. The main goal of the scheme is to fully hide the ratio between the response size (that indicates the number of transactions returned to the client) and the number of scanned blocks. When this ratio is constant, the adversary cannot deduce any meaningful information from the response size.

Figure 5.4 depicts the details of our scanning scheme. The newest block in the blockchain observed by the Bitcoin network is n. A clients request contains an addresses of interest and the number block x indicating how deep the chain should be scanned. The enclave starts scanning from n and moves towards x. It stores intermediate responses and when it reaches block x it

performs a check. The total size of the response, r, is divided by the threshold size, t. The threshold indicates the maximum response size per block such that if we are to scan $n-x$ blocks, the maximum response size for the client can be $r = (n-x) * t$. If the given response size r is greater, then the enclave has to scan up to block y (or $y-x$ more blocks), such that $r = (n-y) * t$. If the response size is smaller, i.e., if after scanning $n-x$ blocks $r \leq (n-x) * t$, we pad the response size such that $r = (n-x) * t$. The exact size of the threshold is empirically determined in Section 5.7.

Side-channel protection. The scanning technique described above prevents leakage from externally observable response sizes. However, if the adversary is able to mount a high-granularity digital side-channel attacks (e.g., one that allows her to observe execution paths with instruction-level granularity), the adversary will be able to determine the transactions that were accessed, and thus infer the client's addresses.

To make our system more robust against such attacks, we optionally add side-channel protections at the expense of performance (cf. Section 5.7). To protect against timing leakage we compute the Merkle path for all transactions in each of the scanned block in contrast to only computing the path for the transactions of client's interest. For protection against control-flow side channels we make use of the `cmov` assembly instruction to hide execution paths. `cmov` is a conditional move such that "*If the condition specified in the opcode (cc) is met, then the source operand is written to the destination operand. If the source operand is a memory operand, then regardless of the condition, the memory operand is read*" [8]. This allows us to replace branches from our code resulting in the same control flow with no leakage.

The same technique is also used in previous side-channel protection solutions like Raccoon [162]. However, since using such a side-channel defense system directly would incur an extremely high performance overhead in our particular setting (due to large amounts of accessed blockchain data), we customize these techniques to our setting. Specifically, we apply the following modifications, as per Figure 5.5:

(**i**) Instead of continuing to scan the chain if the size of the response exceeds the threshold, we stop scanning after the specified number of blocks. If not all transactions fit in the response, the client does not receive all transactions and is informed of this through a flag in the response. This allows the allocation of a response array that does not change size during processing. The client can request the remaining transactions in another request (from the flagged block until the end of the initially specified block).

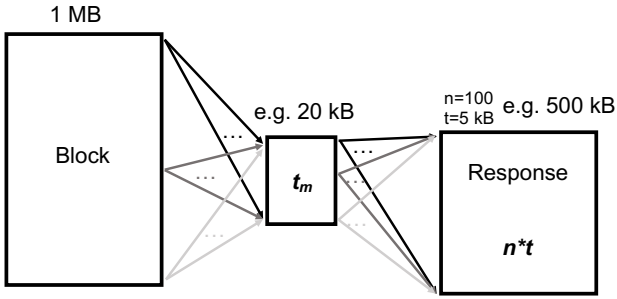

Figure 5.5: ***Oblivious copying in Scanning Window.*** *The data is copied in an oblivious fashion from the block to a temporary array, i.e., every transaction is conditionally moved using* `cmov` *to every possible destination. The data contained in the temporary array is then copied to the response in an oblivious fashion, again using* `cmov` *to conditionally copy everything to all possible locations in the response.*

(**ii**) For each block, we allocate a temporary array of size t_m (see Figure 5.5), where t_m is a threshold that specifies the maximum data per block, as opposed to the threshold t that specifies the average data per block. While the block is parsed, each transaction is moved to the temporary array in an oblivious fashion, i.e., we use the `cmov` instruction to conditionally move each word of each transaction to every entry in the array. This means that for every transaction we access every entry in the array and since the same instruction is used for each possible copy – independent of whether the data is actually copied – even an attacker with an instruction level view of the control flow cannot determine which data is actually copied. After processing the block, the temporary array is traversed and all entries are copied to the response array (see Figure 5.5). This is again done in an oblivious fashion, i.e., each entry is copied conditionally using the `cmov` instruction to every possible position in the response array.

This method of copying transactions from the block to the response is required to efficiently keep the data accesses oblivious. Specifically, for a block of size m, a temporary array of size t_m and n requested blocks, this method requires $\mathcal{O}(m \cdot t_m + t_m \cdot n \cdot t)$ operations instead of $\mathcal{O}(m \cdot n \cdot t)$ operations when naively copying the data in an oblivious fashion from the block to the response directly. Since t_m is usually much smaller than m and $n \cdot t$, this method is in practice orders of magnitude faster, in relation to the data copy in oblivious fashion.

5.5.4 Oblivious Database Variant

In our second variant, we focus on reducing the load of lightweight clients in terms of computation and network while offering even better privacy preservation (namely, the block number that specifies how deep the chain should be searched does not leak). The main idea behind this variant is to allow lightweight clients to send requests containing addresses of their interest and directly receive information regarding unspent outputs, without the need to verify block headers and Merkle tree paths. A comparison of security properties between each of the BITE versions is in Section 5.6.

In order to achieve such verification, a new indexed database of unspent transactions (denoted as *enclave UTXO*) is created and searched for every client request using an Oblivious RAM algorithm. Figure 5.6 shows the operation of this variant, and we describe the details as follows:

Initialization and continuous operation.

(**a**) Similar to a standard full node, on initialization the full node FN_j connects to the peer-to-peer network and downloads and verifies the entire blockchain. After initialization, when new blocks are available in the peer-to-peer network, FN_j downloads and verifies them.

(**b**) During initialization Enclave E_j reads the locally stored blockchain and verifies each block. The enclave builds its own *enclave UTXO* database that is a special version of the original structure present in standard full nodes. In particular, this UTXO set is encrypted on the disk as sealed storage, indexed for easy and fast access depending on the client request, and accessed using ORAM to prevent information leakage through disk accesses. After initialization, the enclave updates this UTXO using ORAM when new blocks are available in the locally stored blockchain.

(**c**) As in the Scanning Window variant, the client obtains the latest block headers from the peer-to-peer network.

Synchronization of Transactions. Clients perform transaction verification as follows:

(**1**) The Lightweight Client LC_i performs an attestation with the secure Enclave E_j residing on the full node FN_j.

(**2**) LC_i establishes a secure communication channel to the E_j using TLS.

(**3**) LC_i sends a request containing the addresses of interest, along with the hash and number of the latest transaction known to the client. The last two parameters are needed in case the number of unspent outputs contained by an address is larger than the maximum size of the message. For example, LC_i receives the first response containing x transaction outputs with an indication

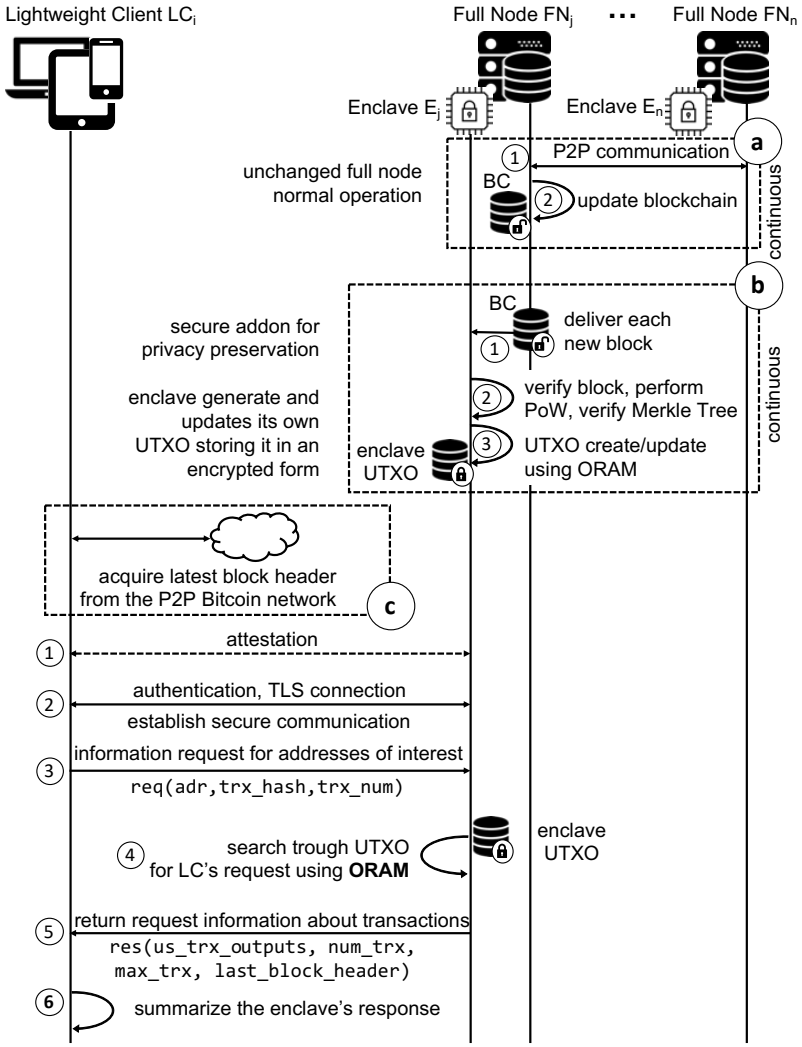

Figure 5.6: ***Oblivious Database operation.*** *Lightweight client sends a request containing its address and the last transaction to an enclave on full node. Enclave queries a specially-constructed version of the UTXO database using ORAM and provides a response back to the client.*

that there is more, and in a consequent request specifies the same address as in the first request along with the $x-th$ transaction hash and transaction number. This gives an indication to the enclave to respond with the second batch of outputs starting from that transaction. The process repeats (possibly with a different node) until the client is satisfied. To prevent information leakage through the message sizes, requests are always of constant size, i.e., the client pads shorter requests and splits up larger queries. The size is defined to accommodate the majority of requests. Since a lightweight client can choose any available node to connect to, she can choose to send requests to different nodes to hide the number of sent requests.

(**4**) The Enclave E_j reads the enclave UTXO database to get the unspent transaction output information in respect to the client's request. E_j uses ORAM and the previously created index to access the enclave UTXO in an oblivious fashion.

(**5**) In preparation of the response, E_j includes the relevant information as explained in step **(3)**, which encompasses the currently included and maximum number of unspent transactions found for a specific address. When these numbers match, the LC_i knows that she has received all the unspent outputs of a specific address. The enclave additionally includes the block hash of the last known block from the local blockchain (longest chain). With this information the client can deduce whether the enclave has been served with the latest block and that the enclave's database is fully updated. Responses are always of constant size, i.e., shorter responses are padded and if a response is too large, the client is informed of missing outputs, such that she can later retrieve the rest of the outputs (e.g., from a different node). The size of the response is chosen such that it accommodates the majority of responses.

(**6**) The Lightweight Client LC_i can summarize the unspent transaction outputs received from the Enclave E_j. The enclave guarantees completeness in terms of transaction confirmation and the current state of the chain, so the client does not have to perform any additional checks by herself. Successful update of the client's internal state results in the connection termination between the enclave and the client.

Oblivious Database details. In this variant, we use the Path ORAM [179] algorithm to protect data access patterns of our enclaves. For readers unfamiliar with this algorithm, we provide a brief description in Section 5.2.3.

Database Initialization. The ORAM database is initialized by creating dummy buckets on disk and filling the *position map* with randomized entries. The *stash* is also filled with dummy chunks. After that the ORAM database

is fully initialized and can be used to add new unspent outputs from the blockchain. To ensure that the enclave always uses the latest version of the sealed enclave UTXO database, SGX counters or rollback-protection systems such as ROTE (described in Chapter 3) can be used.

Database Update. When a new Bitcoin block is added, the enclave first verifies the proof of work. It then extracts all transaction inputs and outputs and bundles them by address. For each address found in the block, the UTXO database entry is requested and then updated with the new information. If too many entries are added, resulting in the chunk getting too big, the chunk is split into two and the index is updated to reflect the changes made to the UTXO database. All accesses are performed using the ORAM algorithm and, therefore, do not leak any information about the access patterns.

Database Access. Accesses to the ORAM database follow the normal procedure described in [179] and in Section 5.2.3.

Side-channel protection. While the usage of ORAM protects against all external leakage, side-channel attacks, and thus, internal leakage remains a challenge. If we consider the most powerful attacker that can perform all digital side-channel attacks (see Section 5.4.3), this variant would be forfeit due to the leakage of the code access patterns, specifically, execution paths in the *if* statements when the stash, indexes and the position map is being accessed. This would leak the exact address which is used to search for the unspent transactions in the internal database.

To remedy internal leakage, we deploy several mechanisms that protect our code and execution. First, when accessing the security critical data structures, specifically, the position map, stash, and the indexes containing information about which chunks contain unspent transactions of a certain Bitcoin address we pass over them entirely in the memory to hide the memory access pattern. Second, to hide the execution paths we remove all branching in the code that accesses these data structures and deploy the `cmov` assembly instruction (see Section 5.5.3). Observation of the control flow and memory access does not leak whether the operation performed by the enclave was a read or a write, and since there is a single control flow without creating multiple branches depending on the condition, we effectively hide the execution and thus protect this variant from internal leakage in full. This protection mechanism has negligible performance overhead (see Section 5.7).

5.6 BITE: Security Analysis

In this section, we provide an informal security analysis. First, we analyze our solution with respect to our adversary model where SGX security enforcements cannot be broken. In particular, we show that our solution ensures confidentiality of the requested client addresses, as the attacker cannot infer the requested address from disk access patterns, response sizes, side-channels, or a combination thereof. Next, we discuss completeness of the responses and finally consider the implication of a potential SGX compromise and show that our solution can handle such cases gracefully.

5.6.1 External Leakage Protection

Scanning Window. This variant scans complete blocks from the blockchain database, instead of accessing individual transactions within them, and thus prevents direct information leakage from disk access patterns. The constant ratio of response size to scanned blocks prevents information leakage from the response size. The adversary may only infer the number of blocks that are accessed and not which addresses are sent by the client or how many transactions are returned.

Oblivious Database. To protect against information leakage attacks on the disk access, our second variant utilizes the well-studied Path ORAM [179] algorithm. Our setting is slightly different than the typical client-server model considered in ORAM. In our case, the enclave corresponds to the client. Because the adversary can run the enclave freely, she can use it as an oracle, i.e., she can influence the data that is written (by delivering blocks to the enclave) and can query for values himself. Regardless of that, due to the unlinkability property of ORAM, the attacker learns nothing about what is accessed and the probability to guess correctly which ORAM block was accessed is equal to that of a random guess, as shown in [179]. As the responses always have a constant size, the adversary cannot learn anything from response sizes either.

5.6.2 Side-channel Protection

Most known side-channel attacks on SGX provide imperfect data-access or control-flow traces and require many repetitions to filter out noise [40, 71, 151, 170]. In BITE, queries from legitimate clients cannot be replayed due to the authenticated TLS channel and since the enclave is either stateless across power cycles or protected against rollback. The adversary can create his own client and send requests to the enclave, but this will not result in any advantage against legitimate clients. For these reasons, mounting side-channel attacks against BITE is more challenging than performing side-channel attacks against

enclaves in general. To analyze our solution against future adversaries that may be able to mount more precise side-channels, below we consider the worst case scenario, i.e., side-channel attacks that obtain perfect data access and control flow traces from enclave's execution.

Scanning Window. To harden our Scanning Window variant against side-channels, we provide optional protections that incur significant performance penalty. When the enclave scans through both the temporary array and the final response array in their entirety, it performs `cmov` operations for all possible transactions. This allows us to replace branches in our code with a single instruction resulting in the same control flow with no leakage to the attacker since all data is accessed and the same operation is executed every time.

Oblivious Database. For our Oblivious Database variant we always include side-channel protections to our solution, since the performance overhead is negligible. When accessing the security critical data structures such as stash, indexes and the position map, we pass over them entirely to hide the memory access pattern. Second, to hide the execution paths, we remove all branching in the code that accesses these data structures and replace them with `cmov` assembly instructions (see Section 5.5.4). Observation of the control flow and memory access does not leak whether the operation performed by the enclave was a read or a write, and since there is a single control flow without creating multiple branches depending on the condition, we effectively hide the execution path and thus protect this variant from internal leakage in full.

The usage of `cmov` for protecting against digital side-channel and internal leakage was previously studied in Raccoon [162] and with respect to protecting ORAM-based systems it was studied in other SGX-related works [16, 167]. These works show the effectiveness of `cmov` in protecting against internal leakage. Our solution uses the same techniques, and thus directly inherits the security guarantees that successfully protect against the same type of attacks, i.e. those based on digital side-channel leakage.

5.6.3 Integrity and Completeness

In the Scanning Window variant, the client herself performs the verification of the blocks, Merkle paths and transactions based on the information received from the full node and can compare the hash of the latest block to its local view of the chain to ensure completeness of the response. In the Oblivious Database variant, the enclave performs all verifications for the client. To ensure completeness, the client can compare the received response to its local view of the chain.

5.6.4 Impact of a Full SGX Compromise

Our adversary model assumes that side-channel leakage from enclave's execution may happen, but the adversary cannot fully break SGX, i.e., the adversary cannot read all enclave's secrets and modify its control flow arbitrarily. However, SGX was never intended to provide tamper resistance against physical attacks and recent research has demonstrated that platform vulnerabilities like Spectre [120] and Meltdown [136] can be leveraged to extract attestation keys from SGX processors [49]. Therefore, it becomes relevant to ask how BITE handles a full SGX compromise.

In the Scanning Window variant, the client only loses the privacy protections provided by our system and all of his funds remain secure. Since the client still performs Simple Payment Verification, the security is otherwise not affected and our system provides the same guarantees as current lightweight clients, i.e., a node may omit transactions, but cannot steal funds or make a client falsely accept a payment.

In the Oblivious Database variant, a compromised enclave could make the client accept false payments by sending invalid UTXOs. However, we argue that this will not be a realistic threat since it would require the client to sell some goods or service to the provider of the node, i.e., this is not a realistic issue for most users. Merchants that see a full break of SGX as a realistic threat can instead use the Scanning Window variant. Additionally, such an attack would be easily detectable after the fact and result in loss of reputation of the provider of our service and would thus likely only be profitable for high value transactions for which most merchants would probably run a full node.

We conclude that BITE can provide as much security and privacy as traditional lightweight clients even given a full break of SGX. This is in contrast to the naive solution of storing the clients' private keys in the enclave and using it as a remote wallet.

5.6.5 Trust Assumptions

In terms of security properties like double-spending protection, Bitcoin relies on the following two trust assumptions: First, there must be an honest majority of mining power. Second, the dissemination of messages broadcast to the peer-to-peer network must be sufficiently good, i.e., no eclipse attacks. BITE relies on the same trust assumptions as Bitcoin for its security properties.

5.7 BITE: Performance Analysis

In this section, we describe our implementation and provide performance evaluation results for BITE.

System	Our implementation		Libraries	Total
	Bitcoin[1]	Network[2]	*mbed-tls*	
Scanning Window	1'876	1'613	53'831	57'320
Oblivious Database	4'117	1'613	53'831	59'561

[1] Processing the Bitcoin blockchain.
[2] Parsing responses from the client over TLS.

Table 5.1: *Trusted Computing Base in LOC.*

5.7.1 Implementation Details

The centerpiece of the Scanning Window and Oblivious Database system variants is an original blockchain parser. For TLS connections we use the *mbed-tls* library from ARM [134]. A comparison between the two systems in terms of Trusted Computing Base is shown in Table 5.1. We differentiate between the code that is used for communication (*Network* in Table 5.1) and the code for processing the blockchain (*Bitcoin* in Table 5.1).

Scanning Window. The implementation of Scanning Window is very small (around 3.5k lines of code without *mbed-tls*) since it only involves scanning the blockchain and does not have to keep state. The network code including the *mbed-tls* library contributes the most to the TCB with over 96%. In order to keep the scanning time per block constant for all requests, the enclave does the same work for matching and non-matching transactions.

The message size per block is calculated to allow for around 5 transactions per block. We believe that this is a reasonable choice that satisfies common usage patterns for lightweight client users. For n included and N total transactions in the block, an upper bound for the Merkle path size is $n * \log(N)$ and each entry in the Merkle path is 32 bytes long. This results in an approximate upper bound of 2.2kB for $N = 4000$, the current limit in Bitcoin. As of today (March 2018) the average transaction size is around 500 bytes, therefore, a message size per block of 5kB is enough to fit around 5 transactions ($5 * 500\text{B} + 2200\text{B} < 5\text{kB}$). If more or larger transactions are found, following from Section 5.5, the enclave scans more blocks of the blockchain until the message size is big enough.

Oblivious Database. The implementation of Oblivious Database is more complex than Scanning Window and contains around 5.7k lines of code without *mbed-tls*. Contrary to the Scanning Window variant, the enclave has to keep state and store a large UTXO set on disk. At the time of writing, the size of the UTXO set (indexed by Bitcoin address) is around 4GB. The bucket size for ORAM is set to $Z = 4$ so one bucket contains 4 chunks.

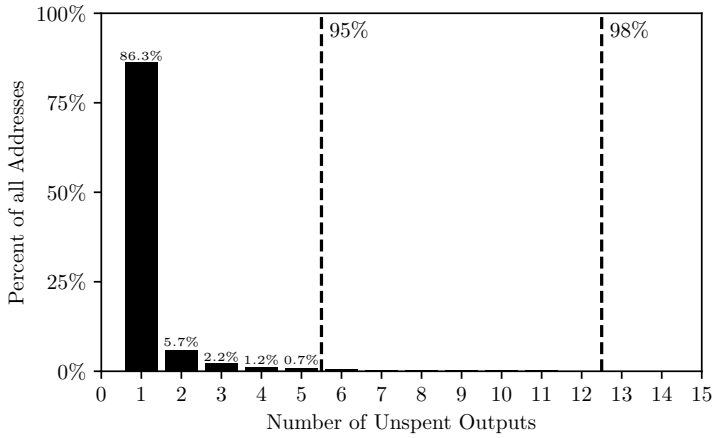

Figure 5.7: *Distribution of the number of unspent transaction outputs per active address in the Bitcoin network.*

We evaluated the ORAM performance for 32kB, 64kB and 128kB chunk sizes in Table 5.2 and settled on 32kB chunk size which then implies a tree height of 16, i.e. 2^{16} buckets. The total resulting file size on disk for the ORAM database amounts to around 8.5GB. With the selected chunk size of 32kB, a single chunk can fill up to 32kB with outputs from one address. If an address has more unspent outputs, the outputs are stored in multiple chunks. Assuming an average output size of 100B, one ORAM read can return up to 320 outputs for one address. The outputs are grouped by the receiving address and then ordered alphabetically. This is necessary in order to keep the size of the index small enough to fit in the enclaves memory. In the worst case we store the lower and upper limits for addresses (20B) and transaction hashes (32B) for every ORAM block resulting in a maximum of $2^{18} * (32\text{B} * 2 + 20\text{B} * 2) \approx 26\text{MB}$.

To set the message size, we analyzed the typical unspent outputs per active address in the Bitcoin network (Figure 5.7) and settled on 12 average outputs per request, resulting in around 1.2kB. This size is big enough to accommodate for more than 98% of all Bitcoin addresses currently in use.

5.7.2 Performance Results and Comparison

In this section, we evaluate both variants of BITE and compare them to current SPV protocols. Note that in all our data points, the TLS handshake times are omitted. In Chapter 4 we report around 100ms for a new handshake and

Chunk size	Size	ORAM
32kB	2^{16}	74.77 ms (±15.52 ms)
64kB	2^{15}	108.26 ms (±33.91 ms)
128kB	2^{14}	172.01 ms (±38.98 ms)

Table 5.2: *ORAM access times for various chunk sizes and the corresponding size (number of entries) needed to store the entire UTXO set. Time measurement averaged over 1000 runs.*

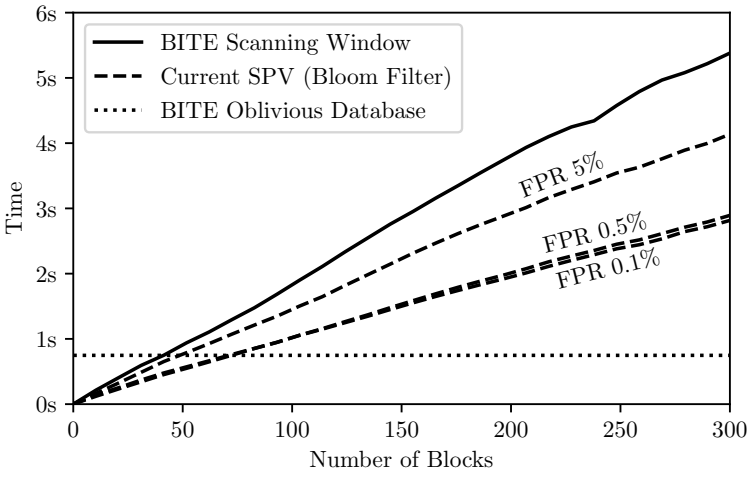

Figure 5.8: *Server time comparison between Scanning Window, Oblivious Database and current SPV protocols using bloom filters. Note that the connection speed is assumed to be infinite.*

<10ms for TLS session resumption using *mbed-tls* in SGX. We do not evaluate the performance of a client since the client-side storage and network overhead are insignificant.

We tested our implementation on an Intel Core i7-8700k processor clocked at 3.70 Ghz. The blockchain and the ORAM database were stored on a Samsung 960 Pro 512GB SSD. To compare with current SPV clients we used *python-bitcoinlib* [188].

Scanning Window. In the Scanning Window variant, our approach is similar to the original SPV procedure. Both systems let the client request filtered blocks which results in scanning the blockchain. Previous work shows

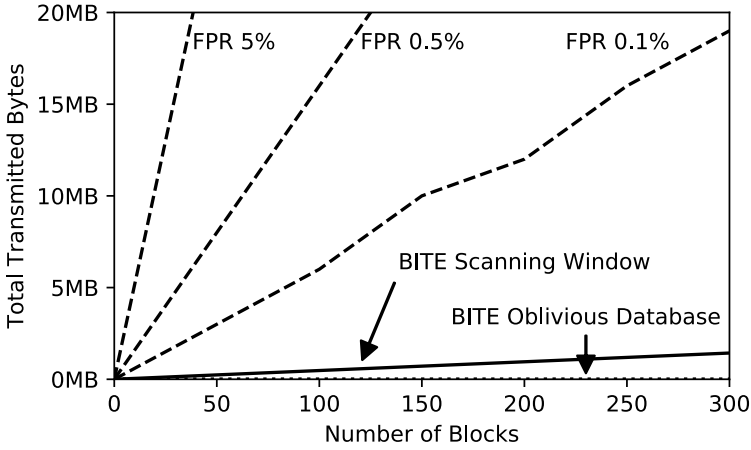

Figure 5.9: *Bandwidth comparison between Scanning Window, Oblivious Database and current SPV protocols using bloom filters.*

		t_m		
		5kB	10kB	20kB
Nr of Blocks	50	0.6s (± 0.2s)	1.2s (± 0.5s)	2.6s (± 0.9s)
	100	0.7s (± 0.2s)	1.3s (± 0.5s)	2.7s (± 0.9s)
	150	0.7s (± 0.2s)	1.4s (± 0.5s)	2.7s (± 0.9s)
	200	0.7s (± 0.2s)	1.4s (± 0.5s)	2.8s (± 0.9s)
	250	0.7s (± 0.2s)	1.4s (± 0.5s)	2.9s (± 0.9s)
	300	0.7s (± 0.2s)	1.5s (± 0.5s)	3.0s (± 0.9s)

Table 5.3: *Processing time per block with oblivious execution for Scanning Window depending on the number of requested blocks and the temporary size, averaged over 100 blocks.*

that Intel SGX imposes significant overhead for copying buffers (reading files) across the trust boundaries [24].

Figure 5.8 shows the time needed to filter blocks by Scanning Window and current SPV protocols. We report an overhead of around 100% (in total the time is 5.3s) in comparison to Bloom filters with a false prositive rate of 0.1% (2.65s) to 0.5% (2.7s). Note that the measurements in Figure 5.8 do not account for the network speed. A device with a decent 4G connection that operates at 100Mbit/s requires additionally around 5s to retrieve 300

blocks with the current SPV protocol and a 0.1% false positive rate. For even higher false positive rates, i.e., 0.5% (the default value of *BitcoinJ*), an SPV client synchronizes in additionally around 8s. Our systems are not significantly impacted by the limited bandwidth of 4G. The synchronizing time for Scanning Window rises by 0.1s to around 5.4s (and Oblivious Database stays constant at 0.5s). Our systems can reduce the required bandwidth because no false positives have to be included to fool the attacker.

The scanning time is impacted significantly by the need to recompute the merkle tree (approx. 6.5ms). Storing the merkle tree for every block could lead to better performance but the required disk space would grow significantly.

Scanning Window with side-channel protection. Oblivious execution and memory access adds a significant overhead to Scanning Window. All branches have to be taken and all in-memory structures have to be touched in their entirety to hide access patterns. The merkle tree has to be recomputed for every block aswell but since generating a merkle tree for the average bitcoin block takes 6.5ms this does not contribute significantly to the runtime of the side-channel free Scanning Window.

Table 5.3 shows the time per block for various number of blocks requested and t_m size. Higher t_m allows to cope with some blocks that have a lot of relevant transactions while others do not, since it limits the amount of transactions of a single block that can be included in a response. Note that the blocks vary in size, and thus the time per block fluctuates a lot leading to a high standard deviation. Synchronizing 300 blocks with $t_m = 10$kB takes around 7.3 minutes corresponding to an overhead of approximately 100x.

Oblivious Database. In this variant, the unspent outputs are directly fetched from the enclave UTXO. Therefore, the time needed is independent of the number of requested blocks, yet only on the ORAM database access times. Figure 5.8 shows the Oblivious Database variant response time for a request containing 10 addresses and a ORAM block size of 32kB. Table 5.4 shows the time an update to the ORAM database takes for various blocks at a time. In order to reach permanent availability we propose the usage of at least 2 systems in parallel which update with an offset between each other. If a user requests the result from a node that is not fully up to date, the remaining blocks can be scanned by utilizing oblivious Scanning Window. The amount of clients that can be served by a single SGX enclave can be estimated by using around 120s for updating the state and then the remaining 8 minutes to continuously answer client requests, leading to an approximate 10000 clients per enclave. The message size is significantly lower than all other variants, since only unspent outputs are included and not the entire transactions and the corresponding partial Merkle path. The message size for a request of

Blocks	ORAM update time
1	78.5s (±13.6s)
3	112.5s (±19.8s)
6	146.7s (±19.8s)

Table 5.4: *Update time for the ORAM of Oblivious Database solution averaged over 100 measurements.*

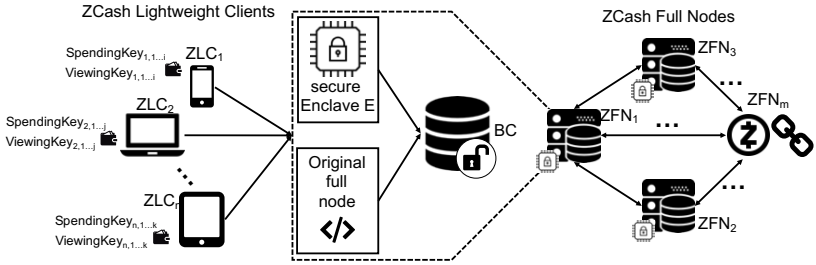

Figure 5.10: ***System model.*** *Lightweight clients request transaction verification and payment issuing service from enclaves hosted on full Zcash nodes.*

10 addresses amounts to $10 * 1.2\text{kB} = 12\text{kB}$. Figure 5.9 shows bandwidth comparison between all discussed protocols.

5.8 ZLiTE: System

In this section, we first present the system model and then a system called ZLITE that realizes our approach securely to enable lightweight Zcash clients in respect to the identified challenges.

5.8.1 System Model

Figure 5.10 represents our system model. The main stakeholders in the system are Zcash Lightweight Clients $ZLC_1 ... ZLC_n$ and Zcash Full Nodes $ZFN_1 ... ZFN_m$. A lightweight client ZLC_a connects to any full node ZFN_b that supports our service by hosting an enclave E when she wants to acquire information regarding transactions and addresses that belong to the client or to issue new transactions towards another Zcash client. ZLC_a can own one or more addresses in her wallet that are also characterized by the $SpendingKey_{a,1...c}$ and the $ViewingKey_{a,1...c}$.

Full nodes maintain the local version of the blockchain (BC) as usual, appending each new confirmed block to the longest chain they have. The blockchain is maintained outside the secure environment, either on the disk

or memory of the platform where the node resides. SGX enclave memory is limited (128MB) and is only suitable for smaller storage related to the currently executed task.

5.8.2 Operation Overview

A client that wants to retrieve transactions, performs remote attestation of the ZLITE enclave and establishes a secure connection (TLS), through which she sends her viewing key and the height h of the last known block B_h. The enclave then scans the blockchain for transactions for this viewing key starting from B_h and obliviously moves them to a temporary *response ORAM* (rORAM) to hide which transactions are of the client's interest. Additionally, the ORAM structure is obliviously serialized in the response buffer sent to the client.

Oblivious Scanning. All processes that rely on secret data, i.e., the client's viewing key, must be performed in an oblivious fashion to prohibit any leakage of sensitive information (see Section 5.9). Finding the transactions that match the clients viewing key clearly depends on the client's secrets. To make block scanning oblivious to a side-channel observer (see the adversary model in Section 5.4.4), processing of each transaction should produce the same side-channel trace. A naive way to solve this is to do a fake copy of each non-matching transaction (viewing key does not result in a valid decryption) to the response buffer as well. However, in that case the response buffer is as big as the scanned blocks (no performance improvement). To improve the performance we use a *response ORAM* to hold all relevant transactions of the current client. The rORAM allows us to perform one ORAM operation per transaction while still hiding if this operation is a write (relevant transaction) or a read (irrelevant transaction). This is achieved by constant-time branchless code using the `cmov` instruction [162] in the similar way as in BITE. In conclusion, the enclave performs the following operations for each transaction:

(**1**) check if the viewing key manages to decrypt the transaction

(**2**) calculate the Merkle tree

(**3**) perform an ORAM operation (write or read transaction into the rORAM depending on the outcome of **(1)**)

Together with the transactions stored in the rORAM, ZLITE delivers the corresponding Merkle paths, all block headers since B_h, and the note commitment tree update for the requested interval (see Section 5.8.4). Below we first describe the details of the ZLITE operation and the retrieval of transactions and then describe how a lightweight client using our system can create new shielded transaction.

5.8.3 Transaction Retrieval

The operation of the synchronization protocol (see Figure 5.11) works as follows:

Initialization and continuous operation.

(**a**) On initialization the Full Node ZFN_j connects to the P2P network (**a-1**) and downloads the full Zcash blockchain (**a-2**). This locally stored blockchain is continuously updated as new blocks are received from the network.

(**b**) When the lightweight client is installed, it contains a checkpoint block header (this can be from a recent date or the genesis block). The client then downloads all newer block headers from the P2P network and verifies them (i.e., the client checks the PoW and that their hash chain leads to the checkpoint). All but a small number of the most recent block headers (to handle shallow forks) can be deleted afterwards. This state is later updated during the synchronization process that the client performs with a ZLiTE node in order to check for received transactions or before sending transactions (see below). This is similar to the operation of existing lightweight clients for other blockchains (e.g. Bitcoin).

Synchronization of Transactions. Clients synchronize with a ZLiTE as follows:

(**1**) The Zcash Lightweight Client ZLC_i performs attestation with the secure Enclave E_j residing on the full node ZFN_j.

(**2**) If the attestation was successful, the Zcash Lightweight Client ZLC_i establishes a secure communication channel to the Enclave E_j using TLS.

(**3**) The Lightweight Client ZLC_i sends a request containing its viewing key and the number of the latest known block.

(**4**) The Enclave E_j creates a temporary in-memory *response ORAM* (rORAM) to store the transactions that will be sent to the client. E_j then *scans* its locally stored copy of the blockchain (BC) starting at the block number specified by the client and decrypts the transactions with the specified viewing key. The decryption will either result in garbage or in a valid plaintext transaction. If the decryption is successful, E_j moves the transaction and the corresponding Merkle paths (for the transaction and for the note commitments) to the response ORAM, and if it is not successful, E_j performs a read operation, thereby performing the move obliviously using the `cmov` technique mentioned in Section 5.8.1 to replace conditional statements.

(**5**) After the scanning operation has finished, the rORAM is serialized by moving the entries to a fixed-size (dependent on the request, i.e., number of

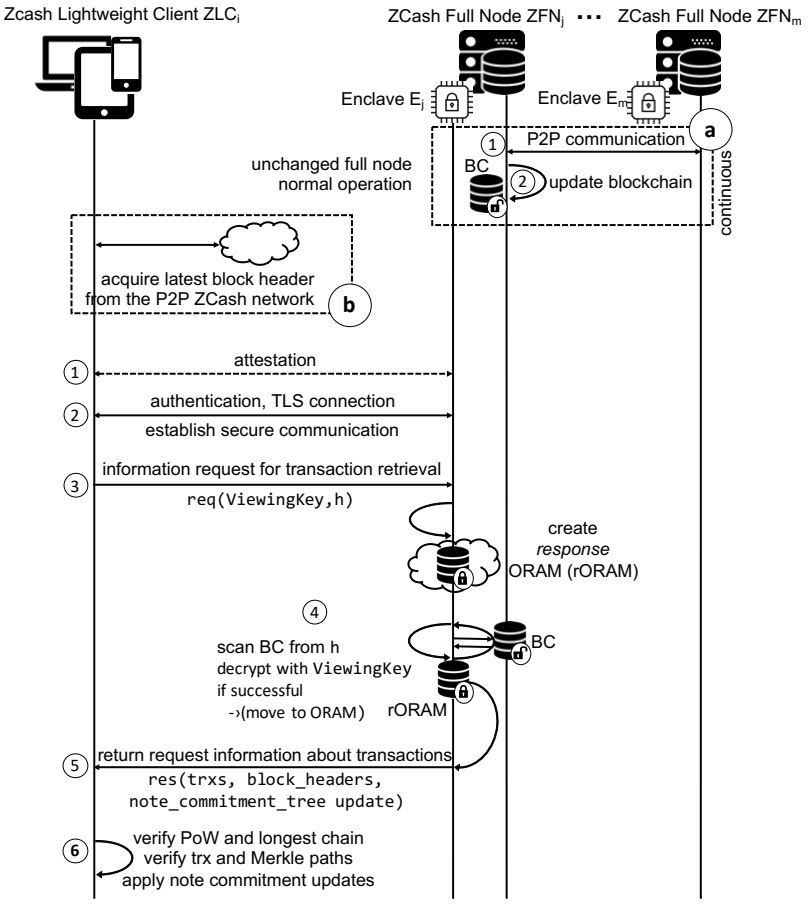

Figure 5.11: ***Synchronization.*** *The lightweight client establishes a secure connection to an enclave on a full node and sends a request that contains its viewing key and latest known block to perform the retrieval of all of her transaction information.*

requested blocks for update) response array that is then sent to the client. In addition, the response contains all of the block headers and the note commitment tree updates (see Section 5.8.4 for details).

(**6**) The Zcash Lightweight Client ZLC_i verifies that the received block headers have a valid proof of work, create a chain to its latest known header and

that the chain is the heaviest chain advertised in the P2P network. For every received transaction, it checks whether the recomputed Merkle root, given the received path, matches the corresponding block header. The client then updates the witnesses for all transactions with the received note commitment tree update and finally deletes old block headers that are no longer needed.

5.8.4 Transaction Creation

The light client receives all information necessary to create shielded Zcash transactions from ZLiTE. Namely, for every output she wants to spend, she requires the witness (when creating the new transaction) of the corresponding note commitment (i.e., its Merkle path in the note commitment tree).

These witnesses could be retrieved from a ZLiTE node at the time of spending. However, this would require the node to retrieve the witness in an oblivious fashion on request, which becomes computationally expensive as the commitment tree gets larger. Instead, when scanning the chain for a client, we additionally supply the witness of a note at the block height where it was created (see Section 5.8.3). When synchronizing, the client then also receives *commitment tree updates*, which allow him to update witnesses for any previous note commitment. In this case, there is no need for oblivious computation since the update only depends on the block height and not on the transaction relevant to the client.

Given a note commitment tree at time t_1 and a note commitment tree at time t_2, to compute the commitment tree update, the enclave starts with an empty list U_{ct} to store the update. Let cm_i be the latest note commitment in the tree at time t_1, i.e., it is the rightmost non-empty leaf. Then, in the tree at time t_2, for every node on the path from cm_i to the root of the tree, add the right child to U_{ct}. A client in the possession of a witness at time t_1 for some note, can then apply the update by replacing any node on the witness with the corresponding node from U_{ct}, if these two nodes have the same location in the Merkle tree. We present a proof in Appendix C.1 that this construction results in a correct witness for the note commitment tree at time t_2.

5.9 ZLiTE: Security Analysis

In this section, we provide an informal security analysis of ZLiTE. We first discuss protection against information leakage, then the completeness of responses, and finally consider the worst-case scenario, i.e., a full SGX break.

5.9.1 Protection Against Information Leakage

Since ORAM reads and writes are indistinguishable, an adversary observing memory access patterns is not able to determine which transactions were written. For ORAM accesses, when accessing the stash, indexes or the position map, every location is accessed to hide memory access patterns.

To protect against side channels (e.g. [40, 71, 151, 170]), conditional statements that depend on transactions (e.g. during the process of moving transactions to the response ORAM) are replaced using the `cmov` instruction. Since this results in the same control flow independent of the transaction, protection against leakage even against an adversary that can observe the control flow with instruction level granularity is guaranteed. The `cmov` instruction has been previously used to protect against side channels by Raccoon [162], Zerotrace [167] and also Obliviate [16] in the context of providing secure ORAM access using SGX. This prior research shows that `cmov` can effectively protect against digital side channels.

Finally, the response size only depends on the number of scanned blocks, i.e., it is independent of how many (or if any) transactions are in the response, and thus does not leak any information about client's keys or transactions.

5.9.2 Integrity and Completeness

The ZLITE node delivers the requested information along with all block information needed for simple payment verification. The client herself then verifies the block headers using the Merkle paths for her transactions. Similar to SPV in Bitcoin lightweight clients [153], this ensures that the server cannot make a client falsely accept payments for which the transactions are not included in the chain. As the client can also check the proof of work and gossips with the P2P network to receive block headers, she can ensure that she receives information from the longest chain. Thus, the server does not have stronger capabilities to eclipse a lightweight client than against a full node.

In contrast to standard SPV (as e.g., in Bitcoin [153]), where the client cannot be sure to have received all of her transactions, the usage of a TEE makes sure that the received response contains all of her transactions for the scanned interval given the ZLITE node's view of the blockchain.

5.9.3 Impact of a Full SGX Compromise

While our adversary model considers side-channel attacks, we do not consider a full compromise of SGX, i.e., forged attestations, arbitrary control flow change or enclave secrets reading. However, recent research has shown that secrets can be read even from the quoting enclave allowing an adversary to

extract attestation keys [49, 191] which makes it necessary to discuss such a worst-case scenario.

While it is obvious that the privacy provided by ZLiTE can no longer hold, if the adversary can read all secrets, or a client connects to a server that uses a forged attestation to impersonate an SGX enclave, such a breach cannot lead to loss of funds. In addition to the loss of privacy, a client also loses completeness, since a node may omit payments. However, because the client's spending key is never sent to a ZLiTE node and the client performs Simple Payment Verification for all of his transactions, a node is not able to steal coins from the client or make him falsely accept a payment.

5.9.4 Trust Assumptions

In terms of security properties like double-spending protection, Zcash relies on the following two trust assumptions: First, there must be an honest majority of mining power. Second, the dissemination of messages broadcast to the peer-to-peer network must be sufficiently good, i.e., no eclipse attacks. ZLiTE relies on the same trust assumptions as Zcash for its security properties.

For privacy, Zcash relies on securely-generated public parameters and hardness of numeric cryptographic assumptions. ZLiTE requires the same assumptions and additional trust in TEEs.

5.10 ZLiTE: Performance Analysis

In this section, we describe our implementation and provide performance evaluation results for ZLiTE.

5.10.1 Implementation Details

Our implementation of ZLiTE is based on the protocol specification of Zcash. It consists of a blockchain parser, an oblivious Path ORAM implementation [179] and it makes use of some bundled cryptographic libraries. We support the current Zcash protocol specification including the 'overwinter' protocol update.

The Trusted Computing Base (TCB) of our implementation can be split up into a network part that is responsible for the communication with a client (around 1.5k LoC) and the blockchain relevant part (around 3.7k LoC). Additionally we use well reviewed crypto libraries like *mbedTLS* (53k LoC) and small libraries that provide crypto primitives: sha256 , blake2b, ripemd160, ChaCha20Poly1305 and ed25519 totaling to around 2.2k LoC. All of the included crypto primitives come from well reviewed sources. We will not go

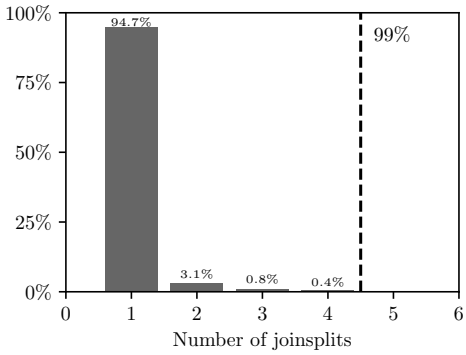

Figure 5.12: *Joinsplit distribution in all shielded transactions up until block 350000.*

into details on the TLS library *mbedTLS* [134] and refer the interested reader to [145, 206] for implementation details and performance results.

5.10.2 Performance Results and Comparison

ZLITE measurements were done on an i7-8700k processor with an SSD. Note that all the reported timing results are without the additional TLS latency. All measurements are according to the blockchain activity as of August 2018.

Lower Bound. Any node that wants to check for new transactions needs to parse the new blocks and test its viewing keys against all transaction in the blocks. This is part of the Zcash specification and implies a lower bound for any full node. Testing viewing keys is computationally intensive because it involves a key exchange based on an elliptic curve for each transaction and viewing key. Our implementation manages to parse blocks of an entire day and test a single viewing key against the transactions within 1.24s compared to fully oblivious operation of ZLITE which takes around 5s. We have to retrieve the Merkle paths and perform at least one ORAM operation per shielded transaction while non oblivious solutions can skip this step for all transaction that are not relevant to the client.

Average Transaction Size. We measured the average number of *joinsplits* in a shielded transaction and show a histogram in Figure 5.12. Around 95% of all shielded transactions only contain one *joinsplit*, thus they have at most 2 shielded inputs and 2 shielded outputs. Every *joinsplit* occupies around 2KB of data. We also have to store the *commitment tree update* (see Appendix C.1) which is around 1KB in size. The average shielded transaction thus requires

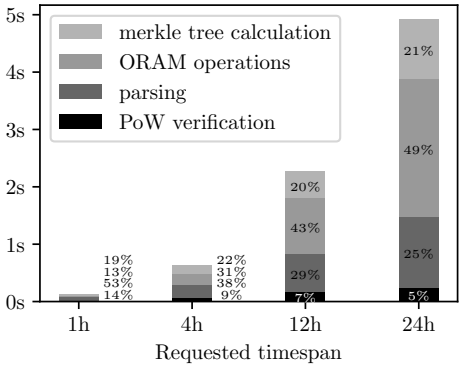

Figure 5.13: *Enclave latency for various request sizes (with 24576B of client data/hour).*

		Client data per hour					
Time	Blocks	6144B		12288B		24576B	
24h	576	4187ms	±504ms	4382ms	±510ms	4967ms	±617ms
12h	288	1875ms	±315ms	2122ms	±364ms	2317ms	±397ms
4h	96	541ms	±75ms	583ms	±92ms	631ms	±104ms
1h	24	123ms	±21ms	129ms	±21ms	130ms	±21ms

Table 5.5: *Total time for various request and response sizes. (100 runs)*

an ORAM operation for around 3KB of data. These measurements allow us to chose optimal ORAM block size for our rORAM of 3KB.

Latency. We measured the time required to fetch various amounts of blocks and show a comparison between different expected client data per hour in Table 5.5. Note that the time per block rises when a client requests a longer time period because the response ORAM is chosen accordingly and a big ORAM database leads to slower accesses. Additionally, slower responses are observed when the client expects a lot of activity and requests a lot of data.

Figure 5.13 shows the latency for a request with 24576B of client data per hour and various requested time spans. The latency is further divided in the four main contributors to the total: parsing the block, proof of work verification, ORAM operations and generating the merkle tree. Note that the ORAM operations start to take the lions share of the latency as soon as longer time spans are requested.

Bandwidth. The required bandwidth can be split into a static part (not dependent on the number of blocks requested) and a dynamic part. The

dynamic part is composed of the blockheader (1487B) and the private data per block that is used to return transactions to the client. For reasonable usage we estimate a lightweight client to have (at most) one transaction every hour occupying 12kB. This results in 1024B of private data per block and the total dynamic bandwidth accumulates to 2511B per block. The static part only consists of the *commitment tree update* and is therefore $29 * 32B = 928B$ large. A client that requests one day of blocks from our system gets a response of 1.38MB. We reason that the clients who excessively use Zcash should already operate a full node.

Increased Blockchain Activity. As of Aug 2018 shielded transactions are not very common on the Zcash blockchain (only 1.5 txs/block). With single steps measurements we estimated ZLITE performance with increased future activity. For 100 shielded transactions per block, a daily request would take 112s, while with an hourly one the latency would shrink to around 750ms.

5.11 Discussion

In this section, we provide additional insight into the reasoning about some of our design choices.

Usage models and long-term privacy for BITE. Lightweight clients can use BITE in different ways and the chosen usage model can have implications on the clients' long-term privacy.

For example, in what we consider *non-recommended usage*, the client (i) performs payment verification requests only when the payment appears in the ledger, (ii) always uses the same full node for verification, and (iii) only uses a single or few Bitcoin address. If all of the above conditions are met, although the adversary controlling the full node does not learn the client's address from a single verification request, he might be able to *correlate* the timing of the verification request events and the Bitcoin addresses visible in the ledger at roughly at the same time, and thus construct a set of candidate addresses that may belong to the served client. We acknowledge that our solution cannot eliminate this type of correlation completely. However, we stress that such correlation would require long-term tracking of verification requests from the adversary and that the same limitation applies to any lightweight client payment verification scheme.

In *recommended* usage of our solution, the client (i) uses different full nodes for payment verification, (ii) regularly uses fresh Bitcoin addresses (e.g., using an HD wallet [201]), and (iii) introduces unpredictability to the timing pattern of payment verification requests like a small number of extra requests

at random points in time. If the client follows such a usage model, the above mentioned correlation becomes very difficult.[2]

Large responses. Some client requests might result in a larger response than our defined threshold for message size. As our performance analysis shows, the number of these requests is almost negligible and represents truly a minority of the complete set of transactions in the blockchain. However, our mechanism still allows these types of request with the distinctive factor that the client would have to request them in batches. For example, in BITE, if a client in the Scanning Window variant requests transactions for 10 of his addresses from the last 300 blocks using the full-side-channel protection, there might be more transactional data then the $300 * t$ kB message size. In this case, the enclave sets a flag indicating there is more information to be delivered. After receiving the response, the client can repeat the request with the defined flag and receive the rest of the information. The protocol operates in the same way, thus no distinction between these two requests can be observed by the attacker. However, the attacker can see that the request is repeated and infer that the specific client has more transactions of interest in the designated blocks. To mitigate this problem one could obfuscate the IP address or change to another enclave for finishing the request.

Deployment models. We consider two deployment models. In the first model, a verifiable company can run our solution as a service offering light clients privacy. In the second model, any full node operator, or a volunteer, can operate our solution. In both deployment models we presume that multiple, or even all nodes, will support this solution, and the scalability depends on that number. One single node can only support around 10000 lightweight clients, and the overall success depends on the supply, demand and acceptance of such service. Such different deployments can enable different authentication models. In the first option, the company could use a PKI which would allow the lightweight clients to recognize which specific enclave they are communicating with. In the second option the clients only know that they are connecting to a correctly attested enclave, but they cannot make distinction between different enclaves. Since successful attestation guarantees expected enclave execution, our solution's privacy properties hold in both cases, unless Intel SGX is broken. We discuss the implication of a full SGX compromise in Section 5.6 for BITE and in Section 5.9 for ZLITE.

[2]To quantify how accurately the adversary can correlate the client's addresses and how difficult such correlation becomes with the above discussed best practices, would be an interesting direction for future work. As building an accurate model would require collecting significant amount data about the behavioral patterns of lightweight clients, we consider this task a research project on its own and outside the scope of this chapter/thesis.

Denial of service. A malicious user might attempt denial of service (DoS) by asking for a very long scan window — incurring large processing times for full nodes and thus making the service momentarily unavailable for other clients. DoS (and spam) are common in systems where there is no significant cost involved (e.g., sending 1M emails is practically free) and hard to prevent when introduction of fees is hard. In our setting, one could easily remedy such denial of service attacks by applying fees based on the nature of the request. Large balance updates for light clients would incur higher costs than frequent updates, thus limiting the attacker from performing "free" DoS attacks.

Unbounded enclave memory. The performance of our system is mostly bounded by the slower disk operations. However, in case that future versions of SGX architecture would allow more enclave memory (i.e., currently the limit is 128MB without the expensive page swapping) ranging up to the RAM limit on the residing platform, one could keep the UTXO database (in case of BITE) and/or all other security critical data in the memory and not on the disk, similar to recently proposed SGX-based in-memory database systems like EnclaveDB [161].

5.12 Related Work

In this section, we review related work that can be classified into three main categories: Bitcoin lightweight client privacy, Zcash scalable clients and SGX information leakage protection.

5.12.1 Bitcoin Lightweight Client Privacy

The idea of light clients for Bitcoin was already included in the Bitcoin paper by Satoshi Nakamoto [153] in the form of *Simple Payment Verification* (SPV). Hearn and Corallo later introduced Bloom filters [35] in BIP 37 [87] that allow a client to probabilistically request a subset of all transactions in a block to mask which addresses are in fact owned by the client. Gervais et al. later showed that the information leaked by the use of Bloom filters in Bitcoin poses a serious privacy risk and can in many cases enable the identification of client addresses [68]. Hearn – who introduced Bloom filters to Bitcoin – later addressed the issues [86], expanded on them, and discussed the difficulties of solving them.

To overcome this privacy issue, Osuntokun et al. recently proposed modifications to Bitcoin nodes and lightweight clients that move the application of the filter to the client [156]. In their protocol, full nodes create a filter (with a low false positive rate) for the set of all transactions in a block. A lightweight client then fetches the filter from one or more full nodes and can then check

whether the block contains transactions that she is interested in. If that is the case, the client will request the full block from any node.

While this approach likely provides more privacy than the protocol using Bloom filters, it still suffers from a number of shortcomings. First, the gained privacy largely depends on the client behavior and how well the client is connected to different entities. If the client does not request the filter headers from multiple entities[3] and uses another entity to then request the blocks, she can be easily tricked into revealing her addresses by using forged filters as follows: A node can prepare a filter that matches half of all addresses and send it to the client. If the client requests the block, at least one of her addresses lies within that set, otherwise all of her addresses lie in the other half. The node can then continue reducing the possible set using a binary search approach by sending modified filters for the following blocks, which allows bitwise recovery of all client addresses. Second, depending on how often a transaction is of interest to the client, she might end up downloading the full blockchain after all. Since the client always either requests the full block or nothing at all, she will download almost every block if a large fraction of blocks contain at least one transaction that is of interest.

Other research on Bitcoin privacy shows that using different heuristics, large parts of the Bitcoin transaction graph can be deanonymized [21, 147]. These techniques are, however, orthogonal to the problem of lightweight client privacy and thus out of scope for our work.

5.12.2 Zcash Scalable Clients

While the main challenge in Bitcoin was to efficiently protect privacy in a system that already provides light clients, with ZLiTE we tackle the problem of enabling light clients in a system that provides privacy, but until now does not support operation of light clients. One notable difference between Bite and ZLiTE is that in Zcash spending previously received funds requires the witness to the transaction's inclusion in the Merkle tree of all transactions, and therefore client must obtain, in an efficient manner, an up-to-date version of this witness to spend the funds.

Several proposals aim to lower the resource requirements for clients in Zcash. While protocol upgrades [12] have reduced the computational resources required to generate a transaction, they have not substantially changed the bandwidth or verification requirements.

[3] Even if the client connects to multiple different nodes to receive the filters, she cannot verify that they are not under the control of the same entity.

Bolt [74] proposes privacy preserving payment channels in which clients conduct most transactions off chain in a fully private manner. However, the current version requires clients to either monitor the blockchain for channel closure using a full-node, or entrust a third party to do so. While this does not violate privacy, failures by the third party can result in monetary loss. No such risk of theft exists with ZLITE even if TEE integrity is violated. Moreover, Bolt requires payers to have an existing relationship with the recipient or an intermediate payment hub. While promising, Bolt is not a full solution for bandwidth limited clients.

In [51], Chiesa et al. explore the use of probabilistic micro-payments as a way of increasing throughput. In this setting, a sequence of, e.g., 100 micro-payments for one cent, is approximated by paying $1 with probability $\frac{1}{100}$. Thus only $\frac{1}{100}$ of transactions are actually issued. However, this is only suitable for small and frequently repeated payments. Moreover, it is unclear if it will reduce the total volume of transactions or simple free up capacity for even more transactions.

5.12.3 SGX Leakage Protection

During the last few years, the research community has studied information leakage from SGX enclaves extensively and proposed a number of defenses. We explain why none of the existing systems solves our problem directly and which prior systems use similar protective primitives as our solution.

The previous work that is probably closest to our solution is a system called Raccoon [162] that addresses both internal and external information leakage for both code and data accesses. For control-flow obfuscation, Raccoon uses taint analysis to determine execution paths that should be hidden and transforms enclave code such that it executes extraneous decoy paths to hide the enclave's actual control flow. The basic building block for such control-flow obfuscation is the `cmov` instruction that we use as well. Raccoon also uses Path ORAM to hide external secret-dependent data accesses and "streaming" over data structures (i.e., accessing every element) in the internal enclave memory. The main difference between Raccoon and our solution is that by tailoring our implementation, we avoid the need for taint analysis and extra decoy paths enabling a more efficient solution.

Other related systems include Cloak [77] that prevents cache leakage using hardware-based transactional memory features in processors; ZeroTrace [167] that provides a library for data structures that are protected using ORAM; DR.SGX [39] that randomizes and periodically re-randomizes all data locations in enclave's memory with cache-line granularity; and, T-SGX [172] and Deja Vu [50] that detect and prevent side-channel attacks based on repeated

interrupts. The main limitation of Cloak is that it requires hardware features that are not available on all SGX CPUs and it only protection cache-based leakage. ZeroTrace is limited to data access protection and it does not prevent leakage from secret-dependent control flow. DR.SGX is also limited to data accesses and imposes high performance overhead when configured to prevent all leakage. T-SGX and Deja Vu are limited to attacks that perform repeated interrupts (subset of known attacks). Recently published Oblix [150] presents a new ORAM algorithm that is designed specifically for SGX. We use well-known Path ORAM, but our solution is agnostic to the used ORAM algorithm and we could easily replace Path ORAM with another algorithm.

5.12.4 Practical ORAM in Trusted Hardware

The Phantom [139] and the Ascend [64] secure processors introduce dedicated implementation of the ORAM techniques directly inside the processor's memory controller. Thus, Phantom and Ascend processors were resilient to the leakage of memory access pattern from the access to the DRAM memory by the secure container. Additional protections have been employed in order to prevent DRAM address bus probing.

The development of an ORAM in the memory controller represents an interesting idea, that could, when combined with the protection of other secure trusted hardware such as Intel SGX, allow for more secure intrinsic properties of a processor. Solving private information leakage is a direct problem for many systems that want to use the benefits of trusted hardware, albeit SGX or some other solution, since most of them do not offer these privacy protections.

5.13 Conclusion

Improved user privacy is one of the main benefits of decentralized cryptocurrencies. However, payment verification requires downloading and processing the entire chain which is impossible for most mobile clients. Therefore, almost all popular blockchains support simplified verification modes where lightweight clients can verify transactions with the help of full nodes. Unfortunately, such payment verification does not preserve user privacy and thus defeats one of the main benefits of using systems like Bitcoin. Zcash provides strong privacy for its users. Shielded transactions, however, require clients to download and process every block which is impractical for devices like smartphones, and consequently no mobile client supporting shielded transactions exists.

We have proposed a new approach to improve the privacy of lightweight clients for Bitcoin using trusted execution. We have shown that our solution provides strong privacy protection and additionally improves performance of

current lightweight clients. We argue that BITE is the first practical solution to ensure privacy for lightweight clients, such as mobile devices, in Bitcoin. On the other hand, in Zcash, we have developed a new solution that enables emergence of light clients to create and receive shielded payments. Usage of trusted execution, obviously, changes the original trust model of Zcash, but we argue that such a solution strikes a balance between the best possible privacy and the range of scenarios where Zcash can be used in practice. Thanks to our solution, development of mobile clients that support shielded transactions becomes possible and more users can benefit from the sophisticated privacy protections of Zcash.

Chapter 6

Closing Remarks

In this final chapter, we summarize the work presented in this thesis and highlight the main findings and results. Additionally, we explore directions for future work and state the lessons learned.

6.1 Summary

We began this thesis by motivating the emergence of trusted execution environments and their influence on the previous, current and future digital security. In Chapter 2, we summarized the state of the art in the field of trusted execution environments, pointing to the whole technology space and as well outlining the benefits and drawbacks. Additionally, we provided the necessary background on a specific TEE, namely Intel SGX, on which the work of this thesis is based on. We show that there is a lot of future potential in actual realization of systems supported by trusted execution in order to provide hardened security with good performance.

In Chapter 3, we identified a vulnerability in Intel SGX that allows a potential adversary to violate the freshness integrity of long-term sealed states in protected enclaves. Our solution, ROTE, designs and implements a secure distributed scheme that enables rollback protection for running Intel SGX enclaves. Unlike other less viable solutions, ROTE allows unlimited state updates occurring in high frequency suitable for all application types that mandate long-term state preservation and protection, and provides *all-or-nothing* guarantees, essentially assuring that no rollback can arise.

In Chapter 4, we explored the usability space of SGX and its influence on the current application world. Namely, we introduced a new concept of

brokered delegation which, in a nutshell, allows users to circumvent mandatory access control, imposed by the service that they use, reducing it to discretionary access control, in a secure manner without losing full control of the service and/or their secret credentials. With this concept, we created a system called DELEGATEE that allows users to safely and selectively delegate their access rights to a certain range of services. We demonstrated the capabilities of the concept and the developed system on 4 different use-case scenarios: delegated email access, brokered payments using credit cards and third party payment systems, and full website access using a HTTPS proxy.

Finally, in Chapter 5, we tackled the problem of privacy preservation for lightweight clients in existing cryptocurrencies using SGX as the backbone. The main challenge and as well the main controversy was the application of a secure, closed-sourced, processor technology owned by a private entity to protect privacy in a wide-scale user space such as Bitcoin and Zcash. Even more, direct application of SGX for privacy has proven to be insecure due to the side-channel vulnerabilities of the technology. However, to reach our goal, we carefully combined techniques from several separate field, including trusted computing, private information retrieval and side-channel protections. Our work resulted in two systems, BITE and ZLITE, where we consider the former to be the first practical solution that provides strong privacy protection for Bitcoin light clients, while the latter represents the first and, currently, only solution that enables the emergence of light clients in Zcash, taking respective privacy requirements of the cryptocurrency itself into account.

6.2 Future Work

In this section, we provide insights for future work in terms of enabling more widespread adoption of Intel SGX and doing further adaptations and continuation of the work that was done in the thesis. The main goal remains the same, eliminate existing vulnerabilities and make the hardened security even *harder* while increasing the capabilities of existing services and allowing the emergence of new ones, which would not be feasible without the existence of TEEs (under the same trust assumptions).

Secure Migration and Rollback Protection for Intel SGX in the Cloud: SGX suffers from an intrinsic shortcoming of being tied to specific hardware on the residing platform, namely the processor, thus diminishing its value of operating in a cloud environment. In the cloud, virtual machines present an overlay to hide the complexity of the hardware and are often, for performance and balancing, migrated from one platform to another inside the cloud

provider's data center. However, moving virtual machines to other hardware makes SGX enclaves unusable since they are tied to the exact processor they are started on. One would need to create an effective migration mechanism coupled with a rollback scheme, as ROTE, that would allow the emergence of effective SGX cloud platforms where the cloud users would not need to rent a complete server but only a virtual machine (as it is most common today).

Prototyping of a Full HTTPS Proxy for Brokered Delegation: In this thesis, we have showed the strengths of brokered delegation through trusted execution environments. However, a major shortcoming when it comes to arbitrary online services and delegation is that specific delegation scenarios have to be implemented as an exact match to the service's operation. Namely, creating a general scheme for delegation that supports arbitrary web technologies (e.g., all types of websites with needed login in order to allow additional content) still remains a challenge. To overcome it, one would need to be able to interpret web technologies, such as javascript, efficiently and securely inside of an SGX enclave. Implementing the interpreters would, first, bloat the trusted computing base and, second, without formal verification (which would be difficult to perform in that large scale) potentially leave vulnerabilities that would forfeit the initial justification of using the trusted execution in the first place. The potential space for research is vast and the end goal would be to find a perfect trade-off that guarantees strong security with a high degree of usability for the end users. Lastly, such a system could be used to increase the security in the business sector, enhancing the intrusion detection systems for protection against inside adversaries, by allowing deep packet Inspection over secure communication channels and encrypted traffic (e.g., TLS).

Trusted Execution in the Blockchain Application Space: During the work of enhancing privacy preservation for Bitcoin and Zcash lightweight clients we observed that trusted execution could be integrated with the already popular space of blockchain applications to increase both security and performance. Shifting trust assumptions towards more trust in technologies like SGX could potentially increase the usability of these systems in a large scale. For example, in Ethereum, smart contracts are executed by all the participating nodes in order to verify the given state transition result. On the other hand, Bitcoin, even though supporting a scripting language for its transactional space, does not enable execution of contextually-rich smart contracts under a Touring-complete language. With the adoption of expressive and powerful TEEs such as Intel's SGX, one could, in the first example, allow off-chain execution

of smart contracts by selected service providers, and use the Ethereum (or similar) cryptocurrency network to publish state changes, thus maintaining the integrity properties of blockchain itself. In the second example, the enclaves could be used to increase the expressiveness of digital currencies by allowing users to execute arbitrary smart contract tied to a specific currency and its blockchain. Moreover, the space of potential future work is ever more growing, from trusted currency exchanges to blockchains built solely on TEEs with various different application goals.

6.3 Final Remarks

In this thesis, the main goal was to explore the new Intel SGX trusted execution environment by identifying potential vulnerabilities and providing solutions for them, investigating the space of possible applications and new concepts that the usage of SGX can allow, and finally, present the benefits of an SGX application in the privacy domain. Specifically, we have identified the vulnerability in long-term sealed enclave storage that lacked the freshness integrity, thus enabling rollback attacks. Our work resulted in a distributed system that offers complete rollback protection for enclave applications. Further on, we have designed a novel concept that we call brokered delegation with which we enable users to securely and selectively delegate their own access rights and credentials to third parties using rich-contextual access control policies, thus tailoring the delegated access to specific needs. The concept was prototyped on top of four real-world use case scenarios in two design spaces (peer-to-peer and centrally brokered) to show the potential of enabling delegation for any existing service, without the support of the service itself. Lastly, we dwell into exploiting SGX for increasing user privacy and develop two privacy-preserving services that allow increased adoption of cryptocurrencies Bitcoin and Zcash, by allowing mobile, resource constrained, clients to use the system under the same trust and privacy assumption that are guaranteed by the currency itself. To achieve privacy we carefully crafted a side-channel free solution using Intel SGX, which is by design, susceptible to it, thus solving another SGX shortcoming for a specific use-case scenario. We conclude that the work of this thesis tackled several different angles of the emergence and application of Intel SGX, and showed that it is a new, prospective TEE on the path of changing how we think and design security.

Appendix A

ROTE

A.1 SGX Counter Analysis

Intel has recently added support for monotonic counters [103] as an optional SGX feature that an enclave developer may use for rollback attack protection. However, the security and performance properties of this mechanism are not well documented. Furthermore, they are not available on all platforms. In this Appendix we outline all executed experiments and evaluate the SGX counter and trusted time service.

SGX counter service. An enclave can query availability of counters from the Platform Service Enclave (PSE). If supported, the enclave can create up to 256 counters. The default owner policy encompasses that only enclaves with the same signing key may access the counter. Counter creation operation returns an identifier that is a combination of the Counter ID and a nonce to distinguish counters created by different entities. The enclave must store the counter identifier to access it later, as there is no API call to list existing counters. After a successful counter creation, an enclave can increment, read, and delete the counter.

According to the SGX API documentation [103], counter operations involve writing to a non-volatile memory. Repeated write operations can cause the memory to wear out, and thus the counter increment operations may be rate limited. Based on Intel developer forums [98], the counter service is provided by the Management Engine on the Platform Control Hub (PCH).

Experiments. We tested SGX counters on five different platforms: Dell Inspiron 13-7359, Dell Latitude E5470, Lenovo P50, Intel NUC and Dell Optiplex 7040. The counter service was not available on Intel NUC. On Dell

laptops a counter increment operation took approximately 250 ms, while on the Lenovo laptop and Dell Optiplex increment operations took approximately 140 ms and 80 ms, respectively. Strackx et al. [182] report 100 ms for counter updates. Counter read operations took 60-140 ms, depending on the platform. As expected, the counter values remained unchanged across enclave restarts and platform reboots. We tested the wear-out characteristics of the counters and found out that on both Dell laptops, after approximately 1.05 million writes, the tested counter became unusable and other counters on the same platform could not be created, incremented or read (all SGX counter operations return `SGX ERROR BUSY`). Additionally, we observed that reinstalling the SGX Platform Software (PSW) or removing the BIOS battery deletes all counters. Finally, to our surprise, we noticed that after reinstalling the PSW, first usage of counter service triggered the platform software to connect to a server whose domain is registered to Intel. If Internet connection is not available, the counters are unavailable. We have no good explanation why a connection to an Intel server is needed after the PSW reinstall. Similarly, we do not know why the SGX counters become unavailable after BIOS battery removal or PSW reinstall.

Performance limitations. An enclave developer could attempt to use SGX counters as a rollback protection mechanism. When an enclave needs to persistently store an updated state, it can increment a counter, include the counter value and identifier to the sealed data, and verify integrity of the stored data based on counter value at the time of unsealing. However, such approach may wear out the used non-volatile memory. Assuming a system that updates one of the enclaves on the same platform once every 250 ms, counters would become unusable in few days. Even with a modest update rate of one increment per minute, the counters are exhausted in two years. Services that need to process tens or hundreds of transactions per second are not possible.

Weaker security model. According to Intel developer forums, counter service is provided by the Management Engine on the PCH (known as "south bridge" in older architectures) [98]. However, to the best of our knowledge, actual location of the non-volatile memory used to store the counters is not publicly stated. Based on Intel specifications [96, 99], the PCH typically does not host non-volatile memory, but it is connected over an SPI bus to a flash memory that is also used by the BIOS. Since Management Engine is an active component, communication between the processor and the Management Engine can be replay protected. However, the SPI flash is a passive component, and therefore any counter stored there is likely to be vulnerable to bus tapping and flash mirroring attacks, as recently demonstrated in the case of mobile devices (inspired by FBI iPhone unlocking debate) [177]. Although the precise

storage location of SGX counters remains unknown at the time of writing, it is clear that if the integrity of enclave data relies on the SGX counter feature, then additional hardware components besides the processor must be considered trusted. This is a significant shift from the enclave execution protection model, where the security perimeter is the processor package [97, p. 30].

Other concerns. The current design of SGX counter APIs makes safe programming difficult. To demonstrate this we outline a subtle rollback attack. Assume an enclave that at the beginning of its execution checks for the existence of sealed state, and if one is not provided by the OS, it creates a new state and counter, and stores the state sealed together with the counter value and identifier. The enclave increments the counter after every state update. Later, the OS no longer provides a sealed state to the restarted enclave. The enclave assumes that this is its first execution and creates a new (second) counter and new state. Recall that the SGX APIs do not allow checking existence of previous counter. The enclave updates its state again. Finally, the OS replays a previous sealed state associated with the first counter. A careful developer can detect such attacks by creating and deleting 256 counters (an operation that takes two minutes) to check if a previous counter, and thus sealed state, exists. A crash before counter deletion would render that particular enclave permanently unusable.

The above attack and availability issues probably could be fixed with better design of SGX APIs and system services, but the performance limitations and the weaker security model are hard to avoid in future SGX versions.

SGX trusted time. Another recently introduced and optional SGX feature is the trusted time service [104]. As in the case of SGX counters, also the time service is provided by the Management Engine. The trusted time service allows an enclave developer to query a time stamp that is relative to a reference point. The function returns a nonce in addition to the timestamp, and according to the Intel documentation, the timestamp can be trusted as long as the nonce does not change [104].

We tested the time service and noticed that the provided nonce remained same across platform reboots. Reinstalling PSW resulted in a different nonce, but the provided time was still correct. The reference point is the standard Unix time. As a rollback protection mechanism the trusted time service is of limited use. Including a timestamp to each sealed data version allows an enclave to distinguish which out of two seals is more recent. However, the enclave cannot know if the sealed data provided by the OS is fresh and latest.

A.2 Identified Vulnerabilites

We analyzed several recent SGX systems through our model and found out that many of them have security issues if the adversarial capabilities that we identified are considered. Below we summarize our findings.

S-NFV [171] tries to preserve the internal state of the Network Function Virtualization (NFV) applications by utilizing the isolation guarantees of SGX. The enclave state could be, e.g., data in a Content Delivery Network (CDN) or routing policies for the internal network. If the adversary creates multiple instances, he can direct all update requests from the external client to one instance and upon client requests, feed stale data from the other one, resulting in a rollback. The system could be protected by using TPM PCR, as outlined in Section 3.7, or by instantiating an SGX monotonic counter and accepting only the first index (counter ID) upon creation.

ICE [180] aims to achieve state continuity for protected applications such as SGX enclaves. ICE enhances the CPU with protected volatile memory (called guard), a power supply and a capacitor that at system shutdown flushes the latest state to non-volatile memory. The adversary model assumes a compromised OS through which the clients communicate with the protected applications. *libice0* and *libicen*, which are similar to the ASEs in our system, rely on a single state-continuous module *ice0*. As an optimization, *libice0* and *libicen* store a cached copy of their state and return this to the client without calling *ice0* if they have not been restarted. The adversary can create a *libicen* instance and let it retrieve its state. Once the retrieval is complete, it can create another instance which will prompt another state retrieval. Now, the attacker can direct all the writes to the second instance and when a read request comes in, it can direct it to the first instance that reports a stale state to the client. The *ice0* module needs to be a single, unique instance for the state continuity guarantees to hold. If the adversary forks the *ice0* module, it is able to compromise all the *libicen* modules' state continuity.

A possible countermeasure to the first attack is to not cache the states at all and request it from *ice0* at every read. This obviously has performance drawbacks given the TPM chip's rate limiting. Another possible defense is to assign specific SGX monotonic counter IDs to specific *libicen* modules. Since the counter IDs share a global namespace, it is possible to see how many counters already exist on the system. This approach has the drawback of losing crash resilience. If a module crashes before deleting its counter, it cannot recover.

VC3 [169] creates a distributed system of nodes that perform MapReduce computation with integrity and confidentiality guarantees. The system

leverages SGX for protected execution and remote attestation. The paper acknowledges that the *in-band* variant of VC3 is vulnerable to replays and suggests using ICE for rollback protection. Thus, the above discussed issues apply also VC3. In the *online* variant users could typically detect replays.

Appendix B

DelegaTEE

B.1 Case by case Security analysis

In this section we show that even in specific cases where the attacker controls machines with compromised SGX or holds secret SGX keys, our system still preserves the relevant security properties. Below we outline assumptions along with the attacker model, followed by a detailed analysis on how the security properties hold.

In both models (P2P and Centrally Brokered), we presume that the attacker has full control of the network described as the standard Dolev-Yao network adversary [58]. Additionally, we trust the service provider since the Owner and the Delegatee are already using its service. There will be no collusion between the service provider and the Delegatee. Also, the external attacker and the Delegatee as an attacker are two separate entities and when one is present the other is not, thus collusion between them is not possible as well. We intrinsically trust Intel, as the manufacturer of processors and SGX creator on top of which the secure enclaves are running. We trust the secure enclaves to execute code under properties of its architecture.

Our system is designed to provide the following main security properties:

(**a**) *First and foremost, the Owner's access credentials remain confidential.*

(**b**) *The use of the delegated credentials is defined by the access control policy which will not be violated.*

(**c**) *Use of the credentials should only be granted to the intended Delegatee, as authorized by the Owner.*

The security analysis is structured by different case scenarios (further referred as Case 1-4) that might occur as an intersection of both the P2P and Centrally Brokered system of DELEGATEE with an external or Delegatee as an attacker. Furthermore, the cases are divided with respect to each of the security guarantees (Subcases a-c).

The attacker can re-route the network traffic, delay or modify the messages, but due to the use of an authenticated and confidential channel the attacker cannot gain any advantage, yet only cause a DoS attack which is out of scope.

1) External Attacker in the P2P system

- We assume that the attacker has access to an arbitrary number of SGX-enabled systems and ***can*** compromise a set of SGX enclaves (i.e., can extract all SGX keys).
- We assume that the attacker ***cannot*** compromise the system of the Owner.
- We assume that the attacker ***cannot*** compromise the systems of the Delegatees. However, the attacker ***can*** have the SGX keys of the Delegatees' machines.

Case 1-(a): In order to obtain the Owner's credentials the external attacker can do the following. She can try to steal the credentials from the Owner at the time of input into the P2P system, however, the attacker is not present on the Owner's software stack, and the connection to the enclave is secured end-to-end. Thus the credentials cannot be stolen. Additionally, she can try to use one of her machines that have compromised SGX in order to trick the Owner into delivering the credentials. However, the Owner and the Delegatee perform an initial agreement over an authenticated and confidential channel (e.g. messenger, a cup of coffee, etc.). They use the channel to exchange a symmetric key (or a certificate; depending on the implementation[1], as discussed in Section 4.3.1) that is used for subsequent authentication when the credentials are delegated. Since the external attacker cannot compromise the system (full software stack) of both the Owner and the Delegatee, the initial exchange of the shared secret and further authentication of the enclave is secure. Thus, the attacker's ability to own a set of machines with compromised SGXs yields no advantage and all re-routed requests to some machines other than the Delegatee's will fail upon authentication. The attacker's knowledge of the Delegatee's SGX keys is useless due to the inability to be present on the Delegatee's software stack. Thus, the attacker cannot decrypt the runtime

[1]The method does not influence the security analysis and in this case we have chosen the pre-shared key mode.

memory where the credentials reside after they are delegated to the enclave since she cannot access that part of the system.

Case 1-(b): The external attacker wants to violate the access control policy defined by the credential Owner to extend the rights and use the credentials in an unwanted way, causing harm to the Owner. Due to the fact that the attacker cannot compromise the full software stack of the Owner, she cannot modify or influence the process of defining the policy. After the policy has been defined, it is delivered to the enclave of the Delegatee and remains confidential. Even though the attacker possesses the SGX keys of the Delegatee's system, she has no access to the software stack of the Delegatee, thus modifying the policy is not possible. Furthermore, as explained, the attacker cannot impersonate the Delegatee by using one of his compromised SGX machines, thereby she cannot gain access to the service even under the defined access control policy, nor perform a violation of it.

Case 1-(c): In order to achieve that an unauthorized user can access some service using the Owner's credentials the attacker might try to steal the exchanged secret key used for authentication. However, if we presume that the attacker does not have the control over the full software stack of the Delegatee and the Owner, she cannot acquire the shared secret that they agreed upon. Without it, the attacker is not able to authenticate to the Owner after the enclave starts even with the machines that have the compromised SGX. Thus, only the authorized Delegatee will be able to use the credentials.

2) Delegatee as Attacker in the P2P system

- We assume that an attacker has access to an arbitrary number of SGX-enabled systems and ***cannot*** compromise a set of SGX enclaves.
- We assume that the attacker ***cannot*** compromise the system of the Owner.
- We assume that the attacker controls the system of the Delegatee. However, the attacker ***cannot*** compromise the enclaves running on the Delegatee's machine nor has access to the SGX keys of that machine.

Case 2-(a): In this case the delegatee/attacker aims to obtain the Owner's credentials. The Delegatee can use any of the machine available to her to initiate the credential delegation according to the P2P system model. However, the attacker cannot compromise the SGX nor does it have the underlying SGX keys of those machines and is, therefore, unable to extract the delivered credentials that remain in the runtime memory of the enclave. Additionally, the attacker does not have access to the Owner's system which prevents her from sniffing

the credentials on input into the enclave. Taking into consideration all given abilities of the attacker, the attacker cannot reveal the confidential credentials if we presume safe delivery of them to the enclave (Owner performs attestation of the enclave after which Delegatee authenticates and secure communication is established from the Owner directly to the enclave, delivering the credentials subsequently...).

Case 2-(b): The Delegatee aims to violate the access control policy set by the Owner in order to use the delegated credentials in ways not defined by the policy. The security reasoning for this case closely follows the above Case 2-(a). Since the adversary cannot compromise the SGX in the machine she uses for delegation, she cannot access the runtime memory of the enclave and perform modifications to the access control policy. Thus, if we presume that the policies are defined correctly, the Delegatee cannot violate them during enclave execution. Additionally, the P2P system model is operational only on a single run, indicating that when credential delegation is executed, the enclave does not store or seal any data to disk. This prevents the adversary from performing a rollback attack [144]. Enclave monitors the usage of delegated credentials and records all activities. If the execution could be stopped and continued at a later time, the attacker could offer an older seal to the enclave thereby evading the limits defined in the policy. We prevent these attacks with a simple constraint that all data is lost after the enclave is stopped and the whole system has to be re-initiated to allow further use of delegated credentials.

Case 2-(c): Here we consider that Delegatee is the attacker. She is therefore already authorized to use the Owner's credentials. This property can be violated only if the Delegatee forwards the shared secret created with the Owner to another user. This action is out of scope.

3) External Attacker in the Centrally Brokered system

- We assume that an attacker has access to an arbitrary number of SGX-enabled systems and ***can*** compromise a set of SGX enclaves.
- We assume that the attacker ***cannot*** compromise the system of the Owner.
- We assume that the attacker ***cannot*** compromise the systems of the Delegatees[2].

[2]Delegatees do not use SGX in the Centrally Brokered system, thus the difference from the external attacker model from the P2P system

• 1st possibility: Attacker ***can*** compromise the full software stack of the Centrally Brokered system. However, we assume that he doesn't have access to the SGX keys and ***cannot*** compromise the system enclaves.

• 2nd possibility: Attacker ***cannot*** compromise the full software stack of the Centrally Brokered system. Thus, he ***cannot*** compromise the enclaves on the servers, but he ***can*** have access to the SGX keys of the server.

Case 3-(a): In order to obtain the Owner's credentials, the attacker has the following choices. She can try to forward all requests addressed to the central system into her own set of machines where the SGX is compromised. Here we have two possibilities. Firstly, the attacker has the ability to compromise the full software stack of the central system but not the SGX enclaves running there. In this way, all requests can be re-routed with ease. However, we use an end-to-end secure connection from the users to the enclaves and the attacker cannot modify the communication. Additionally, users can always attest and authenticate the enclave, verifying its origin[3]. This prevents the attacker from impersonating the server and acquiring confidential credentials into machines with compromised SGX. Secondly, if the attacker cannot compromise the full software stack and the running enclaves, she might possess the SGX keys of the central system. she can try to use these keys in machines with compromised SGX to trick users into believing that they communicate with the correct system. Nevertheless, the authentication step will fail even though the attestation will succeed due to the inability of the attacker to own the secret certificate key (note that without access to the software stack of the central system the attacker has no way of obtaining the full certificate information nor to extract the enclave seal which could be later on decrypted with the SGX keys she owns). Thus, impersonating the central system is not possible by the external attacker. The attacker can try to steal the credentials from the Owner at the time of input to the system but since she does not control the Owner's software stack, credentials remain confidential.

Case 3-(b): In this case the attacker aims to violate the access control policy defined by the credential Owner. The attacker may try to modify the credentials when the Owner is defining them in the system. However, the attacker has no control over the Owner's system and the connection to the central system is secured (see Case 3-(a)). Thus, this attack vector is not successful. The attacker might try to circumvent the policy enforcement in the enclave during execution

[3]The Centrally Brokered system has a public certificate signed by a certificate authority, such as Verisign. Delivering the certificate private key to the enclave is presumed to be secure and works on the trust-on-first-use principle when the central system is bootstrapped.

of some service with the delegated credentials or change the policies inside the central system. In respect to the two possibilities that the attacker has, the policy cannot be violated since the attacker cannot change the defined policy (if he has access to the software stack he does not have the SGX keys and cannot compromise the running enclaves, thus preventing any modification of the policy, while if the attacker has the SGX keys of the central system, she is not present on the system and cannot extract the encrypted sealed nor the runtime memory where the policy resides) and cannot interrupt and compromise the policy enforcement (the attacker cannot compromise running enclaves when the system is compromised and when the system is not compromised she has no access to the execution environment, thus preventing any circumvention of the defined policy).

Case 3-(c): In order to use the Owner's credential without implicit authorization the attacker has the following possibilities. She might try to steal the login information of the authorized Delegatee in order to access the central system or obtain the login information of the Owner to modify the details of the credential delegation and authorize somebody else (i.e., herself). However, since the system uses end-to-end secure connection between all users and enclaves residing on the central system, and since the attacker does not control the software stack of either the Owner nor the Delegatee, she cannot sniff the network traffic or log the user input upon login into the system. Without knowing the login information (i.e., username and the password) of the Owner and the Delegatee, the attacker has no other means of accessing the central system and using the delegated credentials. Additionally, she can try to modify the information about the authorized Delegatees directly on the central system, however, analysis shown in Case 3-(a) implies that the attacker cannot perform any modifications.

4) Delegatee Attacker in the Centrally Brokered system

- We assume that an attacker has access to an arbitrary number of SGX-enabled systems and ***can*** compromise a set of SGX enclaves.
- We assume that the attacker ***cannot*** compromise the system of the Owner.
- We assume that the attacker controls the system of the Delegatee. Furthermore, the Delegatee's SGX system is not involved in the protocol and thus out of scope.
- 1st/2nd possibility: Same as Case 3 (external attacker in the Centrally Brokered system).

Case 4-(a): In this case the Delegatee's role is the same as in the case of an external attacker and the Centrally Brokered system. No other attack capabilities exist (except the control over the full software stack at the Delegatee, which yields no advantage) that extend the external attacker and thus the same security analysis is applied as in the Case 3-(a), the Owner's credentials remain confidential. SGX at the Delegatee's side is not relevant when the Centrally Brokered system is deployed.

Case 4-(b): Similarly as above, the Delegatee's role is the same as of the external attacker with the difference that the Delegatee own login information to connect to the central system. However, this does not enable any new attack vectors that could be used to violate the access control policy. Even the external attacker could register to the central system, thereby acquiring authorized login information. Following the analysis of Case 3-(b), it is straightforward to conclude that the policy referring to the use of Owner's credentials cannot be violated.

Case 4-(c): Here we consider that the Delegatee is the attacker. She is therefore already authorized to use the Owner's credentials. As in Case 2-(c), this property can be violated only if the Delegatee forwards her login information to another user. This action is out of scope in this analysis.

B.2 DelegaTEE Prototype Demo

In this section we show the prototype demo through screenshots when a Delegatee, Alice, is buying something or logging in to a website using DELEGATEE. First, Bob enters his credentials into DELEGATEE and delegates them to Alice. Alice then logs into the browser extension (Figure B.1, Figure B.2) and the new button appears next to the PayPal checkout button (Figure B.3), the credit card/e-banking checkout button (Figure B.4) or the login button (Figure B.5). All the buttons and the dialog get injected to the website by the browser extension. After clicking the DELEGATEE button, Alice is presented with a list of delegated credentials to choose from (Figure B.6). Upon selecting some credentials, the enclave takes over and completes the transaction and Alice is redirected to the confirmation page. If a CAPTCHA has to be solved to continue with the transaction, the user is asked to solve it (Figure B.7).

Receiving and sending emails using delegated credentials can be done with our mail client for DELEGATEE. The client allows viewing of the inbox and reading single mails of the delegated email account (Figure B.8). Sending emails is also supported (Figure B.9).

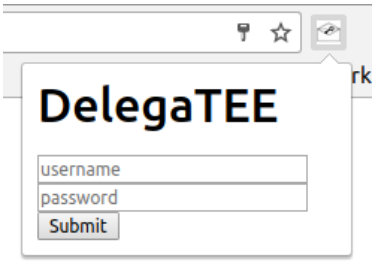

Figure B.1: *Browser extension: Login.*

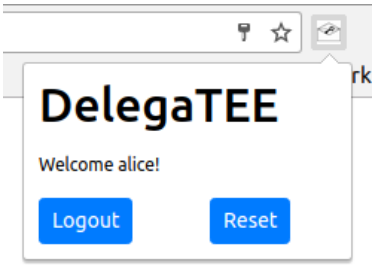

Figure B.2: *Browser extension: Welcome.*

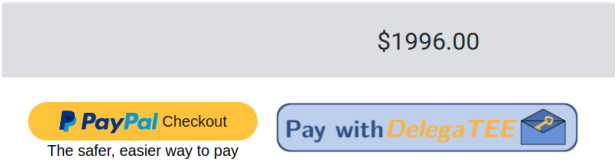

Figure B.3: *Extra button rendered next to the PayPal checkout button.*

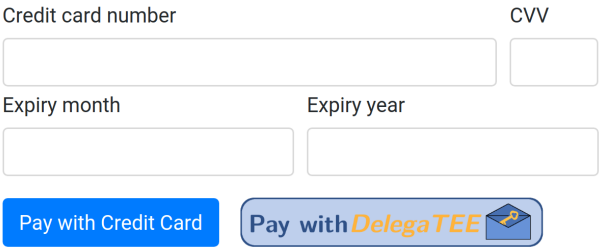

Figure B.4: *Extra button rendered next to the credit card checkout button.*

Figure B.5: *Extra button rendered next to the login button.*

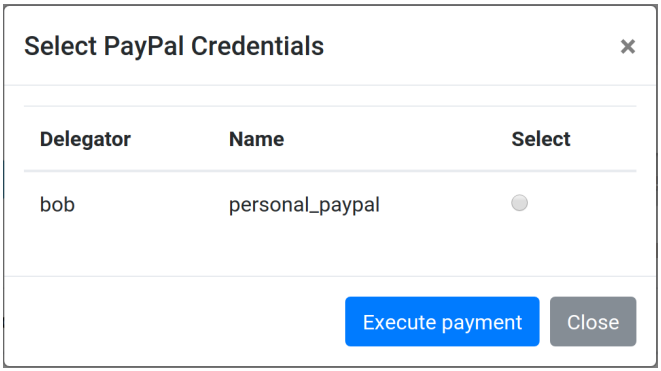

Figure B.6: *Delegated credentials selection.*

Figure B.7: *The Delegatee is asked to solve CAPTCHA.*

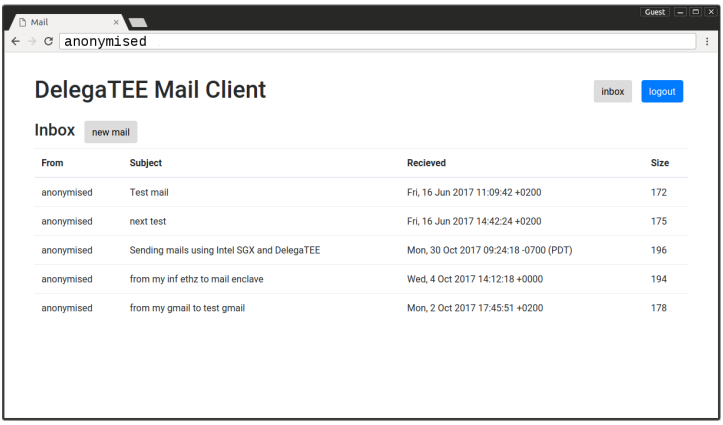

Figure B.8: *Receiving mail client example.*

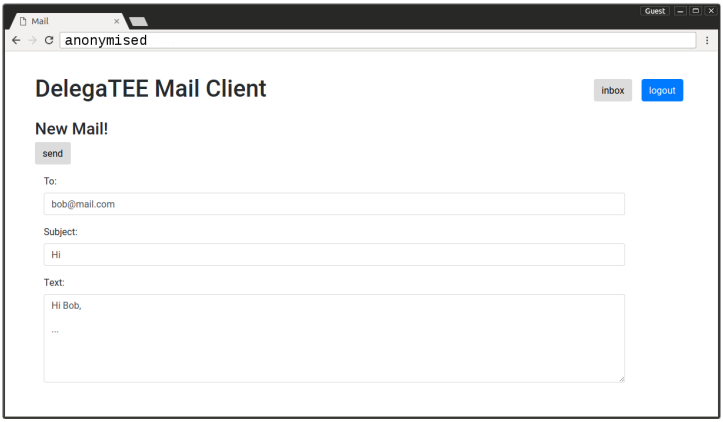

Figure B.9: *Sending mail client example.*

Appendix C

ZLiTE

C.1 Commitment Tree Updates

As described in Section 5.8.4, the commitment tree update U_{ct} for the interval between time t_1 and t_2 consists of the right child of the path from cm_i to the root at time t_2, where cm_i is the rightmost non-empty leaf at time t_1.

In Figure C.1, we show an example for the commitment tree update. In this example, the leaf f is the rightmost non-empty leaf at t_1, i.e., it corresponds to cm_i, which means that the commitment tree update consists of the values of the nodes N11, N5, N3 at time t_2. In the example, the update is applied to the witness of the leaf c (consisting of the nodes d, N8, N5, and N3). In this case, the values of the leaf d and node N8 do not change between time t_1 and t_2, the values of N5 and N3 do, however, and thus the values are contained in the commitment tree update and updated from there.

We now show that given a witness at time t_1 for a commitment cm_j (where $j < i$, i.e., cm_j was added to the tree before cm_i) and the commitment tree update U_{ct}, a client can compute the witness for cm_j at time t_2.

Let A_{ji} be the lowest common ancestor node of cm_j and cm_i in the commitment tree, i.e., cm_j is in the left subtree of A_{ji} and cm_i is in the right subtree. Any node in the left subtree of A_{ji} remains unchanged between t_1 and t_2, i.e., any node from that subtree which is part of the witness for cm_j also remains unchanged. Since none of these nodes changes through the update process, updating the witness with U_{ct} results in the correct values.

Similarly, any node of the witness for cm_j that is a left child of a node on the path from A_{ji} to the root remains unchanged in the Merkle tree at time t_2, since all leafs in any left subtree are already fixed at time t_1 and thus all node

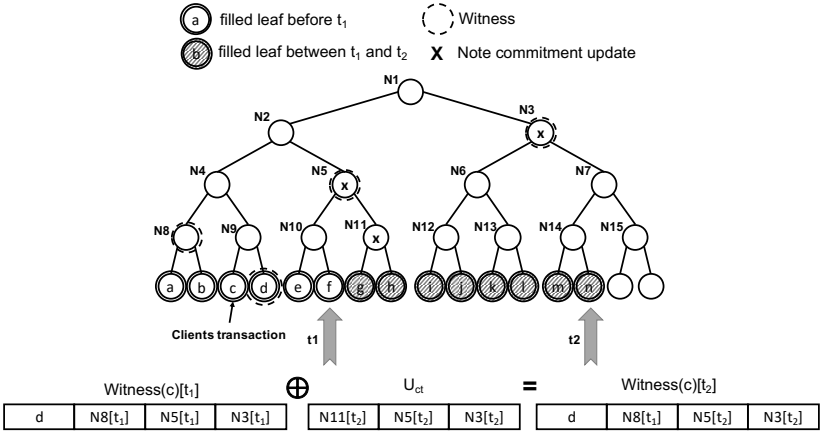

Figure C.1: *At a time t_1 the note commitments Merkle tree is fully updated up to the latest block. A specific client holds a transaction with a note commitment c and knows the witness (i.e., the Merkle path) for it (d, N8, N5, and N3 nodes). After some time the blockchain is updated and new transactions added, thus, the Merkle Tree is updated accordingly (t_2). In order for the client to update the witness of her commitment c, she only needs the updated information from nodes (N11, N5, N3).*

values are already final. Since our update process does not change any left children in the tree, it also leaves these values unchanged and thus results in the correct values.

Finally, any node of the witness for cm_j that is a left child of a node on the path from A_{ji} to the root may change in the Merkle tree at time t_2. Since A_{ji} is an ancestor of cm_i, any such node is included in U_{ct}, i.e., these nodes on the witness are updated in our update process. These values are therefore changed to the correct values from the note commitment tree at time t_2.

It follows that the witness at time t_2 for cm_j can be constructed correctly given the witness at time t_1 and the commitment tree update U_{ct}.

Bibliography

[1] Innovative Technology for CPU Based Attestation and Sealing, 2013. https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing.

[2] BitcoinJ, 2018. https://bitcoinj.github.io/.

[3] Blockchain.info, 2018. https://blockchain.info.

[4] Electrum, 2018. https://electrum.org/#home.

[5] Ethereum, 2018. https://www.ethereum.org/.

[6] Etherscan.io, 2018. https://etherscan.io.

[7] Light Ethereum Subprotocol (LES), 2018. https://github.com/zsfelfoldi/go-ethereum/wiki/Light-Ethereum-Subprotocol-%28LES%29.

[8] OpCodes: CMOV, 2018. http://www.rcollins.org/p6/opcodes/CMOV.html.

[9] PicoCoin, 2018. https://github.com/jgarzik/picocoin.

[10] R3, 2018. https://www.r3.com/.

[11] Ripple, 2018. https://ripple.com/.

[12] Sapling, 2018. https://z.cash/upgrade/sapling.html.

[13] Trusted Computing Group, 2018. https://trustedcomputinggroup.org/.

[14] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and A. Spiegelman. Solidus: An Incentive-compatible Cryptocurrency ased on Permissionless Byzantine Consensus. *Computing Research Repository (CoRR), arXiv abs/1612.02916*, 2016.

[15] A. Adams and M. A. Sasse. Users Are Not The Enemy. *Communications of the ACM*, 42(12):40–46, 1999.

[16] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. OBLIVIATE: A Data Oblivious File System for Intel SGX. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[17] B. Alexander. Introduction to Intel SGX Sealing, 2016. `https://software.intel.com/en-us/blogs/2016/05/04/introduction-to-intel-sgx-sealing`.

[18] T. Alves and D. Felton. TrustZone: Integrated Hardware and Software Security-Enabling Trusted Computing in Embedded Systems, 2004. `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`.

[19] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2013.

[20] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the 13th EuroSys Conference*. ACM, 2018.

[21] E. Androulaki, G. O. Karame, M. Roeschlin, T. Scherer, and S. Capkun. Evaluating User Privacy in Bitcoin. In *International Conference on Financial Cryptography and Data Security*. Springer, 2013.

[22] Apple. iOS Security. Whitepaper, 2017. `https://www.apple.com/business/docs/iOS_Security_Guide.pdf`.

[23] ARM. ARM Security Technology. Building a Secure System using TrustZone Technology, 2009.

[24] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, et al. SCONE: Secure Linux Containers with Intel SGX. In *11th USENIX Symposium on Operating Systems Design and Implementation (USENIX OSDI)*, 2016.

[25] T. W. Arnold, C. Buscaglia, F. Chan, V. Condorelli, J. Dayka, W. Santiago-Fernandez, N. Hadzic, M. D. Hocker, M. Jordan, T. Morris, et al. IBM 4765 Cryptographic Coprocessor. *IBM Journal of Research and Development*, 56(1.2), 2012.

[26] N. Asokan, J.-E. Ekberg, K. Kostiainen, A. Rajan, C. Rozas, A.-R. Sadeghi, S. Schulz, and C. Wachsmann. Mobile Trusted Computing. *Proceedings of the IEEE*, 102(8):1189–1206, 2014.

[27] A. Atamli-Reineh, A. Paverd, G. Petracca, and A. Martin. A Framework for Application Partitioning using Trusted Execution Environments. *Concurrency and Computation: Practice and Experience*, 29(23), 2017.

[28] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[29] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (Im)Possibility of Obfuscating Programs. In *Annual International Cryptology Conference*. Springer, 2001.

[30] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. 37(5):164–177, 2003.

[31] L. Bauer, M. A. Schneider, and E. W. Felten. A General and Flexible Access-Control System for the Web. In *Proceedings of the 11th USENIX Security Symposium (USENIX Security)*, 2002.

[32] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications From An Untrusted Cloud With Haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.

[33] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP)*, 2014.

[34] A. Birgisson, J. G. Politz, U. Erlingsson, A. Taly, M. Vrable, and M. Lentczner. Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[35] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[36] J. Bohannon. Who's Downloading Pirated Papers? Everyone. *American Association for the Advancement of Science*, 2016.

[37] N. Borisov and E. A. Brewer. Active Certificates: A Framework for Delegation. In *Proceedings of the 9th Annual Network and Distributed System Security Symposium (NDSS)*, 2002.

[38] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza. Rollback and Forking Detection for Trusted Execution Environments using Lightweight Collective Memory. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017.

[39] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, U. Müller, and A. Sadeghi. DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization, 2017.

[40] F. Brasser, U. Muller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[41] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Proceedings of the 17th International Middleware Conference*. ACM, 2016.

[42] C. Cachin, D. Dobre, and M. Vukolić. Separating data and control: Asynchronous bft storage with 2t+ 1 data replicas. In *Symposium on Self-Stabilizing Systems*, pages 1–17. Springer, 2014.

[43] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Science & Business Media, 2011.

[44] D. D. Caputo, S. L. Pfleeger, M. A. Sasse, P. Ammann, J. Offutt, and L. Deng. Barriers to Usable Security? Three Organizational Case Studies. *IEEE Security & Privacy*, 14(5):22–32, 2016.

[45] M. Castro, B. Liskov, et al. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (USENIX OSDI)*, 1999.

[46] E. Cecchetti, F. Zhang, Y. Ji, A. E. Kosba, A. Juels, and E. Shi. Solidus: Confidential Distributed Ledger Transactions via PVORM. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[47] D. Champagne and R. B. Lee. Scalable Architectural Support for Trusted Software. In *Proceedings of the 16th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2010.

[48] S. Checkoway and H. Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. *ACM*, 41(1), 2013.

[49] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution. *Computing Research Repository (CoRR), arXiv abs/1802.09085*, 2018.

[50] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjá Vu. In *Proceedings of the 12th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2017.

[51] A. Chiesa, M. Green, J. Liu, P. Miao, I. Miers, and P. Mishra. Decentralized Anonymous Micropayments. In *36th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2017.

[52] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested Append-only Memory: Making Adversaries Stick to their Word. *ACM SIGOPS Operating Systems Review (OSR)*, 41(6):189–204, 2007.

[53] J.-S. Coron, M. S. Lee, T. Lepoint, and M. Tibouchi. Zeroizing Attacks on Indistinguishability Obfuscation over CLT13. In *IACR International Workshop on Public Key Cryptography*. Springer, 2017.

[54] M. Correia, N. F. Neves, and P. Verissimo. How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*. IEEE, 2004.

[55] V. Costan and S. Devadas. Intel SGX explained. In *Cryptology ePrint Archive, Report 2016/086*, 2016.

[56] V. Costan, I. A. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, 2016.

[57] S. A. Crosby and D. S. Wallach. Efficient data structures for tamper-evident logging. In *Proceedings of the 18th USENIX Security Symposium (USENIX Security)*, 2009.

[58] D. Dolev and A. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[59] L. Duflot, D. Etiemble, and O. Grumelard. Using CPU System Management Mode to Circumvent Operating System Security Functions. *CanSecWest/core06*, 2006.

[60] P. Dutta, R. Guerraoui, and M. Vukolic. Best-case complexity of asynchronous byzantine consensus. Technical report, Technical Report EPFL/IC/200499, EPFL, 2005.

[61] S. Egelman, L. F. Cranor, and J. Hong. You've Been Warned: An Empirical Study of the Effectiveness of Web Browser Phishing Warnings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2008.

[62] J. Ekberg, K. Kostiainen, and N. Asokan. The Untapped Potential of Trusted Execution Environments on Mobile Devices. *IEEE Security Privacy*, 12(4), 2014.

[63] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, 2012.

[64] C. W. Fletcher, M. v. Dijk, and S. Devadas. A Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *Proceedings of the 7th ACM workshop on Scalable Trusted Computing*, 2012.

[65] J. A. Garay and K. J. Perry. A Continuum of Failure Models for Distributed Computing. In *International Workshop on Distributed Algorithms*. Springer, 1992.

[66] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-based Platform for Trusted Computing. 37(5):193–206, 2003.

[67] M. Gasser and E. McDermott. An architecture for practical delegation in a distributed system. In *Proceedings of the 1990. IEEE Symposium on Security and Privacy (SP)*, 1990.

[68] A. Gervais, S. Capkun, G. Karame, and D. Gruber. On the Privacy Provisions of Bloom Filters in Lightweight Bitcoin Clients. In *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014.

[69] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[70] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *null*, page 135. IEEE, 2004.

[71] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*. ACM, 2017.

[72] D. Grawrock. Dynamics of a Trusted Platform: A building block approach. Intel Press, 2009.

[73] D. Grawrock. *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 2009.

[74] M. Green and I. Miers. Bolt: Anonymous Payment Channels for Decentralized Currencies. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[75] T. C. Group. TPM Design Principles and Structures. 2003–2011.

[76] T. C. Group and Others. TPM Main Specification, 2003. `http://www.trustedcomputinggroup.org/resources/tpm\_main\_specification`.

[77] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, 2017.

[78] R. Guerraoui, R. R. Levy, and M. Vukolic. Lucky read/write access to robust atomic storage. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 125–136. IEEE, 2006.

[79] R. Guerraoui and M. Vukolić. How fast can a very robust read be? In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 248–257. ACM, 2006.

[80] R. Guerraoui and M. VukoliĆ. Refined quorum systems. *Distributed Computing*, 23(1):1–42, 2010.

[81] D. Gupta, B. Mood, J. Feigenbaum, K. Butler, and P. Traynor. Using Intel Software Guard Extensions for Efficient Two-Party Secure Function Evaluation. In *International Conference on Financial Cryptography and Data Security*. Springer, 2016.

[82] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical Accountability for Distributed Systems. *ACM SIGOPS Operating Systems Review (OSR)*, 41(6):175–188, 2007.

[83] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold-boot Attacks on Encryption Keys. *Communications of the ACM*, 52(5):91–98, 2009.

[84] E. Hammer-Lahav. The Oauth 1.0 Protocol. RFC 5849, 2010. `https://tools.ietf.org/html/rfc5849`.

[85] D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, 2012. `https://tools.ietf.org/html/rfc6749`.

[86] M. Hearn. Bloom Filter Privacy and Thoughts on a Newer Protocol, 2015. `https://groups.google.com/forum/#!msg/bitcoinj/Ys13qkTwcNg/9qxnhwnkeoIJ`.

[87] M. Hearn and M. Corallo. Connection Bloom Filtering. *Bitcoin Improvement Proposal*, 37, 2012. `https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki`.

[88] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg. Eclipse Attacks on Bitcoin's Peer-to-Peer Network. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, 2015.

[89] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using Innovative Instructions to Create Trustworthy Software Solutions. *HASPA ISCA*, 11, 2013.

[90] C. Hoffman. Intel Management Engine Explained, 2018. https://www.howtogeek.com/334013/intel-management-engine-explained-the-tiny-computer-inside-your-cpu/.

[91] J. Howell and D. Kotz. An Access-Control Calculus for Spanning Administrative Domains. *Dartmouth College*, 1999.

[92] B. D. O. Hub. XEN Hypervisor, 2018. https://www.openhub.net/p?ref=homepage&query=xen.

[93] Intel. Intel SGX for Dummies (Design Objectives), 2013. https://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx.

[94] Intel. Intel SGX for Dummies - Part 2, 2014. https://software.intel.com/en-us/blogs/2014/06/02/intel-sgx-for-dummies-part-2.

[95] Intel. Intel SGX for Dummies - Part 3, 2014. https://software.intel.com/en-us/blogs/2014/09/01/intel-sgx-for-dummies-part-3.

[96] Intel. Intel 9 Series Chipset Family Platform Controller Hub (PCH), 2015. http://www.intel.com/content/www/us/en/chipsets/9-series-chipset-pch-datasheet.html.

[97] Intel. Intel SGX, Ref. No.: 332680-002, 2015. https://software.intel.com/sites/default/files/332680-002.pdf.

[98] Intel. Developer Zone Forums, 2016. https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/607330.

[99] Intel. Intel 100 Series and Intel C230 Series Chipset Family Platform Controller Hub (PCH), 2016. http://www.intel.com/content/www/us/en/chipsets/100-series-chipset-datasheet-vol-1.html.

[100] Intel. Intel Software Guard Extensions, 2016. https://software.intel.com/en-us/sgx.

[101] Intel. Intel Software Guard Extensions Developer Guide, 2016. https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_Developer_Guide.pdf.

[102] Intel. Overview of Intel SGX Instructions and Data Structures, 2016. https://software.intel.com/en-us/blogs/2016/06/10/overview-of-intel-software-guard-extensions-instructions-and-data-structures.

[103] Intel. SGX Documentation: sgx-create-monotonic-counter, 2016. https://software.intel.com/en-us/node/696638.

[104] Intel. SGX Documentation: sgx-get-trusted-time, 2016. https://software.intel.com/en-us/node/696636.

[105] Intel. SGX SDK, 2016. https://software.intel.com/en-us/sgx-sdk.

[106] Intel. Software Guard Extensions Tutorial Series, 2016. https://software.intel.com/en-us/articles/introducing-the-intel-software-guard-extensions-tutorial-series.

[107] Intel. Intel Software Guard Extensions - Developer Zone - Details, 2017. https://software.intel.com/en-us/sgx/details.

[108] Intel. Intel Trusted Execution Technology, 2018. https://www.intel.com/content/www/us/en/architecture-and-technology/trusted-execution-technology/trusted-execution-technology-security-paper.html.

[109] Intel Support Forum. Ensuring Only a Single Instance of Enclave, 2017. https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/709552.

[110] P. Jayanti, T. D. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM (JACM)*, 45(3):451–500, 1998.

[111] J. L. Jenkins, B. B. Anderson, A. Vance, C. B. Kirwan, and D. Eargle. More Harm Than Good? How Messages That Interrupt Can Make Us Vulnerable. *Information Systems Research*, 27(4):880–896, 2016.

[112] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen. Intel SGX: EPID Provisioning and Attestation Services, 2016. `https://software.intel.com/en-us/blogs/2016/03/09/intel-sgx-epid-provisioning-and-attestation`.

[113] A. Juels, A. Kosba, and E. Shi. The Ring of Gyges: Investigating the Future of Criminal Smart Contracts. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[114] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: resource-efficient byzantine fault tolerance. In *Proceedings of the 7th European Conference on Computer Systems (EUROSYS)*. ACM, 2012.

[115] G. Kappos, H. Yousaf, M. Maller, and S. Meiklejohn. An Empirical Analysis of Anonymity in Zcash. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[116] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun. Verena: End-to-End Integrity Protection for Web Applications. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (SP)*, 2016.

[117] P. A. Karger and A. J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Security and Privacy, 1984 IEEE Symposium on*, pages 2–2. IEEE, 1984.

[118] B. Kauer. OSLO: Improving the Security of Trusted Computing. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, 2007.

[119] Klarna. Sofort - Einfach und Direkt Bezahlen, 2017. `https://www.klarna.com/sofort/`.

[120] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom.

Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)*, 2019.

[121] K. Kostiainen, N. Asokan, and J.-E. Ekberg. Credential Disabling from Trusted Execution Environments. In *Nordic Conference on Secure IT Systems*. Springer, 2010.

[122] A. Kumar, C. Fischer, S. Tople, and P. Saxena. A Traceability Analysis of Monero's Blockchain. In *European Symposium on Research in Computer Security (ESORICS)*. Springer, 2017.

[123] L. Lamport. On interprocess communication. *Distributed computing*, 1(2):86–101, 1986.

[124] L. Lamport. Lower bounds for asynchronous consensus. In *Future Directions in Distributed Computing*, pages 22–23. Springer, 2003.

[125] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[126] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[127] T. Laurinavicius. What Successful Entrepreneurs Outsource to a Virtual Assistant. Entrepreneur, 2017. `https://www.entrepreneur.com/article/300195`.

[128] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, 2017.

[129] V. Lehdonvirta and E. Castronova. *Virtual Economies: Design and Analysis*. MIT Press, 2014.

[130] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small Trusted Hardware for Large Distributed Systems. In *6thth USENIX Symposium on Networked Systems Design and Implementation (USENIX NSDI)*, 2009.

[131] T. Levin, S. J. Padilla, and C. E. Irvine. A Formal Model for UNIX Setuid. In *Proceedings of the 1989. IEEE Symposium on Security and Privacy (SP)*, 1989.

[132] Y. Li, J. M. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry. MiniBox: A Two-Way Sandbox for x86 Native Code. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, 2014.

[133] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. *ACM SIGPLAN Notices*, 35(11):168–177, 2000.

[134] A. Limited. mbedTLS (formerly known as PolarSSL), 2015. `https://tls.mbed.org/`.

[135] J. Lind, I. Eyal, F. Kelbert, O. Naor, P. Pietzuch, and E. G. Sirer. Teechain: Scalable Blockchain Payments using Trusted Execution Environments. *Computing Research Repository (CoRR), arXiv abs/1707.05454*, 2017.

[136] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[137] J. Liu, W. Li, G. O. Karame, and N. Asokan. Scalable Byzantine Consensus via Hardware-assisted Secret Sharing. *IEEE Transactions on Computers*, 2018.

[138] D. Ma and G. Tsudik. A New Approach to Secure Logging. *ACM Transactions on Storage (TOS)*, 5(1):2, 2009.

[139] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical Oblivious Computation in a Secure Processor. In *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[140] C. Mack. Virtual Goods and Mainstream Game Companies. AdWeek, 2009. `http://www.adweek.com/digital/mainstream-games-companies-virtual-goods/`.

[141] R. Maes and I. Verbauwhede. Physically Unclonable Functions: A Study on the State of the Art and Future Research Directions. In *Towards Hardware-Intrinsic Security*, pages 3–37. Springer, 2010.

[142] D. Malkhi and M. K. Reiter. Secure and scalable replication in phalanx. In *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on*, pages 51–58. IEEE, 1998.

[143] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal byzantine storage. In *Distributed Computing*, pages 311–325. Springer, 2002.

[144] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. ROTE: Rollback Protection for Trusted Execution. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, 2017.

[145] S. Matetic, M. Schneider, A. Miller, A. Juels, and S. Capkun. DELEGATEE: Brokered Delegation Using Trusted Execution Environments. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[146] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *HASP ISCA*, 2013.

[147] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage. A Fistful of Bitcoins: Characterizing Payments among Men with No Names. In *Proceedings of the 2013 conference on Internet Measurement Conference*. ACM, 2013.

[148] F. J. Meyer and D. K. Pradhan. Consensus with Dual Failure Modes. *IEEE Transactions on Parallel & Distributed Systems*, 0(2):214–222, 1991.

[149] I. Miers, C. Garman, M. Green, and A. D. Rubin. Zerocoin: Anonymous Distributed e-cash from Bitcoin. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (SP)*, 2013.

[150] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An Efficient Oblivious Search Index. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP)*, 2018.

[151] A. Moghimi, G. Irazoqui, and T. Eisenbarth. Cachezoom: How SGX Amplifies the Power of Cache Attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017.

[152] M. Möser, K. Soska, E. Heilman, K. Lee, H. Heffan, S. Srivastava, K. Hogan, J. Hennessey, A. Miller, A. Narayanan, et al. An Empirical Analysis of Traceability in the Monero Blockchain. *Proceedings on Privacy Enhancing Technologies*, 2018.

[153] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

[154] B. C. Neuman. Proxy-based Authorization and Accounting for Distributed Systems. In *Proceedings of the 13th IEEE Conference on Distributed Computing Systems*, 1993.

[155] J. Nielsen. Website Response Times. Nielsen Norman Group, 2012. `https://www.nngroup.com/articles/website-response-times/`.

[156] O. Osuntokun, A. Akselrod, and J. Posen. Client Side Block Filtering. *Bitcoin Improvement Proposal*, 157, 2017. `https://github.com/bitcoin/bips/blob/master/bip-0157.mediawiki`.

[157] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical State Continuity for Protected Modules. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (SP)*, 2011.

[158] R. Pass, E. Shi, and F. Tramer. Formal Abstractions for Attested Execution Secure Processors. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2017.

[159] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[160] G. Platform. TEE System Architecture. *Global Platform Technical Overview*, 2011.

[161] C. Priebe, K. Vaswani, and M. Costa. EnclaveDB: A Secure Database using SGX. IEEE, 2018.

[162] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing Digital Side-channels Through Obfuscated Execution. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, 2015.

[163] M. K. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proceedings of the 2nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 1994.

[164] X. Ruan. *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress, 2014.

[165] M. Sabt, M. Achemlal, and A. Bouabdallah. Trusted Execution Environment: What it is, and What it is not. In *Proceedings of the 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2015.

[166] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based Access Control Models. *Computer*, 29(2):38–47, 1996.

[167] S. Sasy, S. Gorbunov, and C. Fletcher. ZeroTrace: Oblivious memory primitives from Intel SGX. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[168] B. Schneier and J. Kelsey. Secure Audit Logs to Support Computer Forensics. *ACM Transactions on Information and System Security (TISSEC)*, 2(2):159–176, 1999.

[169] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (SP)*, 2015.

[170] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017.

[171] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska. S-NFV: Securing NFV states by using SGX. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*. ACM, 2016.

[172] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[173] S. Singh, A. Cabraal, C. Demosthenous, G. Astbrink, and M. Furlong. Password Sharing: Implications for Security Design based on Social Practice. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2007.

[174] A. Sinha, L. Jia, P. England, and J. R. Lorch. Continuous Tamper-proof Logging using TPM 2.0. In *International Conference on Trust and Trustworthy Computing*. Springer, 2014.

[175] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani. Moat: Verifying Confidentiality of Enclave Programs. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[176] H.-S. Siu, Y.-H. Chin, and W.-P. Yang. A Note on Consensus on Dual Failure Modes. *IEEE Transactions on Parallel & Distributed Systems*, 7(3):225–230, 1996.

[177] S. Skorobogatov. The Bumpy Road Towards iPhone 5c NAND Mirroring. *Computing Research Repository (CoRR), arXiv abs/1609.04327*, 2016.

[178] S. W. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8):831–860, 1999.

[179] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[180] R. Strackx, B. Jacobs, and F. Piessens. ICE: A Passive, High-speed, State-continuity Scheme. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*. ACM, 2014.

[181] R. Strackx and N. Lambrigts. Idea: State-continuous Transfer of State in Protected-module Architectures. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*. Springer, 2015.

[182] R. Strackx and F. Piessens. Ariadne: A Minimal Approach to State Continuity. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, 2016.

[183] R. Strackx, F. Piessens, and B. Preneel. Efficient Isolation of Trusted Subsystems in Embedded Systems. In *International Conference on Security and Privacy in Communication Systems*. Springer, 2010.

[184] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing. In *17th annual International Conference on Supercomputing*. ACM, 2003.

[185] G. E. Suh and S. Devadas. Physical Unclonable Functions for Device Authentication and Secret Key Generation. In *Design Automation Conference*. IEEE, 2007.

[186] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. Design and implementation of the aegis single-chip secure processor using physical random functions. In *ACM SIGARCH Computer Architecture News*. IEEE Computer Society, 2005.

[187] T. Thomas. A Mandatory Access Control Mechanism for the Unix File System. In *Proceedings of the 4th Aerospace Computer Security Applications Conference*. IEEE, 1988.

[188] P. Todd. python-bitcoinlib, 2018. `https://github.com/petertodd/python-bitcoinlib`.

[189] F. Tramer, F. Zhang, H. Lin, J.-P. Hubaux, A. Juels, and E. Shi. Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP)*, 2017.

[190] Trusted Computing Group. Trusted Platform Module Library, Part 1: Architecture, Family 2.0, 2014.

[191] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[192] M. van Dijk, J. Rhodes, L. F. G. Sarmenta, and S. Devadas. Offline Untrusted Storage with Immediate Detection of Forking and Replay Attacks. In *ACM Workshop on Scalable Trusted Computing (STC)*. ACM, 2007.

[193] N. Van Saberhagen. Cryptonote v2.0, 2013. `https://cryptonote.org/whitepaper.pdf`.

[194] A. Vasudevan, J. M. McCune, and J. Newsome. *Trustworthy Execution on Mobile Devices*. Springer, 2014.

[195] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *European Symposium on Research in Computer Security (ESORICS)*. Springer, 2016.

[196] O. Weisse, V. Bertacco, and T. Austin. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017.

[197] L. Weston. Why Banks Want You to Drop Mint. Reuters, 2015. `https://www.reuters.com/article/us-column-weston-banks/why-banks-want-you-to-drop-mint`.

[198] M. V. Wilkes and R. M. Needham. The cambridge cap computer and its operating system. 1979.

[199] J. Winter. Trusted Computing Building Blocks for Embedded Linux-based ARM Trustzone Platforms. In *Proceedings of the 3rd ACM workshop on Scalable Trusted Computing*, 2008.

[200] R. Wojtczuk and J. Rutkowska. Attacking SMM Memory via Intel CPU Cache Poisoning. *Invisible Things Lab*, 2009.

[201] P. Wuille. Hierarchical Deterministic Wallets. *Bitcoin Improvement Proposal*, 32, 2012. `https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki`.

[202] K. Wüst and A. Gervais. Ethereum Eclipse Attacks. Technical report, ETH Zurich, 2016.

[203] Y. Xie. The Underground Market For In-Game Virtual Goods. TechCrunch, 2016. `https://techcrunch.com/2016/01/20/virtual-goods-real-fraud/`.

[204] Y. Xu, W. Cui, and M. Peinado. Controlled-channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (SP)*, 2015.

[205] B. Yee. Using Secure Coprocessors. Technical report, PhD Thesis, Carnegie-Mellon University, Pittsburgh, 1994.

[206] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town Crier: An Authenticated Data Feed for Smart Contracts. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[207] F. Zhang and H. Zhang. SoK: A Study of using Hardware-assisted Isolated Execution Environments for Security. In *Proceedings of the Hardware and Architectural Support for Security and Privacy*. ACM, 2016.