

Imperial College London
Department of Computing

Securing Applications Using Trusted Execution Environments

Joshua David Lind

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College,

December 2019

Abstract

Trusted execution environments (TEEs) have recently emerged as a powerful primitive for securing applications and protecting sensitive data. By employing hardware isolation in existing CPUs, TEEs offer a novel application security model: only the CPU and the isolated application code and data need to be trusted; all other software and hardware components are untrusted. This allows applications to execute with confidentiality and integrity guarantees.

However, TEEs are not a panacea: they pose fundamental security challenges that make it difficult for applications to leverage security guarantees: (1) as all code and data isolated by the TEE is trusted to be free from software bugs, applications must minimise the amount of trusted code and data. This reduces the size of the trusted computing base (TCB), and the risk of application vulnerabilities; (2) as all components outside the TEE are unprotected, e.g., the operating system and hardware peripherals, these components are untrusted and potentially malicious. Applications must therefore secure the TEE boundary interfaces between trusted and untrusted components, as these interfaces expose attack vectors; and (3) as TEEs do not provide availability guarantees, and current TEE implementations are vulnerable to attacks, e.g., side-channel attacks, applications must be designed to tolerate TEE failures and compromises.

In this thesis, we explore how applications can overcome the security challenges posed by the use of TEEs. We provide the following contributions:

1. Minimising the trusted computing base (TCB). To minimise the TCB of applications and reduce the risk of vulnerabilities, we propose *automatic application partitioning for TEEs*: a principled approach for partitioning existing applications to be deployed with TEEs. Our approach partitions applications into security-sensitive and security-insensitive parts, and: (i) protects security-sensitive code and data by placing it into the TEE; and (ii) provides guarantees that the security-sensitive code cannot violate the confidentiality or integrity of security-sensitive data, even under attack.

2. Securing TEE boundary interfaces. To secure applications against TEE boundary interface attacks, we propose a set of *TEE interface design principles*. By following these design principles, applications can: (i) better protect the confidentiality and integrity of security-sensitive information that crosses the boundary interface; and (ii) avoid leaking sensitive information about application execution, e.g., by exposing side-channel information through interface calls. By securing TEE boundary interfaces in this way, applications can maintain confidentiality and integrity guarantees even when interacting with untrusted and malicious components.

3. Tolerating TEE failures and compromises. To tolerate TEE failures and compromises, we propose applications employ *TEE-heterogeneity*: the protection of security-critical functionality using multiple heterogeneous TEEs. By employing TEE-heterogeneity, applications: (i) tolerate independent TEE failures and compromises, as they adopt decentralised architectures that avoid relying on a single TEE; and (ii) tolerate global TEE failures and compromises, as they employ heterogeneous TEEs, protecting against vulnerabilities that can simultaneously compromise all TEEs in the deployment.

To my family:

Thank you for your encouragement, support and love.

Acknowledgements

I would like to express my gratitude to the following people. Without you, this thesis would not have been possible:

Professor Peter Pietzuch: First, Peter, thank you for being my supervisor for four long years! Your guidance, support, time and encouragement have been invaluable. Working under you has been a privilege and I have grown so much as a result of your supervision. Thank you for your patience, advice, wisdom and the many different things you've done for me over the years – you have truly gone above and beyond as a supervisor! I would not be where I am today without you.

Professor William Knottenbelt: Will, thank you for your guidance, encouragement and enthusiasm throughout my eight years at Imperial. This has been a long journey, so I want to thank you for all the things you've done for me and the many roles you've played during my studies!

My collaborators and colleagues. To my collaborators and colleagues, whether you're at Imperial, Cornell, Google, the Technion in Israel (and the many other places I haven't mentioned), thank you! I've learnt so much from you all over these past four years, and it's been a real privilege to work alongside you. Thank you for making the journey so fun and rewarding!

My family and friends. To my family and friends (whoever and wherever you may be), thank you for your support, understanding and patience. The journey has been a long one, but I'm blessed to have been able to walk it with so many wonderful people in my life!

My parents and sister. Gavin, Cheri and Lauren, thank you for your unwavering love, encouragement, support, guidance and everything else that you've provided for me over the many years I've been studying. Without you, I would not be where I am today.

My wife. Katie, thank you for loving me, encouraging me, and walking beside me through it all!

Finally, I would like to express my gratitude to God, by Whom all things were created, with Whom all things are possible, and for Whom we are alive.

Declaration of Originality

This thesis presents my work in the Department of Computing at Imperial College London between October 2015 and October 2019.

Parts of the work were done in collaboration with other researchers:

- **Chapter 3.** For Glamdring, all work on static code analysis and integration with a source-to-source code partitioner was my own.
- **Chapter 4.** For Citadel, I led the design and implementation of the oblivious host call interface.
- **Chapter 5.** All work on TEEChan and Teechain was my own, except for the Lightning Network microbenchmarks in the Teechain paper.

I declare that the work presented in this thesis is my own, except where acknowledged above.

Joshua David Lind

December 2019

Publications and Patents

The work presented in this thesis is based on the following publications:

- **Glamdring: Automatic Application Partitioning for Intel SGX.** Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O’Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eysers, Rüdiger Kapitza, Christof Fetzer, Peter Pietzuch. In Proceedings of the 2017 USENIX Annual Technical Conference [ATC’17].
- **Citadel: Securing the Host OS Interface for Trusted Execution.** Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, Peter Pietzuch. [ArXiv Technical Report]
- **Teechan: Payment Channels using Trusted Execution Environments.** Joshua Lind, Ittay Eyal, Peter Pietzuch, Emin Gün Sirer. Presented at the 4th Workshop on Bitcoin and Blockchain Research [BITCOIN’17].
- **Teechain: A Secure Payment Network with Asynchronous Blockchain Access.** Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, Peter Pietzuch. In Proceedings of the 27th ACM Symposium on Operating Systems Principles [SOSP’19]

The following publications have also been co-authored over the course of this thesis, but do not directly contribute to the work presented here:

- **SCONE: Secure Linux Containers with Intel SGX.** Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O’Keeffe, Mark L Stillwell, David Goltzsche, Dave Eysers, Rüdiger Kapitza, Peter Pietzuch, Christof Fetzer. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation [OSDI’16].
- **LibSEAL: Revealing Service Integrity Violations using Trusted Execution.** Pierre-Louis Aublin, Florian Kelbert, Dan O’Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eysers, Peter Pietzuch. In Proceedings of the 13th EuroSys Conference [EUROSYS’18].
- **XCLAIM: A Framework for Blockchain Interoperability.** Alexei Zamyatin, Dominik Harz, Joshua Lind, Panayiotis Panayiotou, Arthur Gervais, William J Knottenbelt. In Proceedings of the 40th IEEE Symposium on Security and Privacy [S&P’19].

The following patent applications have been created based on the work presented in this thesis:

- **Blockchain Payment Channels with Trusted Execution Environments.** Ittay Eyal, Emin Gün Sirer, Peter Robert Pietzuch, Joshua David Lind. US Patent App. 16/100,689

Copyright Declaration

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.

"No one lights a lamp and puts it in a place where it will be hidden, or under a bowl. Instead they put it on its stand, so that those who come in may see the light."

Luke 11:33

Contents

1	Introduction	1
1.1	The need for trusted computing	1
1.2	Trusted computing	2
1.3	Trusted execution environments (TEEs)	3
1.4	Problem statement	4
1.4.1	Minimising the trusted computing base (TCB)	7
1.4.2	Securing TEE boundary interfaces	7
1.4.3	Tolerating TEE failures and compromises	8
1.5	Research contributions	9
1.5.1	Automatic application partitioning to minimise the TCB	9
1.5.2	Interface design principles for secure boundary interfaces	11
1.5.3	TEE-heterogeneity for TEE failures and compromises	13
1.6	Thesis outline	14
2	Background	16
2.1	Securing applications in untrusted environments	16
2.2	Trusted computing	17
2.2.1	A brief history and overview	18
2.2.2	A survey of trusted computing architectures	21
2.3	Trusted execution environments (TEEs)	28
2.3.1	A definition and overview	28
2.3.2	A comparison of TEEs	31
2.4	Open security challenges for TEEs	38
2.5	Summary	41
3	Minimising the Trusted Computing Base (TCB)	42
3.1	Overview	43
3.2	Background: Securing applications with TEEs	44
3.2.1	Threat model	44
3.2.2	Security with Intel SGX	44
3.2.3	Design space for SGX applications	45
3.3	Glamdring design	48
3.3.1	Design overview	49

3.3.2	Code annotation phase	50
3.3.3	Code analysis phase	51
3.3.4	Code partitioning phase	52
3.3.5	Security discussion	53
3.4	Code generation and hardening	54
3.4.1	Code transformation	54
3.4.2	Code hardening	56
3.5	Evaluation	58
3.5.1	Security evaluation	60
3.5.2	Performance evaluation	65
3.6	Related work	66
3.7	Limitations	69
3.8	Summary	70
4	Securing TEE Boundary Interfaces	71
4.1	Overview	72
4.2	Background: Host OS interfaces for TEEs	73
4.2.1	Security goals and threat model	73
4.2.2	Host interfaces of existing TEE runtime systems	74
4.2.3	Designing a secure host interface	79
4.3	Minimising the host interface	80
4.3.1	Citadel host interface	80
4.3.2	In-enclave OS functionality	81
4.4	Protecting the host interface	84
4.4.1	Protecting disk I/O calls	84
4.4.2	Protecting network I/O calls	85
4.4.3	Protecting event and time calls	86
4.5	Making the host interface oblivious	87
4.5.1	Disk I/O calls	87
4.5.2	Hiding disk I/O accesses	87
4.5.3	Hiding disk I/O parameters	88
4.5.4	Other host calls	90
4.6	Evaluation	90
4.6.1	Application performance	91
4.6.2	Disk I/O performance	93
4.6.3	Network I/O performance	94

4.7	Related Work	94
4.8	Limitations	96
4.9	Summary	96
5	Tolerating TEE Failures and Compromises	98
5.1	Overview	99
5.2	Background: TEEs for secure blockchain payment networks	100
5.2.1	Blockchains and cryptocurrencies	100
5.2.2	Payment networks and channels	101
5.2.3	Limitations of payment networks	102
5.2.4	TEEs as a root-of-trust	103
5.2.5	Threat model	104
5.3	Teechain design	105
5.3.1	Design overview	105
5.3.2	Treasuries	107
5.3.3	Treasury committees	108
5.3.4	Threat analysis	109
5.4	Payment Protocol	110
5.4.1	Allocating dynamic deposits	111
5.4.2	Using payment channels	112
5.4.3	Executing multi-hop payments	113
5.4.4	Payment protocol security	115
5.5	Treasury committees	117
5.5.1	Force-freeze chain replication	117
5.5.2	Treasury committee configurations	118
5.5.3	Treasury committees with secure persistent storage	119
5.6	Evaluation	120
5.6.1	Performance of payment channels	121
5.6.2	Performance of multi-hop payments	123
5.6.3	Performance of payment networks	124
5.7	Related Work	126
5.8	Limitations	127
5.9	Summary	128
6	Conclusion	129
6.1	Thesis summary	129
6.2	Future work	131
6.3	Closing remarks	134

References**135**

List of Tables

2.1	Summary of the TEE building blocks and functionalities offered by different TEE implementations.	31
3.1	TCB sizes, enclave interfaces and enclave crossings for Glamdring applications (Application requests are: (i) get , set for Memcached; (ii) sign for LibreSSL; and (iii) seed , sign , random for Digital Bitbox).	59
3.2	The TCB sizes and performance for Memcached for Glamdring, SCONE and Graphene.	64
4.1	The security breakdown of parameters in host interface calls for existing TEE runtime systems.	75
4.2	The host interface exposed by Citadel.	79
5.1	The Teechain API (The API allows users to manage fund deposits, create payment channels, execute payments, perform payment channel settlement, and leverage chain replication.)	107
5.2	The maximum performance of payment channels for both the Lightning Network and Teechain.	122
5.3	The performance of API operations in the Lightning Network compared to Teechain.	124

List of Figures

1.1	A high-level overview of the security model offered by TEEs (We make three security observations about this model: (1) the sensitive code and data inside the TEE is trusted to be free from software bugs. If a vulnerability in the trusted code and data can be exploited, the security guarantees of the TEE can be violated; (2) the TEE boundary interface by which sensitive code and data interacts with untrusted components forms an attack vector: attackers can use this interface to try and compromise application confidentiality or integrity; and (3) TEE implementations themselves are often vulnerable to hardware or software based attacks. Such attacks can be used to violate the security guarantees of the TEE in order to attack the sensitive code and data inside the TEE directly.)	5
2.1	The XOM [LTM⁺00] platform architecture (In this model, containers are encrypted and stored in untrusted main memory, before being decrypted and executed in the secure execution engine.)	22
2.2	The AEGIS [SOD05] platform architecture (In this model, the tamper-resistant processor chip separates trusted components, e.g., the security kernel and encryption engine, and untrusted components, e.g., main memory, peripherals, and the untrusted OS.)	24
2.3	The IBM 4765 secure co-processor architecture [ABC⁺12] (In this model, all components are encapsulated into a single tamper-resistant enclosure.)	25
2.4	The TyTan [BEMS⁺15] architecture (In this model, we see the clear separation between trusted components, e.g., the EA-MPU and the root of trust measurement, and untrusted components, e.g., the OS and untrusted tasks.)	27
2.5	The TEE building blocks (Here, we see the various TEE building blocks outlined by Mohamed et al. [SAB15]; these are the building blocks required to operate TEEs on a computing platform.)	30
2.6	The TEE architecture specification proposed by the GlobalPlatform [Pla11] (The platform is divided into the Rich Execution Environment (REE), containing untrusted OS components and applications, and the Trusted Execution Environment (TEE), containing trusted OS components and applications.)	32
2.7	The Intel SGX [MAB⁺13] architecture (Here, we see the layout of Processor Reserved Memory (PRM), the Enclave Page Cache (EPC) and Enclave Page Cache Map (EPCM) in untrusted DRAM.)	34
2.8	The Sanctum [CLD16] architecture (Here, we see the hardware and software layers required by Sanctum to execute an enclave-based host application.)	37
3.1	The design alternatives for the use of enclaves (These designs include: (a) a complete enclave interface, where the entire application is executed in the enclave; (b) a predefined enclave interface, where the application conforms to a specific enclave interface; and (c) an application-specific interface, where the application is partitioned into trusted and untrusted application code and data, with a corresponding interface between the two.)	46
3.2	Overview of the Glamdring framework (Here, we see: (i) the application source code as an input; (ii) the four phases of execution, from code annotation to automatic code generation; and (iii) the resulting output: a set of trusted and untrusted application code and data files.)	49

3.3	The partitioned Memcached architecture (Here, we see the separation between security-sensitive code and data placed inside the enclave for Memcached, and non-sensitive code and data outside the enclave.)	61
3.4	The partitioned LibreSSL architecture (Here, we see the separation between security-sensitive code and data placed inside the enclave for LibreSSL, and non-sensitive code and data outside the enclave.)	62
3.5	The partitioned Digital Bitbox architecture (Here, we see the separation between security-sensitive code and data placed inside the enclave for Digital Bitbox, and non-sensitive code and data outside the enclave.)	63
3.6	The throughput versus latency for Memcached native, with SCONE and with Glamdring, for different types of application workloads.	65
3.7	Throughput and latency overhead for Digital Bitbox and LibreSSL when performing various operations.	67
4.1	The Citadel architecture	81
4.2	Training throughput with TensorFlow	91
4.3	Inference throughput with TensorFlow	92
4.4	Execution overhead for PARSEC workloads	92
4.5	Disk performance with encryption and integrity protection	93
4.6	Network I/O throughput for different buffer sizes	94
5.1	A payment channel in operation (Here, we see: (1) funds are deposited into the payment channel; (2) <i>A</i> pays <i>B</i> using the channel; (3) <i>B</i> pays <i>A</i> using the channel; and (4) the channel is settled writing the final balances to the blockchain.)	102
5.2	The Teechain overview (Teechain nodes operate <i>treasuries</i> to store and manage funds. Users construct <i>payment channels</i> between nodes to exchange funds directly, and execute <i>multi-hop</i> payments along concatenated payment channels. <i>Committee chains</i> with multiple treasuries replicate and protect state.)	106
5.3	The protocol for multi-hop payments in Teechain.	115
5.4	The machines, network and evaluation setup of our experiments.	121
5.5	The performance of multi-hop payments.	125
5.6	The hub-and-spoke network topology.	125
5.7	The throughput for the complete topology.	125
5.8	The throughput for the hub-and-spoke topology.	125

1

Introduction

Software plays a critical role in the world around us. It drives cars, flies planes, operates life-saving medical devices and even controls nuclear facilities [Sto96]. As a society, we have never been more reliant on software than we are right now. However, *software is only as secure as the environment in which it executes*. This is the observation that motivates my work.

1.1 The need for trusted computing

Software deployment models today require trust: applications trust the security of the environments in which they are deployed. For example, an application deployed on an operating system (OS), e.g., Windows or Linux [SGG18], trusts the OS to protect the application from other software, e.g., by managing memory and enforcing access control policies. Likewise, the application trusts the OS not to be under adversarial control, otherwise the adversary could attack the application directly, e.g., by reading or modifying memory. If any of these assumptions are violated, the security of the application is compromised. As a result, to completely secure a software application, you must secure all components that the application relies on. This forms the *trusted computing base* (TCB) of the application, i.e.,

the set of components that together ensure the security of the application.

Unfortunately, for most application deployment models today, the environment in which applications execute is incredibly complex, and applications rely on a plethora of different services. For example, runtime environments [LYBB14], the OS, the hypervisor, the CPU, and the even the BIOS [BHBN14] affect the functionality and correctness of application execution. As such, the TCB of most applications today extend far beyond the application itself, with most TCBs consisting of several millions of lines of code [SPHH06] and various hardware components.

Software is therefore only as secure as the environment in which it executes. This observation has significant consequences when we consider different application deployment models in use today, for example, *cloud* [FGJ⁺09], *volunteer* [Sar01] and *outsourced* [MKL⁺02] computing. In these scenarios, applications are outsourced to third-party providers, where users have very little insight and control over what software or hardware is operating. As a result, they must often simply trust the environment to be secure. This is difficult when you consider the size of the TCB in these environments and the various threats that they face.

Take cloud computing [FGJ⁺09] for example: cloud providers not only employ OSes and hypervisors on which applications run, but they also employ additional layers of software, such as software for resource management, scheduling [SC16], and security, e.g., containers [BJJ10]. In addition, because cloud environments support multi-tenant deployment models, i.e., where applications share the same physical machines, malicious applications can target other co-located applications to violate security, e.g., through side-channel and hardware based attacks [KPMR12]. This creates many different attack vectors by which an adversary can target an application in order to compromise security.

1.2 Trusted computing

To better secure the TCB of applications, *trusted computing* [Mit05] has been proposed. Trusted computing is the idea of adding trusted components to a computing platform in order to create a foundation of trust for applications [Mit05]. This gives the entities who interact with an application, e.g., users and other software, assurances and guarantees about the security of the application's execution [Pea02]. The goal of trusted computing is to develop technologies which can ensure the correct behaviour of an application, even when the application executes in an *untrusted environment*, i.e., an attacker or adversary controls the software and hardware in that environment.

To secure the TCB of applications in untrusted environments, trusted computing has proposed many

different solutions over the years, including trusted computing architectures that use secure software, trusted hardware, or a co-design of both [SGR09]. While trusted computing has made great strides towards protecting security-critical applications, it has yet to see widespread adoption. This is because early trusted computing architectures, such as the trusted platform module (TPM) [Pea02], and IBM’s secure co-processor [ABC⁺12], suffered from limitations that made them impractical for general purpose use.

For example, the TPM and the IBM secure co-processor both require auxiliary hardware to be purchased and placed alongside the CPU to perform security-critical operations. Requiring such auxiliary hardware was often expensive and suffered from performance overheads due to having to move data back and forth between the central CPU and the trusted hardware device during execution. Moreover, these solutions offered limited runtime abstractions for execution and provided little or no support for attestation. For example, the TPM only provided secure bootstrapping and measurement during the boot process, but offered no support for attesting the state of the application at runtime. This meant that these solutions were only suitable for a narrow set of applications, but not practical in general.

1.3 Trusted execution environments (TEEs)

To overcome the limitations of existing trusted computing architectures, *trusted execution environments* (TEEs) have been proposed. TEEs offer powerful primitives for protecting the integrity and confidentiality of arbitrary applications without requiring additional auxiliary hardware. Instead, TEEs leverage hardware isolation in general purpose CPUs to protect code and data. The isolation offered by TEEs includes separation from higher-privileged system software, such as the OS, the hypervisor and the BIOS. As such, TEEs avoid having to trust these components in order to execute correctly. This provides a novel root of trust that allows new software deployment models today: by only trusting the CPU and the isolated code and data, applications can be deployed in untrusted environments with a small TCB.

In contrast to previous trusted computing architectures, TEEs do not need to move data back and forth between different CPUs or hardware devices. As a result, TEEs offer improved performance and lower overheads than previous solutions. Moreover, TEEs are also widely available in general purpose commodity CPUs today, and offer powerful abstractions and implementations for software attestation at runtime. For example, current implementations [Int14, ARM19, Ope18, Lin14, Hex18] typically support remote attestation [AGJS13], integrity measurements of code and data, and data sealing functionalities that allow TEEs to store and retrieve data in persistent storage.

Intel’s Software Guard Extensions (SGX) [Int14] and Arm TrustZone [ARM19] are two notable examples of TEEs currently available. Intel SGX [Int14], for instance, is a set of instruction extensions available in all Intel CPUs from the Skylake generation onwards. Intel SGX provides secure TEEs, called *enclaves*. Applications can instantiate enclaves to protect memory regions within the address space of the application. The processor prevents unauthorised accesses to enclave code and data, even for privileged software such as the OS and BIOS. All memory held in an enclave is encrypted when written to RAM or disk, and is only ever decrypted inside the processor before being used.

We believe that TEEs hold the potential to revolutionise the state of application security and privacy in the world around us. However, despite their wide-spread availability and general practicality, TEEs have yet to see mainstream adoption. This is because TEEs are not a panacea: applications that wish to use TEEs must address several security challenges posed by the TEE security model. In this thesis, we explore how applications can be designed and modified to address the security challenges posed by TEEs. We present novel techniques for overcoming these challenges.

1.4 Problem statement

Figure 1.1 shows an overview of the security model offered by TEEs. TEEs provide security through hardware isolation: the TEE isolates security-sensitive application code and data, separating it from untrusted components in the system, including non-sensitive application code. The TEE also allows sensitive application code to communicate with untrusted components via the *TEE boundary interface*: an interface explicitly defined by the application inside the TEE. By combining strong isolation with a flexible communication interface, TEEs allow applications to execute with strong security guarantees.

However, applications that wish to employ TEEs must address three security challenges posed by this security model. These challenges are shown in Figure 1.1. For each challenge, we identify a corresponding research question that we aim to answer in this thesis. We present these challenges and research questions in detail below:

1. First, we observe that all security-sensitive code and data must be placed into the TEE to leverage confidentiality and integrity guarantees (❶, Figure 1.1). Naturally, this code and data is trusted by the TEE to be free from software bugs: if a bug exists in trusted code, it can be exploited by an adversary to compromise the security of the application. This raises a question for developers who wish to secure applications using TEEs: *what application code and data should be placed into the TEE?*

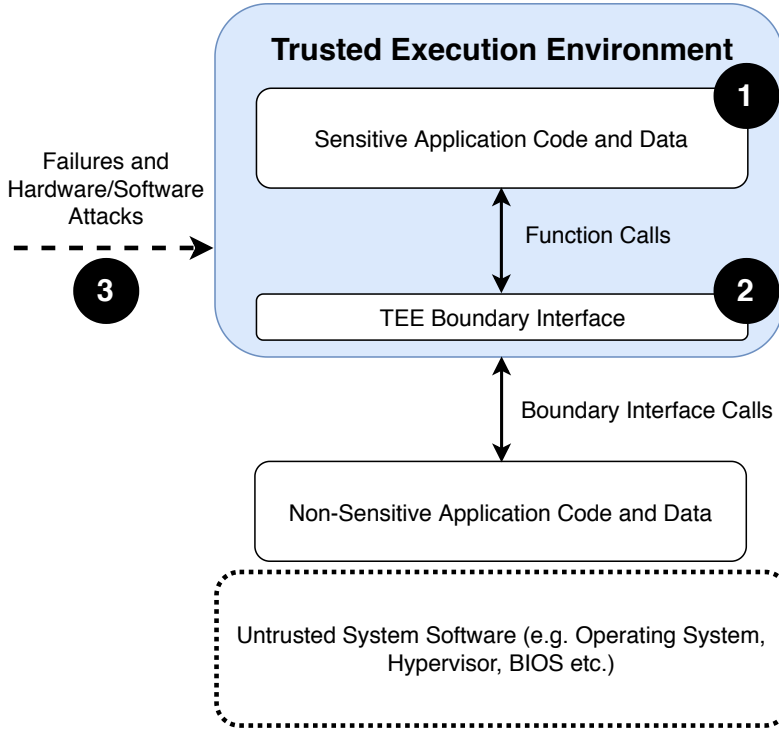


Figure 1.1: **A high-level overview of the security model offered by TEEs** (We make three security observations about this model: (1) the sensitive code and data inside the TEE is trusted to be free from software bugs. If a vulnerability in the trusted code and data can be exploited, the security guarantees of the TEE can be violated; (2) the TEE boundary interface by which sensitive code and data interacts with untrusted components forms an attack vector: attackers can use this interface to try and compromise application confidentiality or integrity; and (3) TEE implementations themselves are often vulnerable to hardware or software based attacks. Such attacks can be used to violate the security guarantees of the TEE in order to attack the sensitive code and data inside the TEE directly.)

An obvious, but naive, answer to this question is to simply place the entire application into the TEE. Existing work [BPH15, TPV17] has done this, e.g., for Intel SGX enclaves. Although this provides an easy deployment model for developers, it encompasses a trade-off: by placing the entire application into the TEE, developers increase the amount of code and data that needs to be trusted, i.e., the TCB. This increases the likelihood of vulnerabilities inside the TEE. As such, to reduce the possibility for exploits and better protect the application, it is essential to minimise the code placed into the TEE. This leads us to the first security challenge: minimising the TCB of existing applications deployed using TEEs. We discuss this challenge and the corresponding research question in Section 1.4.1 below.

2. Second, we observe that trusted code, executing inside the TEE, and untrusted code, executing outside the TEE, are separated by the TEE’s isolation boundary. As the isolation boundary is implemented in hardware, this prevents cross-boundary communications via traditional mechanisms, e.g., direct function calls. As a result, TEEs allow applications to define explicit boundary interfaces to allow cross-boundary communication (②, Figure 1.1). This raises a question for ap-

plication developers: *how should the TEE boundary interface be designed?*

While it may appear trivial to design a boundary interface based purely on functionality, the interface itself has significant security implications. This is because the interface secures sensitive information and execution flows across the boundary. As such, adversaries can often target the boundary interface as an attack surface. For example, an adversary may manipulate sequences or parameters of boundary interface calls in an attempt to trigger vulnerabilities inside the TEE or violate integrity [CS13b]. Moreover, an adversary may observe the frequencies or sequences of boundary interface calls to leak sensitive data, thus forming a *side-channel* [KB07]. As a result, it is essential to protect the TEE boundary interface from attacks, and this leads us to our second security challenge: securing TEE boundary interfaces. We discuss this challenge and the corresponding research question in Section 1.4.2 below.

3. Finally, we observe that applications rely on: (i) the TEE to provide availability guarantees for application execution; and (ii) the security of the TEE for confidentiality and integrity guarantees. As such, if the TEE fails, e.g., loses power, or the TEE is compromised, e.g., the implementation is faulty or insecure, the availability and security of the application can be trivially broken (❸, Figure 1.1). Unfortunately, current TEEs do not provide availability guarantees. Moreover, recent work has shown that current TEE implementations are vulnerable to a wide range of software and hardware based attacks [WCP⁺17, JLLK17, CZWW17, MIE17a]. For example, Foreshadow [VBMW⁺18] and Spectre [OMA⁺18], which exploit vulnerabilities in speculative execution to leak private keys for the Intel Attestation Service (IAS) [AGJS13] and expose side-channels in Intel SGX enclaves. This raises a question for developers: *how can applications be protected against TEE failures and compromises?*

The current approach to addressing this question is for developers and service providers to try and minimise the likelihood of any failures or TEE compromises. For example, by employing TEE best practices and defences, such as: (i) maintaining backup sources of power for TEEs and machines; (ii) keeping TEE implementations up-to-date with security patches and fixes, e.g., microcode updates [Int18]; and (iii) employing software defences against known attacks [OTK⁺18a]. However, while this approach is beneficial, it cannot ensure that TEEs are always available or secure. This is because these defences can only be employed against known failures and attacks. However, unexpected failures and unknown attacks on the TEE can still exist and be exploited by adversaries. As such, applications should never assume that TEEs are free from failures or attacks. Instead, they must be designed to tolerate TEE failures and compromises, even when defences are actively deployed. This leads us to our third and final security challenge: tolerating TEE failures and compromises for applications. We discuss this challenge and the corresponding

research question in Section 1.4.3 below.

1.4.1 Minimising the trusted computing base (TCB)

While it is easy to secure existing applications by placing all application code and data into the TEE [BPH15, TPV17], this approach is problematic for two reasons: (i) prior work has shown that the number of bugs in an application is generally proportional to the amount of code that application contains [MB03]. Therefore, by increasing the amount of code placed into the TEE, this approach increases the likelihood of vulnerabilities; and (ii) by placing the entire application into the TEE, this approach violates the *principle of least privilege* [SS75], i.e., allows all code access to sensitive data in the TEE, regardless of whether or not that code actually requires access in order to function. As such, this approach faces the issue that an adversary need only exploit one vulnerability inside the application to compromise all security guarantees offered by trusted execution.

To overcome this, we argue that it is not always necessary to protect all code and data in an application. Rather, it is often the case that only a subset of the application is security-sensitive and needs to be protected; an observation often exploited by existing work on privilege separation [BS04, SPHH06, ST06a]. For example, in the case of a database application, it might only be necessary to protect data and query execution, while protecting less-sensitive metadata, such as query statistics, may not be necessary. Prior work has leveraged this observation for applications in the context of TEEs, by showing that complex applications can be partitioned by hand to operate with minimal TCBs [BWL⁺16, PPFF16]. However, these solutions propose manual application partitioning, which is often tedious, error-prone, time consuming and application specific. Instead, we want to explore a solution that can be applied to a wide range of existing applications. As such, we ask the question: **"Can we use principled techniques to automatically partition existing applications for TEEs, in order to minimise the TCB?"** This is the first research question that we aim to answer in this thesis.

1.4.2 Securing TEE boundary interfaces

Although TEE boundary interfaces enable communication across the TEE boundary, it is necessary to secure sensitive information that flows through these interfaces. Otherwise it is possible for an adversary to exploit boundary interfaces as an attack surface. Existing work [BPH15, TPV17, SLTTS17] has identified the need to protect boundary interfaces from attacks, and also proposed a limited number of solutions to achieve this. However, these solutions suffer from two primary limitations: (i) the solutions do not systematically secure the entire interface against attacks, but rather only propose a small number

of adhoc solutions for specific boundary interface calls, e.g., using encryption to protect sensitive network data that passes through the interface [BPH15]; and (ii) they do not consider the boundary interface to expose side-channel information about TEE execution. By observing the frequencies, sequences and parameters of boundary interface calls, an adversary can learn sensitive information about application execution, thus violating application confidentiality. As such, these solutions do not offer protection against all TEE boundary interface attacks.

To overcome this, we argue that TEE boundary interfaces should be designed in a way that systematically protects the application against all TEE boundary interface attacks, i.e., applications should consider boundary interfaces as explicit attack vectors and sources of side-channel information about application execution. As such, we ask the question: **"Can we identify a set of interface design principles that prevent TEE boundary interfaces from violating application integrity or leaking information about application behaviour?"** This is the second research question we aim to answer in this thesis.

1.4.3 Tolerating TEE failures and compromises

Despite the best efforts of hardware manufacturers, TEEs do not provide availability guarantees. Moreover, current TEE implementations are vulnerable to a wide range of software and hardware based attacks [WCP⁺17, JLLK17, CZWW17, MIE17a] . As such, it is naive for application developers to assume: (i) that TEEs cannot fail; and (ii) that TEEs are secure against vulnerabilities and compromises. In addition, given that the majority of TEE implementations [Int14, ARM19, KPW16] are proprietary and released as black-boxes, i.e., without exposing how they are implemented, it is nearly impossible for developers to verify or audit the security of TEE implementations for themselves. As a result, it still remains a challenge today to determine whether or not existing TEEs are truly reliable and trustworthy [IL07].

To overcome this, we argue that until TEEs can provide availability guarantees and offer open-sourced, trustworthy and publicly verifiable implementations, applications should be designed with TEE failures and compromises in mind. That is, developers should not assume that TEEs are always available or completely secure; instead, applications should be tolerant against availability failures and attacks that violate TEE confidentiality and integrity. Moreover, we further argue that applications should be *TEE-agnostic*: designed to be compatible with different TEE implementations. This allows applications to protect against faulty or vulnerable TEEs, avoiding being locked into a single TEE implementation. We therefore ask the question: **"Can applications employ multiple TEE instances and implementations to provide fault tolerance against TEE failures and compromises?"** This is the

third and final research question we aim to answer in this thesis.

1.5 Research contributions

In this thesis, we propose solutions to the security challenges and research questions presented above. We outline these solutions in detail below.

1.5.1 Automatic application partitioning to minimise the TCB

To minimise the TCB of existing applications and answer the research question posed in Section 1.4.1, we propose *automatic application partitioning for TEEs*: a principled approach for automatically partitioning existing applications to be deployed with TEEs. Our approach protects security-sensitive code and data by placing it into the TEE, and provides guarantees that the security-sensitive code cannot violate the confidentiality or integrity of security-sensitive data, even under attack. The rest of the application, and non-sensitive code and data, are untrusted and remain outside the TEE.

To automatically partition existing applications for TEEs, we face three challenges: (1) we must identify which code and data in the application is security-sensitive, as this will be application and functionality specific; (2) we must partition the application and define a secure TEE boundary interface between trusted and untrusted code and data; and (3) we must transform the application to adopt the new partitioned architecture. As such, this work makes the following contributions:

1. *Code annotations and static program analysis for code and data*: Deciding which code and data to protect using a TEE is application-specific and depends on the functionality that needs to be protected. As such, we propose that application developers annotate the source code of their applications to explicitly mark components that are security-sensitive, e.g., variables or values returned by functions. Static program analysis can then be applied to these annotations to ensure the confidentiality and integrity of the annotated components. To ensure confidentiality, *static dataflow analysis* [RHS95] can be used to identify all code and data that uses or references the annotated components. To ensure integrity, *static backward slicing* [Wei81] can be used to identify all code and data that contributes to the value of the annotated components.

Identifying security-sensitive code and data in this manner prevents application developers from having to do it manually. Moreover, by using static program analysis, our approach is conservative, meaning that all sensitive code and data is identified, regardless of the application input or workload.

2. *Function granularity partitioning for secure TEE boundary interfaces*: To partition existing applications into security-sensitive and non-sensitive code and data, we propose applications be split at the *function granularity*. This: (i) reduces the amount of code changes that need to occur to the application; and (ii) produces a natural TEE boundary interface, as applications already maintain a well-defined function interface with clear control flows [All70], clear inputs, i.e., function parameters, and clear outputs, i.e., function return values. As such, we partition applications by moving security-sensitive functions into the TEE, and leaving non-sensitive functions outside the TEE.

To achieve this, every function that contains one or more security-sensitive statements, as identified by static program analysis, is marked security-sensitive. Functions that do not contain any security-sensitive statements are non-sensitive. A TEE boundary interface can thus be defined based on this partitioning: a new boundary interface call can be declared where: (i) a security-sensitive function calls a non-sensitive function; and (ii) a non-sensitive function calls a security sensitive function. Moreover, security-sensitive data that crosses the boundary interface, i.e., via one of these function calls, can be identified and transparently encrypted and integrity protected. This allows the resulting partitioned architecture to be more easily reasoned about, and produces a clear and well-defined boundary interface between trusted and untrusted code and data.

3. *Source code generation to transform the application*: To transform the application to follow the new partitioned architecture, we propose using automatic source code generation. For this, a code generator analyses the source code of the original application, and knowing which functions are security-sensitive, performs a source-to-source transformation: the code generator splits the original source code into a set of trusted and untrusted source files based on the security-sensitive functions, and generates: (i) the necessary boundary interface calls to allow efficient interaction, e.g., function invocations, between trusted and untrusted code; (ii) modifies the allocation, scope and lifetime of variables in the trusted and untrusted code; and (iii) performs code hardening to protect and secure the TEE boundary interface from various attacks. The code generator then compiles the trusted and untrusted code into a single application, with all trusted code executing inside a TEE.

We present **Glamdring**, a framework that implements our contributions to automatically partition existing C applications for Intel SGX enclaves. To evaluate our approach, we apply Glamdring to three different applications: the *Memcached* key/value store [Fit04], the *LibreSSL* library [Bec14], and the *Digital Bitbox* Bitcoin wallet [Shi16]. Our experiments show that Glamdring creates partitioned versions of these applications with TCBs that contain 22% – 40% of the lines of code of the applications.

Despite their strong security guarantees, the partitioned applications execute with between $0.3x - 0.8x$ of the performance of the original versions.

1.5.2 Interface design principles for secure boundary interfaces

To protect applications against TEE boundary interface attacks and answer the question posed in Section 1.4.2, we propose that TEE boundary interfaces be designed with security as the primary focus. This is in contrast to existing approaches [BPH15, TPV17], where the interface is merely just a consequence of the application’s design. To achieve this, we propose three *TEE interface design principles* that any TEE boundary interface should employ:

1. **Minimality:** the interface should be as small as possible, to narrow the attack surface, reduce verification complexity, and allow all interface calls to be systematically protected [Här02].
2. **Protection:** as sensitive data often flows through boundary interfaces, encryption and integrity-protection should be employed to achieve confidentiality and integrity for all sensitive or critical data when it leaves or enters the TEE boundary.
3. **Obliviousness:** as interface calls can be observed by the adversary at runtime, the frequencies, sequences and parameters of these calls expose a side-channel. To prevent this, interface calls should execute in an oblivious fashion, i.e., the behaviour and execution of interface calls should not be affected by the application, or the workload being run.

To illustrate how these design principles can secure TEE boundary interfaces from attack, we design a secure *host OS interface for TEEs*. This interface employs minimality, protection and obliviousness to separate and secure applications executing inside a TEE from an untrusted host OS. To achieve this, our design makes the following contributions:

1. *A library OS for minimality:* To create a minimal interface between the application and the untrusted host OS, we propose placing a complete library OS inside the TEE, alongside the application. This reduces the amount of functionality, e.g., POSIX [AAG⁺16] functionality, the application requires from the host OS, as the library OS can provide this functionality directly. As a result, the library OS reduces the size of the boundary interface between the application and the untrusted host, as interface calls only need to be defined for functionality that cannot be provided by the library OS, e.g., low-level disk and network I/O. This achieves a minimal TEE boundary interface.

2. *Transparent encryption and integrity verification for protection:* To create a protected boundary interface between the application and the untrusted host OS, we propose: (i) encrypting any data that is passed to the host OS via the interface; and (ii) decrypting all data and verifying its integrity when it returns to the TEE via the interface. This can be done by the trusted code inside the TEE, thus offering transparent protection for all applications. For example, to protect POSIX functionality provided by the host OS, such as file and disk I/O, we can use an encrypted root file system image stored outside the TEE. This ensures the confidentiality and integrity of any sensitive data that is written to, or read from, the disk, without requiring the application to protect this data manually. This secures the application against attacks where the host OS manipulates data crossing the boundary interface.

3. *Discrete and deterministic interface calls for obliviousness:* To create an oblivious boundary interface between the application and the untrusted host OS, we propose executing interface calls independently of the application or workload being run inside the TEE. This prevents the frequencies, sequences and parameters of host interface calls from exposing side-channel information. To achieve this, we: (i) discretise the execution of interface calls into fixed time-interval rounds, and add dummy interface calls to each round. This results in a deterministic and fixed batch of interface calls being executed every round, hiding the frequencies and sequences of real interface calls; and (ii) use probabilistic encryption [GM84a] to hide sensitive parameters passed to interface calls, making them appear random to an adversary outside the TEE. By applying these two techniques, we ensure that host interface calls execute in an oblivious fashion, preventing information leakage.

We present **Citadel**, a TEE runtime system that implements our contributions to offer a secure host OS interface. Citadel leverages minimality, protection and obliviousness to protect Linux binaries executing inside Intel SGX enclaves from an untrusted host OS. Citadel employs the Linux Kernel Library (LKL) [PGT10] as the library OS inside the enclave, and exposes only 7 host interface calls between the application and the untrusted host OS. To evaluate the overhead of our approach, we measure the performance of Citadel: with emulated SGX enclaves (to ignore current SGX memory limits), Citadel trains deep neural networks with an overhead of 14%, and 21% with oblivious calls. With current SGX hardware, Citadel executes benchmarks that fit into SGX memory with an average overhead of 1.5x and 3.1x with and without oblivious calls, respectively.

1.5.3 TEE-heterogeneity for TEE failures and compromises

To tolerate TEE failures and compromises and answer the question posed in Section 1.4.3, we propose applications use *TEE-heterogeneity*: the protection of security-critical functionality using multiple heterogeneous TEEs. We argue that: (i) rather than using a centralised TEE architecture, e.g., securing private keys with a single TEE instance, applications should adopt a decentralised architecture, where multiple TEE instances are employed. This allows applications to protect against independent TEE failures and compromises in the deployment; and (ii) instead of using homogeneous TEEs to secure applications, i.e., a set of identical TEE implementations, applications should employ heterogeneous TEEs. This provides protection against global TEE failures and compromises, where an entire class of TEEs might fail or be compromised simultaneously, e.g., due to a faulty or vulnerable TEE design. By combining decentralisation with heterogeneous TEEs, applications can achieve fault tolerance against TEE failures and compromises.

To leverage TEE-heterogeneity, applications must move away from centralised TEE architectures and instead be designed to operate in a decentralised and TEE-agnostic way. This requires overcoming two challenges: (1) security-critical functionality in the application must be distributed across multiple heterogeneous TEEs; and (2) TEEs must maintain agreement, i.e., consensus, on the current state of the application, despite possible TEE failures and compromises. Our work makes the following contributions:

1. *Threshold cryptography for decentralised TEEs*: To decentralise security-critical functionality across heterogeneous TEEs, we propose applications use *threshold cryptography* [Des94]. Threshold cryptography distributes the role of a single entity in a cryptosystem to multiple different entities. As such, applications can employ threshold cryptography to distribute the role of a single TEE to multiple heterogeneous TEEs. For example, to achieve data confidentiality using TEEs, applications can employ secret sharing algorithms [KGH83], such as *Shamir's Secret Sharing* [Sha79], to share a secret among n different TEEs, where at least k of those TEEs are required to reconstruct the secret. This provides confidentiality even if $k - 1$ TEEs are compromised. Likewise, to ensure integrity for a TEE-based application, you could require at least k different instances of the TEE to agree, i.e., achieve consensus, on the state of the application. This provides integrity even if $k - 1$ TEEs are compromised or misbehave.
2. *Force-freeze chain replication for TEE consensus*: By decentralising security-critical functionality across multiple different TEEs, the application must provide a way for the TEEs to maintain agreement, i.e., consensus, on the current application state. This should occur despite TEE

failures and compromises. To overcome this, we propose a simple yet efficient replication protocol between TEEs based on *chain replication* [VRS04a]. Our proposal, called *force-freeze chain replication*, ensures that all TEEs, protecting the same functionality in an application, maintain consistency, despite failures or attacks on the TEEs. Force-freeze chain replication offers high throughput and strong security guarantees for TEE state replication, without requiring all TEEs to communicate with each other directly; an attractive property for applications deployed in networks where not all nodes are directly connected, e.g., peer-to-peer networks [LCC⁺02].

To illustrate why TEE fault tolerance is so important, and how applications can employ our contributions, we design a TEE-based application that operates in the context of a blockchain-based cryptocurrency, such as Bitcoin [Nak08] and Ethereum [The17b]. In cryptocurrency settings, parties exchange funds across a peer-to-peer network, where each party is mutually distrusting of the other and potentially malicious. As such, applications operating in these settings must be designed to assume that parties will always misbehave or act maliciously if it benefits them. Therefore, any applications that wish to employ TEEs in these environments, e.g., using the TEE to secure funds or private keys, must be designed to tolerate TEE failures and compromises, as the TEE becomes a primary target for attack.

We present **Teechain**, a blockchain payment network that leverages TEEs to execute high performance cryptocurrency payments. Teechain exploits TEE-heterogeneity to provide fault tolerance against TEE failures and compromises. To do this, Teechain implements our contributions: (i) Teechain uses threshold cryptography to protect cryptocurrency funds by requiring m -out-of- n different TEEs to sign any transactions before funds can be spent or accessed; and (ii) Teechain employs force-freeze chain replication to ensure all TEEs in the application maintain consensus about the state of the funds and balances. To evaluate the overhead of our approach, we implement Teechain using Intel SGX enclaves operating on the Bitcoin blockchain. Our experiments show that Teechain achieves at least a 33x higher transaction throughput than state-of-the-art payment networks; a 30-machine Teechain deployment can handle over 1 million Bitcoin transactions per second.

1.6 Thesis outline

The remainder of this thesis is organized as follows:

Chapter 2 provides background and preliminary materials for the concepts used in this thesis. We begin the chapter by discussing the challenges of securing applications in untrusted environments. Next,

we provide a brief history of trusted computing and present early trusted computing architectures. We then continue by providing a definition for trusted execution environments (TEEs) and comparing existing TEE architectures. Finally, we outline the open security challenges faced by TEEs and conclude the chapter. This chapter acts as a general background for the concepts used in this thesis; we present chapter specific background in each of the technical chapters that follow.

Chapter 3 explores the first security challenge faced by TEE-based applications: minimising the trusted computing base (TCB). To overcome this challenge, we propose automatic application partitioning for TEEs: a principled approach for automatically partitioning existing applications to be deployed with TEEs. We begin the chapter by providing an overview of our approach. Next, we present background on securing applications with TEEs. We then continue by presenting the design, implementation and evaluation of Glamdring: the first application partitioning framework that employs our contributions to minimise the TCB of C applications for Intel SGX enclaves. Finally, we discuss related work, highlight limitations of the approach and conclude the chapter.

Chapter 4 explores the second security challenge faced by TEE-based applications: securing TEE boundary interfaces. To overcome this challenge, we propose a set of TEE interface design principles for protecting applications against boundary interface attacks. We begin the chapter by providing an overview of these design principles. We then present background on host OS interfaces for TEEs. Next, we present the design, implementation and evaluation of Citadel: the first TEE runtime system that employs our interface design principles to offer a secure host OS interface. Finally, we discuss related work, highlight limitations of the approach and conclude the chapter.

Chapter 5 explores the third security challenge faced by TEE-based applications: tolerating TEE failures and compromises. To overcome this challenge, we propose applications employ TEE-heterogeneity: the protection of security-critical functionality using multiple heterogeneous TEEs. We begin the chapter by providing an overview of TEE-heterogeneity. We then present background on secure blockchain payment networks using TEEs. Next, we present the design, implementation and evaluation of Teechain: the first blockchain payment network that employs TEE-heterogeneity to tolerate TEE failures and compromises. Finally, we discuss related work, highlight limitations of the approach and conclude the chapter.

Chapter 6 concludes by summarizing the contributions of this thesis and presenting directions for future work.

2

Background

This chapter provides general background and preliminary materials for the concepts used in this thesis. We begin the chapter by discussing the challenges of securing applications in untrusted environments. Next, we provide a brief history of trusted computing and present early trusted computing architectures. We then continue by providing a definition for trusted execution environments (TEEs) and comparing existing TEE architectures. Finally, we outline the open security challenges faced by TEEs and conclude the chapter.

We present additional chapter specific background in each of the technical chapters that follow.

2.1 Securing applications in untrusted environments

Today, many applications are deployed in untrusted environments, where a third party controls the software and hardware in the environment, e.g., public clouds [FGJ⁺09] and volunteer computing [Sar01, MKL⁺02]. In these settings, the underlying infrastructure on which applications run, e.g., the operating system (OS) and the hypervisor, are untrusted by the application owner and may be under

adversarial control. As a result, security incidents have become more frequent in recent years, with data breaches [Ide16] and integrity violations [Sof16] having a major impact on users [Gem14] and user data [Pon13, Ins14].

As such, it has become increasingly important to secure applications and protect sensitive data in untrusted environments. However, this is a complex and challenging goal to achieve: applications today contain large trusted computing bases (TCBs) and rely on many different software and hardware components to execute securely. As a result, they expose large attack surfaces for adversaries and are vulnerable to both software and hardware based attacks.

For example, applications deployed in a public cloud [FGJ⁺09], rely not only on the OS and the hypervisor for security, but also rely on: (i) complex layers of software, such as software for resource management, scheduling [SC16], runtime environments [LYBB14] and deployment, e.g., containers [BJJ10]; and (ii) various hardware components, such as the CPU, main memory, the system bus, and I/O peripherals. If any of these components is under adversarial control, the application can be trivially compromised. As such, to secure applications and protect data in untrusted environments, the TCB of the application must be completely secure against attacks.

In this chapter, we present background and existing work on securing applications and protecting sensitive data in untrusted environments. We explore solutions that: (i) consider both software and hardware based attacks; and (ii) target application and data confidentiality and integrity.

2.2 Trusted computing

To secure applications and protect sensitive data, we introduce trusted computing [Pea02]. The idea behind trusted computing is to develop technologies that give entities who interact with an application, e.g., users or other software, guarantees about the behaviour of the application. More specifically, an application or piece of software is said to be trusted if it *always behaves in the expected manner for the intended purpose* [Mar08]. This means that even if the application is deployed in an untrusted environment, where an adversary has control over the environment, they cannot force the application to misbehave. As a result, the entire trusted computing base (TCB) of the application must be secure.

One of the first observations made in regards to trusted computing is that software-only approaches are generally insufficient for achieving both confidentiality and integrity against a powerful adversary. This is because they cannot protect against attacks where the adversary has complete control over the system, e.g., physical access to the machine or control over the OS. As such, even though several

trusted computing approaches have been proposed based on software exclusively [MPB10, ASAP13, SP12], these cannot provide the same guarantees as hardware-based approaches. This is because it is considered much more difficult for an attacker to modify the functionality of hardware than it is software, to the extent that hardware is generally considered immutable. Therefore, to fully protect an application, any trusted computing platform will always have some form of trusted hardware in which trust is said to be *rooted*. As such, we consider hardware-based trusted computing architectures in our background and discuss these proposals in depth below.

2.2.1 A brief history and overview

Trusted computing was originally proposed by the *Trusted Computing Platform Alliance* (TCPA), an industry alliance created to develop and standardize trusted technology [Pea02]. The TCPA was formed in 1999, by Compaq, Hewlett-Packard, IBM, Intel, and Microsoft, with the goal to combat the growing number of software security threats. The TCPA released its very first *trusted platform* specification in 2001, describing a new component to enable trusted computing called a *trusted platform module* (TPM) [Pea02]. The TPM was a security module, such as a cryptoprocessor placed next to the CPU, that offered trusted functionality to protect security-sensitive information, such as cryptographic keys. The TCPA later became known as the *Trusted Computing Group* (TCG) in 2003, and continued to develop specifications for trusted platforms and computing devices. This includes expanding into trusted computing for mobile devices [Pea02].

Trusted computing functionalities: In their first specification, the TCG argued that any trusted computing platform should provide at least three basic features or functionalities [GM09]: (i) *authenticated boot* to enable a platform’s state to be measured and recorded securely; (ii) *protected storage* to allow data to be stored securely; and (iii) *platform attestation* to allow the platform’s state to be reported securely. The TCG later proposed two additional functionalities that trusted platforms should also provide [Ver06]: (iv) *secure boot* to allow a platform’s state to be securely measured, verified and recorded; and (v) *software isolation* to allow software to run securely without other software or components interfering with execution. We discuss each of these five functionalities below, and outline the components required to achieve them [GM09]:

- **Authenticated boot:** Peinado, England and Chen [PEC05] proposed using *integrity measurements* to securely measure the state of a platform. An integrity measurement is a cryptographic digest or hash computed over the platform, providing a secure and efficient mechanism for identification. For example, to measure an application or a piece of software, you would compute a

hash over the set of instructions in the application, alongside the input given to the application and the initial state of the application. This would form an integrity measurement over the application, allowing you to determine exactly what application is being run.

To compute an integrity measurement securely, it is necessary to define a *root of trust for measurement* (RTM). The RTM is a component typically embedded directly in the platform that is explicitly trusted to compute integrity measurements correctly. The RTM is often implemented in the normal platform processor, controlled by the *core root of trust for measurement* (CRTM). To secure the CRTM, the TCG proposed that: (i) the CRTM be immutable, so that it can only be replaced or modified by the platform manufacturer alone; and (ii) the CRTM be tamper-evident, so that any physical attempts to modify the CRTM can be visibly seen.

- **Protected storage:** To enable the trusted platform to store sensitive data securely, the TCG argued that the platform requires a *root of trust for storage* (RTS). The RTS is defined as a collection of trusted components on the platform that allow it to protect the confidentiality and integrity of specific pieces of data, in both non-volatile and volatile memory. For example, the RTS typically maintains an accurate summary of the platform integrity measurements once they are computed by the RTM.
- **Platform attestation:** To allow the state of the platform to be reported securely, the TCG argues that the platform requires a *root of trust for reporting* (RTR). The RTR is formed by a collection of trusted components that allow it to report integrity measurements securely to various external entities, such as remote applications. For example, in the case of the TPM, the TCG define the RTR to be provided by a number of functional components embedded in the TPM. These include: (i) special purpose registers for recording platform state; (ii) a means of securely reporting this state to remote entities; (iii) secure volatile and non-volatile memory; and (iv) a trusted engine for performing cryptographic operations, such as computing hashes, asymmetric key generation and encryption.

The TCG argue that any TPM should provide at least authenticated boot, protected storage and attestation. To ensure the security of these components, the TCG argue that: (i) the TPM must be completely protected against software attacks, i.e. the RTM, RTS and RTR must be immutable; and (ii) the TPM should provide some degree of protection against physical attacks, e.g., be tamper-evident.

- **Secure boot:** The TCG later extended the definition of authenticated boot to secure boot, in the context of mobile devices. Secure boot extends the original definition by including a process for boot verification. This allows the platform to not only conduct measurements during the boot

process, as defined by authenticated boot, but to compare the measurements to the expected values, in order to determine if the platform has been modified or tampered with. Secure boot, as defined by the TCG, requires a *root of trust for verification* (RTV). The RTV is defined as a computing engine capable of verifying the platform’s integrity measurements against the expected values, termed the *reference integrity measurements*.

- **Software isolation:** Finally, to protect the execution of software on various trusted platforms, the TCG define the use of software isolation, or *isolation execution environments* (IEE). An IEE protects the software from external interference, while allowing secure communication between software in different IEEs. A common way to achieve this is through the use of a *virtual machine monitor* (VMM) hosted on the OS [GM09]: software and applications can be run on their own virtual machines, managed independently by appropriate guest OSs executing in each virtual machine.

Trusted computing outside the TCG: Although trusted computing has now become synonymous with the TCG, it is worth noting that other trusted computing architectures and specifications were also proposed in the early 2000’s. These originated from sources outside the TCG, including both industry and academia. Examples include [GM09]: (i) the work done by Microsoft under the name *Palladium*, and now the *Next Generation Secure Computing Base* [PCEM04] in 2004, which provides a secure OS architecture using various trusted platform features; (ii) *Terra* [GPC⁺03] in 2003, which proposes a *trusted virtual machine monitor* (TVMM) that partitions the computing platform into several isolated, virtual machines; (iii) *Perseus* [SS03] in 2003 and the *Open Trusted Computing architecture* [KLR⁺06] in 2006, which use virtual machine monitors and secure microkernels to provide isolated software execution; and (iv) the *European Multilaterally Secure Computing Base* [SSP05] in 2005, which employs a microkernel-based isolation layer.

While many of these works originate from different sources entirely, they all face the same fundamental security issues when designing a trusted computing platform. As such, they all provide similar functionalities and features. For example, the solutions mentioned above, although independent of the TCG, provide: (i) software isolation through various isolation layers and compartments; and (ii) assume the use of a trust-enhanced hardware platform providing features similar to the TPM. It is therefore important to note that while the specifications proposed by the TCG have created a solid foundation for trusted computing today, most solutions follow specifications and designs of their own. As a result, we discuss various trusted computing architectures in more detail in Section 2.2.2 below.

2.2.2 A survey of trusted computing architectures

In this section, we present and discuss several trusted computing architectures that have been proposed over the past 20 years. We critique these architectures with respect to the goals outlined in Section 2.1: securing applications and protecting sensitive data in untrusted environments. We defer a more detailed discussion and comparison of trusted execution environments (TEEs) to Section 2.3.

The architectures presented here are by no means a complete list of all trusted architectures available today; rather, they serve only to provide examples of the different solutions that have been proposed. We present these in chronological order:

- **Trusted Platform Module** [Pea02] (TPM): The TPM was one of the first trusted computing architectures proposed, with the TPM implementation following the specifications defined by the TCG [Pea02]. The TPM is a secure co-processor placed on the motherboard, alongside the CPU; it is capable of securely storing cryptographic keys and performing attestation. As the TPM does not require modifications to the CPU, it has become popular, being widely deployed on commodity desktops and servers [Baj02].

The TPM is passive by design: software executing on the CPU must interact with it explicitly. To perform attestation, any software that runs on the platform, e.g., the boot loader and OS, needs to be measured by the TPM, and thus forms part of the TCB. Attestation relies on the CPU to report its own measurement from each of the phases during boot. During each of these phases, the CPU sends a hash to the TPM. All hashes are tied to the previous hashes by the TPM to prevent malicious software at a specific boot stage from concealing itself. Tampering with code is only detected during measurement, and all software components are trusted after being loaded. The TPM offers limited protection against physical attacks, because all buses that connect the TPM and the CPU are considered part of the TCB. Therefore, an adversary with physical access to the machine can compromise software integrity and confidentiality by tapping these buses [KSP05].

At manufacturing time, an Endorsement Key (EK) is generated and written to persistent memory within the TPM. The EK is unique to every chip and hidden from users. It provides the master key for all TPM operations. Attestation Identity Keys (AIKs) and storage keys can be generated when needed using the EK and stored in volatile memory within the TPM. AIKs are used for all operations directly involving cryptographic signatures, while the storage keys are used for the secure storage of data through encryption and integrity protection, i.e., *data sealing*.

While the TPM has become widely deployed for protecting sensitive data, e.g., cryptographic

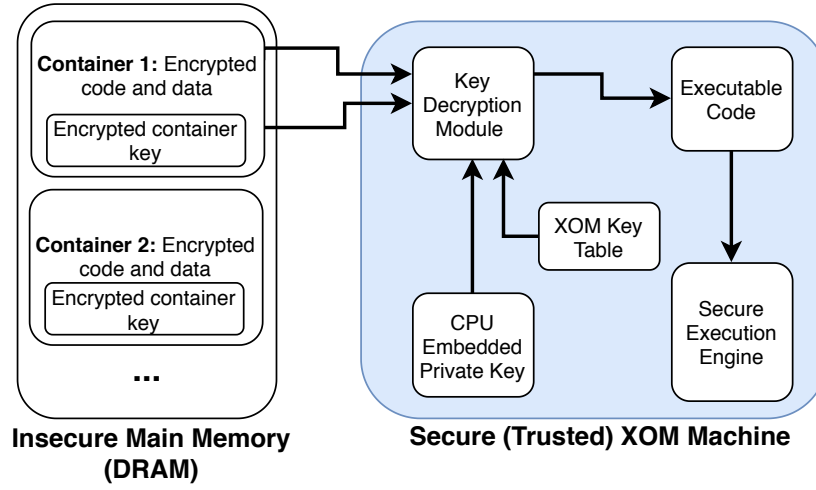


Figure 2.1: **The XOM [LTM⁺00] platform architecture** (In this model, containers are encrypted and stored in untrusted main memory, before being decrypted and executed in the secure execution engine.)

keys, it offers limited support for protecting applications in untrusted environments, e.g., public clouds. This is because: (i) the TPM requires a large TCB, i.e., due to having to measure the entire software stack on boot, such as the boot loader and the OS; and (ii) the TPM offers no hardware protections, making it vulnerable to attackers with physical access to the machine.

- **XOM [LTM⁺00]:** The XOM architecture, which stands for eXecute-Only Memory, was one of the first architectures to use containers to secure code and data on an untrusted host. XOM argues that hardware solutions are more efficient than software solutions, and as such, XOM uses hardware protections and access control to enforce security.

Figure 2.1 shows the XOM architecture. The main features of this architecture are: (i) data and program partitioning between secure containers; and (ii) hardware enforced encryption and integrity protection of data when stored in DRAM, i.e., insecure main memory. First, the XOM architecture protects the memory of a single container through automatic encryption, decryption and HMAC protection. This is implemented in hardware and can be seen in Figure 2.1, where each container is stored in main memory in encrypted format: whenever a container is brought back onto the CPU, it is decrypted securely using a private key stored and owned by the CPU. After decryption, the container can then be executed by the secure execution engine. Second, in the processor package, each cache line is tagged with an identifier that specifies the identifier of the corresponding container that uses the cache line. This identifier corresponds to the encrypted container key in Figure 2.1. When a container wishes to access data, the identifier of the container, i.e., the encrypted container key, must match the identifier stored with the cache line. This means that containers are only allowed to read and modify the data they own.

By integrating encryption and HMAC functionality in the processor’s memory controller, XOM protects container memory from physical attacks on DRAM. The encryption and HMAC functionality is used for all cache line evictions and fetches, and the error-correcting code bits in DRAM chips are overwritten to store HMAC values.

However, despite offering some protection against physical attacks, XOM faces several significant limitations when protecting applications in untrusted environments: (i) XOM cannot guarantee memory freshness in DRAM, so software is vulnerable to replay attacks; (ii) XOM does not protect memory access patterns, meaning that it is vulnerable to cache timing attacks; and (iii) XOM does not provide software attestation support, i.e., no platform measurements are performed when a container is deployed on the platform. This makes it difficult to verify the integrity of any application executing in an untrusted environment.

- **AEGIS** [SOD05]: AEGIS is another example of the early trusted computing architectures, proposed in 2003. AEGIS is a single-chip architecture, providing programs with a tamper-evident and tamper-resistant environment, where memory tampering, either through software or physical attacks, can be detected. Figure 2.2 shows the AEGIS platform architecture. In AEGIS, the chip is trusted, as well as part of the OS called the security kernel. The security kernel operates with privileged protection, provided by processor isolation, protecting it from attacks from other parts of the OS. The security kernel handles tasks such as virtual memory management and hardware exceptions. The rest of the OS, external memory and peripherals are untrusted, placed outside the TCB. To protect off-chip memory, AEGIS uses: (i) integrity-verification to protect data integrity outside of the chip; and (ii) symmetric encryption with random initialization vectors to protect data confidentiality.

AEGIS provides support for software isolation through secure containers that are isolated by the security kernel. Container isolation is achieved by configuring page tables for address translation, preventing untrusted software from accessing the contents of secure isolated containers. Even though the untrusted OS can evict pages from container memory, the security kernel verifies the correctness of these operations. Moreover, the kernel uses integrity-verification, encryption and freshness protection for container related pages swapped in and out of the memory.

Finally, AEGIS also supports platform attestation, through an attestation procedure that includes the security kernel’s cryptographic hash in the measurement. In order to perform attestation, AEGIS can use either a protected private key fused in the processor for signing, or a set of Physically Unclonable Functions (PUFs) [SOSD05].

While AEGIS offers strong software and hardware protections, it suffers from several limitations

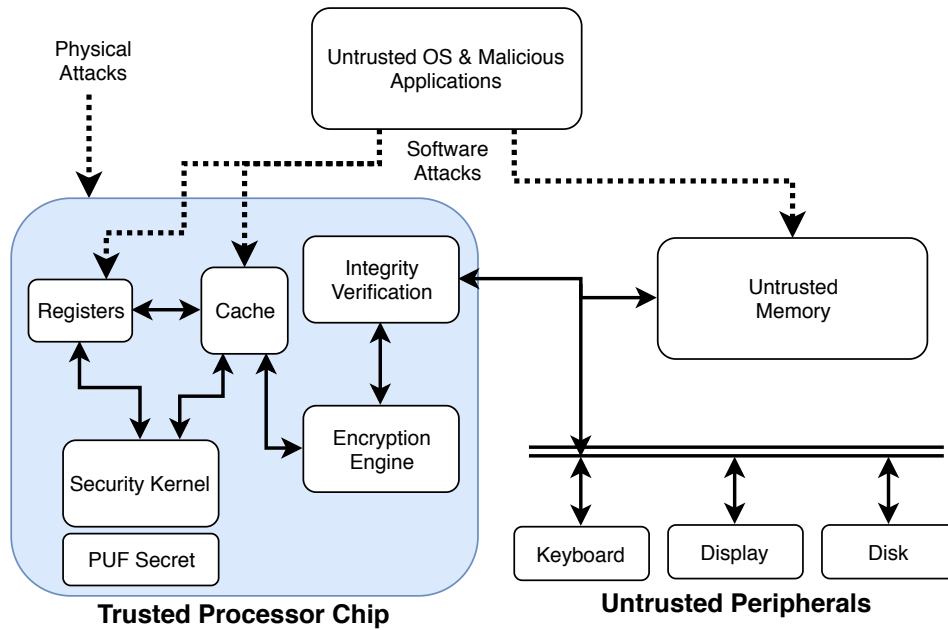


Figure 2.2: **The AEGIS [SOD05] platform architecture** (In this model, the tamper-resistant processor chip separates trusted components, e.g., the security kernel and encryption engine, and untrusted components, e.g., main memory, peripherals, and the untrusted OS.)

when protecting arbitrary applications in untrusted environments: (i) due to the untrusted OS being able to handle pages, AEGIS leaks memory access patterns and is also vulnerable to cache-timing attacks [SOSD05]; and (ii) AEGIS requires a trusted security kernel, which significantly increases the TCB of the application.

- **IBM 4765 Secure Co-processor [ABC⁺12]:** The idea behind a secure co-processor is to encapsulate an entire computing platform into a single enclosure. This includes the CPU, caches, DRAM, and I/O controllers. Secure co-processors place all of these components into a tamper-resistant environment with sensors to detect physical attacks or tampering. If such attacks are detected, the co-processors may destroy any secrets or confidential information contained within the enclosure. This provides strong security against physical attacks. However, building such enclosures is often expensive [And01].

The IBM 4765 [ABC⁺12] and its various predecessors [DLP⁺01] are notable examples of secure co-processors. The IBM 4765, as seen in Figure 2.3, relies on physical isolation in a secure tamper-resistant enclosure to guarantee security. The IBM 4765 secures software by separating the system software from application software and running them on different CPUs: system software executes on the service CPU, while application software executes on the application CPU. These CPUs are connected by a system bus controller that prevents application software from accessing privilege resources owned by the system software, such as secrets stored in battery-backed RAM.

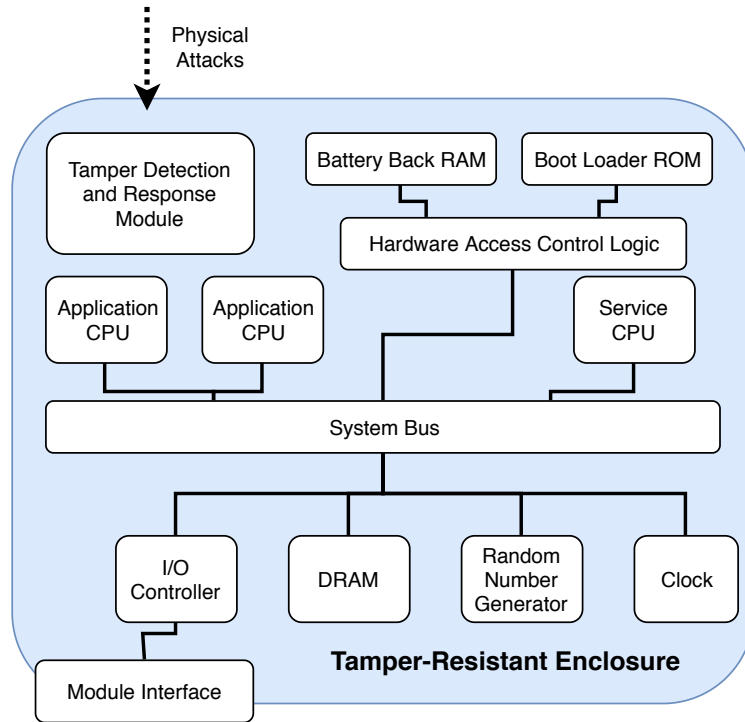


Figure 2.3: **The IBM 4765 secure co-processor architecture [ABC⁺12]** (In this model, all components are encapsulated into a single tamper-resistant enclosure.)

Finally, the IBM 4765 also supports software attestation. The attestation key is stored in battery-backed memory that is only accessible to the service CPU. To perform attestation, the service CPU executes a bootloader stored in read-only-memory (ROM), which measures and loads system software. After the system software has been loaded, application code can then be measured and loaded into DRAM. The system software provides attestation services to the application directly.

While the IBM 4765 offers strong security guarantees and useful features, its high production price means that it is not widely appropriate for securing applications in untrusted environments. Moreover, because the IBM 4765 requires measuring system software for attestation, e.g., on boot, this increases the size of the TCB for applications.

- **Intel Trusted Execution Technology (TXT) [Gra09]:** Intel TXT, formerly known as *La-Grande* technology, aims to defend trusted platforms from software attacks, including attacks against the firmware and BIOS. At a high-level, TXT constructs a chain of trust from an on-board TPM that extends to the Virtual Machine Monitor (VMM) to enable trusted hypervisor and OS boot. Intel TXT combines the TPM’s software attestation model and tamper-resistant chip with hardware virtualization from the CPU, i.e., Intel Virtualization Technology [UNR⁺05]. This enables Intel TXT to reduce the software inside the secure container to a virtual machine, i.e., guest OS and application.

Intel TXT isolates the software executing inside the container by ensuring that the container has control over the entire platform while running. This is achieved by an authenticated initialization code module that performs a partial system reset before starting the container's VM. For deployment, Intel TXT requires a TPM with an extended register set. This register set contains: (i) a static root of trust measurement (SRTM) for the platform on measured boot; and (ii) a dynamic root of trust measurement (DRTM) for the VM being executed.

While Intel TXT is wide availability on commodity Intel CPUs today, it suffers from several significant limitations that make it challenging to use in untrusted environments: (i) Intel TXT does not provide DRAM protections, such as encryption and integrity verification; and (ii) Intel TXT contains a large hardware TCB, comprising the TPM, CPU chipset, motherboard and system buses. This makes applications secured using Intel TXT vulnerable to a wide range of physical attacks.

- **Bastion** [CL10]: Bastion is a hardware and software architecture. It relies on a trusted hypervisor together with a modified processor to ensure confidentiality and integrity for software modules. To provide protection through the hypervisor, each software module has a security segment that lists the virtual addresses and permissions of all the container's pages in the hypervisor. These segments associate each physical page to a specific container and virtual address space. At runtime, every memory lookup is checked against the rules specified in the Translation Lookaside Buffer (TLB). The hypervisor checks whether the virtual address responsible for the access corresponds to the one associated with the physical page and specific software module. 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 matches the expected values associated with a specific security segment of a container.

Bastion tolerates physical attacks on all hardware components apart from the CPU package, and offers memory protection. Since everything apart from the CPU and the hypervisor is untrusted, Bastion does not trust the platform's firmware, and instead computes the cryptographic hash of the hypervisor only after the firmware finishes booting. The hypervisor's hash is included in the measurement reported by software attestation. Bastion was the first architecture to offer direct hardware protection of the hypervisor from both software and physical attacks. Bastion demonstrates the feasibility of bypassing an untrusted OS to provide security with better performance than the TPM. Similar to AEGIS, Bastion also allows the untrusted OS to manage pages of the secure containers; these pages are encrypted, authenticated and freshness protected.

While Bastion tolerates a wide range of software and hardware based attacks, it suffers from several disadvantages when protecting applications in untrusted environments: (i) as the OS can

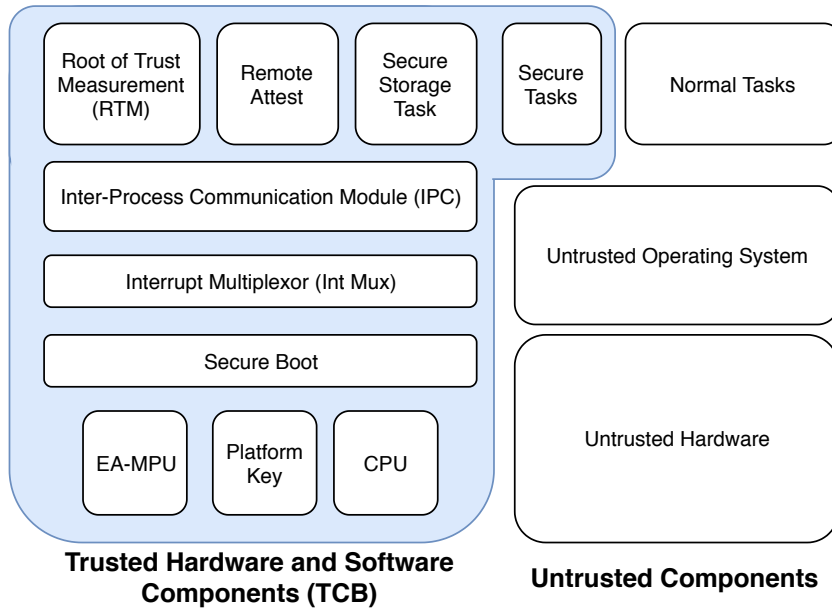


Figure 2.4: **The TyTan [BEMS⁺15] architecture** (In this model, we see the clear separation between trusted components, e.g., the EA-MPU and the root of trust measurement, and untrusted components, e.g., the OS and untrusted tasks.)

evict pages, Bastion leaks memory access patterns at the page level; (ii) Bastion is vulnerable to cache side-channel attacks; and (iii) Bastion requires a large TCB, as the hypervisor is trusted.

- **TyTan [BEMS⁺15]**: TyTan is a trusted computing architecture for lightweight devices. It offers isolation between different tasks, using a hardware and software co-design. TyTan argues that it is the first security architecture for low-end embedded systems, providing all of: (i) hardware-assisted dynamic root of trust, which allow secure task loading at runtime; (ii) secure inter-process communication (IPC) between the different tasks; and (iii) local and remote software attestation to allow locally run processes and remote processes to verify the security of a TyTan task. TyTan is implemented as an extension to Intel’s Siskiyu Peak architecture, and uses the FreeRTOS real-time OS [ACCSB09].

TyTan is specifically designed for multi-tenant deployment scenarios, allowing for secure execution of mutually distrusting tasks on the same platform. Figure 2.4 shows the TyTan architecture. TyTan employs an Execution-Aware Memory Protection Unit (EA-MPU), which enforces program counter-based isolation and enables IPC. The EA-MPU ensures that each task can only access the appropriate memory regions, thus enforcing task isolation. The EA-MPU also ensures that tasks can only be invoked or run through dedicated entry points.

Several software components form part of the TyTan TCB. Each of these trusted components is isolated from the rest of the platform by the EA-MPU. The EA-MPU supports loading and unloading of trusted tasks at runtime. These trusted tasks include: (i) the Secure Boot task which

is responsible for loading all other trusted software components at startup; (ii) the Root of Trust for Measurement (RTM) task which is used to attest other tasks, by computing cryptographic hashes of the binary code of each task; (iii) the IPC Proxy task, which offers Secure IPC by forwarding messages from sender tasks to receiver tasks; (iv) the Secure Storage task, which seals data by storing it encrypted in non-secure memory. Tasks communicate with the secure storage via secure IPC; and (v) a trusted Interrupt Multiplexor (Int Mux) task which is used to securely save the context of an interrupted task to its stack and clear the CPU registers before control is passed to the interrupt handler.

Although TyTan ensures strong isolation between mutually distrusting applications executing on the same platform, TyTan requires a hardware and software co-design. As such, this results in a large TCB, making it challenging to secure applications and protect sensitive data in untrusted environments.

2.3 Trusted execution environments (TEEs)

Trusted execution environments (TEEs) are a specific subclass of trusted computing architectures that offer confidentiality and integrity guarantees for arbitrary code and data. In contrast to other trusted computing architectures, TEEs rely on hardware isolation and cryptographic primitives, e.g., encryption and integrity protection, to protect code and data in general-purpose platforms. This is done without requiring auxiliary CPUs or expensive hardware. In this section, we provide a definition for TEEs, identify the building blocks that make a TEE, and discuss and compare several different TEEs. We conclude the section by discussing the security challenges faced by TEEs.

2.3.1 A definition and overview

There have been many different TEEs proposed over the past 20 years [SAB15]. As a result, there is some ambiguity about what exactly constitutes a TEE [SAB15]. For the purpose of this thesis, and our discussions, we adopt the broad definition proposed by Mohamed et al. [SAB15]: a "*Trusted execution environment (TEE) is a tamper-resistant processing environment that runs on a separation kernel. It guarantees the authenticity of executed code, the integrity of state (e.g. CPU registers, memory and sensitive I/O), and the confidentiality of code, data and state stored on persistent memory. In addition, it should also provide remote attestation [...] for third-parties. The content of TEEs are not static; they can be securely updated. The TEE resists against all software attacks as well as physical attacks performed on main memory [...]*".

Under this definition, Mohamed et al. [SAB15] define the *separation kernel* as a fundamental component necessary to support TEEs. The primary purpose of the separation kernel is to enable different systems with different security requirements to co-exist on the same platform in isolated partitions. Unlike other security kernels, such as the OS and hypervisor, the separation kernel provides both *time and space partitioning*. That is, separation kernels ensure strict isolation between each partition, meaning: (i) data cannot be read or modified by other partitions; (ii) security breaches in one partition should not affect other partitions; (iii) communication between partitions cannot occur unless explicitly permitted; and (iv) shared resources, such as caches or shared memory, cannot leak information about other partitions.

Using the separation kernel, Mohamed et al. [SAB15] outline the general building blocks that together constitute a TEE. These building blocks can be seen in Figure 2.5. At a high-level: (i) the TEE is separated from the rest of the platform via the separation kernel; (ii) the TEE has several trusted components that allow it to provide confidentiality and integrity to code and data, including secure boot, secure provisioning and secure storage; and (iii) the TEE can communicate directly with the rest of the platform via the inter-environment communication interface and a trusted I/O path for peripherals.

We briefly describe each of the TEE building blocks in detail below. We also note the similarities these building blocks have with the trusted computing functionalities originally outlined by the TCG in Section 2.2.1:

- **Secure boot** ensures that only trusted and verified code can be loaded onto the platform and run. If the code is modified by an adversary or the adversary attempts to execute their own code or application, secure boot prevents this from happening. Secure boot is often achieved in practice by performing integrity measurements during the boot process, chaining each of the measurements together to form a chain of trust. We note that secure boot is one of the properties outlined by the TCG in Section 2.2.1 as one of the core functionalities for any trusted platform.
- **Secure scheduling** allows applications inside the TEE to execute without affecting the responsiveness of the main platform OS. This offers execution fairness between the TEE and the rest of the platform. In most cases, this is done via the untrusted OS as the TEE is designed to support preemption, i.e., the OS has control over scheduling decisions and can preempt the TEE. However, because the OS is often untrusted, it means that TEEs do not typically provide availability guarantees to applications as a malicious OS may choose not to schedule the TEE at all. Similarly, hardware failures, such as power cuts or faults, may also prevent the entire platform from running.

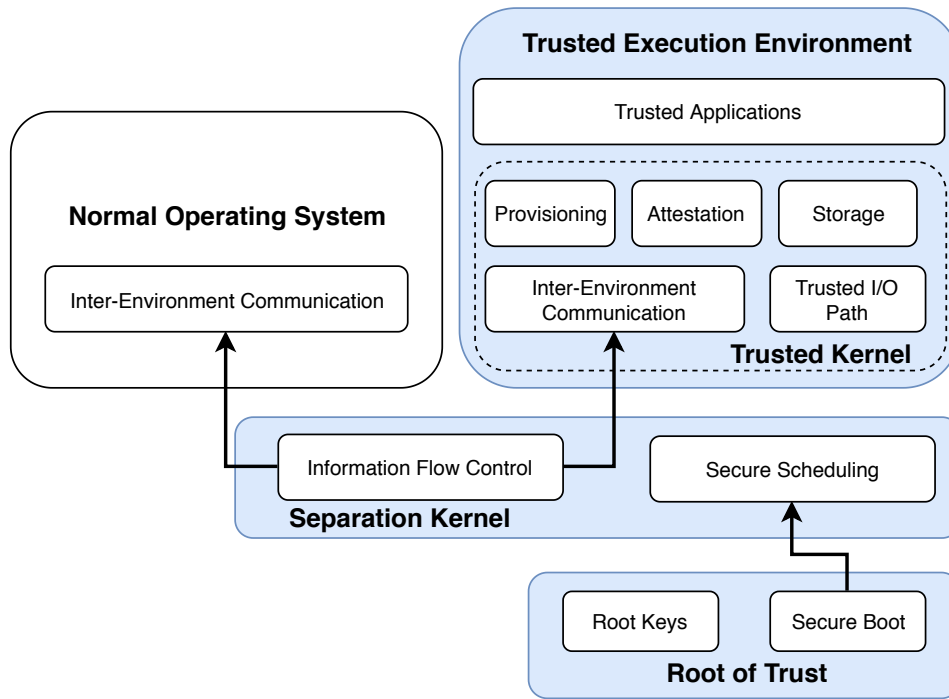


Figure 2.5: **The TEE building blocks** (Here, we see the various TEE building blocks outlined by Mohamed et al. [SAB15]; these are the building blocks required to operate TEEs on a computing platform.)

- **Inter-environment communication** allows a TEE to communicate with the rest of the computing platform. This is often required if the TEE needs to perform inter-process communication (IPC), or interface with an untrusted application. Despite the need for inter-environment communication, communication interfaces typically open attack vectors, as an adversary might exploit these interfaces to attack the TEE. Communication interfaces therefore need to be secured.
- **Secure storage** allows data to be stored securely on the platform, only accessible to authorized entities, ensuring the confidentiality, integrity and freshness of the data. A common way to implement secure storage in TEEs is to provide *sealed storage*, that is, use a cryptographic key that can be accessed only by the TEE to encrypt, integrity protect and version data written to the persistent storage, e.g., a hard drive or disk. Data versioning requires a non-volatile storage area only accessible by the TEE. We note that secure storage is analogous to protected storage, as outlined by the TCG in Section 2.2.1.
- **Trusted I/O paths** protect the integrity and confidentiality of any communication between the TEE and external peripherals in the platform (e.g., the network, keyboard or monitor). Examples of where this is necessary and useful include: (i) the TEE needing to display confidential information to the user on the screen; and (ii) the user needing to type in a confidential password that can be forwarded directly to the TEE. Trusted I/O paths enable stronger platform security

for human-computer interaction.

2.3.2 A comparison of TEEs

In practice, most TEEs are designed with specific use cases in mind, e.g., mobile or cloud computing. As such, they only implement a subset of the TEE building blocks outlined above (in Section 2.3.1). In this section, we critique several TEE implementations with respect to our goals: securing applications and protecting sensitive data in untrusted environments (see Section 2.1). Table 2.1 shows a summary of the building blocks implemented by the TEEs we discuss. The TEEs presented here are not a complete list of solutions; rather they serve only to provide examples of notable TEE implementations. We discuss these solutions in chronological order below.

	Secure boot	Secure scheduling	Inter-environment communication	Secure storage	Trusted I/O paths
ARM TrustZone [ARM09]	✓	✗	✓	✗	✓
Intel SGX [Int14]	✓	✓	✓	✓	✗
Iso-X [EEO ⁺ 14]	✓	✓	✓	✗	✗
Sanctum [CLD16]	✓	✓	✓	✗	✗

Table 2.1: Summary of the TEE building blocks and functionalities offered by different TEE implementations.

- **ARM TrustZone** [ARM09]: TrustZone was one of the first TEEs commercially available. It offers a collection of hardware modules that can be used to partition the platform into a TEE, known as the *secure world*, and a REE (Rich Execution Environment), known as the *normal world*, where untrusted components execute, such as the untrusted OS. TrustZone is an implementation of the TEE specification proposed by *GlobalPlatform* in 2011 [Pla11]. In the specification, GlobalPlatform argued that REE resources, e.g., memory, should be accessible from the TEE, but that TEE resources should be protected from the REE, unless explicitly allowed. This ensures that only trusted applications can access trusted resources. Figure 2.6 shows the TEE architecture specification proposed by GlobalPlatform and implemented by TrustZone.

TrustZone is implemented for system-on-chip (SoC) architectures currently used in a large number of existing mobile devices. The secure world in TrustZone provides protection for trusted hardware and software resources, while the normal world is prevented from accessing secure world resources. This is enforced through two hardware-based mechanisms: (i) the SoC’s bus ensures that secure world resources cannot be accessed from normal world resources; and (ii), a TrustZone-enabled processor core exclusively executes code in either the secure or normal world. To enforce isolation between trusted hardware resources on the bus, a control bit known as the

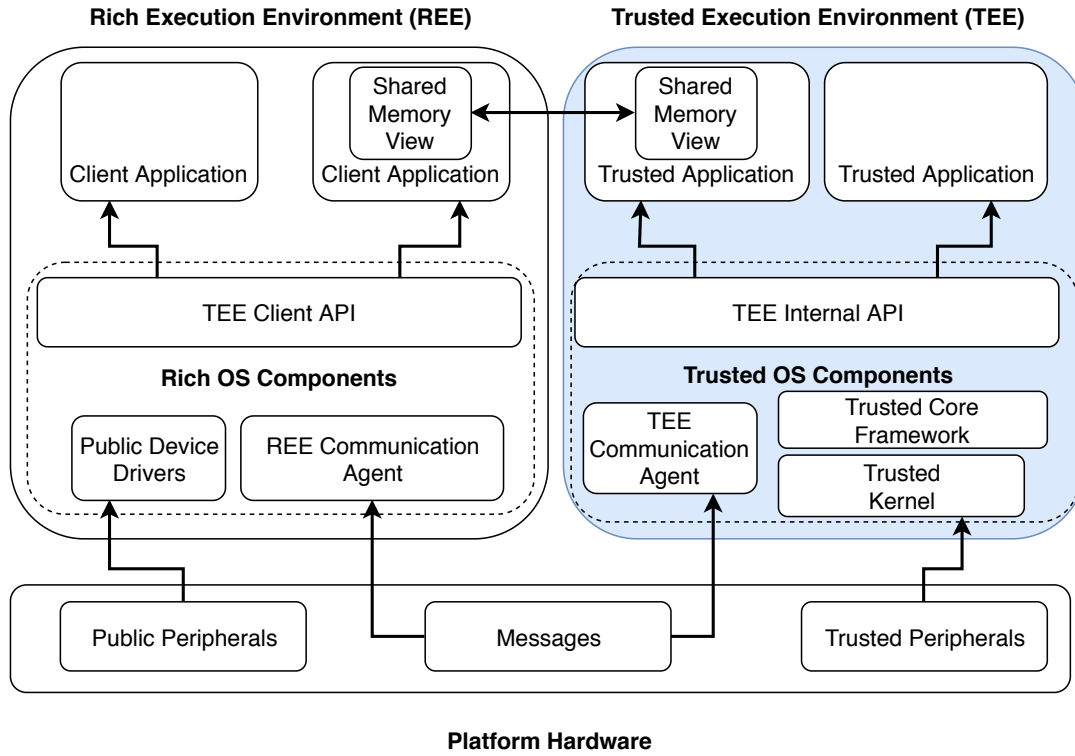


Figure 2.6: **The TEE architecture specification proposed by the GlobalPlatform [Pla11]** (The platform is divided into the Rich Execution Environment (REE), containing untrusted OS components and applications, and the Trusted Execution Environment (TEE), containing trusted OS components and applications.)

Non-Secure (NS) bit was added to the AXI specification [ARM09]. TrustZone cores can switch between worlds at runtime. When the processor core is in the secure world, it generates AXI transactions with the NS bit set to zero, allowing it to access resources in both security domains. When a core is in the normal world, and the NS bit is set to one, the core can only access resources in the normal world.

To perform a context switch between worlds, the processor core passes through monitor mode, which serves as a gatekeeper. Monitor mode saves the state of the current world and restores the state of the world being switched to. The address mappings in the memory management unit (MMU) can be configured independently for each world. This allows the OS in each world to enforce its own memory management. IPC with small messages can be done by placing the message inside registers and then switching worlds. For larger messages, it is recommended to use shared memory.

During boot, a chain of trust is formed by verifying the integrity of the trusted second stage boot loader and trusted OS before execution. The firmware of the first stage boot loader is implicitly trusted, and is typically located in read-only-memory (ROM). The firmware verifies the integrity

of the second stage boot loader, and the second stage boot loader verifies the secure world OS. Some implementations of the secure OS also verify the integrity of trusted applications before loading them. TrustZone uses signatures to perform verification of each stage of the boot process: a vendor signs the code, and each of the verifying software components, e.g., the firmware, use the public key to verify the signature at runtime.

Table 2.1 shows a summary of the TEE building blocks and functionalities offered by TrustZone. TrustZone provides: (i) secure boot through platform measurements; (ii) inter-environment communication via message exchanges; and (iii) trusted I/O paths by assigning peripherals to the secure world exclusively. However, TrustZone does not offer: (i) secure scheduling, as the secure world and normal world execute different OSs, and as such, TrustZone doesn't provide a way to fairly time slice between the two worlds; and (ii) secure storage, as TrustZone does not provide hardware support for data sealing.

Despite TrustZone being widely deployed in mobile devices today, it has several limitations that make it challenging to protect applications in untrusted environments: (i) TrustZone violates the principle of least privileged as trusted applications from different vendors run in the same secure world. This means any trusted application can compromise the entirety of the secure world; (ii) TrustZone does not completely separate the cache between worlds, meaning the normal world can evict cache lines to perform cache timing attacks on applications in the secure world; (iii) TrustZone components offer limited protections against physical attacks, and thus are vulnerable to adversaries with physical access to the device; and (iv) TrustZone requires a large TCB, as secure boot builds a chain of trust through several different software components, e.g., the firmware, boot loader and the OS.

- **Intel Software Guard Extensions** [MAB⁺13]: Intel SGX, first announced in 2013 [MAB⁺13], is one of the most notable TEE implementations currently available in commodity CPUs today. Intel SGX offers secure containers called *enclaves*, which protect the confidentiality and integrity of applications without modifying the CPU's execution path. To achieve this, Intel SGX adopts an approach that trusts the security of the CPU package and microcode, while relying on the untrusted OS to manage enclave memory and page tables.

Similar to the design of Bastion (see Section 2.2.2), Intel SGX allows the untrusted OS to evict enclave pages. The CPU's microcode ensures the confidentiality, integrity and freshness of enclave pages through hardware enforced encryption, integrity protection and versioning. This is provided by the Memory Encryption Engine (MEE) — a secure hardware unit added to the processors memory controller. The MEE transparently: (i) encrypts memory pages that leave the confines of the processor package, e.g., when flushed to DRAM, and; (ii) decrypt and integrity check

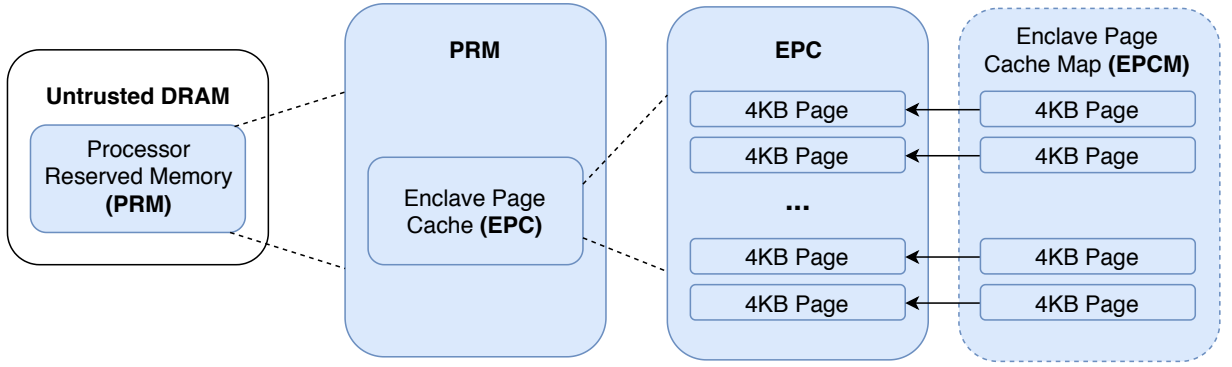


Figure 2.7: **The Intel SGX [MAB⁺13] architecture** (Here, we see the layout of Processor Reserved Memory (PRM), the Enclave Page Cache (EPC) and Enclave Page Cache Map (EPCM) in untrusted DRAM.)

memory pages when brought back on chip, e.g., when paged in from DRAM.

All accesses to enclave memory adopt the same address translation rules as normal memory. However, Intel SGX ensures that untrusted code outside the enclave cannot access enclave pages or memory. This includes isolation from higher-privileged software, such as the OS and hypervisor. To achieve this, enclave code and data is stored in Processor Reserved Memory (PRM), which can be seen in Figure 2.7. The PRM is a subset of DRAM that cannot be directly accessed by other software, including system software. The Enclave Page Cache (EPC) and the Enclave Page Cache Map (EPCM), both subsets of the PRM, are protected by the CPU against untrusted accesses. The EPC stores enclave pages, i.e., trusted enclave code and data, and the EPCM manages meta-data about the pages within the EPC, e.g., enclave page access rights and virtual addresses. If the capacity of the EPC is exhausted, i.e., too many enclave pages exist in the EPC, the pages are transparently encrypted and integrity protected by the MMU before being flushed out to DRAM by the OS.

All enclave operations, including creation, initialization, and destruction, are performed by system software, while entering and leaving the enclave occurs at the application layer: the enclave provides a single entry point into the enclave, and dedicated instructions can be called to enter and leave. Upon leaving the enclave, the context of the current thread is saved within an EPC page and all registers are cleared. The context is loaded again when the enclave is entered. If an interrupt occurs during enclave execution, an Asynchronous Enclave Exit (AEX) occurs by the CPU, which saves the current enclave execution context.

When an enclave is initialized, all enclave page contents, including code and static data, are measured, creating a *report*. The report can then be signed by a private key burnt into the CPU, verifying that the report belongs to a valid Intel SGX processor. This forms an enclave *quote*. Quotes provide a means to attest to the correctness and validity of enclaves and the hardware

on which they run. If an enclave quote does not match the expected value, the enclave can be terminated or prevented from accessing sensitive data during provisioning.

In addition to local and remote attestation using reports, data produced by an enclave can also be securely stored, or sealed to the enclave and written to stable storage. Sealed data can serve as permanent storage beyond the lifetime of a single enclave. Local attestation, remote attestation, and sealing all rely on the non-forgability of the initial enclave measurement. Intel SGX also provides *monotonic counter* support, to allow data to be versioned, preventing roll-back attacks when an enclave is shutdown and restarted.

Table 2.1 shows a summary of the TEE building blocks and functionalities offered by SGX. SGX provides: (i) secure boot through remote attestation, as application code and data can be precisely measured and attested before being provisioned with sensitive data; (ii) inter-environment communication, as enclave code can access untrusted data, allowing shared memory communications; (iii) secure storage, as SGX provides support for data sealing to stable storage with monotonic counters to ensure freshness; and (iv) secure scheduling, as the untrusted OS controls exactly what processes run, thus it can preempt any enclave applications. However, SGX does not offer trusted I/O paths, as all peripherals are managed by the untrusted OS, so enclave applications must delegate communication with peripherals to the OS.

Compared to other existing TEEs, e.g., Arm TrustZone, Intel SGX has two disadvantages: (i) as the untrusted OS manages pages, Intel SGX is vulnerable to memory access pattern leakage, e.g., the OS can identify enclave’s memory accesses at page granularity, and other software can perform cache timing attacks; and (ii) as Intel SGX does not provide support for trusted I/O, applications must communicate with the untrusted host OS to perform this functionality. This requires applications to employ defences to protect against attacks from a malicious host OS.

However, despite these limitations, we believe Intel SGX offers a suitable and attractive solution for securing applications and protecting sensitive data in untrusted environments. This is because: (i) applications can be deployed with a small TCB, i.e., only the CPU and the application code and data need to be trusted; (ii) SGX offers hardware protections against attackers with physical access to the machine, e.g., protecting data in untrusted DRAM; (iii) SGX offers an efficient mechanism for performing secure boot and remote attestation: the application code and data can be measured without having to trust the BIOS or the OS; and (iv) SGX is widely available in commodity CPUs today. As such, we deem it a viable solution for achieving our goals.

For more information about Intel SGX, we refer the reader to a detailed report [CD16].

- **Iso-X** [EEO⁺14]: Iso-X is a flexible, software and hardware co-design that provides isolated

execution environments for security-sensitive code and data. Iso-X achieves isolated execution through secure compartments that use page tables to dynamically map and maintain memory pages. Execution compartments are protected from higher-privileged system software, allowing Iso-X to provide confidentiality and integrity guarantees even in the presence of an untrusted OS or hypervisor. Iso-X requires only six additional instruction set architecture (ISA) instructions for compartment management and two optional ISA instructions to support page swapping, resulting in an efficient hardware implementation.

In the Iso-X framework, application developers partition their applications into an untrusted partition (UP) and several trusted partitions (TPs). The UP contains untrusted code and data, including system software and libraries. The TPs contain security-sensitive code and data, along with memory regions to provide a secure execution environment. The Physical Page Compartment Membership Vector (CMV) tracks whether a memory page is assigned to a compartment and the Compartment Table (CT) records the compartment's characteristics, e.g., compartment size. In addition, each compartment is also assigned a Compartment Page Table (CPT), which maps virtual addresses to physical addresses. This avoids having to trust the OS's page tables.

Similar to Intel SGX, memory pages in Iso-X are only added to compartments after initialization. Compartments can be entered through untrusted code using dedicated entry points. Iso-X also supports remote attestation: compartments can be measured to produce secure compartment certificates that can be used to prove the compartment's integrity to an external verifier. This certificate is signed with the CPU's private key. Iso-X also provides a secure swapping mechanism to support memory management: before giving the OS access to a page, Iso-X hashes and encrypts the page before being swapped out, and decrypts and verifies the page when swapped back in by the OS.

Table 2.1 shows a summary of the TEE building blocks and functionalities offered by Iso-X. Iso-X provides: (i) secure boot through remote attestation, as compartment code and data can be measured and attested before being provisioned; (ii) inter-environment communication, as compartments also maintain a library for performing secure interaction with the rest of the system; and (iii) secure scheduling, as the untrusted OS controls exactly what processes run, thus it can preempt isolated compartments. However, Iso-X does not offer: (i) secure storage, as Iso-X provides no support for data sealing or versioning across many lifetimes of the application; and (ii) trusted I/O paths, as all peripherals, e.g., network and disk, are managed by the untrusted OS, so trusted code must delegate communication with these devices to the OS.

Compared to other TEEs, e.g., Intel SGX, Iso-X has several disadvantages that make it challenging to secure applications and protect data in untrusted environments: (i) Iso-X does not assume

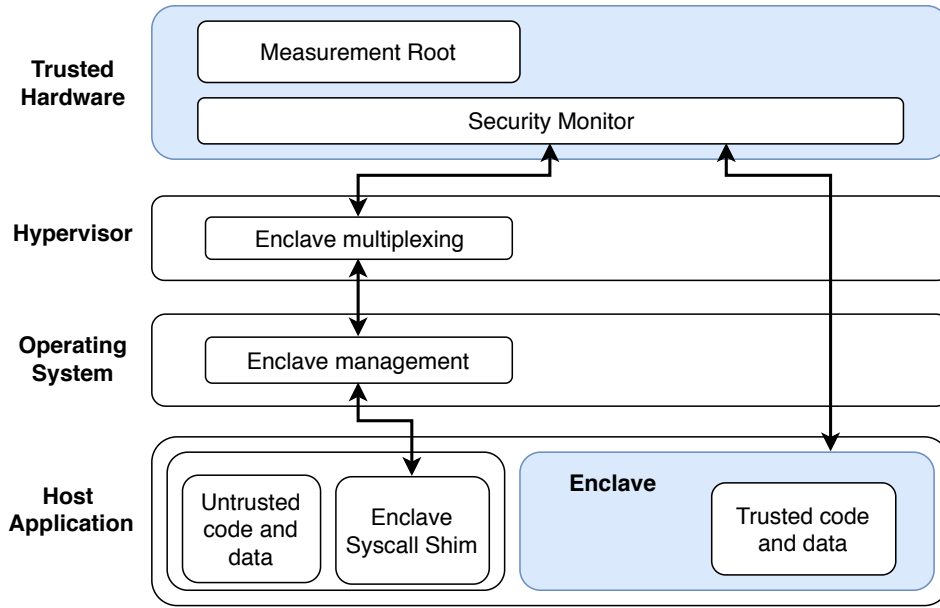


Figure 2.8: **The Sanctum [CLD16] architecture** (Here, we see the hardware and software layers required by Sanctum to execute an enclave-based host application.)

hardware attacks to be part of the threat model. As such, Iso-X provides little physical protection to snooping on the memory bus or probing the physical memory; (ii) applications using Iso-X must be partitioned in such a way that no data flow exists from a trusted to an untrusted partition. For this reason, Iso-X is more applicable to cloud environments, where applications are designed to execute in complete isolation; and (iii) Iso-X is vulnerable to cache timing attacks.

- **Sanctum [CLD16]:** Sanctum provides a software and hardware co-design that yields similar protection against software attacks as Intel SGX, but offers improved protection against memory access pattern leakage, such as page fault monitoring and cache timing attacks. At a high-level, Sanctum uses a cache partitioning scheme, where DRAM is split into disjoint regions, and each region uses distinct sets in the shared last-level cache (LLC). By allocating each DRAM region to exactly one isolated enclave, enclaves are protected in both DRAM and the LLC. To protect enclaves in other caches, e.g., the L1 and L2 caches, Sanctum performs cache flushes on context switches.

Figure 2.8 shows the software stack on a Sanctum machine. To provide security guarantees, Sanctum employs a trusted software component, called the security monitor. The security monitor is the first piece of firmware executed by the processor and is measured on boot by the measurement root. The security monitor allows Sanctum to provide a chain of trust for software attestation. The security monitor then provides an API for creating and destroying enclaves, and also manages transitions into and out of enclaves, as well as handles interrupts.

During execution, the security monitor is protected from compromised system software, by ex-

executing at the highest privilege level on the hardware. The monitor ensures that no DRAM region is ever accessible to two different enclaves. Each enclave manages its own page tables, and handles page faults. As such, the OS cannot learn virtual addresses that would cause a page fault in the enclave. Hardware modifications proposed by Sanctum work in combination with the security monitor to ensure that an enclave's page tables only reference memory inside the enclave's DRAM regions. As enclaves execute in a restricted environment, they need to exit to perform system calls and I/O.

Table 2.1 shows a summary of the TEE building blocks and functionalities offered by Sanctum. Sanctum provides: (i) secure boot through attestation, with the chain of trust starting at the security monitor; (ii) inter-environment communication, e.g., enclaves can communicate with each other directly via shared memory; and (iii) secure scheduling, as the untrusted OS controls exactly what processes run, thus it can preempt different enclaves. However, Sanctum does not offer: (i) secure storage, as it provides no cryptographic support for data sealing; and (ii) trusted I/O paths, as enclaves need to exit in order to perform system calls or execute I/O.

One of the unique benefits offered by Sanctum is that it provides an open-source design and implementation using a RISC-V processor. This allows the security community to audit the implementation and reason about Sanctum's security properties. However, in comparison to other TEEs, e.g., Intel SGX, the biggest limitation faced by Sanctum's design is that it does not offer protection from any physical attacks. For instance, attacks on DRAM cannot be prevented. As a result, this makes it difficult to secure applications and protect data in untrusted environments. However, the authors argue that Sanctum could be combined with physical protections proposed by other systems e.g., AEGIS (see Section 2.2.2), to provide protection against physical attacks. If such defences are employed, we believe Sanctum offers another attractive solution for securing applications and data in untrusted environments.

2.4 Open security challenges for TEEs

Despite the differences between the TEE implementations presented in Section 2.3.2, there are three fundamental security challenges that affect all TEEs when protecting applications in untrusted environments. These are: (i) minimizing the TCB (Section 1.4.1); (ii) securing TEE boundary interfaces (Section 1.4.2); and (iii) tolerating TEE failures and compromises (Section 1.4.3). In this section, we briefly discuss how each of these challenges affects the TEE implementations presented above. We further illustrate why these challenges must be overcome to provide confidentiality and integrity guarantees to applications and data:

(i) Minimising the TCB: TEE-based applications require sensitive code and data to be partitioned from the rest of the untrusted platform and executed within the confines of the TEE. For example: (i) ARM TrustZone requires executing security sensitive code and data within the confines of the secure world; (ii) Intel SGX and Sanctum require sensitive code and data to be executed in the enclave and identified on enclave creation; and (iii) Iso-X requires applications to be partitioned into trusted and untrusted compartments. Isolating security-sensitive code and data in this way forms the fundamental basis for how TEEs operate.

However, one of the challenges faced by this approach is that all isolated code is implicitly trusted to be free from software bugs [NZSS17]. As such, if a bug resides within the TEE, e.g., a buffer overflow or synchronization bug [WKPK16], an adversary can exploit this to violate security guarantees. For example, in the case of a buffer overflow, an adversary can exploit this to: (i) corrupt the integrity of the application by overwriting sensitive data in the TEE; and (ii) corrupt the confidentiality of sensitive data by forcing the TEE to write that data to untrusted memory. These types of attacks affect all TEEs regardless of the implementation. For example: (i) as TrustZone runs all trusted code in the secure world, a trusted application containing a buffer overflow can compromise the security of the entire world, including other trusted applications; and (ii) for Intel SGX and Sanctum, a buffer overflow in enclave code can compromise the security of the entire enclave.

As the number of bugs in an application is generally proportional to the amount of code that application contains [MB03], it is therefore essential to minimise the amount of code and data placed into a TEE for any given application; this reduces the TCB of the application. Moreover, it is essential to employ the principle of least privilege [SS75] when protecting applications using TEEs; otherwise vulnerabilities can compromise the entire application. This means: (i) for TrustZone, reducing the size and number of applications executing in the secure world; (ii) for Intel SGX and Sanctum, reducing the amount of code and data placed into the TEE; and (iii) for Iso-X, partitioning applications efficiently into trusted and untrusted compartments. Minimising the TCB better protects applications against software bugs when deployed in untrusted environments.

(ii) Securing TEE boundary interfaces: Applications executing within the confines of a TEE often require communication with the outside world. This is necessary, for example, when: (i) the application needs to perform I/O but the TEE does not provide trusted I/O paths, thus delegating this functionality to the untrusted OS, e.g., Intel SGX and Sanctum; and (ii) when trusted application code needs to communicate with untrusted application code via inter-environment communication (Section 2.3.1), e.g., TrustZone applications in the secure world communicating with applications in the normal world. As such, all TEEs must define boundary interfaces to communicate with the outside

world, i.e., interfaces that separate trusted code and data from untrusted code and data.

However, the problem with boundary interfaces is that they create an attack surface by which adversaries can target the TEE. Specifically, they expose two attack vectors: (i) adversaries can read and modify the parameters that cross the TEE boundary interface to cause application confidentiality and integrity violations. For example, in Intel SGX and Sanctum, an adversary can modify the results of system calls returned by the untrusted OS to hijack the control flow of the application [CS13b]; and (ii) the frequencies, sequences and parameters of interface calls expose side channels [HCP17, CZRZ17] to an adversary, i.e., expose application or data dependent information; this can violate application or data confidentiality in the TEE. For instance, the adversary can differentiate between applications or workloads based on system call behaviours in Intel SGX and Sanctum, e.g., observe the disk I/O access patterns.

As such, all applications that communicate with the outside world, must secure TEE boundary interfaces from attacks, regardless of the TEE implementation being employed. This means: (i) for TrustZone, securing all communication between applications in the secure and normal worlds; (ii) for Intel SGX and Sanctum, securing all communication between trusted code and data inside the enclave, and untrusted code and data outside the enclave; and (iii) for Iso-X, securing communication between trusted partitions and system software. This is required to protect the confidentiality and integrity of applications in untrusted environments.

(iii) Tolerating TEE failures and compromises: As highlighted by the discussions in Section 2.3.2, TEEs do not provide availability guarantees, e.g., all TEE implementations will fail by simply removing the power source from the device. Moreover, no single TEE implementation is free from security vulnerabilities or attacks: all TEE implementations have weaknesses. For example: (i) TrustZone, Iso-X and Sanctum offer limited protection against an attacker with physical access to the TEE. As such, they are vulnerable to a number of hardware attacks, e.g., attacks that read or modify sensitive data in DRAM or on the system bus; and (ii) Intel SGX enclaves are vulnerable to side channel attacks through memory access pattern leakage and cache attacks; these attacks can violate application confidentiality.

As such, TEEs are not silver bullets, and they cannot be assumed to provide complete security in all situations. As a result, to secure applications in untrusted environments, TEEs should only be deployed using a defence in depth approach, i.e., applications should employ TEEs as another layer of defence but assume adversaries can compromise the security guarantees of the TEE with enough effort and/or cost. As such, applications need to be fault tolerant against TEE failures and compromises.

2.5 Summary

In this chapter, we explored background on securing applications and protecting sensitive data in untrusted environments. We began with basic concepts and preliminaries for trusted computing, including: (i) a brief history and overview of trusted computing; (ii) an introduction to the Trusted Computing Group; and (iii) the five functionalities for trusted computing platforms. We then discussed several different trusted computing architectures that have been proposed over the past 20 years, including: the TPM [Pea02], XOM [LTM⁺00], AEGIS [SOD05], IBM 4765 [ABC⁺12], Intel TXT [Gra09], Bastion [CL10] and TyTan [BEMS⁺15].

We continued the chapter by taking a closer look at trusted execution environments (TEEs). We provided an informal definition for TEEs and identified the five core building blocks required to provide TEE functionalities. We then took a closer look at various TEEs that have been proposed over the past 20 years, and compared their designs, including: ARM TrustZone [ARM09], Intel SGX [MAB⁺13], Iso-X [EEO⁺14] and Sanctum [CLD16].

Finally, we highlighted three fundamental security challenges facing all applications secured with TEEs. These challenges are: (i) minimizing the TCB; (ii) securing TEE boundary interfaces; and (iii) tolerating TEE failures and compromises. We continue this thesis by exploring each of these challenges in the technical chapters that follow.

3

Minimising the Trusted Computing Base (TCB)

All application code and data isolated by the TEE is trusted to be free from software bugs. If this assumption is violated, an adversary can target these bugs to violate the confidentiality and integrity of the application. As such, developers must minimise the trusted computing base (TCB) of applications when deployed with TEEs (see Section 2.4). To achieve this, we propose automatic application partitioning for TEEs: using principled techniques to partition existing applications into security-sensitive and non-sensitive code and data.

We begin the chapter by providing an overview of our approach (Section 3.1). We then present relevant background on securing applications with TEEs (Section 3.2); this includes a discussion of our assumptions and threat model. Next, we present Glamdring: the first application partitioning framework that minimises the TCB of C applications for Intel SGX enclaves (Sections 3.3 and 3.4). We continue by offering a detailed security and performance evaluation of our approach (Section 3.5). Finally, we discuss related work to Glamdring (Section 3.6), highlight limitations of the approach (Section 3.7) and conclude the chapter (Section 3.8).

3.1 Overview

To minimise the TCB of applications, we propose automatic application partitioning for TEEs: protecting security-sensitive code and data by placing it into the TEE, and providing guarantees that the security-sensitive code cannot violate the confidentiality or integrity of security-sensitive data, even under attack. By placing only security-sensitive code and data into the TEE, and leaving the rest of the application outside the TEE, this approach: (i) avoids bloating the TCB with unnecessary code; and (ii) leverages privilege separation, as trusted code executes securely inside the TEE, while the rest of the application remains outside the protection domain.

To automatically partition existing applications for TEEs, we face three challenges: (i) we must identify which code and data in the application is security-sensitive, as this will be application and functionality specific; (ii) we must partition the application and define a secure TEE boundary interface between trusted and untrusted code and data; and (iii) we must transform the application to adopt the new partitioned architecture.

To overcome these challenges, we make three contributions: (i) we use source code annotations and static program analysis to identify security-sensitive code and data in the application. This prevents application developers from having to do it manually; (ii) we partition the application at the function granularity to produce a natural and well-defined TEE boundary interface. This interface employs encryption and integrity protection to protect sensitive data; and (iii) we use automatic source code generation to transform the application and harden the generated code against various attacks. This avoids having to rewrite the application by hand.

To illustrate our contributions, we present **Glamdring**, an automatic application partitioning framework for Intel SGX. Glamdring implements our contributions to partition existing C applications at the source code level for Intel SGX enclaves. We apply Glamdring to three different applications: the *Memcached* key/value store [Fit04], the *LibreSSL* library [Bec14], and the *Digital Bitbox* Bitcoin wallet [Shi16]. Our experiments show that Glamdring creates partitioned versions of these applications with TCBs that contain 22% – 40% of the lines of code of the applications. Despite their strong security guarantees, the partitioned applications execute with between 0.3x – 0.8x of the performance of the original versions. Glamdring demonstrates that our approach is not only feasible, but practical, as it can be successfully applied to several existing real-world applications with acceptable performance overheads.

3.2 Background: Securing applications with TEEs

In this section, we explore and compare the different approaches to securing applications with TEEs. We consider application code to be security-sensitive if it accesses sensitive data directly or impacts the confidentiality or integrity of data indirectly; we assume that only a subset of application code and data is security-sensitive. For example, in the Memcached [Fit04] store, assuming that key/value pairs are sensitive, functions that manipulate and handle key/value pairs are security-sensitive, while ones for network handling are not.

3.2.1 Threat model

When deploying applications in an untrusted environment, the adversary’s goal is to either disclose confidential data or violate data integrity. We consider a powerful and active adversary, such as a malicious system administrator, who has control over the hardware and software of the machine executing the application. The adversary may therefore: (i) access or modify any data in memory or disk; (ii) view or modify the application code; and (iii) modify the OS or other system software.

We consider denial-of-service (DoS) attacks outside the scope of this work. This is because an adversary with full control over the machine can simply decide not to run the application (see Section 2.3.2). Such attacks can be detected and potentially mitigated using various forms of replication [DM04]. We also consider side-channel attacks that exploit timing effects [WKPK16] or page faults [XCP15] of the TEE outside the scope of this work. As discussed in Section 2.3.2, some TEEs, such as Sanctum, avoid these attacks, while others, such as Intel SGX, are vulnerable to them. In the case that the TEE is vulnerable to these attacks, there exist dedicated mitigation strategies [BGNS06, CHB⁺15] that can be employed. As such, we exclude side-channels from our threat model.

3.2.2 Security with Intel SGX

While there are many different TEE architectures currently available, Intel SGX offers one of the most prominent and mature TEEs for commodity CPUs (see our comparison in Section 2.3.2). As such, in this chapter, we specifically employ Intel SGX TEEs to illustrate how the TCB of existing applications can be minimized through application partitioning. We note however that the high-level contributions presented here are TEE-agnostic and can be applied to different TEE architectures. We refer the reader to Section 2.3 for an in-depth discussion and comparison of other TEE technologies to which our contributions can be applied.

The Intel SGX SDK [Int16a] allows developers to create enclave libraries which can be loaded into an enclave and executed by a CPU with SGX support. A developer defines the interface between the enclave code and other, untrusted application code: (i) a call into the enclave is referred to as an *enclave entry call* (*ecall*). For each defined ecall, the SDK adds instructions to marshal parameters outside, unmarshal the parameters inside the enclave and execute the function; and (ii) *outside calls* (*ocalls*) allow enclave functions to call untrusted functions outside. For each defined ocall, the SDK adds instructions to leave the enclave, unmarshal the parameters, call the function, re-enter the enclave and continue execution of the caller’s code. Any ecalls and ocalls introduce a performance overhead because the hardware must perform certain actions to maintain the security guarantees of SGX. Enclave code must also verify the integrity of accessed data, such as parameters of ecalls, return values of ocalls, and data read from untrusted memory.

The Intel SGX SDK also supports remote attestation: users can remotely attest the integrity of the enclave memory and the presence of a genuine SGX-capable CPU. For this, the enclave can request a measurement report from the CPU that contains a digest over the current enclave content. The report is then signed by a platform-provided quoting enclave which indicates the trustworthiness of the platform, before it is sent to the user.

3.2.3 Design space for SGX applications

We now explore the design space for securing applications and data using Intel SGX TEEs. We discuss the trade-offs of different application designs with respect to: (i) the size of the TCB; (ii) the complexity of the enclave boundary interface; (iii) the development effort required by developers; and (iv) the generality of the approach to different applications.

With Intel SGX, the TCB consists of all code and data loaded into the enclave, as well the hardware used to implement the enclave itself. Following the principle of least privilege [SS75], only the parts of an application that require access to sensitive data should be executed within an enclave. As studies have shown [SYTP85, MB03], the number of software bugs, and thus potential security vulnerabilities, increases proportionally with the code size. This makes it important to minimize the size of the TCB.

The complexity of the enclave boundary interface, i.e., the interface between trusted code inside the enclave and untrusted code outside the enclave, impacts the security of enclave code and data. For example, security-sensitive application code inside the enclave must still interact with the untrusted environment, such as the OS, to perform I/O (see Section 2.3.2). Return values from system calls must therefore be checked to protect against *Iago attacks* [CS13b], where an attacker compromises the OS

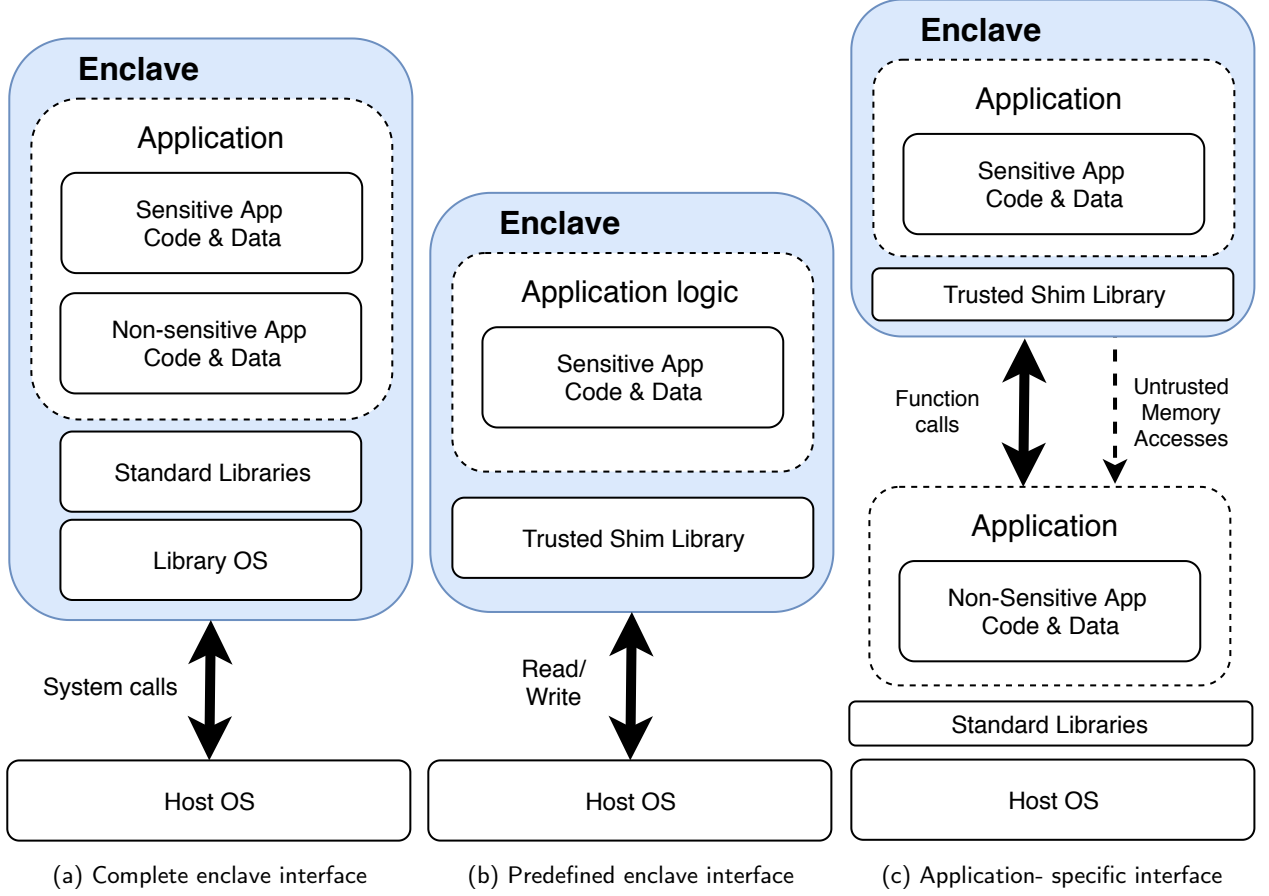


Figure 3.1: **The design alternatives for the use of enclaves** (These designs include: (a) a complete enclave interface, where the entire application is executed in the enclave; (b) a predefined enclave interface, where the application conforms to a specific enclave interface; and (c) an application-specific interface, where the application is partitioned into trusted and untrusted application code and data, with a corresponding interface between the two.)

kernel and manipulates the return values of system calls to force enclave code to disclose or modify sensitive enclave data. Creating a principled enclave interface thus makes it easier to reason about the feasibility and potential for particular attacks.

Other important factors that determine the adoption of a given approach for securing applications and data with Intel SGX enclaves are: (i) the development effort required in order to develop secure applications; and (ii) whether the approach is generally applicable to any and all applications.

Figure 3.1 shows three possible design alternatives for protecting applications using Intel SGX enclaves. We discuss each of these three designs in detail below:

Complete enclave boundary interface. As shown in Figure 3.1a, the approach adopted by other existing systems such as Haven [BPH15], SCONE [ATG⁺16] and Graphene [TAB⁺14, Osc17] provides isolation at a coarse granularity by executing a *complete* application inside an enclave. Haven runs

unmodified Windows applications using the Drawbridge library OS [PBWH⁺11a]; Graphene uses a library OS in the enclave to run Linux applications; and SCONE places a modified version of the standard C library in the enclave for supporting recompiled Linux applications. For these systems, both security-sensitive and insensitive application code and data reside within the enclave, increasing the size of the TCB.

The enclave boundary interface of these systems supports a complete set of system/hyper calls, which cannot be handled inside the enclave. The interface is application-independent, but its complexity (in terms of number of distinct calls and their input parameters) depends on the adopted system abstraction. The required system support within the enclave further adds to the TCB size.

Predefined enclave boundary interface. Figure 3.1b shows a different approach in which applications must adhere to a *predefined* restricted enclave boundary interface [SCL⁺16, SRSV15, SCF⁺15]. For example, VC3 [SCF⁺15] protects map/reduce jobs using enclaves and forces map/reduce tasks to interact with the untrusted environment only through a specified interface. In these systems, the enclave contains a small trusted shim library, resulting in a smaller TCB compared to the previous approach offering a complete enclave interface, i.e., Figure 3.1a.

This approach results in a minimal enclave boundary interface—VC3’s interface consists of only two calls, one to read encrypted key/value pairs and another to write them as the job output. This limited interaction of the enclave with the outside simplifies protection: it is possible to add dynamic checks that enforce security invariants [SCL⁺16], e.g., preventing enclave code from accessing untrusted memory except through the enclave interface. However, the security benefits of this approach are offset by its limited applicability. Given the predefined enclave interface, the approach can only be used with applications that interact with the untrusted environment in specific and constrained ways, such as map/reduce tasks, that do not issue arbitrary system calls. To support this approach, most applications have to be rewritten.

Application-specific enclave boundary interface. Our goal in this work is to explore another design point for SGX-based applications. We exploit the fact that, for many applications, only a subset of code handles sensitive data, while other code is not security-sensitive and does not need protection [SPHH06, ST06b, BWL⁺16]. As shown in Figure 3.1c, this makes it possible to partition the application to reduce the TCB size, leaving code and data that is not security-sensitive outside the enclave.

Existing work has shown that application partitioning can be done by hand so that complex applications can exploit enclaves [BWL⁺16, PPFF16]. Instead, we want to explore the hypothesis that it is feasible

to use *principled* techniques, e.g., program analysis and code generation, to automatically partition existing applications for secure enclaves, and provide *security guarantees* about the enclave code and its boundary interface to the untrusted environment. This avoids having to partition and protect existing applications by hand.

With this approach, the enclave boundary interface now becomes *application-specific*: a set of ecalls and ocalls is required between trusted and untrusted application code. In contrast to a complete enclave boundary interface as in Figure 3.1a, fewer system calls need ocalls because application code that is placed outside the enclave can issue system calls directly.

Since application data now also exists outside the enclave, enclave code must be allowed to access untrusted memory. This means that it is no longer possible to prohibit all memory accesses, as with the predefined enclave boundary interface [SCL⁺16] in Figure 3.1b. Instead, it is important to give security guarantees that, despite the richer application-specific enclave boundary interface, the untrusted environment cannot disclose sensitive enclave data or compromise its integrity.

3.3 Glamdring design

To target an application-specific enclave boundary interface for existing applications, i.e., achieve the interface shown in Figure 3.1c from Section 3.2.3, our partitioning approach must satisfy four requirements. First, it must protect the *confidentiality* of sensitive *input data* and the *integrity* of sensitive *output data* (*R1*). This ensures an attacker cannot interfere with application execution or compromise security-sensitive data. Second, it must apply the principle of least privilege, minimising the code that can access sensitive data (*R2*). This reduces the TCB of the application and protects against software bugs in the TEE. Third, it must automate changes to the application code (*R3*). This avoids developers having to modify the application by hand; a process which is tedious and error-prone. Fourth, it must impose an acceptable performance overhead (*R4*), otherwise the approach will not be feasible to protect applications in the real world.

We present Glamdring, a framework for partitioning existing C applications to execute security-sensitive code in an Intel SGX enclave. Glamdring targets requirements *R1-R4* defined above. To partition applications using Glamdring, application developers first annotate input and output variables in the source code that are security-sensitive, i.e., variables containing sensitive data that require confidentiality and integrity. Glamdring then employs *static program analysis* [Ern03] to identify security-sensitive application code and data based on these annotations. Static program analysis is workload-independent and makes conservative decisions about data and control flow dependencies. As

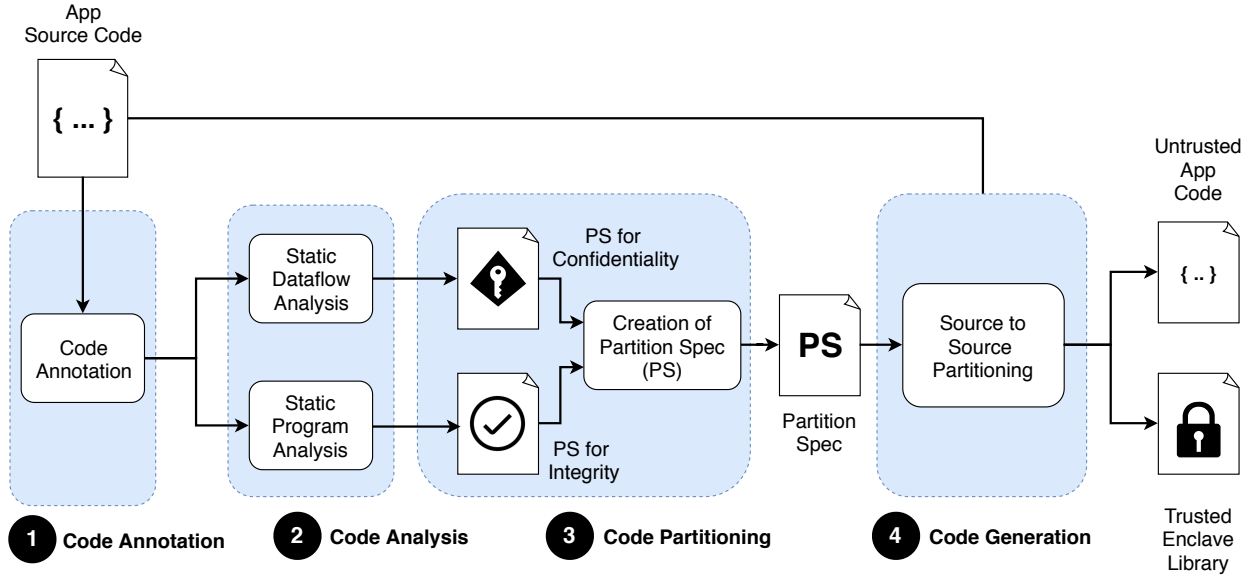


Figure 3.2: **Overview of the Glamdring framework** (Here, we see: (i) the application source code as an input; (ii) the four phases of execution, from code annotation to automatic code generation; and (iii) the resulting output: a set of trusted and untrusted application code and data files.)

such, Glamdring protects security-sensitive code and data regardless of the application input data, or workload at runtime. This is in contrast to dynamic program analysis [KL88], which is workload-dependent.

In this section, we present the design of Glamdring and discuss how Glamdring achieves its goals.

3.3.1 Design overview

To achieve requirements $R1-R4$, Glamdring operates in four phases. These can be seen in Figure 3.2. We discuss each of these phases in detail below:

① Code annotation phase: Glamdring must know which application data is security-sensitive in order to protect it (R1). To achieve this, the developer provides information about the *sources* (inputs) and *sinks* (outputs) of security-sensitive data by annotating variables whose values must be protected in terms of confidentiality and integrity (see Section 3.3.2).

② Code analysis phase: Based on the annotated source code, Glamdring identifies a subset of code that is security-sensitive (R2). It uses automatic static program analysis (R3) to find control and data dependencies on security-sensitive data. Glamdring thus obtains the minimal set of statements that either handle confidential data or affects its integrity (R1/R2) (see Section 3.3.3).

③ Code partitioning phase: Next Glamdring creates a *partition specification* (PS) that defines

which parts of the code must be protected by the enclave. The PS enumerates the functions, memory allocations and global variables that are security-sensitive based on the program analysis. This defines the enclave boundary interface of the partitioned application, which includes ecalls, ocalls, and direct accesses to untrusted memory (see Section 3.3.4).

④ **Code generation phase:** Finally, Glamdring uses a source-to-source compiler that, based on the PS, partitions the code into a secure enclave library and untrusted code. The enclave boundary interface is hardened with runtime checks that enforce invariants on the program state (see Section 3.4).

3.3.2 Code annotation phase

The security-sensitive data that must be protected by the TEE is application-dependent. To identify it, a developer must therefore annotate the source code. Glamdring relies on the fact that security-sensitive data is protected when it is exchanged between a trusted client and the application. For example, if data is received via the network, Glamdring requires the client to encrypt and sign the data. Both the client and the enclave code use symmetric AES-GCM encryption [Dwo07], where the key is established upon enclave creation.

When encrypted security-sensitive data reaches the application through a *source*, such as an I/O channel, or leaves the application through a *sink*, a developer must annotate the corresponding variable using a compiler pragma. The annotation `sensitive-source` identifies a variable at a given source code location where security-sensitive data enters the application; analogously, the annotation `sensitive-sink` indicates a variable at which security-sensitive data leaves the application.

For example, for Memcached we assume that the security-sensitive data is the type of command submitted by the client (`get/set`) and its associated key/value data. This data is encrypted and signed by the trusted client when sent to the application. Using Glamdring, Memcached then requires two annotations, which can be seen in Listing 3.1.

An obvious location for the `sensitive-source` annotation might be the `socket read()` call from which a client request is received. However, this would be unnecessarily conservative because it would denote all network data as security-sensitive (and thus encrypted). Instead, the annotation in line 1 marks the content of the parameter `command`, which holds the request command and data, as security-sensitive. The `sensitive-sink` annotation in line 12 specifies that the output buffer for the client response also contains security-sensitive data.

```

1 #pragma glamdring sensitive--source(command)
2 static void process_command(conn *c, char *command) {
3     token_t tokens[MAX_TOK];
4     size_t ntokens;
5     ...
6     ntokens = tokenize_command(command, tokens, MAX_TOK);
7     ...
8     process_update_command(c, tokens, ntokens, comm, false);
9     ...
10 }
11
12 #pragma glamdring sensitive--sink(buf)
13 static int add_iov(conn *c, void *buf, int len) {
14     ...
15     m = &c->msglist[c->msgused - 1];
16     m->msg_iov[m->msg_iovlen].iov_base = (void *) buf;
17     ...
18 }

```

Listing 3.1: The pragma annotations added to the Memcached source code in order to identify security-sensitive sources and sinks (as required by Glamdring).

3.3.3 Code analysis phase

Next, the code analysis phase identifies all security-sensitive statements in the program that have dependencies on the set of all annotated statements S_A . This combines: (i) for *confidentiality*, the set of all statements that are influenced by the ones in S_A ; and (ii) for *integrity*, the set of all statements that influence the ones in S_A .

Glamdring uses static program analysis to identify all security-sensitive statements. Static analysis is workload-independent and hence makes conservative decisions about dependencies. To ensure that an attacker cannot violate the invariants that static analysis infers from the untrusted code, Glamdring adds runtime checks during code generation (see Section 3.4).

Glamdring’s analysis uses a *program dependence graph* (PDG) [FOW87], referred to as P , in which vertices represent statements, and edges are both data and control dependencies between statements. PDGs are effective representations for program slicing [OO84, HRB88]. Using P , Glamdring finds the set of all security-sensitive statements as follows:

1. **Static dataflow analysis for confidentiality [RHS95].** Given S_A and P , Glamdring uses graph-reachability to find a subgraph P_c of P that contains all statements with a transitive control/data dependence on statements in S_A (i.e., vertices reachable from statements in S_A via edges in P).

For statements in S_A that are annotated as a **sensitive-sink**, Glamdring encrypts/signs the data before the statement inside the enclave, making it unnecessary to perform dataflow analysis

from these statements.

2. **Static backward slicing for integrity [Wei81].** Given S_A and P , Glamdring uses static backward slicing to find a subgraph P_i with all statements in P on which statements in S_A have a control/data dependence (i.e., all vertices from which statements in S_A are reachable via P).

For these statements in S_A that are annotated as **sensitive-source**, Glamdring employs client-side encryption of the data, making it unnecessary to perform backwards slicing from these statements.

Finally, the set of all security-sensitive statements S_s is obtained by combining P_c and P_i .

3.3.4 Code partitioning phase

Although S_s enumerates security-sensitive statements, Glamdring partitions the application at the granularity of functions rather than statements. There are two reasons for this: (i) function-granularity partitioning reduces the amount of code changes that need to occur to the application (R3) (see Section 3.4); and (ii) function-granularity partitioning produces a natural TEE boundary interface, as applications already maintain a well-defined function interface with clear control flows [All70], clear inputs, i.e., function parameters, and clear outputs, i.e., function return values.

Glamdring produces a *partition specification* (PS) from S_s with the set of security-sensitive functions, memory allocations and global variables to protect:

1. *functions*: the PS includes all functions whose definitions contain *at least* one statement in S_s ;
2. *memory allocations*: the PS must identify allocated memory for security-sensitive data. Statements in S_s with calls to `malloc` (or similar) are enumerated in the PS, and these allocations are placed inside the enclave;
3. *global variables*: the PS lists all global variables accessed in statements in S_s , and these are allocated inside the enclave. Special accessor ecalls (with checks) are provided to the untrusted code to access these globals if needed. The PS specifies if the global was part of P_c or P_i or both, which determines what type of access (read, write or none) the outside code has.

Enclave boundary relocation (EBR). Glamdring’s code analysis phase produces a lower bound on the code that must be inside the enclave to guarantee security. In practice, however, a partitioning may prove costly in terms of performance if program execution must frequently cross the enclave

boundary interface, i.e., as `ecalls` and `ocalls` incur significant performance overheads in Intel SGX (see Section 2.3.2).

To overcome this, Glamdring improves the performance of the partitioned application by moving additional functions into the enclave in order to reduce the number of enclave crossings, i.e., the number of `ocalls` executed at runtime. To achieve this, Glamdring uses a representative workload and the output of the `gcov` runtime profiling tool [Fre17]: Glamdring assigns a cost to each enclave boundary function according to the number of invocations. Up to a configurable threshold, Glamdring adds functions to the enclave. Adding extra functions to the enclave cannot violate the security guarantees of Glamdring, but it does increase the TCB size. As such, there is a trade-off between the number of enclave crossings and the TCB of the resulting partitioning.

3.3.5 Security discussion

The security guarantees of Glamdring rely on several different aspects of the approach: (i) the soundness of the static analysis; and (ii) the modelling of external library calls whose source code is unavailable. We discuss each of these in more detail below:

Static analysis. To be tractable, static analysis infers invariants on program state based on the source code. These invariants must also hold at runtime, even when the untrusted code is under control of an attacker. Glamdring ensures this by adding runtime invariant checks to the enclave boundary. We discuss these invariant checks in Section 3.4.2.

Static pointer analysis is undecidable for C programs [Ram94] and thus fundamentally imprecise [HL11, GKM16]. While static analysis may introduce false positives, this does not compromise soundness: a false positive will simply assign more functions to the enclave than necessary; thus increasing the size of the TCB, but not violating security guarantees. We note that because static analysis is conservative, it will never produce false negatives, thus Glamdring will never exclude security-sensitive functions from the enclave.

Modelling external library calls. Static analysis must model the behaviour of all invoked functions to compute all code and data dependencies. However, some program functions may not be available during the code analysis phase, e.g., functions in external libraries with unavailable definitions. For these functions, Glamdring takes a conservative approach, making all output parameters of the undefined function dependent on all input parameters. As such, this may produce false positives in the static analysis, thus assigning more code to the enclave than necessary. However, this does not compromise the soundness of the approach. We note that more precise models can consider actual function

behaviour to specify dependencies [GH12, BFM⁺08].

3.4 Code generation and hardening

With the partition specification (PS) produced by the first three phases in Glamdring (see Figure 3.2), the fourth and final phase is code generation. Code generation produces a source-level partitioning of the application based on the PS (§3.4.1). In addition, it *hardens* the enclave boundary interface against malicious input, ensuring that the enclave upholds the confidentiality and integrity guarantees for sensitive data (§3.4.2). This results in a set of enclave and non-enclave source files, along with an enclave specification, which can be compiled using the Intel SGX SDK.

3.4.1 Code transformation

Code transformation of the existing application must: (i) handle calls into and out of the enclave; and (ii) change the allocation, scope and lifetime of variables and functions in the generated enclave and non-enclave versions of the code.

Glamdring provides a *code generator* that relies on the LLVM/Clang compiler toolchain [LA04, Cla16] to rewrite the preprocessed C source code. It uses the Clang libraries to parse source code into an abstract syntax tree (AST), and traverses the AST to analyse and modify the source code. In addition to the enclave and non-enclave source files, it produces an interface specification in the *enclave definition language* (EDL) required by the Intel SDK [Int16a]. The code generation then proceeds in three steps:

1. **Moving function definitions into the enclave.** For each source file, the code generator creates an enclave and a non-enclave version, which contain a copy of the original preprocessed input file. From the enclave version, it removes all functions not listed in the PS and from the non-enclave version it removes all listed enclave functions.
2. **Generating ecalls and ocalls.** Based on the set of enclave functions, the code generator identifies the ecalls and ocalls that are part of the enclave boundary interface. It traverses the direct call expressions in each function: (i) if the caller is an untrusted function and the callee is an enclave function, the callee is made an ecall; or (ii) if the caller is an enclave function and the callee is an untrusted function, the callee is made an ocall.¹

¹Pointers passed outside the enclave are only deep-copied if data in enclave-allocated memory needs to be declassified—the programmer needs to implement this manually.

```

1  /* Initialised to func_A and func_B outside */
2  int (*addrof_func_A) (int);
3  int (*addrof_func_B) (int);
4
5  int jump_to_func(int (*fptr) (int), int x) {
6      if (fptr == addrof_func_A)    return ocall_func_A(x);
7      else if (fptr == addrof_func_B) return ocall_func_B(x);
8  }
9
10 int ecall_enclave_func(int (*fptr) (int), int y) {
11     return jump_to_func(fptr, y);
12 }

```

Listing 3.2: The trampoline code generated by Glamdring to handle ecalls and ocalls when jumping to functions.

Adding stubs for encryption/decryption. As mentioned in Section 3.3.2, the security-sensitive data received from (and returned to) clients is encrypted (and integrity-protected) using a shared AES-GCM key. The code generator adds code to: (i) decrypt security-sensitive data entering the enclave at locations annotated as `sensitive-source`, and (ii) encrypt the security-sensitive data leaving the enclave at locations annotated as `sensitive-sink`. The application client must be modified to handle the corresponding encryption/decryption operations.

Handling C library functions. Calls to C library functions are handled separately. A subset is supported by the Intel SDK inside the enclave and is handled in a polymorphic manner: the enclave and non-enclave code call their respective versions.² For unsupported library functions, e.g., those making system calls, the code generator creates ocalls to the corresponding library function linked to the outside code. These ocalls violate the enclave boundary identified through static analysis and hence will be hardened with runtime checks (see Section 3.4.2).

Handling function pointers as interface arguments. Function pointer arguments to ecalls and ocalls are special cases because the target function may not exist at the point of invocation of the function pointer. For example, if an ecall passes a function pointer targeting a function on the outside, the program will fail when the enclave attempts to call that function pointer directly. Glamdring employs a static function pointer analysis [ZZ15] to identify the possible target functions of function pointer arguments passed to ecalls and ocalls. The code generator then creates ecalls or ocalls for the target functions and uses a trampoline to jump to the correct one, as shown in the `jump_to_func` function in Listing 3.2.

3. Handling memory allocation. The code generator also uses the PS to decide which memory allocations must be placed inside the enclave. For the memory allocations listed in the PS,

²Linked calls to the few stateful C library functions (e.g., `strtok`) typically do not span multiple functions, making it unlikely that such calls get partitioned into different regions.

nothing needs to be done because a `malloc` call inside the enclave allocates memory inside; for other memory allocations, a function must allocate memory outside, and the `malloc` is replaced by an ocall to the outside. This arises when placing non-sensitive code into the enclave when: (i) partitioning at the function instead of statement level; and (ii) moving functions into the enclave using EBR (see Section 3.3.4). We note that because the memory allocation was not included in the PS, it does not contain sensitive data, and thus allocating the memory outside the enclave (via an ocall) does not impact the security of the approach.

3.4.2 Code hardening

Next we analyse the attack surface of the enclave boundary interface and describe the protection techniques of the code generation phase against attacks (this is R1 from Section 3.3).

Boundary interface attacks. The security of the enclave code depends on the inputs that it receives from the enclave boundary interface. However, an attacker may manipulate the parameters to ecalls, the results of ocalls, and accesses to global variables.

Secure by construction: The enclave code is, by construction, immune to input manipulation attacks. As long as Glamdring’s static analysis is sound, it transitively identifies all code that can affect the confidentiality and integrity of security-sensitive data annotated by the developer, placing it inside the enclave (see §3.3.3).

However, static analysis infers invariants about the possible values of program variables at different program points, permitting it to prune unfeasible program paths from analysis. The soundness of the static analysis therefore depends on these invariants holding at runtime. Any invariant that relates to untrusted code or data outside the enclave may be compromised by an attacker. The code snippet in Listing 3.3 gives an example of this, a debug option that is deactivated in the source code.

Static analysis infers that the value of `dump_flag` cannot be 1, making it impossible to take the branch that does not include the `declassify()` call. Since the value of `dump_flag` does not affect the control flow leading to sensitive data release, Glamdring would allocate it outside the enclave. An attacker could set `dump_flag` to any value at runtime, including 1, to cause data disclosure.

Runtime invariant checks. To prevent such attacks, Glamdring enforces the invariants assumed by the static analysis at runtime. It does this by extracting invariants from the analysis phase and adding them as runtime checks in the code generation phase. Glamdring applies checks on global variables and parameters passed into and out of ecalls and ocalls. In the above example, Glamdring adds a

```

1  /* Outside code */
2  int dump_flag = 0; // Can be modified by attacker.
3
4  /* Enclave code */
5  int ecall_enclave_func(int dump_flag) {
6      char* dump_data = malloc(...);
7      if (dump_flag == 1)
8          memcpy(dump_data, sensitive_data);
9      else
10         memcpy(dump_data, declassify(sensitive_data));
11     write_to_untrusted(dump_data);
12 }

```

Listing 3.3: An example of an invariant that relates to code or data that resides outside the enclave and that may be compromised by an attacker at runtime.

check `assert(dump_flag == 0)`.

Checks are also applied to *pointers*. The static analysis infers the subset of `malloc` calls that may allocate memory pointed to by each pointer. Glamdring distinguishes between two cases: (i) the analysis infers that a pointer may only point to untrusted memory. A runtime check upholds this and any other invariants on pointer aliasing; or (ii) the pointer may point to enclave memory. Here, Glamdring’s invariant checks prevent pointer-swapping attacks (i.e., a trusted pointer being replaced by another trusted pointer): Glamdring instruments the `malloc` calls inferred for that pointer inside the enclave, storing the addresses and sizes of allocated memory. When a trusted pointer is passed to the enclave via an `ecall`, it is checked to ensure that it points to a memory region allocated by one of the statically inferred `malloc` calls for that pointer. This upholds the results of the static pointer analysis at runtime with enclave checks.

For checks on *global variables* allocated outside, before each use, Glamdring copies the value inside and applies the check to the local copy.

Enclave call ordering attacks. By construction, Glamdring prevents an attacker from subverting the security guarantees by changing the *order* in which `ecalls` are invoked. The transitivity of static analysis ensures that all functions that have a data/control flow dependence relationship (in either direction) with security-sensitive data are placed inside the enclave. Therefore, any change in the ordering of `ecalls` cannot affect the security guarantees as long as the statically-inferred enclave boundary is enforced. The EBR operation does change this boundary, but only by placing extra functions inside, and therefore cannot violate the security guarantees.

Iago attacks. For applications that use C library functions unavailable in the Intel SGX SDK, Glamdring adds `ocalls`. The arguments to such `ocalls` may expose security-sensitive data or their results may cause integrity violations, leading to Iago attacks [CS13b]. For these functions, Glamdring

enforces statically inferred invariants on the return values at runtime. Further protection, such as encryption and integrity protection, could be done, similar to I/O shields in SCONE [ATG⁺16].

Replay attacks. An attacker may tamper with the program state assumed by the enclave by replaying previously issued ecalls. Glamdring guarantees the freshness of encrypted sensitive data that is passed to ecalls. The client affixes a *freshness* counter to security-sensitive data as part of its encryption (see Section 3.3.4). The enclave stores the latest freshness counter for each data item, and validates freshness at ecalls. After an enclave restart, the freshness counters must be restored to their latest values [SP16a].

3.5 Evaluation

We want to evaluate the security and performance of our approach. For this, we ask two questions: (i) how effectively does Glamdring reduce the size of the TCB for various applications? and (ii) how does Glamdring affect the performance of the partitioned application?

To answer these questions, we apply Glamdring to three real-world applications: (i) the Memcached key/value store [Fit04]; (ii) the LibreSSL library [Bec14]; and (iii) the Digital Bitbox bitcoin wallet [Shi16]. Section 3.5.1 describes the security objectives, the source code annotations and the resulting partitioning and its interface. The TCB in lines of code (LOC) identified by Glamdring varies between 22% and 40%, and the size of the interface between 41–171 ecalls and 51–615 ocalls for the three applications. Section 3.5.2 presents performance results on SGX hardware: the partitioned applications execute with $0.3\times$ – $0.8\times$ of the native performance.

Glamdring implementation. Glamdring uses the Frama-C Aluminium [Fra16a] static analysis framework, with the “Impact Analysis” [Fra16b] and “Slicing” [Fra16c] plug-ins and CodeSurfer 3.0.0 [Gra16]. The Glamdring code generator uses LLVM/Clang 3.9 and has approximately 5,000 LOC.

Memcached [Fit04] is a distributed key/value store. It supports several operations: `set(k,v)`, `get(k)`, `delete(k)`, and `increment/decrement(k,i)`. We apply Glamdring to Memcached 1.4.25 which includes libevent 1.4.14 [PM03], an asynchronous event library for socket polling and network I/O. As Table 3.1 shows, Memcached has 31,100 LOC and 655 functions.

LibreSSL [Bec14] is a fork of the OpenSSL cryptographic library [CEH⁺02], with the goal to provide a simpler and more secure implementation. We apply Glamdring to LibreSSL 2.4.2 to secure its functionality when serving as a certificate authority (CA). As Table 3.1 shows, LibreSSL has 176,600 LOC

Application	LOC	Functions	Global variables	Security-sensitive LOC	Security-sensitive functions	Security-sensitive global variables	Ecalls	Ocalls C lib.	App. ocalls	Ecall crossings per application request	Ocall crossings per application request
Memcached	31,100	655	119	12,474 (40%)	273 (42%)	81 (68%)	41	146	82	64	
Memcached w/o EBR	31,100	655	119		272 (42%)	81 (68%)	41	147	82	65	
Memcached v1.4.25	13,800	247	84		215 (87%)	72 (85%)					
libevent v1.4.14	17,300	408	35		57 (14%)	9 (26%)					
LibreSSL	176,600	5,508	1,034	38,291 (22%)	918 (17%)	163 (16%)	171	613	23	312	
LibreSSL w/o EBR	176,600	5,508	1,034		916 (17%)	163 (16%)	171	615	23	314	
libcrypto v2.4.2	124,800	4,550	833		654 (14%)	91 (11%)					
libssl v2.4.2	24,300	628	42		83 (13%)	7 (17%)					
apps v2.4.2	27,500	330	159		179 (54%)	65 (41%)					
Digital Bitbox	23,300	873	105	8,743 (38%)	365 (42%)	55 (52%)	114	51	20	31	
Digital Bitbox w/o EBR	23,300	873	105		361 (42%)	55 (52%)	118	55	20	35	
Digital Bitbox v2.0.0	7,900	382	81		195 (51%)	48 (60%)					
Secp256k1 v1.0.0	12,900	112	9		52 (46%)	1 (11%)					
Yajl v2.1.0	2,500	379	15		114 (30%)	6 (40%)					
				seed	sign	random	seed	sign	random	seed	sign
				23	4	7	4	0	0	0	
				3,252	6,937	672	59	12	11		

Table 3.1: **TCB sizes, enclave interfaces and enclave crossings for Glandring applications** (Application requests are: (i) get, set for Memcached; (ii) sign for LibreSSL; and (iii) seed, sign, random for Digital Bitbox).

and 5,508 functions, which are divided into three libraries, `libcrypto`, `libssl` and `apps/openssl`. We compile LibreSSL without inline assembly because our static analysis does not support it.

Digital Bitbox [Shi16] is a bitcoin wallet designed for high-security USB microcontrollers. It supports: (i) hierarchical deterministic key generation; (ii) transaction signing; and (iii) encrypted communication. We apply Glamdring to Digital Bitbox 2.0.0 with Secp256k1 1.0.0, a cryptographic library, and Yajl 2.1.0, a JSON library. As Table 3.1 shows, Digital Bitbox has 23,300 LOC and 873 functions.

3.5.1 Security evaluation

We evaluate the security of the three partitioned application in terms of: (i) the *TCB size*, i.e., the total LOC, number of global variables and number of functions placed inside the enclave; and (ii) the exposed *enclave boundary interface*, i.e., the number of ocalls and ecalls exposed by the enclave. We look at each application in turn:

Memcached distributed key/value store [Fit04]

Security objectives: We want to protect the integrity and confidentiality of all key/value pairs in an untrusted Memcached deployment, preventing an attacker from reading or modifying the stored key/value data. For this, we use the source code annotations described in Section 3.3.2.

Security-sensitive code: Table 3.1 shows that Glamdring places 40% of LOC, 42% of functions and 68% of global variables of Memcached inside the enclave. EBR moves a single additional function into the enclave, reducing the ocall crossings by an order of magnitude for `get` and `set` operations. We conclude that a large portion of the Memcached codebase (without libevent) is security-sensitive, as 87% of its functions and 85% of its global variables are assigned to the enclave.

Partitioned architecture: Figure 3.3 shows the Memcached architecture produced by Glamdring. Glamdring places the following functionality inside the enclave: (i) binary/ASCII protocol handling functions; (ii) slab and cache memory management functions that manipulate the data structures responsible for the internal storage of key/value pairs; (iii) the hash functions over key/value pairs; and (iv) some request processing and utility functions. The functionality placed outside includes: (i) thread initialization and registration functions; (ii) libevent functions for socket polling and network I/O; and (iii) signal handlers and string utility functions.

Enclave interface: Table 3.1 shows that the enclave interface has 41 ecalls and 146 ocalls. Out of these, 82 ocalls are to C library functions unavailable inside the enclave; 64 ocalls are to application functions.

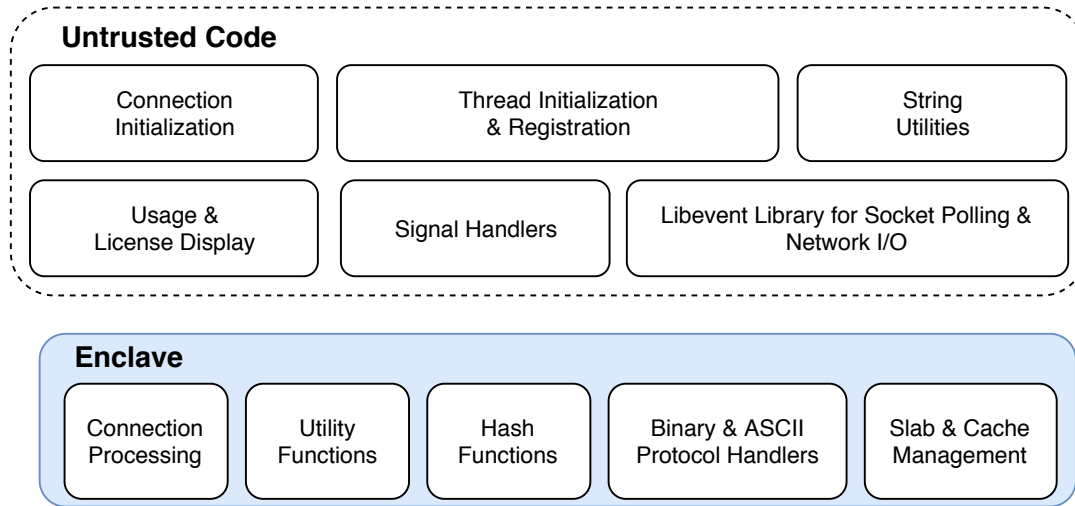


Figure 3.3: **The partitioned Memcached architecture** (Here, we see the separation between security-sensitive code and data placed inside the enclave for Memcached, and non-sensitive code and data outside the enclave.)

```

1 int ca_main(int argc, char** argv) {
2     ...
3     #pragma glamdring sensitive--source(pkey)
4     pkey = load_key(bio, keyfile, keyform, 0, key, "...");
5     ...
6 }

```

Listing 3.4: The pragma annotations added to the LibreSSL source code in order to identify the private key as a security-sensitive source.

To protect the security-sensitive data between the Memcached client and the enclave interface, Glamdring encrypts the following parameters at the client for each request: (i) the **operation** to perform; (ii) the **key**; and (iii) the **value**. The **keys**, **values** and the request **outcome** are encrypted in the client response.

LibreSSL cryptographic library [Bec14]

Security objectives: Our goal is to protect the confidentiality of the private key of the root certificate of the LibreSSL CA. Listing 3.4 shows the annotations added to the private key in the LibreSSL source code.

Security-sensitive code: Table 3.1 shows that Glamdring places 22% of LOC, 17% of functions and 16% of global variables inside the enclave. EBR moves 2 functions into the enclave, thereby: (i) more than halving the number of ecall crossings; and (ii) reducing the number of ocall crossings by an order of magnitude for **sign** requests. The majority of functions and global variables assigned to the enclave originate from the libcrypto library, which contains most of the certificate signing logic.

Partitioned architecture: Glamdring places only a subset of LibreSSL into the enclave (see Figure 3.4):

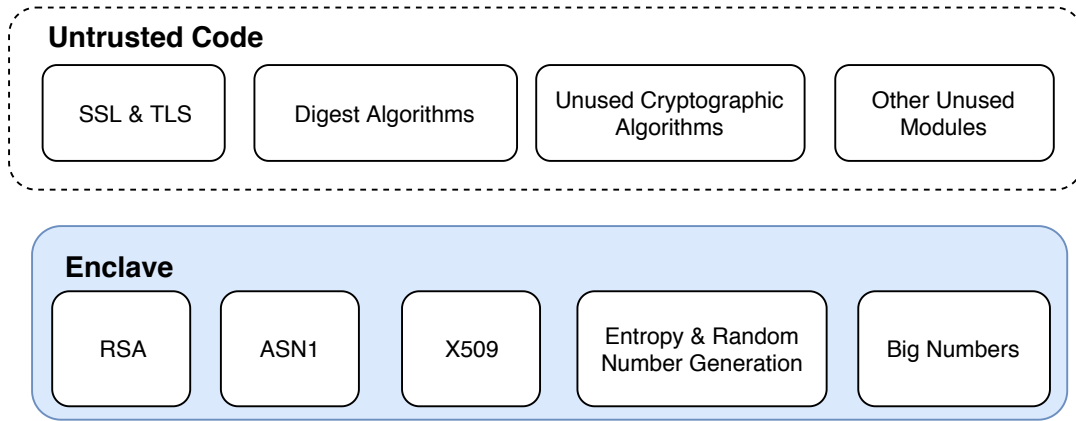


Figure 3.4: **The partitioned LibreSSL architecture** (Here, we see the separation between security-sensitive code and data placed inside the enclave for LibreSSL, and non-sensitive code and data outside the enclave.)

(i) the entropy/random number generator; (ii) the RSA and Big Numbers module; and (iii) the X509 module, which stores the certificates. The functionality placed outside includes: (i) the TLS/SSL modules for secure communication; (ii) digest algorithms (MD5, SHA256); and (iii) cryptographic protocols unrelated to certificate signing (DSA, AES)

Enclave interface: LibreSSL exposes 171 ecalls and 613 ocalls (see Table 3.1). Out of those, only 23 ocalls provide access to C library functions; 49% of ocalls provide access to global variables; and the remaining 278 ocalls are used to execute outside LibreSSL functions.

Glamdring places the private key of the root certificate and any variables that depend on it inside the enclave. The communication between the client requesting a certificate signature and the enclave involves: (i) reading the certificate to be signed; and (ii) outputting the signature. We assume that the root certificate and its private key are given to the enclave during initialisation [Joh16]. Since the signed certificate is not confidential, no explicit declassification is needed before writing it to disk via an ocall.

Digital Bitbox hardware wallet [Shi16]

Security objectives: We want to secure Digital Bitbox in a remote deployment, such as an online bitcoin service. An attacker must not: (i) read or modify the private keys in the wallet; and (ii) issue commands such as creating and signing transactions.

We consider three API calls security-sensitive: (i) `seed()` to create a new wallet; (ii) `sign()` to sign a transaction and return the signature; and (iii) `random()` to return a random number. We annotate these API calls with security annotations. Listing 3.5 shows the annotation added to protect the transaction signature returned to the user for the `sign()` API call:

```

1 int wallet_sign(char *message, char *keypath) {
2     uint8_t sig[64];
3     ...
4     ecc_sign_digest(node.private_key, data, sig);
5     ...
6     #pragma glamdring sensitive--sink(sig)
7     return commander_fill_signature_array(sig, pub_key);
8 }

```

Listing 3.5: The pragma annotations added to the Digital Bitbox source code in order to identify the transaction signature as a security-sensitive sink.

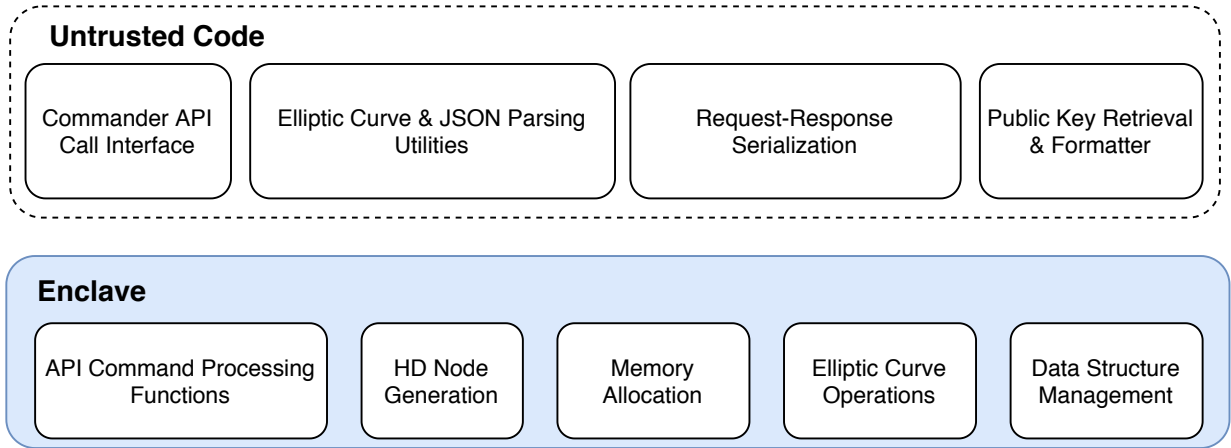


Figure 3.5: **The partitioned Digital Bitbox architecture** (Here, we see the separation between security-sensitive code and data placed inside the enclave for Digital Bitbox, and non-sensitive code and data outside the enclave.)

Security-sensitive code: Glamdring places 38% of LOC, 42% of functions and 52% of global variables inside the enclave (see Table 3.1). EBR increases the TCB by 4 functions, reducing the number of ecall and ocall crossings at runtime by between 1 and 3 orders of magnitude, for the `seed`, `sign` and `random` API calls. Only half of the Digital Bitbox code itself is security-sensitive: 51% of functions and 60% of global variables.

Partitioned architecture: Figure 3.5 shows the partitioned Digital Bitbox architecture. The functionality placed inside the enclave includes: (i) command processing functions for specific API calls; (ii) code for generating seeds (using the SGX-provided hardware random generator); and (iii) elliptic curve operations for transaction signing. The functionality placed outside includes: (i) wallet management functions for retrieving the public key and address formats; (ii) the command interface for handling API calls and constructing responses; and (iii) elliptic curve and JSON parsing utility functions in Secp256k1 and Yajl.

Enclave interface: Digital Bitbox exposes 114 ecalls and 55 ocalls (see Table 3.1). 36% of ocalls are to C library functions unavailable inside the enclave; 64% are to application functions outside the enclave.

Secure application approaches	LOC	Binary Size	Throughput
Memcached with Glamdring	42,800	770 kB	160 kreq/s
Security-sensitive Memcached	12,450		
Glamdring code generation & hardening	5,662		
Intel SGX SDK	24,688		
Memcached with SCONE	149,298	3.3 MB	270–330 kreq/s
Memcached	28,807		
Musl lib. C	105,885		
Stunnel (network encryption)	14,606		
Memcached with Graphene	746,716	4.1 MB	65–95 kreq/s
Memcached	28,807		
Graphene	693,221		
Intel SGX SDK	24,688		

Table 3.2: The TCB sizes and performance for Memcached for Glamdring, SCONE and Graphene.

To protect the security-sensitive data between the client and the application, Glamdring encrypts: (i) the command to execute (`seed()`/`sign()`); (ii) the user-provided entropy for `seed()`; (iii) the transaction data for `sign()`; (iv) the value of `seed()`; (v) the signature of `sign()` returned to the client; and (vi) the generated random number. Performing data protection at this granularity prevents an attacker from issuing commands to Digital Bitbox, and permits Glamdring to move the majority of the JSON parsing functions outside the enclave, as only a subset of the API request/response is security-sensitive.

Security discussion

Our security evaluation has led to several insights: First, Glamdring achieves small enclave sizes, protecting security-sensitive functionality for real-world applications. Table 3.2 compares the TCB for Memcached of Glamdring with SCONE [ATG⁺16] and Graphene [TAB⁺14, Osc17], which place the whole application inside the enclave. As can be seen, Glamdring is one-third the size of SCONE, and one order of magnitude smaller than Graphene in terms of enclave LOC; around 6,000 LOC are added by Glamdring to the TCB through the code generator and enclave interface hardening. For SCONE, musl places 5× the LOC into the enclave compared to Memcached. For Graphene, the library OS places over 20× the LOC into the enclave compared to Memcached. In binary sizes, Glamdring is 4× and 5× smaller than SCONE and Graphene, respectively.

Second, EBR is effective at reducing the number of `ecall` and `ocall` crossings at runtime, despite only moving a few additional functions into the enclave. In the case of Digital Bitbox, moving four functions into the enclave reduces the number of enclave boundary crossings by up to three orders of magnitude.

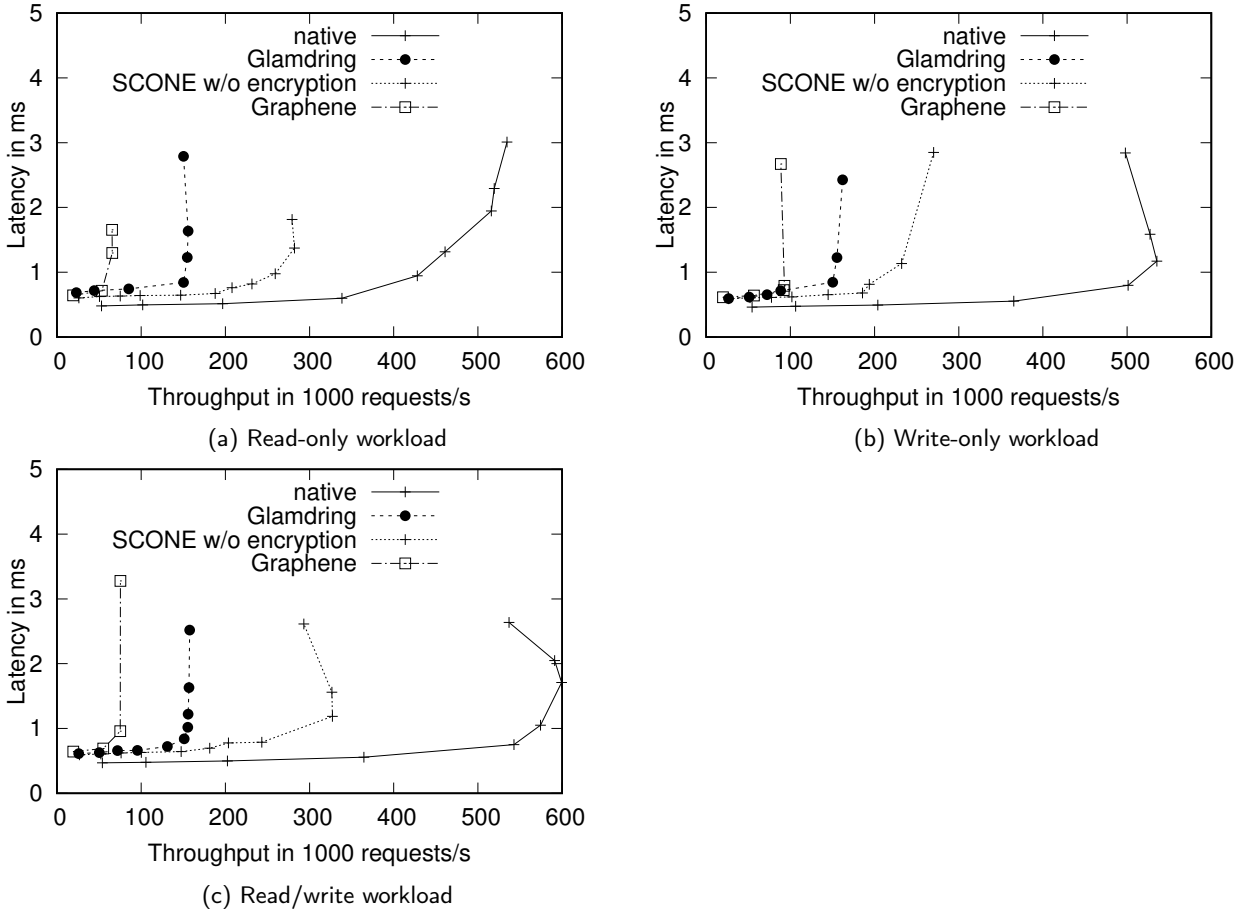


Figure 3.6: The throughput versus latency for Memcached native, with SCONE and with Glamdring, for different types of application workloads.

3.5.2 Performance evaluation

We evaluate the performance of the three partitioned applications in terms of throughput and latency.

Experimental set-up. All experiments are executed on an SGX-supported 4-core Intel Xeon E3-1280 v5 at 3.70 GHz with 64 GB of RAM, running Ubuntu 14.04 LTS with Linux kernel 3.19 and the Intel SGX SDK 1.7. We deactivate hyper-threading and compile the applications using GCC 4.8.4 with `-O2` optimisations.

Application benchmarks. We evaluate Memcached with the YCSB benchmark [CST⁺10]. Clients run on separate machines connected via a Gigabit network link. We increase the number of clients until the server is saturated. Memcached is initialised with the YCSB default of 1000 keys with 1 KB values. We then vary the percentage of `get` (read) and `set` (write) operations.

For LibreSSL, we measure the throughput and latency when signing certificates using SHA-256 and a 4096-bit RSA key. For Digital Bitbox, we observe the performance for the `seed`, `sign`, and `random` API calls using workloads from the Digital Bitbox test suite: (i) `tests_sign` seeds a wallet and signs

64-byte transactions; (ii) `tests_aes_cbc` seeds a wallet with user-provided entropy, sets passwords and performs encryption/decryption with AES-256; and (iii) `tests_random` returns random numbers.

Results. We measure the throughput and latency for Memcached: (i) partitioned by Glamdring; (ii) executed by SCONE (without network encryption); (iii) by Graphene; and (iv) natively, as the request rate is increased. We consider three workloads: `read-only`, `write-only` and `50%/50% read/write`.

Figure 3.6 shows that all three variants exhibit consistent behaviour across the workloads. Glamdring shows a throughput of 160k requests/s; SCONE (without encryption) achieves between 270k–330k requests/s; Graphene between 65k–95k requests/s; and the native Memcached achieves around 530k–600k requests/sec.

The reason for Glamdring’s lower throughput compared to SCONE is that SCONE avoids all enclave transitions; it trades off TCB size for performance (see Section 3.2.3) and requires user-level enclave threading to avoid transitions. Enclave transitions dominate the cost of processing a request; by batching multiple `get` requests together using `multi-get`, Glamdring achieves over 210k requests/sec. However, Glamdring has only a third of the TCB of SCONE (see Table 3.2). The overhead of a library OS means that Memcached with Graphene exhibits worse performance than Glamdring.

Figure 3.7 shows the performance of LibreSSL and Digital Bitbox with Glamdring compared to their native versions. The throughput of certificate signing in LibreSSL is $0.6\times$ compared to native execution, decreasing from 63 to 36 signatures per second on each CPU core. The performance of LibreSSL is limited by a single ecall (`bn_sub_part_words`), which is central to the RSA algorithm and accounts for 95% of all enclave transitions. As for Digital Bitbox, compared to native execution, the relative throughput is between $0.7\times$ and $0.8\times$; the relative latency is between $1.3\times$ and $1.4\times$.

Effect of EBR. By comparing the performance of partitioned applications before and after applying the EBR optimisation, we found that the EBR optimisation increased the throughput by $1.6\times$ to $4.0\times$ for the three applications, at the cost of at most 4 additional functions in the enclave.

3.6 Related work

We discuss work related to Glamdring in detail below:

Privilege separation. The attack surface of applications can be reduced in many different ways [BS04, Rub16, WSLD13, GWA⁺15, GPKK12, LZC⁺15]. For example, PrivTrans [BS04] partitions applications into a privileged monitor and an unprivileged slave component based on the principle of least

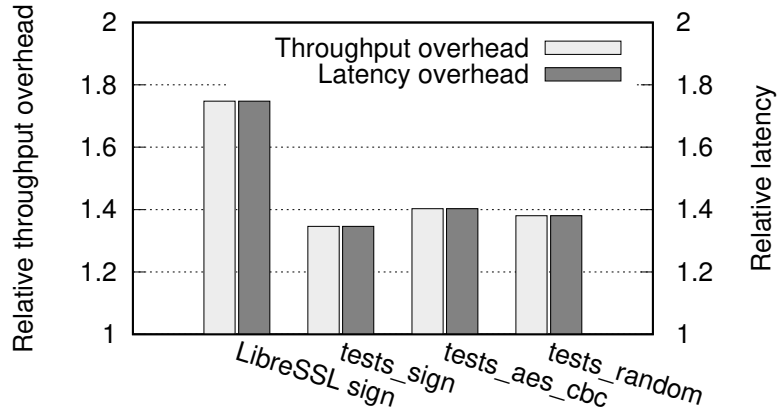


Figure 3.7: Throughput and latency overhead for Digital Bitbox and LibreSSL when performing various operations.

privilege. To achieve this partitioning, PrivTrans uses static analysis, however, PrivTrans does not consider the integrity of sensitive application data when performing the partitioning. ProgramCutter [WSLD13] and Wedge [BMHK08a] rely on dynamic analysis to partition applications, which is not as conservative as static analysis. SeCage [LZC⁺15] combines static and dynamic analysis to partition applications. However, the isolation between different partitions is enforced using CPU virtualisation features. In contrast to these works, Glamdring does not need a trusted OS or hypervisor and respects the constraints enforced by trusted execution.

SOAAP [GWA⁺15] helps developers to reason about the potential compartmentalisation of applications based on source annotations and static analysis. Unlike Glamdring, it does not support automated code partitioning. Rubinov et al. [Rub16] propose a partitioning framework for Android applications. It refactors the source code and adds a set of privileged instructions. However, it only supports type-safe Java applications and requires users to re-implement the security-sensitive functionality in C.

Protecting applications from an untrusted OS. A number of approaches have been proposed to deal with an untrusted OS that spans millions of LOC. NGSCB [PCEM04] and Proxos [TMLL06] execute both a trusted and an untrusted OS using virtualisation, where security-sensitive applications are managed only by the trusted OS. The TCB, however, still includes a full OS. In more recent work, Overshadow [CGL⁺08], SP³ [YS08], InkTag [HKD⁺13] and Virtual Ghost [CDA14] protect application memory from an untrusted OS by extending the *virtual machine monitor* (VMM). In contrast to Glamdring, these approaches put trust in the VMM, and cannot protect against attackers with privileged access, such as system administrators. Moreover, these works do not offer protections against an attacker with physical access to the machine.

Trusted hardware. Solutions based on the use of trusted hardware, such as *secure co-processors* [DLP⁺01] and *trusted platform modules* (TPM) [Baj02], can protect against attackers with physical access. A

TPM can measure system integrity and provide remote attestation to verify the software stack [GPC⁺03]. Since the TPM measurement will always include the OS and any system libraries, the TCB is greater than that offered by Glamdring.

Flicker [MPP⁺08] achieves an extremely small TCB of just 250 LOC, in addition to hardware and the application code itself, by using the *Late Launch* feature provided by recent processors. However, Flicker provides no relevant system support for applications and suffers from slow TPM operations. TrustVisor [MLQ⁺10] is a special-purpose VMM that uses software-based μ TPMs for application integrity checking, but it focuses on small pieces of application logic and requires a trusted hypervisor. CloudVisor [ZCCZ11] provides integrity and confidentiality protection for virtual machines using nested virtualization, but this leads to VM-sized TCBs.

Trusted execution environments (TEEs). Trusted execution allow systems to avoid placing trust in software layers such as a VMM. For example, *Haven* [BPH15] demonstrates shielded execution of unmodified Windows applications inside of SGX enclaves, but suffers from a large TCB size. *SCONE* [ATG⁺16] provides a modified C library that allows unmodified Linux applications to run on top of SGX without the TCB overhead of a full library OS. *VC3* [SCF⁺15] shows how map/reduce jobs can be protected by enclaves. It executes all data processing tasks inside an enclave and secures data exchange using encryption. However, neither Haven, SCONE, or VC3 consider the automated partitioning of legacy applications to achieve a small TCB while still keeping the cost of transitions between trusted and untrusted execution context at balance.

Application partitioning. A number of application partitioning frameworks have been proposed that focus on aspects other than security. For example, *Pyxis* [CMAM12] automatically partitions database applications based on profiling and static analysis to convert parts of the application into *stored procedures* in order to improve performance. *Virtual application partitioning* [GPKK12] profiles and monitors legacy applications to automatically partition applications at runtime. Based on events such as user authentication or file accesses, it enables or disables security mechanisms dynamically to minimise performance overhead. In contrast to Glamdring, these approaches do not focus on security, feature TCB reduction or offer hardware-based isolation. PSWD [SGZ16] partitions and partly encrypts legacy applications at the binary level for Intel SGX enclaves to fight software piracy. Unlike Glamdring, PSWD neither focuses on reducing the TCB size nor does it follow the principle of least privilege.

3.7 Limitations

In this section, we briefly discuss the limitations of our approach:

Correctness of developer annotations. The approach offered by Glamdring requires developers to correctly annotate security-sensitive sources and sinks in the application. If a developer provides incorrect annotations, e.g., fails to mark a security-sensitive source, the partitioned application produced by Glamdring will not correctly protect sensitive data using the TEE. As such, the security of our approach: (i) requires domain specific knowledge about the application, e.g., which variables are security-sensitive; and (ii) relies heavily on the correctness of the annotations provided by developers. While this problem is not exclusive to Glamdring, i.e., many security tools take the same approach [SYM⁺14, GWA⁺15], we note that it is a limitation of this work.

Feasibility of static analysis. The approach offered by Glamdring requires static program analysis to identify security-sensitive code and data in the application. For example, Glamdring employs: (i) static dataflow analysis to identify statements that impact confidentiality; and (ii) static backward slicing to identify statements that impact integrity. As such, Glamdring needs to be able to reason about application code in order to produce an appropriate partitioning. However, it is still an open research challenge to apply static program analysis to large-scale, complex applications [AB01]. Moreover, as static pointer analysis is undecidable in C [Lan92], static analysis may produce false positives. This means Glamdring may: (i) not be able to produce a partitioning for some applications; or (ii) mark large parts of an application as security-sensitive unnecessarily. As such, our approach is limited by the feasibility of performing static analysis on the application.

Manual effort for EBR. Glamdring improves the performance of partitioned applications by proposing enclave boundary relocation (EBR): moving additional functions into the enclave in order to reduce the number of enclave crossings at runtime. While Glamdring has shown EBR to be highly effective at improving performance with a minimal increase in the application TCB (see Section 3.5.2), developers have to perform this optimisation manually. This requires: (i) finding an appropriate workload to run on the partitioned application; and (ii) iteratively moving functions into the enclave to find an optimal trade-off between performance and the size of the TCB. However, such manual effort is both tedious and time consuming. Moreover, it is subjective as to what constitutes an appropriate or optimal trade-off between the performance and size of the TCB for the application. As such, these issues still remain limitations of our approach.

3.8 Summary

In this chapter, we proposed a solution to the first security challenge faced by TEE-based applications: minimizing the TCB (Section 1.4.1). To do this, we proposed automatic application partitioning as a way to reduce the amount of code and data placed into the TEE when an application is deployed. This reduces the amount of code and data that needs to be trusted for the application to operate securely, and leverages the principle of least privilege.

To perform automatic application partitioning for TEEs, we make three contributions: (i) using source code annotations and static program analysis to identify security-sensitive code and data; (ii) partitioning the application at the function granularity to produce a secure TEE boundary interface; and (iii) using automatic source code generation to transform the application. We employed these contributions for Glamdring, the first application partitioning framework for existing C applications protected by Intel SGX enclaves. Glamdring demonstrates that our approach is not only feasible, but practical, as it can be successfully applied to several existing real-world applications with acceptable performance overheads.

While Glamdring targets applications written in C for Intel SGX enclaves, the contributions made by this chapter are fundamentally application and TEE-agnostic: our approach does not rely on any application or TEE-specific features, but can be applied to a wide range of applications and TEEs. As such, we argue that application partitioning for TEEs is a realistic and efficient way to protect applications in untrusted environments.

4

Securing TEE Boundary Interfaces

Applications executing within the confines of a TEE must be able to communicate with the outside world. As such, applications must define TEE boundary interfaces between trusted components inside the TEE, and untrusted components outside the TEE. As these interfaces expose attack vectors, developers must secure these interfaces against adversaries (see Section 2.4). To achieve this, we propose a set of TEE interface design principles for protecting applications against boundary interface attacks.

We begin this chapter by providing an overview of our interface design principles (Section 4.1). We then present relevant background on host OS interfaces for TEEs (Section 4.2); this includes our threat model and existing approaches for host OS interfaces. Next, we present Citadel: the first TEE runtime system that employs our interface design principles to offer a secure host OS interface (Sections 4.3 to 4.5). We continue the chapter by offering a detailed performance evaluation of our approach and contributions (Section 4.6). Finally, we discuss related work to Citadel (Section 4.7), highlight limitations of the approach (Section 4.8) and conclude the chapter (Section 4.9)

4.1 Overview

To protect TEE boundary interfaces from attacks, applications must consider two possible attack vectors: (i) adversaries can read and modify the parameters that cross the TEE boundary interface to cause application integrity and confidentiality violations [CS13c]; and (ii) the frequencies, sequences and parameters of interface calls expose side channel information to adversaries [HCP17, CZRZ17]. This information can violate confidentiality (see Section 2.4).

To address this, we propose applications follow three TEE interface design principles: (i) minimality: the interface should be as small as possible, to narrow the attack surface and allow all interface calls to be systematically protected [Här02]; (ii) protection: all sensitive data that flows across the boundary interface should be protected to ensure confidentiality and integrity, i.e., prevent an adversary from reading or modifying this data; and (iii) obliviousness: to prevent information leakage, i.e., side-channels, interface calls should execute in an oblivious fashion, i.e., the behaviour and execution of interface calls should not be affected by the application, or workload being run.

To illustrate these design principles, we design a TEE boundary interface that leverages minimality, protection and obliviousness to secure communication between an application and an untrusted host OS. To achieve this, we make three contributions: (i) we propose that applications employ a library OS inside the TEE to reduce the amount of functionality required by the host OS. This helps achieve a minimal boundary interface; (ii) we propose transparently encrypting and integrity protecting all data that crosses the TEE boundary interface to the host OS. This prevents an attacker from reading or modifying this data when it leaves the TEE, thus protecting sensitive application data and avoiding application integrity violations; and (iii) we propose making host OS interface calls discrete, to hide timing information, and deterministic, to hide application behaviour. This avoids leaking information about application execution to an adversary and achieves an oblivious boundary interface.

To evaluate our contributions, we present **Citadel**, the first TEE runtime system for applications that protects against an untrusted host OS. Citadel implements our contributions to protect Linux applications executing inside Intel SGX enclaves: (i) for minimality, Citadel employs the Linux Kernel Library (LKL) [PGT10] as the library OS inside the enclave, thus exposing only 7 host interface calls; (ii) for protection, Citadel ensures that all I/O operations across the host interface are transparently encrypted and integrity-protected, e.g., Citadel uses an encrypted Linux file system to protect disk I/O; and (iii) for obliviousness, Citadel makes host interface calls discrete and deterministic, executing them independently of the application workload inside the enclave.

To measure the performance and overhead of our contributions, we evaluate Citadel with real-world applications and benchmarks: with emulated SGX enclaves (to ignore current SGX memory limits), Citadel trains deep neural networks with an overhead of 14%, and 21% with oblivious interface calls. With current SGX hardware, Citadel executes benchmarks that fit into SGX memory with an average overhead of 1.5x and 3.1x with and without oblivious calls, respectively.

4.2 Background: Host OS interfaces for TEEs

In this section, we analyse the current host OS interfaces offered by existing TEE runtime systems. We assume that applications are deployed in cloud environments, where the cloud provider offers TEEs, and the host OS is untrusted. In this work, we focus on a *lift-and-shift* model [Net] for applications: users deploy unmodified Linux binaries in the cloud. To achieve this, binary compatibility requires a *TEE runtime system* [BPH14, ATG⁺16, TPV17], which provides POSIX abstractions to the application. In TEE runtime systems, functionality that cannot be implemented within the trust domain of the TEE, e.g., I/O operations, must be delegated to the untrusted host OS. As such, we refer to the interface between the TEE runtime system and the host as the *host interface*.

4.2.1 Security goals and threat model

Our goal is to prevent an adversary from compromising the confidentiality and integrity of: (i) the application code; (ii) the application input and output data; and (iii) all computations executed inside the TEE.

We consider all software outside the TEE as untrusted and under adversarial control, including the host OS kernel. We assume that the TEE itself is trustworthy, i.e., it is outside the scope of this work to tolerate attacks against the TEE, such as Spectre [LSD19], Foreshadow [VBMW⁺18], Zombieload [SLM⁺19], and cache side channel attacks [MIE17b, BMD⁺17a, GESM17]. While mitigations for these types of attacks exist, e.g., hardware and/or microcode changes [Int19], compiler techniques [OTK⁺18b, BCD⁺17, CWC⁺18] and defensive programming [OST06, SGF18, CBZ⁺18], tolerating such attacks are orthogonal to our work. We defer the reader to Chapter 5 for our contributions on tolerating TEE failures and compromises.

Our threat model focuses on attacks against the host interface, as these attacks are easy to carry out and do not assume implementation specific behaviour of the TEE. To attack the host interface, an adversary can: (i) compromise confidentiality, by observing the parameters, frequencies and sequences

of host interface calls. For example, they can observe the disk I/O access pattern to determine the application or workload being run, e.g., linear scans and repeated accesses are easy to identify via the host interface. This side channel discloses information about the TEE execution; and (ii) compromise integrity, as the adversary may modify the input/output parameters of host calls, repeatedly perform arbitrary host calls or interfere with their execution outside of the TEE. For example, an adversary may hijack an enclave’s control flow or trick it into exposing confidential data via Iago attacks [CS13c].

4.2.2 Host interfaces of existing TEE runtime systems

We analyse the security of the host OS interfaces offered by three existing TEE runtime systems. These systems are:

1. *Panoply* [SLTTS17] minimises the size of the trusted computing base (TCB) inside the TEE. To achieve this, it places the *glibc* standard C library [pro19] outside the TEE and implements a shim layer that forwards C library calls. Panoply exposes the largest host interface because it delegates all C library calls to the host OS.
2. *Graphene-SGX* [TPV17] reduces the size of the host interface compared to Panoply by implementing a partial library OS inside the TEE. However, Graphene-SGX relies on the host OS for file system, network stack, and threading implementations.
3. *Haven* [BPH14] uses the Drawbridge library OS [PBWH⁺11b] inside the TEE to execute Windows applications. Drawbridge provides file system and network stack implementations inside the TEE, reducing the size of the host interface.

Types of host interface parameters. To analyse the security of existing host OS interfaces, we target the parameters that cross the TEE boundary interface. This is because these parameters can leak information to the host OS and/or compromise TEE integrity. We categorise parameters into five types, ordered by the difficulty of protection from easiest to hardest: (1) *variable-sized buffers* pass a user-defined byte array across the host interface. They are used in file/network I/O operations, such as the `buf` and `count` parameters in the `read()` system call [Rea19]; (2) *address ranges* represent parameters that refer to regions of untrusted or trusted memory. Examples are the parameters passed to `mmap()` and the return value of `malloc()`; (3) *pure identifiers* point to entities, e.g., a file descriptor; (4) *impure identifiers* also point to entities, however impure identifiers additionally disclose information about the entity, e.g., a path name; and (5) *semantic parameters* refer to parameters with opaque semantics specific to the host call, such as `mode` and `flags` for file access operations.

TEE runtime system	Function of host calls	Out parameters (impact confidentiality) More difficult to protect →						In parameters (impact integrity) More difficult to protect →					
		Number of host calls			Variable size buffer			Pure identifier			Address range		
Panoply [SLTTS17]	I/O	239	11	17	96	49	139	10	2	22	24	74	
	Events	22	-	-	5	-	22	-	-	2	-	8	
	Time	12	-	-	-	-	7	-	-	-	-	10	
	Threading	29	1	-	10	-	17	2	-	4	-	10	
Graphene-SGX [TPV17]	I/O	29	3	1	18	6	19	3	2	2	4	7	
	Events	1	-	-	1	-	1	-	-	-	-	1	
	Time	2	-	-	-	-	1	-	-	-	-	2	
	Threading	6	-	-	-	1	4	1	-	1	-	1	
Haven [BPH14]	I/O	11	1	3	6	1	7	1	-	3	-	1	
	Events	6	-	-	5	-	1	-	-	1	-	3	
	Time	1	-	-	-	-	-	-	-	-	-	1	
	Threading	6	-	-	1	-	2	-	-	1	-	-	
Citadel	I/O	4	2	-	-	-	2	2	-	-	-	-	
	Events	2	-	-	-	-	1	-	1	-	-	2	
	Time	1	-	-	-	-	-	-	-	-	-	1	
	Threading	-	-	-	-	-	-	-	-	-	-	-	

Table 4.1: The security breakdown of parameters in host interface calls for existing TEE runtime systems.

Table 4.1 shows the total number of host calls for each TEE runtime system, categorised according to the following functions: I/O, events, time and threading.¹ Panoply exposes the largest host interface, with 302 host calls in total; Graphene-SGX and Haven require 38 and 24 host calls, respectively.²

We break down host interface parameters according to the five types outlined above and distinguish between: (i) *out* parameters, which are passed out to the host OS and may compromise confidentiality; and (ii) *in* parameters, which are passed into the TEE and may affect integrity.

Confidentiality attacks. To learn sensitive information, an adversary may observe the *out* parameters of host interface calls. The amount of data disclosed depends on the type of the parameter:

1. *Variable-sized buffers* may hold security-sensitive data. To protect the confidentiality of this data, these buffers must be encrypted and padded to a fixed length, otherwise an adversary can still infer sensitive information. For example, in a machine learning application, such as image classification [KSH17], if the outcome of the classifier is held in a variable-sized buffer and this buffer is not encrypted, the adversary can read the classification result, learning sensitive information about the input image. Likewise, if the outcome of the classifier is encrypted, but not padded to a fixed length, the adversary can still infer the result by comparing buffer lengths.

Table 4.1 shows that Panoply, Graphene-SGX and Haven pass 11, 3 and 1 variable-sized buffers, respectively, to the host for I/O operations; Panoply also uses a variable-sized buffer to share messages between threads. To protect these buffers, most of the TEE runtime systems provide transparent encryption when the buffer crosses the TEE boundary interface. However, all of these systems still disclose the size of the encrypted buffer to the host. For example, for disk operations: (i) Haven implements an in-enclave file system that encrypts data written to the host, but still exposes the size of the encrypted data; (ii) SCONE divide files into fixed-size encrypted chunks, but still reveals the number of chunks written to the host, thus exposing the file size; and (iii) Graphene-SGX and Panoply do not provide transparent file encryption, thus exposing sensitive file data. As such, all of these systems expose potential attack vectors through variable-sized buffers.

2. *Address ranges* passed from the TEE to the host OS point to continuous memory regions in untrusted memory. As such, adversaries can observe the *access pattern* of these address ranges to attack the confidentiality of the application. For example, Panoply uses untrusted memory

¹For Panoply and Graphene-SGX, the host calls are taken from the GitHub source code [Gra19, Pan19]; for Haven, they are obtained from the paper [BPH14].

²We do not include a breakdown of SCONE’s [ATG⁺16] host interface as its implementation is not publicly available. However, as it forwards most system calls to the host OS, its interface can be assumed to be similar to that of Panoply.

address ranges for communication between enclaves. An adversary can observe the access pattern of these memory ranges to reveal application-specific control flows, e.g., a secret-dependent inter-enclave call. To prevent address range access patterns from exposing confidential information, accesses must be made oblivious, or avoided entirely. However, Panoply, Graphene-SGX, and Haven do not protect against such attacks. Instead, they expose 17, 1 and 3 address ranges, respectively. As such, all of these address ranges can be used to target application confidentiality.

3. *Pure/impure identifiers*. Similar to address ranges, passing pure and impure identifiers to the host OS reveals resource access patterns, which can be used to violate the confidentiality of security-sensitive data. For example, in the case of a database protected by a TEE, Cash et al. [CGPR15] have shown that an adversary can infer which queries are being executed by observing the accesses of records outside the TEE. Preventing information leakage by pure and impure identifiers can therefore only be done on a case-by-case basis, e.g., by replacing file names with hashes, and making accesses oblivious. In general, pure identifiers will reveal less information than impure identifiers, as pure identifiers are often chosen randomly, e.g., file descriptors. However, both of these identifiers expose access patterns.

SCONE, Panoply, Graphene-SGX, and Haven use a large number of identifiers for I/O operations, event handling and threading. These systems: (i) do not provide any protection for impure identifiers; and (ii) do not protect the access patterns of pure and impure identifiers. As such, these identifiers can be used to target application confidentiality.

4. *Semantic parameters* have a context-specific meaning and therefore cannot be encrypted transparently. Existing runtime systems rely on a large number of outgoing unprotected semantic parameters. For example, Graphene-SGX, Panoply, and Haven all rely on the host for thread synchronisation. Graphene-SGX exposes address, operation, value, and timeout parameters of a `futex` host call. This gives an adversary detailed information about the current state of the application and facilitates possible attacks, e.g., *AsyncShock* [WKPK16] which exploits the host's control of enclave threads. As such, these systems are vulnerable to confidentiality attacks through semantic parameters.

Integrity attacks. To compromise integrity, an adversary might tamper with the *in* parameters of host interface calls. The difficulty of integrity-protecting these parameters varies according to the parameter type:

1. *Variable-sized buffers* passed into the TEE must be integrity protected. Otherwise, they expose two types of attacks: (i) an adversary may modify the contents of the buffer to violate data

integrity; or (ii) an adversary may modify the buffer count to trigger an overflow in the TEE. To protect against these attacks, the contents of the buffer must be protected cryptographically, e.g., using an HMAC, and the TEE must check buffer lengths. The TEE must also ensure the freshness of the HMAC, otherwise an adversary could swap two valid buffers or perform a roll-back attack.

For I/O related calls, Panoply, Graphene-SGX, and Haven pass 10, 3 and 1 parameters with variable-sized buffers, into the TEE, respectively. Panoply and Graphene-SGX also use variable-sized buffers for inter-enclave communication. All analysed TEE runtime systems check buffer lengths to avoid variable-sized buffers from triggering buffer overflow attacks. Moreover, Panoply protects the integrity of inter-enclave communication using TLS, but does not protect file I/O; Graphene-SGX ensures the integrity of file I/O by maintaining a Merkle tree of file chunk hashes inside the enclave. However, Panoply, Graphene-SGX, and Haven not protect the integrity of network I/O for applications without built-in TLS support. As such, these parameters expose potential attack vectors for integrity.

2. *Address ranges* passed into the TEE are difficult to integrity check: an adversary may manipulate addresses by reusing old ranges to roll back data, swap ranges, or modify address ranges to corrupt data.

For I/O operations, Panoply and Graphene-SGX pass 2 address ranges into the TEE. They check that the ranges are entirely inside or outside of TEE memory, thus preventing an adversary from hijacking an enclave’s control flow [BCD⁺18, LJJ⁺17]. However, all three runtime systems are vulnerable to an adversary reusing old ranges, swapping ranges and modifying ranges; the only effective mitigation is to avoid relying on untrusted address ranges.

3. *Pure/impure identifiers*. Similar to address ranges, pure/impure identifiers passed into the TEE are difficult to integrity check. For example, adversaries may pass invalid or manipulated identifiers into the TEE, such as swapping two file descriptors, causing the TEE to access incorrect files. In some cases, such malicious activity can be detected, e.g., Graphene-SGX maintains per-file HMACs, which reveal wrong file descriptors, and incorrect network sockets can be detected by TLS. However, securing against such attacks in general is difficult.

Panoply passes 46, 2, and 4 identifiers into the TEE for I/O, events, and threading, respectively; Graphene-SGX passes 6 identifiers for I/O and 1 for threading; Haven passes 3, 1, and 1 identifiers for I/O, events and threading, respectively. These systems implement few integrity checks on pure/impure identifiers. As such, an adversary can target these identifiers to violate integrity.

4. *Semantic parameters* must have their integrity verified on a case-by-case basis. For semantic

	Call	Description
I/O	<code>disk_read(offset, buf, len)</code>	Read <code>len</code> bytes from disk image file at offset <code>offset</code> ; returns bytes read
	<code>disk_write(offset, buf, len)</code>	Write <code>len</code> bytes from <code>buf</code> to disk image file at offset <code>offset</code> ; returns bytes written
	<code>net_read(buf, len)</code>	Read at most <code>len</code> bytes from network device into <code>buf</code> ; returns bytes read
	<code>net_write(buf, len)</code>	Write <code>len</code> bytes from <code>buf</code> to network device; returns bytes written
Events	<code>net_poll(eventmask) → reventmask</code>	Wait for <code>eventmask</code> events on network device; returns occurred events
	<code>forward_signal(num, code, addr)</code>	Forward signal <code>num</code> with description <code>code</code> occurred at <code>addr</code> to enclave
Time	<code>time_read() → time</code>	Read time from untrusted vDSO memory region

Table 4.2: The host interface exposed by Citadel.

parameters with few valid values, such as `errorcode` and `signum`, the TEE can perform explicit checks; for ones with a larger domain, such as `size`, bounds checks are possible. None of these checks can establish semantic correctness however.

All three systems use many semantic *in* parameters for I/O, events, time and threading operations. Since both Panoply and Graphene-SGX rely on the host file system, they can only check for the plausibility of returned file meta-data, such as `st_size` and `st_blocks`. As such, these parameters are difficult to check and expose potential attack vectors for adversaries.

4.2.3 Designing a secure host interface

From the above discussion, we observe that it is non-trivial to: (i) protect all *out* parameters from leaking information and exposing access patterns; and (ii) verify the correctness of all *in* parameters. As such, each parameter that crosses the boundary interface has the potential to violate application confidentiality or integrity. Therefore, the first step in designing a secure host OS interface is to keep the interface as narrow as possible and minimise the number of *in* and *out* parameters that cross the interface. This reduces the size of the attack surface for the application. As such, it is critical that only functionality that cannot be provided within the TEE should be delegated to the host OS.

In addition, it is important to reduce the complexity of host interface call parameters. By building on low-level host calls, instead of high-level POSIX abstractions, the parameter types exposed to the host become simpler, thus making it easier to reason about security implications. For example, Panoply, SCONE, and Graphene-SGX rely on the host file system implementation and thus expose impure identifiers to the host, such as file paths and file names. This leaks a large amount of sensitive information to the host. Instead, a secure host OS interface should avoid delegating such resource

management to the host as much as possible, and rather perform such management inside the TEE. This exposes less information to the host. For example, complex parameters related to file I/O, e.g., impure identifiers for file names, can remain within the TEE, and simpler parameters, e.g., pure identifiers for file descriptors, can be passed to the host instead.

Based on these principles, Table 4.1 shows the host OS calls and parameter types of Citadel. Citadel is our TEE runtime system that targets Intel SGX enclaves. To reduce the attack surface of the host interface, Citadel avoids using address ranges and pure/impure identifiers. Instead, it only requires a small number of simple semantic parameters, of which all, but one (`time`), have a fixed set of valid values or can be bounds-checked. As such, these parameters can be clearly reasoned about and protected against attacks.

4.3 Minimising the host interface

Table 4.2 shows the Citadel host interface. This interface is minimal by design, only exposing seven interface calls to the untrusted host OS. These calls relate to functionality that cannot be provided inside an Intel SGX enclave: (i) as Intel SGX does not support trusted I/O paths, Citadel must expose disk and network I/O operations to the host; and (ii) as Intel SGX does not provide in-enclave support for events and time, such functionality must be delegated to the host. We defer the reader to Section 2.3.2 for background on Intel SGX. We discuss the host OS interface, and the corresponding in-enclave functionality required to achieve this interface, below.

4.3.1 Citadel host interface

The Citadel host interface exposes calls for:

I/O operations. To reduce the complexity of the host interface, Citadel targets a low-level I/O interface. For disk I/O, `disk_read()` and `disk_write()` read from and write to a persistent block device, respectively. Each call takes an `offset` into the block device, a pointer `buf` to a buffer, and the length `len` of the data. For network I/O, `net_read()` and `net_write()` receive and send network packets. Both calls take a pointer `buf` to a buffer. For `net_read`, `len` contains the buffer size; for `net_write`, it is the number of bytes to be written.

Events. To support event related functionality, Citadel exposes `net_poll()`, which passes an `eventmask` to the host with the network events that Citadel is waiting for. The call blocks until network packets are available to be read or outgoing packets can be sent. It returns which events have occurred.

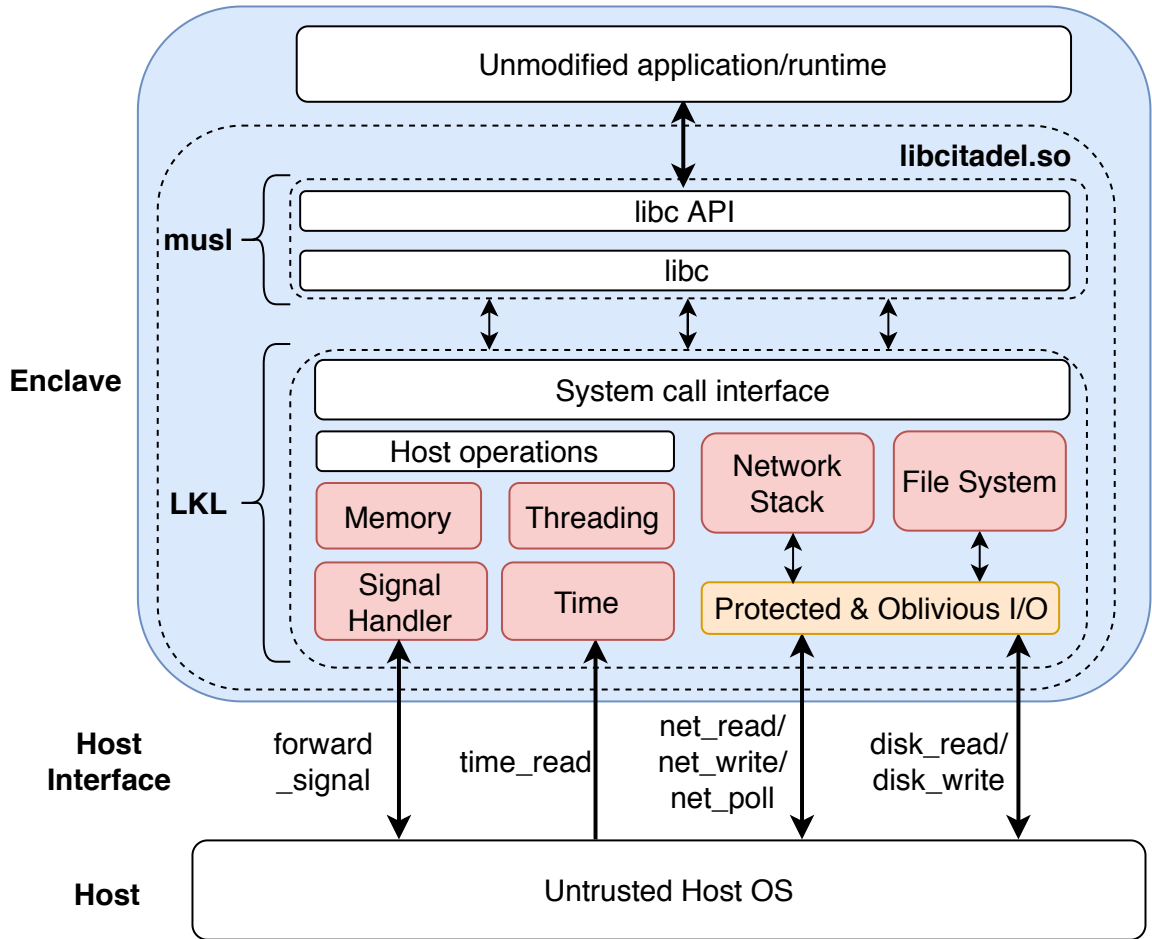


Figure 4.1: The Citadel architecture

Citadel must also handle hardware exceptions, such as page access violations or illegal instructions. An exception causes an enclave exit and transfers control to the host kernel. A `forward_signal()` call provides the signal description to the enclave: the signal number `num`, the cause `code`, and the associated memory address `addr`. The exception is then either processed by Citadel directly or forwarded to the application if it has registered a corresponding signal handler.

Time. To provide application support for time in Citadel, a call `time_read()` reads the time from different clock sources, such as the real-time and monotonic clocks. It is used by application code requesting the current time and by Citadel itself, e.g., to generate timer interrupts required by LKL (see Section 4.3.2).

4.3.2 In-enclave OS functionality

The host interface described above allows Citadel to access low-level host resources, but Linux applications require higher-level POSIX abstractions. To bridge this gap, Citadel provides the following OS functions inside the enclave: (i) file system implementations; (ii) a TCP/IP network stack; (iii) thread-

ing and scheduling; (iv) memory management; (v) signal handling; and (vi) time. These OS functions are typically made accessible to userspace through system calls but, rather than invoking system calls directly, applications link against a C standard library (*libc*). Similar to other TEE runtime systems [ATG⁺16, TPV17], Citadel includes a *libc* implementation to support unmodified binaries that are dynamically linked against *libc*.

Figure 4.1 shows the Citadel architecture. We describe the OS functionality provided by Citadel inside the enclave. This functionality enables Citadel to offer a minimal host OS interface; components discussed in this section are shown in red. The components responsible for protecting the host interface and for making the interface oblivious (shown in orange) are described in Section 4.4 and Section 4.5, respectively.

When an application is deployed, Citadel creates an SGX enclave and loads the enclave library *libcitadel.so*; this library runs alongside the application within the enclave. *libcitadel.so* includes a modified *musl* [Mus19] C standard library that redirects system calls to an in-enclave library OS provided by the *Linux Kernel Library* (LKL) [PGT10]. LKL is an architecture port of the Linux kernel to userspace. It enables Citadel to make components such as the file system and network stack implementations, available inside the enclave. As it is intended to run in userspace, LKL expects a set of *LKL host operations*, e.g., to create threads or to allocate memory. Therefore, *libcitadel.so* includes further components for memory management, user-level threading, signal handling and time.

File system implementation. Existing TEE runtime systems [ATG⁺16, TPV17, SLTTS17] forward POSIX file operations to a host file system implementation. Citadel cannot adopt this approach because it would expose security-sensitive metadata, such as file names, file sizes, file permissions, and directory structures. Instead, it provides complete file system implementations inside the TEE via the Linux *virtual file system* (VFS) layer [BC02]. The VFS layer only requires two host operations for block-level disk I/O (*disk_read()* and *disk_write()*, see Table 4.2).

Applications operate on files through file descriptors as usual, which are handled by the *ext4* file system implementation of LKL. On the host, the reads and writes are made to a single *ext4* disk image file, via the *disk_read()* and *disk_write()* calls. This approach has two advantages: (i) it maintains a small host interface with only two disk I/O calls; and (ii) it ensures that individual file accesses are not visible to the host, which can only observe reads/writes to disk block offsets.

TCP/IP network stack. To provide a POSIX socket API, Citadel uses LKL’s TCP/IP stack to process packets within the TEE. This has three advantages: (i) it minimises the host interface because only access to a virtual network device to send/receive Ethernet frames is needed; (ii) it enables Citadel

to support any transport protocol, e.g., TCP or UDP, without extra host calls; and (iii) it exposes Linux networking features such as packet encryption (see Section 4.4.2).

To be notified about incoming/outgoing packets, Citadel issues a `net_poll()` request. The return value indicates if the device is ready for reading/writing packets using `net_read()` and `net_write()`.

Memory management. Citadel does not interact with the host for memory allocations/deallocations. However, a current limitation of SGX version 1 is that the enclave size must be fixed at initialisation time. Citadel therefore pre-allocates enclave memory and provides low-level memory management primitives inside the enclave. When an enclave is created, it contains uninitialised heap area. The heap area is exposed through both LKL and higher-level *libc* allocation functions, such as `malloc()` and `free()`, as well as `mmap()`, `mremap()`, and `munmap()` which are directly implemented by Citadel. Citadel supports both variable- and fixed-address anonymous mappings, and tracks free pages via a heap allocation bitmap. It implements `mmap()` by scanning the bitmap for consecutive free pages large enough for the requested allocation.

Thread management. While Intel SGX supports concurrency by allowing multiple host threads to enter an enclave, the maximum number of *enclave threads* must be specified at enclave creation time, which conflicts with dynamic thread creation. In addition, having a one-to-one relation between enclave and application threads means that the creation, joining, or exiting of threads as well as thread synchronisation requires host OS support, which poses a security threat [WKPK16].

Therefore, Citadel implements user-level threading based on the *lthread* library [lth19] and provides synchronisation primitives inside the enclave. A fixed number of host threads are assigned to enclave threads, which enter the enclave at startup and only leave when idle. Application threads and LKL kernel threads are *lthreads*, managed via the standard *pthread*s interface. Citadel implements *futex* calls in-enclave to provide synchronisation primitives such as mutexes and semaphores.

Signals. Applications can register custom signal handlers to handle exceptions, interrupts, or user-defined signals. Some signals, such as `SIGALRM`, can be handled entirely within the enclave; others such as `SIGSEGV` or `SIGILL` are caused by hardware exceptions and result in an enclave exit and return control to the host OS. Citadel must forward these signals from the untrusted host to custom signal handlers inside the enclave.

During initialisation, Citadel registers signal handlers for all catchable signals with the host. All signals are forwarded via `forward_signal()`. It also hides application-specific handlers. Citadel then checks for a corresponding application-registered signal handler and, if present, delivers the signal, or ignores it otherwise. Since application-registered signal handlers are managed within the TEE, calls such as

`sigaction()` and `sigprocmask()` are supported without host interaction.

Time. Applications, libraries and LKL frequently access time information. In addition, Citadel reads the current time between context switches to reschedule blocked threads to trigger timer interrupts. Current SGX implementations, however, do not offer a high-performance in-enclave time source, and Citadel relies on time provided by the host. Therefore, instead of issuing expensive individual host calls, Citadel uses the *virtual dynamic shared object* (vDSO) [vDS19] mechanism of the Linux kernel. The kernel maps a small shared library to the address space outside of the enclave. On each clock tick, the host kernel updates a shared memory location with the current time for various clock sources, which are read from within the TEE when a time-related call is made.

For high precision, the vDSO mechanism requires the `RDTSCP` instruction to adjust for the time passed since the last vDSO update. In SGX version 1, this instruction is not permitted inside enclaves. The accuracy of `clock_gettime()` in Citadel thus depends on the frequency of vDSO updates; SGX version 2 does not have this limitation.

Illegal instructions. Applications might use the `RDTSC` instruction to read the timestamp counter or `CPUID` to discover CPU features. These instructions, however, are illegal inside SGX version 1 enclaves. As Citadel must support unmodified binaries, illegal instructions cannot be replaced by corresponding host calls. Instead, Citadel catches the resulting `SIGILL` exception and emulates the instructions: `RDTSC` is executed outside the TEE, and the result is forwarded via the `forward_signal()` call; for `CPUID`, Citadel caches all `CPUID` information during TEE set-up. This eliminates the need for an extra host call and also hides the `CPUID` information requested by an application.

4.4 Protecting the host interface

We now describe how Citadel protects host interface calls from attacks.

4.4.1 Protecting disk I/O calls

Host disk blocks must have their confidentiality and integrity protected. For this, Citadel uses the Linux *device mapper* subsystem [Dev19], which maps lower-level virtual block devices to higher-level devices and allows for I/O data to be transformed along the way. VFS file systems such as *ext4* use the virtual block device exposed by the device mapper. Data can be encrypted and integrity protected transparently before it reaches the underlying device.

Citadel uses different device mapper targets: (i) *dm-crypt* [DMC19] provides full-disk encryption using AES in XTS mode with the sector number as an initialisation vector; (ii) *dm-verity* [DMV19] offers volume-level read-only integrity protection through a Merkle tree of disk block hashes stored on the virtual disk with the root node stored in memory; and (iii) *dm-integrity* [DMI19] provides key-based block-level read/write integrity protection. *dm-crypt* and *dm-integrity* can be combined to provide full disk encryption and read/write integrity protection via AES-GCM. For both AES-XTS and AES-GCM, Citadel uses hardware acceleration through Intel’s AES-NI instruction set extension.

Citadel combines the different targets to provide both confidentiality and integrity for block reads/writes, depending on the security requirements of the application. For example, for an in-memory key-value store such as *Memcached* [Mem19], it is sufficient to protect the integrity of a read-only disk with *dm-verity*: the Memcached binary stored on disk must have its integrity protected, but no further sensitive data is stored on the disk. If the application itself is confidential or other application data is written to disk, *dm-crypt* can be used with either of the integrity protection targets. Citadel’s use of LKL also allows it to support other device mapper based protection targets such as *dm-x* [CJK⁺17] which provides full volume-level read/write integrity as well as replay protection. The use of different targets affects I/O performance, as we explore in Section 4.6.2.

4.4.2 Protecting network I/O calls

Citadel must guarantee the confidentiality and integrity of all network data. Existing TEE runtime systems either require applications to have built-in support for network encryption [BPH14, TPV17, SLTTS17] or use TLS [ATG⁺16]. Not all application-level protocols, however, can use TLS transparently.

Instead, Citadel exploits having a complete TCP/IP network stack inside the enclave, which enables it to provide transparent low-level network encryption. All data received and sent via the `net_read()` and `net_write()` host calls is automatically encrypted, authenticated and integrity-protected. To protect all network traffic, Citadel uses *Wireguard* [Wir19], a layer 3 virtual private network (VPN) protocol, currently proposed for inclusion in the Linux kernel [LKM19].

Citadel sets up Wireguard at initialisation time and exposes the VPN to the application through a network interface with its own IP address. An application binding to this IP address is only reachable by trusted nodes in the VPN. Each Wireguard *peer* has a public/private key pair, which is bound to a VPN IP address and an endpoint, an (IP, port) pair through which the VPN is accessible. Wireguard uses the asymmetric key pairs to establish ephemeral symmetric session keys to protect messages

using authenticated encryption, and nonces to prevent replay attacks. In contrast to TLS, which uses certificates, Wireguard identifies parties through public keys. It does not perform key distribution—Citadel binds keys to enclave identities and supports provisioning of peers' keys.

4.4.3 Protecting event and time calls

For the remaining calls in Table 4.2, Citadel must ensure that an adversary cannot learn confidential data or compromise integrity by providing invalid data.

`net_poll()`. While the `eventmask` reveals if the enclave wants to receive/send packets, this is already disclosed by the presence of `net_read` and `net_write` calls. An adversary can return a wrong `eventmask`: as a result, either the `net_read` call fails, which can be handled transparently, or an invalid packet is read that fails Wireguard's integrity protection (see Section 4.4.2).

`forward_signal()`. Citadel must ensure that signals correspond to genuine events with valid signal descriptions—otherwise an adversary can cause an application signal handler to execute with invalid signal data. For signals triggered by hardware exceptions, Citadel ensures that the passed address lies within the enclave range (e.g., `SIGSEGV`) or replaces the address with the current instruction pointer for signals that refer to a faulting instruction (e.g., `SIGILL` and `SIGFPE`). Citadel can be configured to ignore user-controlled signals (e.g., `SIGINT`).

`time_read()`. A challenge to integrity is that the returned time cannot be trusted. An adversary may return a timestamp that is not monotonically increasing and thus cause an underflow when an application calculates a timespan. Citadel therefore checks for monotonicity for `CLOCK_MONOTONIC_*` clock sources.

We note that the official Intel SGX SDK provides *trusted time* for enclaves based on the Intel Management Engine (ME). However, access to the ME is slow and requests can be arbitrarily delayed by an adversary. As a result, Citadel does not currently support trusted time through the ME. However, future TEE implementations may provide practical trustworthy time sources. As such, Citadel could use these to avoid having to delegate time related functionality to the host, thus reducing the size of the host interface.

4.5 Making the host interface oblivious

As explained in Section 4.2.2, an adversary can compromise the confidentiality of an application by observing the execution of host calls. There are three characteristics of host calls that expose side channels: (i) *frequencies* of calls, i.e., their number and time intervals. For example, there may be more host calls when an application processes data on disk, thus revealing information about its execution; (ii) *sequences* of calls, i.e., the order in which host calls are executed and the relationships between them. For example, an application may always execute reads before writes, allowing an adversary to make inferences about the execution; and (iii) *parameters* of calls. Although Citadel encrypts data blocks written to the disk (Section 4.4.1), the block offset reveals which file location is accessed. Observing the same offset repeatedly discloses the data layout on disk.

The main idea to mitigate against these side-channels is to make the host call interface *oblivious* [GO96], i.e., ensuring that the sequence and frequency of calls as well as any observable parameters appear to be workload independent. We focus on the disk I/O host calls and discuss other calls in Section 4.5.4.

4.5.1 Disk I/O calls

Citadel exposes two disk I/O calls to the host: `disk_read()` and `disk_write()`. These read and write fixed-size encrypted blocks at the specified offset (see Section 4.4.1). To prevent these calls from revealing information, Citadel employs several techniques: (i) regarding frequencies, Citadel *discretises* the execution of read/write calls into fixed time-interval rounds. In each round, it executes a fixed number of calls in a *batch*, potentially adding indistinguishable *dummy* calls; (ii) regarding sequences, Citadel makes the order of read/write calls *deterministic* per batch by issuing them in a predefined order, e.g., always executing reads before writes; and (iii) regarding parameters, Citadel ensures that all call parameters appear *random*, obscuring patterns. Citadel makes accesses *oblivious* by using seemingly random *offset* parameters. Repeated accesses to the same disk block appear indistinguishable.

4.5.2 Hiding disk I/O accesses

Citadel discretises the execution of all calls into fixed interval batches. Every t time units, Citadel performs a single `disk_read()` followed by a `disk_write()`. When the LKL filesystem layer (see Section 4.3.2) issues a disk I/O call, instead of submitting it directly to the host, it is added to a host call

queue. Citadel’s `lthread` scheduler checks if enough time has elapsed, and then issues the next call from the queue.

If there are too many read/write calls in the queue, the remaining calls are delayed until the next batch. Conversely, if there are not enough read/write calls from the application, Citadel issues *dummy* calls to random blocks in order to pad that batch. This is done through dummy files in the file system and reading/writing blocks of those files.

The timing parameter t must be tuned for good performance because applications issue system calls at different rates: if t is lower than the application’s call rate, Citadel will issue more batches with dummy calls; if t is too high, the application will make slow progress. In our experiments, we use $t = 0.1$ ms, which works well across a range of applications workloads.

4.5.3 Hiding disk I/O parameters

Although Citadel hides the number of calls, the call parameters (see Table 4.2) still leak information: (i) the data being read or written in `buf`; (ii) the disk `offset`; and (iii) the buffer length `len`. To hide the `buf` parameter, Citadel uses *probabilistic encryption* [GM84b], which re-encrypts data before writing it (see Section 4.4.1). This prevents an adversary from discerning if written data is old or new, thus tracing data movement.

To hide the `offset` parameter, two traces of block access patterns must be indistinguishable from each other. File I/O calls made by the application pass through the file system layer, which can handle them from the LKL page cache, which is opaque to the host, or expose them to the host. The file system layer stores the mapping from logical to physical disk blocks. To hide repeated block accesses from the adversary, Citadel must therefore: (i) randomise the disk layout on application deployment, to hide the initial block mapping from the adversary; and (ii) *shuffle* blocks by changing this logical to physical block mapping in a manner opaque to the adversary.

Existing techniques from oblivious RAM (ORAM) [GO96, SvDS⁺13, WCS15, DvDF⁺16, SGM⁺16, LWZY19] can be used to provide oblivious access to blocks. LKL’s page cache makes it well-suited to employ square root ORAM [GO96]—the page cache is analogous to a “shelter”. As it is private to the enclave, it precludes, however, the need to scan it in its entirety for each access. and allows data to be accessed directly. Once a block has been added to the page cache, subsequent accesses to the same block are served from the page cache, until the page is removed, or flushed to disk. If such evicted blocks are requested again, the adversary could observe this. Therefore, Citadel must *obviously reshuffle* blocks before continuing with execution [GO96].

Algorithm 1: Oblivious ext4 file sytem shuffle

```
Data:   FDs: file descriptors of files to be shuffled
         LP: logical to physical block mapping for file system
Result: LP_n: new logical to physical block mapping for file system
1 for fd in FDs do // find file with largest size
2   | file_blk := num_blocks(fd);
3   | max_blk := max(max_blk, file_blk);
4   | num_shuff_blk := num_shuff_blk + file_blk;
5 end
   /* create donor files to be used for the swap */
6 free_blk := get_num_free_blocks_in_fs();
7 num_donors := free_blk / max_blk;
8 donor_files := init_donors(num_donors);
   /* generate new random permutation of source blocks */
9 src_blks_perm := map_src_blk_to_num(FDs);
10 src_blk_new_perm := FisherYatesShuffle(src_blk_perm);
11 for id in (0 .. num_shuff_blk) do // get fd & block no. for id
12   | fd := fd_for_id(src_blk_new_perm, id);
13   | block_no := blk_no_for_id(src_blk_new_perm, id);
14   | donor = pick_rand_donor(num_donors);
   /* swap source block with donor block */
15   | fetch_source_block(block_no);
16   | ioctl(EXT4_IOC_MOVE_EXT, fd, donors_files[donor], block_no);
17 end
18 unlink_all_files(donor_files);
```

An *oblivious shuffle* moves blocks around so that an adversary cannot correlate the blocks before and after the shuffle. Citadel uses a *k-oblivious shuffle* [PPY18]. This assumes that k of the n source blocks are in the private cache. The algorithm then sequentially moves the n blocks from the source (either from the disk or from the cache) to the destination according to a randomly generated permutation of the blocks.

Algorithm 1 describes the shuffle algorithm. Instead of shuffling the full disk, we assume a list of files that need to be shuffled, which can be obtained from a trace of files opened by the application. We divide the disk blocks into two parts: (i) blocks that belong to files that must be shuffled, i.e., the source array of size n ; and (ii) a destination array of n unallocated blocks.

First, the algorithm determines the size *max_blk* of the largest file to be shuffled (lines 1–4), the number of free blocks (line 6). It uses this information to create a set of donor files (line 8), each of size *max_blk*, to fill up the free blocks. This set of donor files is used for swapping blocks with the source files. Next, the algorithm assigns an ordering to the blocks of the source files that need to be shuffled (line 9), mapping the set of source file blocks to natural numbers. It uses a Fisher-Yates shuffle [Knu98] to produce a random permutation of the source file blocks (line 10). The algorithm then performs an iteration per source block to be shuffled (line 11). At each iteration, it determines the source file descriptor and file block number that maps to the current index according to the new permutation (lines 12–13). If the source block is already in the page cache, an unread source block, if

any, is accessed and brought into the page cache (line 15).

Finally, a random donor file is selected, and the source block is swapped with the block from the donor file (line 16). The algorithm is implemented at the LKL ext4 layer because the shuffled blocks must be decrypted/encrypted by *dm-crypt*, as discussed in Section 4.4.1. The ext4 layer also stores the logical/physical block mapping, which needs to be modified when blocks are swapped. Citadel uses the `EXT4_IOC_MOVE_EXT ioctl()` call for this, which exchanges the blocks belonging to two different files, while also updating the underlying metadata. Once all the block have been swapped, the dummy files are unlinked, and the corresponding blocks become unallocated again.

4.5.4 Other host calls

Network I/O. To hide access patterns in network traffic, noise in the form of dummy network packets can be added to generate a constant stream of fixed size packets sent between participating trusted nodes [VDHLZZ15, TGL⁺17]. To an adversary, regular packets become indistinguishable from dummy packets, which are dropped by receivers. To scale to more clients, traffic could be routed through a trusted proxy, which establishes a single connection with each Citadel enclave.

Signals. Signals caused by the application disclose sensitive information. For example, if an application executes an illegal instruction, the resulting signal is observable. As described in Section 4.4.3, Citadel hides what instruction caused the signal but cannot hide the exception itself. This is a limitation of Intel SGX—future TEE implementation may handle exceptions without host involvement.

Time. The host kernel updates a vDSO memory region with the current time (see Section 4.3.2). It is read by the enclave, which is observable by an adversary. However, the Citadel scheduler accesses time frequently on each context switch, and an adversary cannot distinguish between accesses from Citadel and the application, hiding application-specific access patterns.

4.6 Evaluation

To understand the performance impact of Citadel’s design choices, we evaluate it with real application workloads and microbenchmarks. We conduct all the experiments on SGX-enabled machines with Intel Xeon E3-1280 4-core CPUs with 8 MB LLC and SGX 1 support, 64 GB RAM, and a 10-Gbps network card. The machines run Ubuntu Linux 18.04, Linux Kernel 4.15.0-46, with the Intel SGX driver version 2.5.

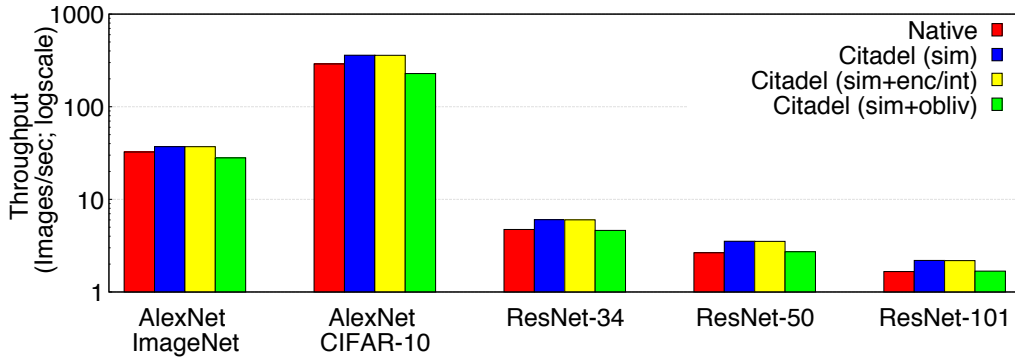


Figure 4.2: Training throughput with TensorFlow

Our SGX version has an EPC size of 128 MB, and only around 90 MB are available to user applications. Since this limit is likely to increase in the future, we evaluate Citadel in both simulation and hardware modes to ignore the overhead due to EPC paging. When using the oblivious host interface, we randomise the disk image and set the page cache size to be large enough for the input data.

4.6.1 Application performance

We evaluate the performance of Citadel with two data-intensive workloads, TensorFlow [ABC⁺16] and PARSEC [BKSL08].

TensorFlow. We use TensorFlow (TF) [ABC⁺16] for training and inference. The models are selected from TF’s benchmark suite [Ten19], representing different type of networks, both small (ResNet-34) and large (AlexNet, ResNet-101) as well as deep and low-dimensional (ResNet-50). The input datasets are CIFAR10 [KNH19] and a subset of ImageNet [Ima19]. We use the default benchmark suite settings. All experiments are executed natively and in Citadel with and without disk encryption and oblivious host calls, respectively. To avoid SGX paging effects, we run in simulation mode with an enclave size of 12.5 GB.

In Figure 4.2, we report the training throughput with Citadel in different modes compared to native execution. Citadel outperforms native in simulation mode: the training throughput of ResNet-34, ResNet-50 and ResNet-101 is higher than native by 28%, 33% and 32%, respectively. Citadel achieves better performance due to LKL’s implementation of system calls as standard function calls. The results also show that the overhead of disk encryption and integrity checking is negligible.

With oblivious host calls with disk encryption, Citadel experiences a slow-down: the training throughput for AlexNet decreases by 24% (ImageNet) and 37% (CIFAR10) compared to simulation mode, or 14% and 21% compared to native execution. The impact of oblivious host calls on the ResNet models is less significant: the training performance of Citadel is almost equal to native. This is due to the

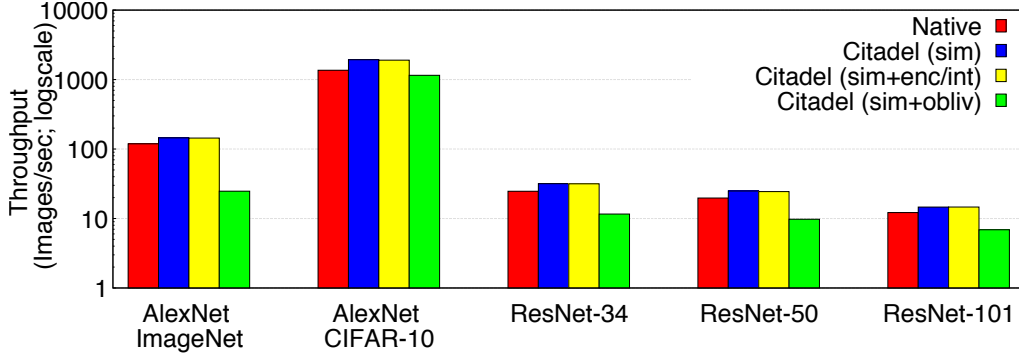


Figure 4.3: Inference throughput with TensorFlow

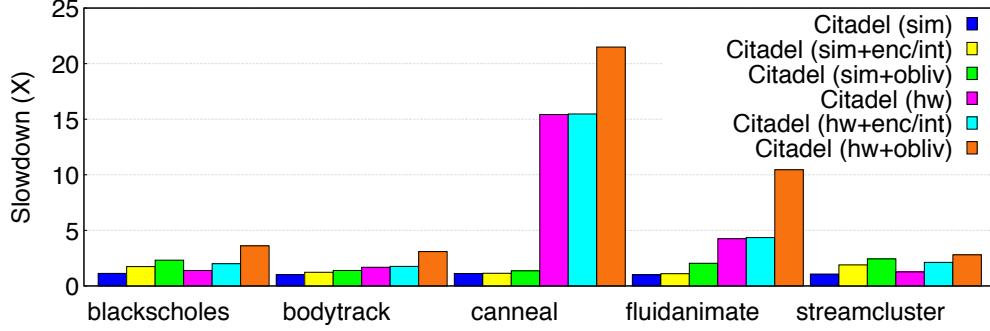


Figure 4.4: Execution overhead for PARSEC workloads

different I/O characteristics of the networks: AlexNet processes more images than ResNet per second, thus stressing the oblivious host calls and requiring more shuffle operations.

In Figure 4.3, we also explore inference throughput with Citadel in different modes compared to native. The results show the same trend as training: the inference throughput is higher than the native system by 22% for AlexNet with ImageNet, 42% for AlexNet with CIFAR10, 29% for ResNet-34, 26% for ResNet 50, and 20% for ResNet 101. The throughput is lower than native with oblivious host calls by 79% (AlexNet with ImageNet), 15% (AlexNet with CIFAR-10), 53% (ResNet-101), 50% (ResNet-50), and 43% (ResNet-34). We conclude that CPU-intensive networks are less affected by the oblivious call interface, while I/O-intensive networks are more affected.

PARSEC. We use five PARSEC benchmarks [Uni19] with different working set sizes to explore the performance of Citadel in SGX hardware mode. All workloads execute with four threads, and the input size is “simlarge”.

Figure 4.4 shows the execution with Citadel with different security guarantees. Without oblivious host calls, Citadel in simulation mode is $1.05\times$ slower than native. The overhead for **streamcluster** is $1.07\times$; **fluidanimate** has the same performance as native; for **canneal** and **blacksholes**, the slowdown is $1.12\times$ and $1.13\times$, respectively. We conclude that, without EPC size limitations, the performance of Citadel without oblivious host calls is comparable to native execution.

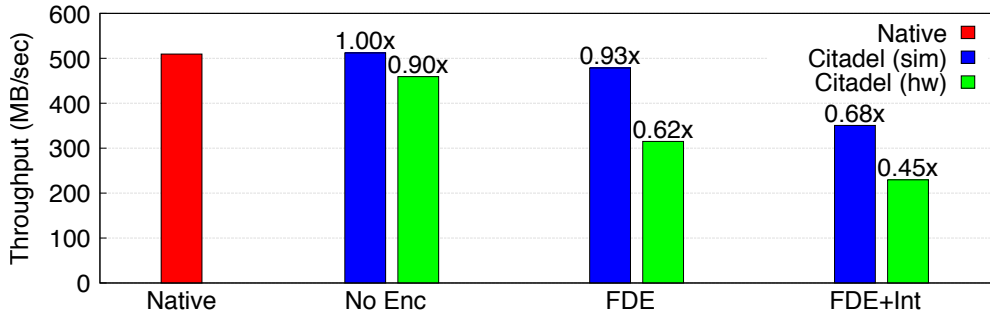


Figure 4.5: Disk performance with encryption and integrity protection

In hardware mode, the overhead of workloads with the small to medium working set sizes is $1.3\times$ (*streamcluster*), $1.4\times$ (*blackscholes*) and $1.7\times$ (*bodytrack*). With large working sets, *fluidanimate* and *canneal* are slower by $4\times$ and $15\times$, respectively, which is due to the limited EPC size.

With oblivious host calls, *streamcluster* shows similar overheads in simulation ($2.4\times$) and hardware modes ($2.8\times$). Since *streamcluster* does not have input files, no shuffling is required, and the overhead solely comes from the dummy calls; for benchmarks with input files, e.g., *blackscholes*, the overhead in hardware mode is higher ($3.6\times$) due to the shuffling. For *fluidanimate* and *canneal*, which have large memory footprints, we observe an overhead of up to $2\times$ in simulation; in hardware mode, the overhead rises to $10\times$ and $21\times$, respectively, due to SGX paging.

In summary, we see that without EPC size limitations: (i) Citadel with disk encryption and integrity protection offers similar performance to native execution; and (ii) the overheads of disk encryption and integrity protection are out-weighed by the cost to perform oblivious host calls. However, when EPC size limitations of SGX are considered, the overheads due to SGX paging are much more significant than those introduced by oblivious host calls. As such, the performance of Citadel for real-world and data-intensive applications are bound by the cost to perform SGX paging.

4.6.2 Disk I/O performance

We evaluate disk I/O performance by measuring the sequential read throughput when reading an uncached 1 GB file from an SSD. We consider different configurations: (i) unencrypted; (ii) with full disk encryption (FDE) via *dm-crypt* using AES-XTS; and (iii) with FDE and integrity protection via *dm-crypt* and *dm-integrity* using AES-GCM.

Figure 4.5 shows that Citadel achieves near native performance without encryption or integrity protection, fully saturating the SSD bandwidth of approx. 510 MB/sec. With FDE, the throughput decreases to 320 MB/sec (62%) of native throughput in hardware mode. Enabling integrity protection further

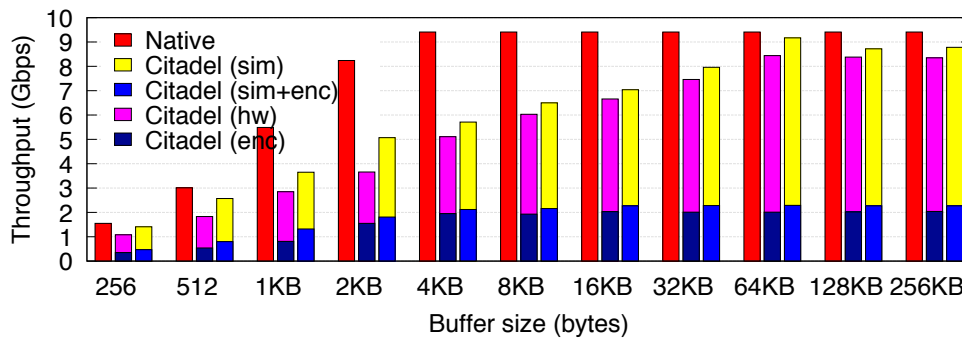


Figure 4.6: Network I/O throughput for different buffer sizes

reduces throughput to around 230 MB/sec (45%).

Given that the overhead of FDE with integrity protection for an uncached 1 GB file on an SSD is only around $2\times$, this shows the benefit of using x86-specific cryptographic instructions.

4.6.3 Network I/O performance

We evaluate network performance by measuring throughput for different buffer sizes with *iperf* version 3.1.3 [IPe19].

As Figure 4.6 shows, for both native and Citadel, throughput increases with larger buffer sizes. For 256 byte packets, Citadel reaches about $0.7\times$ and $0.9\times$ of native throughput in hardware and simulation modes, respectively. For small buffers, the throughput is client-bound. Native execution saturates the network bandwidth of about 9.4 Gbps with 4 KB buffers. Here Citadel reaches a throughput of 5.1 Gbps ($0.5\times$) in hardware mode, and 5.7 Gbps ($0.6\times$) in simulation mode. Due to its support for TCP segmentation offloading, Citadel performs better for larger buffer sizes: with 64 KB buffers, Citadel reaches a maximum throughput of 8.4 Gbps ($0.9\times$) and 8.8 Gbps ($0.9\times$) for hardware and simulation modes, respectively.

With Wireguard and 256 byte buffers, throughput for hardware and simulation mode is 0.44 Gbps and 0.54 Gbps, respectively; with buffers larger than 4 KB, the throughput increases to 2.0 Gbps and 2.2 Gbps, respectively. Here Wireguard decryption becomes the bottleneck.

4.7 Related Work

TEE runtime systems. As mentioned in Section 4.2.2, Haven [BPH14], Graphene-SGX [TPV17], SCONE [ATG⁺16], and Panoply [SLTTS17] are all existing TEE runtime systems. However, in com-

parison to Citadel, each of these runtime systems: (i) expose a wider host interface than Citadel, e.g., Citadel only exposes 7 host interface calls, while each of these systems expose between 24 and 302 host calls; and (ii) do not support an oblivious host interface, i.e., each of these systems expose side-channel information about the application through interface call sequences, frequencies and parameters. Similar to Citadel, SCONE [ATG⁺16] also implements user-level threading inside the enclave to reduce costly enclave transitions of threads. While Ryoan [HZX⁺18] uses SGX enclaves to sandbox data processing and make the data flow independent from the input data, it does not make the host interface oblivious. As such, Citadel is the first TEE runtime system to offer an oblivious host OS interface.

ORAM with TEEs. TEE-based ORAM can conceal memory access patterns from an adversary, e.g., an untrusted host OS. For example, ZeroTrace [SGF18] combines Intel SGX enclaves with Path and Circuit ORAM [SvDS⁺13, WCS15] to offer oblivious memory primitives. OblIDB [EZ17] and Oblix [MPC⁺18] use Path ORAM with Intel SGX to hide the access patterns of SQL queries at runtime. In Raccoon [RLT15], Path ORAM hides array accesses that are data depend on in-enclave secrets, e.g., cryptographic keys. By using ORAM, these approaches can hide memory access patterns. However, in comparison to Citadel, adversaries can still mount Iago attacks [CS13c] when returning data blocks to the enclave. This can violate the integrity of the application.

Oblivious file systems. Oblivate [AKSL18] is an oblivious file system for Intel SGX enclaves. It adapts an ORAM protocol, e.g., Path ORAM, to hide reads and writes to a data file outside the enclave. Privatefs [WST12] is an ORAM-based parallel oblivious file system, supporting concurrent accesses, i.e., reads and writes, to data files. Oladi [CBC⁺18] is a cloud-based key/value store that supports oblivious transactions while protecting file access patterns from cloud providers. Bittau et al. [BEM⁺17] introduce an oblivious architecture for monitoring the behaviour of software clients while protecting user privacy. In contrast to Citadel, all of these approaches do not offer a file system abstraction with POSIX semantics. Moreover, these works do not protect other POSIX related host calls, e.g., network and time host calls, from an untrusted host OS.

Unobservable communication. For message systems, hiding which users communicate is a challenging goal. Anonymous networks such as Tor [DMS04] are susceptible to traffic analysis attacks [HVCT10, MD05]. Dining Cryptographers (DC) networks [Cha88, WCGFJ12, SGRE04] provide stronger guarantees by broadcasting messages to all users, but incur a high overhead. More recently, Private Information Retrieval (PIR) [CGKS95, CGBM15, AS16] and differential privacy techniques [VDHLZZ15, BKM⁺13, TGL⁺17] add noise to hide the metadata. These approaches are orthogonal to the contributions made by Citadel. As such, they can be leveraged by Citadel to blind network communications.

4.8 Limitations

In this section, we briefly discuss the limitations of our approach:

Impact on the TCB. Citadel places a library OS inside the TEE to reduce the functionality required by the host OS. While this helps to achieve a minimal host OS interface (see Section 4.3), it increases the TCB size of the application. This is because the library OS executes alongside the application: if a vulnerability exists in library OS code, e.g., a buffer overflow, this can be exploited by an adversary to compromise the confidentiality or integrity of the application.

Although Citadel aims to reduce the likelihood of software bugs residing in library OS code, i.e., by using a port of the time-tested Linux Kernel for the library OS, Citadel ultimately trades-off the size of the TCB for the minimality of the host interface. This increases the TCB of the application by around 1 million LOCs; a limitation of our approach. We note, however, that: (i) the TCB increase due to LKL is on the same order of magnitude as Graphene, and lower than Haven, i.e., Drawbridge has more than 5 million LOCs; and (ii) this trade-off is unavoidable: the OS functionality must be provided to the application. If the OS functionality is placed outside the TEE, it results in a host interface that is difficult to secure; if the functionality is placed inside the TEE, it increases the TCB of the application (as seen with Citadel).

Assuming large EPC sizes. To evaluate the performance overhead of Citadel, we perform experiments in both SGX simulation and hardware modes. This allows us to examine the effects of SGX paging due to current EPC memory limitations (see Section 4.6). Our results show that when Citadel executes large benchmarks in hardware mode, i.e., with SGX paging, performance exhibits up to a $21\times$ slowdown. Without SGX paging, performance is only around $2\times$ - $4\times$ slower. This illustrates the significant effects paging has on Citadel’s performance. As such, this is a current limitation of our approach. However, we note that: (i) current EPC sizes of Intel SGX enclaves have already increased, and will continue to do so in the future. As such, this will make Citadel more practical for real-world deployments; and (ii) SGX paging affects the performance of all SGX applications with large memory footprints, not just Citadel.

4.9 Summary

In this chapter, we proposed a solution to the second security challenge faced by TEE-based applications: securing TEE boundary interfaces (Section 1.4.2). To do this, we proposed applications

follow three TEE interface design principles: minimality, protection and obliviousness. These principles protect applications from TEE boundary interface attacks and allow applications to interact with untrusted code and data securely.

To illustrate these design principles, we design a TEE boundary interface that secures communication between an application and an untrusted host OS. To do this, we make three contributions: (i) for minimality, we propose applications employ a library OS inside the TEE to reduce the amount of functionality required by the host OS; (ii) for protection, we propose transparently encrypting and integrity protecting all data that crosses the TEE boundary interface; and (iii) for obliviousness, we propose making host OS interface calls discrete and deterministic, to avoid leaking information about application execution.

To evaluate our approach, we implemented these contributions in Citadel, the first TEE runtime system that offers a secure host OS interface for applications. Citadel protects Linux binaries executing inside Intel SGX enclaves from boundary interface attacks. Citadel demonstrates that our approach is practical for real world applications: with emulated SGX enclaves, Citadel trains deep neural networks with an overhead of 14%, and 21% with oblivious calls. With current SGX hardware, Citadel executes benchmarks that fit into SGX memory with an average overhead of 1.5x and 3.1x with and without oblivious calls, respectively.

While Citadel targets Linux applications for Intel SGX enclaves, the interface design principles and contributions offered by this chapter are fundamentally TEE-agnostic: our approach does not rely on any TEE-specific features, but can be applied to a wide range of TEEs and applications.

5

Tolerating TEE Failures and Compromises

TEEs are not a silver bullet: current implementations do not provide availability guarantees and are vulnerable to a wide range of software and hardware-based attacks. As such, developers must design applications to tolerate TEE failures and compromises (see Section 2.4). To achieve this, we propose that applications employ TEE-heterogeneity: the protection of security-critical functionality using multiple heterogeneous TEEs.

We begin this chapter by providing an overview of TEE-heterogeneity (Section 5.1). We then present relevant background on secure blockchain payment networks using TEEs (Section 5.2.2): this includes our goals and threat model for TEEs. Next, we present Teechain: the first blockchain payment network that employs TEE-heterogeneity to tolerate TEE failures and compromises (Sections 5.3 to 5.5). We continue the chapter by offering a detailed performance evaluation of our approach and contributions (Section 5.6). Finally, we discuss related work to Teechain (Section 5.7), highlight limitations of our approach (Section 5.8) and conclude the chapter (Section 5.9).

5.1 Overview

To tolerate TEE failures and compromises, we propose that applications employ TEE-heterogeneity: protecting security-critical functionality using multiple heterogeneous TEEs. This allows applications to: (i) protect against independent TEE failures and compromises, by avoiding centralised application architectures, i.e., deployments where only a single TEE is used; and (ii) protect against global TEE failures and compromises, where an entire class of TEEs fail or are compromised simultaneously, e.g., due to a faulty or vulnerable TEE implementation.

To leverage TEE-heterogeneity, applications must move away from centralised TEE architectures and instead be designed to operate in a decentralised and TEE-agnostic way. This requires overcoming two challenges: (i) to avoid centralised TEE architectures, security-critical functionality in the application must be distributed across multiple heterogeneous TEEs; and (ii) to maintain agreement on the current state of the application, all TEEs must achieve consensus despite possible failures and compromises.

To overcome these challenges, we make two contributions: (i) we propose applications use threshold cryptography to distribute security-critical functionality from a single TEE to multiple heterogeneous TEEs. This allows the application to avoid relying on a single TEE in the deployment; and (ii) we propose force-freeze chain replication: an efficient and practical way to maintain consensus between different TEEs. This allows the state of the application to be agreed upon, despite TEE failures and compromises in the deployment.

To illustrate our contributions, we present **Teechain**, the first TEE-based application that employs TEE-heterogeneity to tolerate TEE failures and compromises. Teechain employs TEE-heterogeneity in the context of a secure payment network for blockchain-based cryptocurrencies: an application that allows parties to execute high-performance cryptocurrency payments asynchronously from a blockchain. As blockchains and cryptocurrency are notorious for being adversarial and untrusted [LJC⁺17, LL17], Teechain implements both of our contributions to protect against TEE failures and compromises in these environments: (i) Teechain uses threshold cryptography to protect cryptocurrency funds by requiring multiple different TEEs to sign any transactions before funds can be spent or accessed; and (ii) Teechain employs force-freeze chain replication between TEEs. This ensures TEEs in the application maintain consensus about the state of the funds and balances.

To evaluate the performance and overhead of our contributions, we implement Teechain using Intel SGX enclaves for the Bitcoin blockchain. Our experiments show that: (i) the overheads introduced by TEE-heterogeneity are practical and reasonable given the fault tolerance benefits; (ii) despite hav-

ing to perform force-freeze chain replication between TEEs, Teechain achieves at least a 33x higher transaction throughput than state-of-the-art payment networks; and (iii) a 30-machine Teechain deployment can handle over 1 million Bitcoin transactions per second. Thus, Teechain demonstrates that TEE-heterogeneity is a general and practical approach to securing applications against TEE failures and compromises; even in adversarial settings, such as blockchains and cryptocurrencies.

5.2 Background: TEEs for secure blockchain payment networks

To tolerate TEE failures and compromises, Teechain employs TEE-heterogeneity in the context of a blockchain payment network. While existing payment networks avoid using TEEs entirely, they face several significant security limitations. As such, in this section, we justify why Teechain combines TEEs with payment networks, and explore the security benefits of this approach. We then discuss why these TEEs must be protected to maintain the security guarantees of the payment network. This highlights the need for fault tolerance against TEE failures and compromises.

We begin this section by providing general background on blockchains (Section 5.2.1) and payment networks (Section 5.2.2). Next, we discuss the limitations of current payment networks (Section 5.2.3), and explore how TEEs can overcome these limitations (Section 5.2.4). Finally, we present our threat model and justify the need to tolerate TEE failures and compromises (Section 5.2.5).

5.2.1 Blockchains and cryptocurrencies

Blockchain-based cryptocurrencies, such as Bitcoin [Nak08], Ethereum [The17b] and Zerocash [SCG⁺14], allow mutually distrusting parties to pay each other without trusting a third party or central authority. Instead, in a blockchain-based cryptocurrency, a set of nodes connect over a peer-to-peer network to operate as a replicated state machine. This state machine maintains an append-only *ledger* that contains the history of all transactions in the system, where each transaction is a payment from one system user, i.e., a *party*, to another, secured cryptographically. This ledger is a chain of *blocks*, i.e., forming a *blockchain*, such that each block contains a list of transactions.

Each transaction in the blockchain is a list of instructions that update the state of the blockchain. Different cryptocurrencies implement payment transactions that move funds differently: Bitcoin [Nak08] follows an *unspent-transaction-output* (UTXO) model in which transactions consume, or use as *input*, a set of previously unspent transactions, where the *output* of those transactions are owned by the sender. A payment transaction therefore consumes unspent input transactions and generates new out-

put transactions that recipients can spend; Ethereum [The17b] uses an *account model* in which a user's account balance is represented as an integer stored on the blockchain and updated by transactions.

Users in a blockchain are represented by cryptographic public keys. A user's transaction is validated with a cryptographic signature produced by the matching private key. To prevent users from *double-spending*, i.e., signing multiple transactions that spend the same funds, blockchains enforce that funds can only be spent once by making double-spending transactions *conflict*: only one transaction in a set of conflicting transactions can be written to the blockchain. Transactions may also support more elaborate conditions such as *m-out-of-n multi-signatures* that require signing by multiple users: such transactions must be signed by any m keys from a set of n keys.

In blockchains, nodes must agree on the order of transactions, i.e., they must reach consensus. The details of blockchain consensus are immaterial to this work—we treat consensus as a black box. Consensus, however, limits transaction throughput [Vuk15] and incurs high storage costs. In Bitcoin, global throughput is limited to 7 transactions per second [Nak08], and the total size of the blockchain is 100s of GBs [DW15]. Due to consensus, transactions may also take arbitrarily long to be written to the blockchain, e.g., minutes or even days [blo18].

5.2.2 Payment networks and channels

Payment networks [MMSK⁺17], such as Lightning [PD16] and Raiden [The17c], try to overcome the performance limitations of blockchains by allowing parties to exchange funds directly, *off-chain*. To execute a transaction, they establish a point-to-point *payment channel* [LEPS16, MBKM17, PB17, DW15, PD16]. A payment channel is a protocol between two parties, A and B , that updates their balances directly through message exchange. When a payment channel is closed, the payment network *settles* the channel by writing the final balances of A and B back to the blockchain using a *settlement* transaction. Since payment networks do not write to the blockchain for each transaction, their transaction throughput is higher and latencies lower compared to on-chain payments [PD16]. Payment networks also reduce the number of transactions stored on the blockchain because only final balances are recorded [PD16, The17c].

To establish a payment channel c , as shown in Figure 5.1, one or both of A and B write *fund deposit* transactions to the blockchain. These place funds into a 2-out-of-2 *multi-signature* account [The17a] owned by both parties, and requires both A and B to cryptographically sign any transaction in order to spend the funds. A creates a fund deposit d of \$1000 for c (step ❶). Using the fund deposits, A and B can then execute payment transactions: a new payment transaction is generated and signed

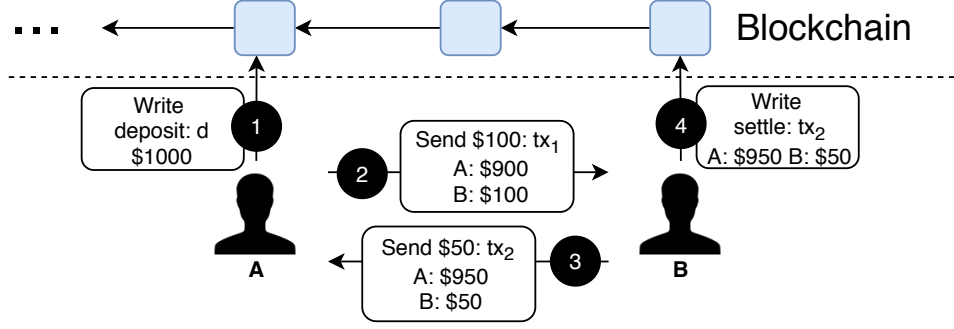


Figure 5.1: **A payment channel in operation** (Here, we see: (1) funds are deposited into the payment channel; (2) A pays B using the channel; (3) B pays A using the channel; and (4) the channel is settled writing the final balances to the blockchain.)

by both parties, spending from the channel deposits and reflecting the new balances. For example, A pays B \$100 using tx_1 signed by both A and B (step ②), and B , whose balance is now \$100, sends A \$50 using tx_2 , also signed by both parties (step ③). Note that tx_1 and tx_2 do not require interaction with the blockchain and that each payment takes into account all previous payments and updates the current state of the payment channel. At any time, either A or B may close the channel by writing the most recent payment transaction to the blockchain: B settles the channel by writing tx_2 to the blockchain with their final balances (step ④).

Payment networks also support *multi-hop* payments [PD16, MMSK⁺17, MBKM17] in which multiple payment channels, c_1 to c_n , are concatenated to form a payment path. This allows for payments between parties that do not have a direct payment channel. This makes payment networks useful in practice, because it allows payments between parties without long-lived financial relationships, e.g., e-commerce buyers and sellers who conduct transactions via intermediaries [ACM17] such as Amazon [Ama19] and eBay [Eba19]. Similar to a single payment channel, any party along the path can unilaterally settle its channels. The added guarantee is atomicity: either all channels c_1 to c_n are settled at the state after the multi-hop payment, or all settle before it.

5.2.3 Limitations of payment networks

To avoid fund theft or loss, payment networks must only settle channels with the most recent payment transaction; otherwise a malicious party can launch a *roll-back* attack in which they settle the channel at a previous payment transaction with a stale balance. For example, in Figure 5.1, step ④, if B settled c using tx_1 instead of tx_2 it would allow B to steal \$50 from A .

Existing payment networks [PD16, DW15, PB17] overcome this problem by requiring parties to detect misbehaviour using information available on the blockchain: when using a payment channel, each party

monitors the blockchain for a settlement transaction written by its counterparty to settle the channel. If an old settlement transaction is written, the party negates its effect by writing the most up-to-date settlement transaction to the blockchain within a bounded *reaction time* Δ .

For this mechanism to work, the payment network must assume that parties can read and write transactions on the blockchain within the fixed upper bound Δ . We refer to this assumption as *synchronous blockchain access*.

In practice, it is not always possible to ensure synchronous blockchain access during payment channel operation. The load on the blockchain may result in long queues to write transactions [blo18]. Moreover, an attacker may delay transaction writes deliberately, such as by controlling the order in which transactions are written [Jos17, Jor15, sup17], or censoring transactions [SEC18, JP 17, Dwa16]. Attackers may also partition victims from the network [MHG18], preventing them from accessing the blockchain at all. Current payment networks therefore face a trade-off when selecting the reaction time Δ : a short Δ allows for quick settlement but facilitates the above attacks.

The requirement for synchronous blockchain access also prevents parties from using payment channels when they are *disconnected* from the blockchain. This negates one of the benefits of payment networks: parties can no longer exchange payments directly with only point-to-point network connections. For example, it becomes impossible to use a payment channel between two devices that are directly connected, but do not have connectivity to the Internet and thus the rest of the blockchain [Dry16].

5.2.4 TEEs as a root-of-trust

The requirement for synchronous blockchain access in existing payment networks comes from the fact that their protocols use the blockchain as a root-of-trust: parties executing the payment protocol monitor the blockchain to discover when other parties deviate from the protocol, and react appropriately (see Section 5.2.3).

To overcome this, we propose introducing a separate root-of-trust that, independently of the blockchain, ensures the faithful execution of a payment protocol. Our idea is for the payment network to use TEEs [Int14, KPW16] during the execution of a payment protocol. As discussed in Section 2.3, TEEs are encrypted and integrity-protected memory regions, which are isolated by the CPU hardware from the rest of the software stack, including higher privileged system software. By using TEEs as an independent root-of-trust, we aim to only require *asynchronous blockchain access*, i.e., the payment protocol must not assume that transaction reads and writes to the blockchain complete within a fixed upper bound, but only complete *eventually*.

However, while TEEs may offer an independent root-of-trust outside the blockchain, it is clear from the discussions in Section 2.4 that current TEE implementations may fail or be compromised by adversaries, e.g., through software or hardware based attacks. As such, it is naive to assume that a single TEE can provide security in all deployments. Given that blockchains and cryptocurrencies are particularly adversarial and untrusted environments for applications [LJC⁺17, LL17], our design must therefore provide fault tolerance against TEE failures and compromises. This is to ensure that the TEEs in the payment network can remain secure and independent roots-of-trust; otherwise, adversaries will target the TEEs in these environments to violate fund security.

We define the security of funds in the payment network in terms of *balance security*: at any time during the payment protocol execution, each party should be able to perform a finite set of actions that eventually results in them receiving their *perceived balance* on the underlying blockchain. A party's perceived balance is their initial balance on the blockchain plus any payments received in the payment network, minus any payments made.

5.2.5 Threat model

We assume that mutually distrusting parties use a blockchain to exchange funds and that their machines have TEEs. Parties trust their own machines, including the hardware and software, but distrust the machines of others [GMSR⁺19]. We assume that TEEs on machines are normally trustworthy, but that a subset of TEEs may suffer failures, e.g., accidental or deliberate crashes, and arbitrary integrity and confidentiality compromises, e.g., through software and hardware attacks on the TEE. We assume that TEEs may be compromised by other parties in the blockchain payment network, or by external adversaries who want to violate security.

We further assume that parties are rational, selfish and potentially malicious, i.e., they may attempt to steal funds and misbehave, if it benefits them. We also assume that parties may collude with one another. Parties are connected via a network, with some behind firewalls or network address translation (NAT). Parties may drop, modify and replay messages. An attacker may delay or prevent others from accessing the blockchain for an unbounded amount of time, but we assume this cannot occur indefinitely. Likewise, an attacker may prevent communication between parties for an unbounded amount of time, but we also assume this cannot occur indefinitely.

5.3 Teechain design

To satisfy the goals and threat model defined in Section 5.2, Teechain’s design must be fault tolerant against TEE failures and compromises. As such, it should avoid a centralized architecture that employs only a single TEE for security. Instead, Teechain should employ TEE-heterogeneity: leveraging multiple different TEEs to protect the security of funds in the payment network.

To achieve this, Teechain implements our contributions defined in Section 5.1: (i) Teechain leverages threshold cryptography to introduce *treasury committees*: sets of heterogeneous TEEs, called *treasuries*, that jointly own funds together. In each treasury committee, a threshold number of treasuries in the committee must agree before funds can be accessed. This allows Teechain to protect against fund loss and theft as a result of TEE failures and compromises; and (ii) Teechain implements force-freeze chain replication within each committee. This ensures treasuries in the committee agree on the state of funds and balances, despite possible failures and compromises. Thus, maintaining consensus between TEEs.

To explore how treasury committees and force-freeze chain replication offer TEE fault tolerance in a payment network, we present a high-level overview of Teechain’s design (Section 5.3.1). We then present treasuries in more detail (Section 5.3.2), followed by committees (Section 5.3.3). Finally, we analyse how Teechain handles different threats and attacks on TEEs (Section 5.3.4).

5.3.1 Design overview

Figure 5.2 shows the design of Teechain. Teechain constructs a peer-to-peer payment network in which parties operate Teechain *nodes*, e.g., node n_A is operated by party A . Each node comprises: (i) an API for parties to interact with the payment network; (ii) an interface through which to read and write blockchain transactions; and (iii) a TEE-protected *treasury* that securely holds and manages parties’ funds.

Treasuries ensure the faithful execution of the payment protocol. They are external to the blockchain and manage payment channels, execute payment transactions and control the access to funds. To avoid blindly trusting treasuries to behave honestly, Teechain uses TEEs to ensure the confidentiality and integrity of treasuries. By using TEEs, Teechain achieves asynchronous blockchain access because treasuries operate correctly, autonomously and protect against misbehaviour by parties without having to interact with the blockchain.

As current TEE implementations may suffer from confidentiality, integrity and availability failures

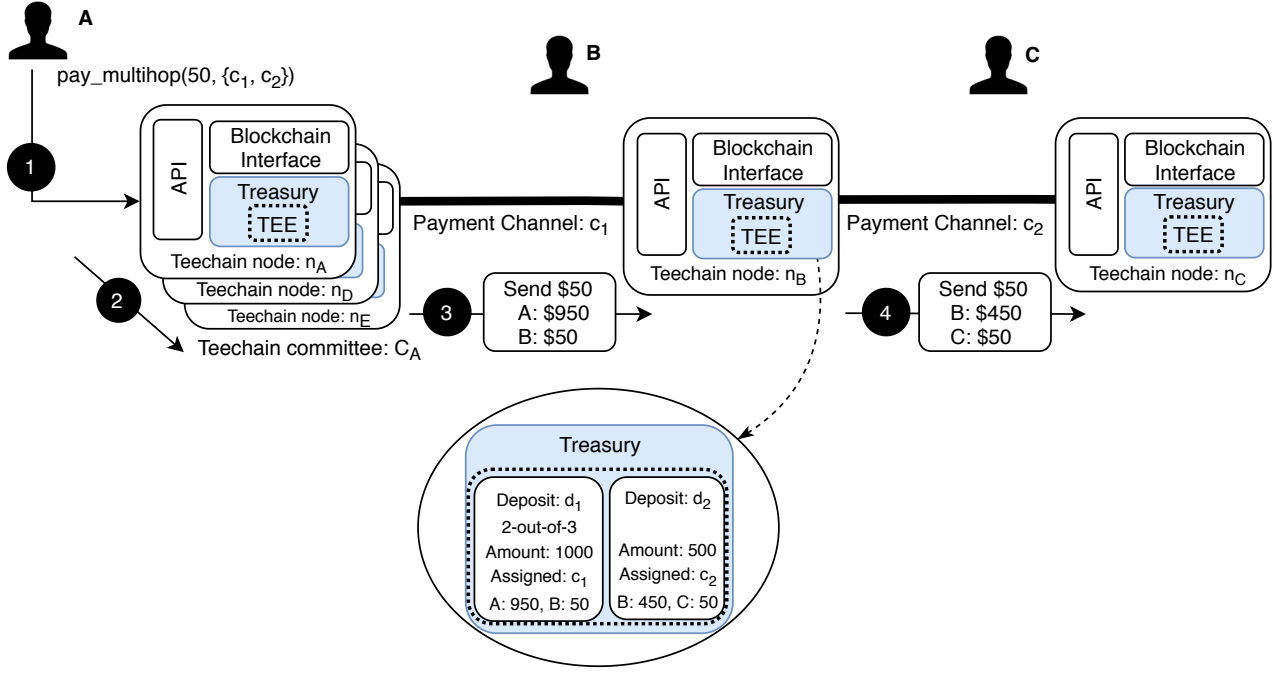


Figure 5.2: **The Teechain overview** (Teechain nodes operate *treasuries* to store and manage funds. Users construct *payment channels* between nodes to exchange funds directly, and execute *multi-hop* payments along concatenated payment channels. *Committee chains* with multiple treasuries replicate and protect state.)

(see Section 2.4), Teechain avoids trusting individual TEEs for security. Instead, Teechain operates *committees* of treasuries: these are groups of treasuries that use threshold cryptography to manage a single collection of funds together. Figure 5.2 shows a committee C_A that constitutes the treasuries at nodes n_A , n_D and n_E . Within each committee, a treasury must have approval from a subset of other treasuries to make an off-chain transaction or settle a payment channel. By distributing the responsibilities of fund management to multiple treasuries, Teechain avoids fund loss or theft if a subset of treasuries in the committee are compromised, or have failed.

Table 5.1 shows the API that Teechain provides to parties. It supports (i) creating deposits (section 5.4.1), (ii) operating payment channels (section 5.4.2) and (iii) constructing committees (section 5.5). Teechain generates unique identifiers for each deposit and channel, e.g., when a deposit is created (`new_deposit`), a unique identifier is returned as a handle to be used in subsequent API calls. Treasuries are identified through unique public keys.

To execute payments, Teechain forms *payment channels* between nodes with network connectivity. Treasuries communicate via these channels to update payment balances. Figure 5.2 shows channel c_1 between A and B ; and c_2 between B and C .

Multi-hop payments can be executed along payment channel paths. In Figure 5.2, a payment from

Teechain API	Inputs	Outputs	API Description
Deposits (section 5.4.1)			
new_deposit	tx, pub ₁ . . . pub _n	d_id	Creates a new fund deposit (d_id) using a transaction (tx) and a set of treasury public keys
release_deposit	d_id	tx	Refunds an unassociated fund deposit (d_id) by generating and returning a transaction (tx)
approve_deposit	d_id, pub	T ⊥	Requests approval for a deposit (d_id) from a specific treasury (pub)
Payment channels (section 5.4.2)			
new_pay_channel	pub	c_id	Creates a new payment channel (c_id) with a given treasury (pub)
associate_deposit	d_id, c_id	T ⊥	Associates an approved fund deposit (d_id) with an existing payment channel (c_id)
dissociate_deposit	d_id, c_id	T ⊥	Dissociates a previously associated fund deposit (d_id) from a payment channel (c_id)
Payments (section 5.4.2)			
pay_channel	val, c_id	T ⊥	Pays an amount (val) to the other user in a payment channel (c_id)
pay_multihop	val, c_id ₁ . . . c_id _n	T ⊥	Executes a multi-hop payment of an amount (val) along a given path of payment channels
Settlement (section 5.4.2)			
settle_channel	c_id	tx	Settles a payment channel (c_id) by generating a settlement transaction (tx)
eject_multihop	c_id	tx	Settles a payment channel (c_id) during the execution of a multi-hop payment
eject_multihop popt	c_id, popt	tx	Settles a payment channel (c_id) using a <i>PoPT</i> (popt) during a multi-hop payment
Chain replication (section 5.5)			
assign_comm_chain	pub	T ⊥	Assigns this treasury to a committee chain by joining the last treasury (pub) in the chain

Table 5.1: **The Teechain API** (The API allows users to manage fund deposits, create payment channels, execute payments, perform payment channel settlement, and leverage chain replication.)

A to C is executed: A invokes the API to execute a multi-hop payment of \$50 along path c_1 – c_2 to C (step ❶); node n_A notifies the treasuries of its committee of the upcoming balance update (step ❷); the treasuries for nodes n_A and n_B update the balances of A and B in c_1 (step ❸); and the treasuries for nodes n_B and n_C update the balances of B and C in c_2 (step ❹). The final state is that A ’s balance has been deducted \$50 in c_1 , B ’s balance incremented by \$50 in c_1 and decremented by \$50 in c_2 , and C ’s balance incremented by \$50 in c_2 .

5.3.2 Treasuries

To achieve asynchronous blockchain access, Teechain employs TEE-based treasuries as an independent root-of-trust (see Section 5.2.4). Treasuries generate public/private key pairs for *treasury addresses*, which are cryptocurrency addresses owned exclusively by a treasury. They are generated securely inside each TEE, and their private keys are stored in TEE memory.

Parties can send funds to these addresses in the form of fund *deposits*. A call to `new_deposit` from

Table 5.1 creates a deposit. It requires a deposit transaction tx , which sends funds to a set of treasury addresses, identified by the treasury public keys, $\text{pub}_1 \dots \text{pub}_n$. In Figure 5.2, deposit d_2 sends \$500 to the treasury at node n_B . Deposits can be associated by a treasury with a payment channel, thus incrementing the balance of that party in the channel. Figure 5.2 shows two deposits registered with the treasury of node n_B : d_1 of \$1000 assigned to channel c_1 ; d_2 of \$500 assigned to channel c_2 .

Parties must verify the integrity of treasuries before trusting them with funds; Teechain uses the remote attestation support of TEEs for verification [Joh16, Int16b]. A TEE (i) measures the treasury code; (ii) cryptographically signs the measurement and the treasury’s public key; and (iii) provides the signed measurement and public key to the remote party. The remote party then verifies the attestation, i.e., the remote party ensures that the attestation is correctly signed by the TEE hardware and that the measurement corresponds to a known treasury implementation. Parties can thus verify that a specific treasury, identified by its public key, is operating genuine TEE hardware.

Users without a TEE-enabled node of their own can use a remote node to manage their funds through *treasury outsourcing*. For this, the party attests a remote treasury and provisions it with a secret key, giving it the same abilities as a local party. To avoid having to trust a single remote treasury, Teechain constructs committees with multiple remote treasuries (Section 5.3.3).

5.3.3 Treasury committees

To avoid having to rely on a single treasury for fund security, committees are groups of treasuries that jointly manage fund deposits together. For each deposit owned by a committee, a minimum number of committee members are required to sign transactions before that deposit can be spent, thus tolerating a fixed number of TEE failures and compromises. For this, Teechain employs threshold cryptography through the multi-signature support provided by the blockchain [The17a]: each fund deposit is paid into an *m-out-of-n* treasury address, where m treasury signatures are required to spend the deposit. The n committee members correspond to the n public keys provided to `new_deposit` in Table 5.1, when the deposit is created. By employing treasury committees, Teechain offers security against $m - 1$ TEE compromises, and $n - m$ TEE failures.

All committee members must agree on the proportion of each deposit owned by the parties in a payment channel. To achieve agreement, Teechain uses a variant of *chain replication* [VRS04b], which offers strong consistency without requiring all nodes to communicate directly. This is beneficial in a payment network because parties may not have direct connectivity due to network address translation (NAT) and firewalls.

With chain replication, Teechain must prevent *roll-back* and state *forking* attacks [BCLK17] in which an attacker partitions the committee members into disjoint subgroups that can settle a channel at different balances using different deposit states. Forking a treasury committee in this way would allow attackers to roll-back to old payment states to steal funds. To overcome this, Teechain employs *force-freeze chain replication*: if any committee member fails or refuses to update to the latest agreed upon state, the replication chain is broken, freezing all nodes at the current state. This prevents future state updates and requires that all channels are settled and unused deposits released. Thus, maintaining consensus between the committee members. We describe force-freeze replication in more detail in Section 5.5.

5.3.4 Threat analysis

In this section, we analyse the various threats posed to Teechain’s design, and discuss how treasury committees and force-freeze chain replication protect against these threats.

Malicious parties. Teechain assumes parties are rational and selfish, i.e., parties behave in their best financial interest (section 5.2.5). We consider two possible cases: (i) A is a malicious local party; and (ii) B is a malicious remote party. In the case of a local malicious party A , Teechain requires parties to encrypt and sign all API calls made to a local (or outsourced) treasury (table 5.1). A only has access to their own funds but cannot affect other funds, as enforced by the local treasury.

In the case of a malicious remote party B who wishes to steal A ’s funds, B must either interact with the Teechain API to force a protocol deviation or drop/replay/modify messages on the network. Teechain secures funds with treasuries and uses TEEs to ensure faithful protocol execution. Treasuries use encrypted, authenticated and freshness-protected messages.

Compromised treasuries. Current TEE implementations are vulnerable to attacks, e.g., through side-channels [VBMW⁺18, BMD⁺17b, MIE17a], and Teechain assumes that treasury compromises are possible (see Section 5.2.5). We therefore consider two cases of a compromised treasury T , which wishes to attack A : (i) T_L is a local treasury that A interacts with directly (i.e., the treasury at node n_A in fig. 5.2); and (ii) T_R is a remote treasury on another node in the Teechain network.

A compromised local treasury T_L cannot steal A ’s funds due to Teechain’s m -out-of- n committees for deposits. To steal a deposit, T_L would need to compromise $m - 1$ treasuries in the committee. To prevent T_L from deceiving A when interacting with the Teechain API, Teechain requires the results of API calls to be signed by all n committee treasuries, except when an API call returns a blockchain

transaction, which only requires m signatures. If T_L fails to coordinate correctly with the committee, A settles channels and returns deposits.

Mitigating a compromised remote treasury T_R is similar to the local case above: committees protect deposits, and thus T_R needs to compromise $m - 1$ other treasuries. Note that, although the requirement for n signatures on API calls means that T_R can force channel settlements, it does not gain financially from this. Similar to prior work [BDW17], Teechain assumes committee members are paid fees for participation.

Global TEE compromises. To mitigate global TEE compromises, in which many treasuries are compromised simultaneously, Teechain is designed to be TEE-agnostic, thus avoiding dependencies on a single TEE implementation. This allows parties in the network to protect deposits using committees of *heterogeneous* TEEs. Under global TEE compromises, e.g., when a specific TEE vendor leaks hardware private keys or a batch of TEEs are found to be faulty, parties can lower their risk by including sufficiently heterogeneous TEEs in their committee chains.

Compromises to the attestation mechanism of a particular TEE implementation, e.g., as done by the Foreshadow [VBMW⁺18] attack against the Intel attestation service, do not affect funds held by committees. As described in Section 5.3.2, remote attestation ensures that a specific treasury, identified by its public key, operates genuine TEE hardware. Even if remote attestation has been compromised, an attacker can only create new malicious treasuries, but cannot spoof other treasuries or committee members in the network. To steal funds, an attacker would need to bias the selection of future committee members. We discuss committee member selection in Section 5.5.2.

5.4 Payment Protocol

As discussed in Section 5.2.4, Teechain leverages TEEs as an independent root-of-trust to offer asynchronous blockchain access for payment networks. To achieve this, Teechain must provide payment protocols that ensure balance security regardless of blockchain access times: treasuries in the payment network cannot assume that transaction reads and writes to the blockchain complete within a fixed upper bound. Rather, these operations only complete eventually (see Section 5.2.5).

This section presents the payment protocols offered by Teechain to ensure balance security under asynchronous blockchain access. We begin the section by describing deposit allocation in Teechain (Section 5.4.1). We then present the payment channel protocol (Section 5.4.2) and multi-hop payment protocol (Section 5.4.3). Finally, we sketch Teechain’s security proofs (Section 5.4.4).

Algorithm 2: Teechain payment protocol executed by the treasury at each node
 (Based on the API shown in Table 5.1. For brevity, we omit the collection of committee member signatures at the end of each API call (see Section 5.3.4).)

```

1  on new_deposit(tx, pub1...pubn):
2    verify_tx(tx, pub1...pubn)
3    d ← create_new_deposit(tx)
4    deposits[d.id] ← d /* store dep */
5    write_to_blockchain(tx)
6    return d.id /* return deposit id */

7  on release_deposit(d_id):
8    d ← deposits[d_id]
9    assert(d.chan = ∅) /* unassoc */
10   tx ← gen_deposit_refund(d)
11   deposits[d.id] ← ∅ /* clear dep */
12   write_to_blockchain(tx)
13   return tx /* return refund */

14 on approve_deposit(d_id, pub):
15   d ← deposits[d_id]
16   appr ← ask_approve_remote(d, pub)
17   d.apprv[pub] ← appr
18   return appr /* return approval */

19 on new_pay_channel(pub):
20   c ← create_channel_with(pub)
21   (c.my_bal, c.rem_bal) ← (0, 0)
22   channels[c.id] ← c
23   return c.id /* return channel id */

24 on associate_deposit(d_id, c_id):
25   d ← deposits[d_id]
26   c ← channels[c_id]
27   assert(d.chan = ∅) /* unassoc */
28   assert(d.apprv[c.pub])
29   d.chan ← c /* add assoc */
30   c.my_bal ← c.my_bal + d.val
31   send_assoc_to_remote(d, c)

32 on dissociate_deposit(d_id, c_id):
33   d ← deposits[d_id]
34   c ← channels[c_id]
35   assert(d.chan = c)
36   send_dissoc_to_remote(d, c)
37   d.chan ← ∅ /* remove assoc */
38   c.my_bal ← c.my_bal - d.val

39 on pay_channel(val, c_id):
40   c ← channels[c_id]
41   assert(c.my_bal ≥ val)
42   c.my_bal ← c.my_bal - val
43   c.rem_bal ← c.rem_bal + val
44   send_pay_to_remote(c, val)

45 on settle_channel(c_id):
46   c ← channels[c_id]
47   if neutral_balance(c) then
48     /* terminate off-chain */
49     dissociate_all_deposits(c);
50     channels[c.id] ← ∅
51     return ∅
52   else
53     /* terminate on-chain */
54     tx ← get_settle_for_bals(c)
55     send_settle_to_remote(c, tx)
56     channels[c.id] ← ∅
57     write_to_blockchain(tx)
58     return tx

59 on pay_multihop(val, c_id1...c_idn):
60   c1 ← channels[c_id1]
61   ...
62   cn ← channels[c_idn]
63   lock(val, c1, ..., cn) /* Alg.2 */
64   wait_for_unlock()
65   return T /* payment done */

66 on eject_multihop(c_id):
67   c ← channels[c_id]
68   s ← c.state
69   if s = lock ∨ s = sign ∨
70      s = postpayment ∨
71      s = unlock then
72     return settle_channel(c_id)
73   return c.τ /* settle all */

74 on eject_multihop(c_id, popt):
75   s ← popt.state
76   if s = lock ∨ s = sign then
77     return settle_prepay(c_id)
78   if s = postpayment ∨
79      s = unlock then
80     return settle_postpay(c_id)

```

5.4.1 Allocating dynamic deposits

Deposits can be created at any time and associated/dissociated with payment channels dynamically. Algorithm 2 shows the protocol executed by treasuries for the API calls from Table 5.1.

To construct a new deposit d , parties invoke `new_deposit` (Algorithm 2, line 1) and present a deposit transaction tx and the list of treasury public keys forming the committee that tx sends funds to. The treasury then verifies that tx sends funds to an m -out-of- n multi-signature address using the committee members' public keys, $\text{pub}_1 \dots \text{pub}_n$, and notifies the committee of the new tx (see Section 5.5). The treasury then constructs a new deposit d , forwards tx to the blockchain, and returns d 's unique identifier to the requester (line 6), signed by all committee members—we omit signature collection for brevity.

A payment channel c may contain zero or more deposits through deposit association. The sum of the deposits associated with c must be equal to the sum of the balances of A and B in c , i.e., deposits are distributed to A and B . Before a deposit d can be associated with c , it must be approved by the remote party in c (e.g., B if A requests approval) using `approve_deposit` (line 14). Approval contacts the remote

party via its treasury and queries if the deposit is eligible for association with c . Deposit approval therefore allows A and B to define which deposits can be associated with c . Due to our assumption of asynchronous blockchain access, this may take unbounded time. Deposits need to be approved only once.

Approved deposits can be *associated* with a single channel using `associate_deposit`, and *dissociated* using `dissociate_deposit` (lines 24 and 32). When deposit d is associated with c by A , the treasuries increase A 's balance by the deposit amount (lines 30 and 31); dissociation decrements A 's balance (lines 36 and 38). Disassociation can only be done if the participant's balance is greater than or equal to the deposit amount. *Unassociated* deposits are deposits not associated with any channel. They can be returned upon request through `release_deposit` (line 7): a new transaction tx is generated and signed by the appropriate committee treasuries, and written to the blockchain; d is then removed from the treasury.

5.4.2 Using payment channels

To create payment channels between treasuries without blockchain interaction, participants call `new_pay_channel` and provide the public key of the treasury with which to create the channel (Algorithm 2, line 19). The two treasuries then establish a secure communication channel using authenticated Diffie-Hellman [Kra03] for key provisioning and remote attestation. Using the secure channel, the treasuries assign a unique channel identifier to the channel c , initialize both participant's balances to 0, and return the channel identifier.

To execute a payment along a channel, the sender calls `pay_channel` (line 39), which specifies the amount to send and the channel identifier. The sender's treasury first ensures that the sender has sufficient funds before decrementing the sender's balance and incrementing the recipient's balance locally (lines 42 and 43). It then forwards the payment to the recipient's treasury to update balances. If the payment is not received by the recipient, e.g., due to a network failure, the sender settles the channel and writes the balances to the blockchain to allow the remote party to see the final state of the channel. This prevents balance inconsistencies.

As deposits can only be associated with a single channel, participants may suffer from *deposit lock-in*: when a large deposit is added to a channel but only a small fraction is spent, it leaves the remaining locked-in until the channel is settled. In a channel c with deposit d_x of amount a_x , after payments of value p_x have been made, the locked-in funds f_x are $a_x - p_x$. If f_x is large, there is a high fund lock-in. To avoid this, participants can perform *deposit rebalancing*: they associate another deposit d_y

Algorithm 3: Teechain multi-hop payment protocol (For brevity, we omit the messages exchanged between treasuries after each step, i.e., the messages in Figure 5.3. Payment channels in the path are denoted: $c_1 \dots c_n$. Treasuries in the path are numbered $1 \dots n+1$. pos denotes a treasuries' position.)

```

1  on ❶ lock(val, c1, ..., cn):
2    if pos ≤ n then
3      assert(cpos.my_bal ≥ val)
4      lock_channel(cpos, val)
5    if pos > 1 then
6      lock_channel(cpos-1, val)
7  on lock_channel(c, val):
8    (c.state, c.val) ← (lock, val)
9  on ❷ sign( $\bar{r}$ , c1, ..., cn):
10   if pos > 1 then
11     sign_channel(cpos-1,  $\bar{r}$ )
12   if pos ≤ n then
13     sign_channel(cpos,  $\bar{r}$ )
14  on sign_channel(c,  $\bar{r}$ ):
15    $\bar{r} \leftarrow \text{add\_chan\_settle\_post}(\bar{r}, c)$ 
16   c.state ← sign
17  on ❸ pre( $\bar{r}$ , c1, ..., cn):
18   if pos ≤ n then
19     pre_channel(cpos,  $\bar{r}$ )
20   if i > 1 then
21     pre_channel(cpos-1,  $\bar{r}$ )
22  on pre_channel(c,  $\bar{r}$ ):
23   c. $\bar{r} \leftarrow \bar{r}$  /* store  $\bar{r}$  for if settle */
24   c.state ← prepayment
25  on ❹ inter(c1, ..., cn):
26   if pos > 1 then
27     increase_my_bal(cpos-1)
28   if pos ≤ n then
29     decrease_my_bal(cpos)
30  on increase_my_bal(c):
31   c.my_bal ← c.my_bal + c.val
32   c.rem_bal ← c.rem_bal - c.val
33   c.state ← inter
34  on decrease_my_bal(c):
35   c.my_bal ← c.my_bal - c.val
36   c.rem_bal ← c.rem_bal + c.val
37   c.state ← inter
38  on ❺ post(c1, ..., cn):
39   if pos ≤ n then
40     post_channel(cpos)
41   if pos > 1 then
42     post_channel(cpos-1)
43  on post_channel(c):
44   c. $\bar{r} \leftarrow \emptyset$  /* not needed */
45   c.state ← postpayment
46  on ❻ unlock(c1, ..., cn):
47   if pos > 1 then
48     cpos-1.state ← idle
49   if pos ≤ n then
50     cpos.state ← idle

```

of value v_y , where $v_x > v_y \geq p_x$, with c and dissociate d_x from c . This reduces the lock-in.

At any time, either party may settle the channel using `settle_channel` (line 45). If the balances of the parties are *neutral*, i.e., equivalent to their deposits as if no payments were made, the treasuries can terminate the channel off-chain by simply disassociating all deposits from the channel. Off-chain termination avoids writing a settlement transaction to the blockchain; otherwise, the local treasury generates a blockchain transaction tx using the deposits and balances in the channel, collects signatures from the committee members, and writes tx to the blockchain.

5.4.3 Executing multi-hop payments

To do a multi-hop payment across multiple payment channels, parties invoke `pay_multihop` (Algorithm 2, line 59) with the payment amount and the channel identifiers of the path.¹ All channels in the path must update their balances atomically otherwise intermediaries could lose funds. For example, B in Figure 5.2 retains the same total funds post-payment, i.e., their balance is incremented by \$50 in c_1 and decremented by \$50 in c_2 ; if c_1 is not updated and only c_2 updates, B pays C personally.

One approach to ensure atomic channel updates is to freeze channels by preventing parties from settling them until the multi-hop payment completes. This has the problem that if a failure occurs along the path, channels are frozen indefinitely. To overcome this, Teechain allows parties to settle channels even if a multi-hop payment is being executed. Teechain achieves this using a *proof of premature*

¹We assume that participants determine paths in Teechain out-of-band.

termination (*PoPT*). When a party prematurely settles a channel c during a multi-hop payment, the settlement transaction $\mathbf{tx}$ can be used by other parties in the path to determine the state s of settlement: c was either settled pre-payment ($s = \text{pre}$), i.e., before the payment has occurred, or post-payment ($s = \text{post}$), i.e., after the payment has occurred. The parties can present $\mathbf{tx}$ to their treasuries as a *PoPT* to settle all channels in the same state s .

Teechain enforces that settlement transactions in state *pre* will *conflict* (Section 5.2.1) with those in state *post*. If a channel in the path is terminated prematurely using `eject_multihop` (Algorithm 2, line 66), the first settlement transaction $\mathbf{tx}$ to be written to the blockchain determines the state at which all channels are settled. If a channel in a different state tries to settle afterwards, the transaction is rejected by the blockchain. The party can present $\mathbf{tx}$ to its treasuries as *PoPT* through `eject_multihop` (Algorithm 2, line 72), which generates a settlement transaction without conflict. Conflicts prevent Teechain from assuming how long settlements take to be written to the blockchain.

For blockchains with expressive transactions [The17b], smart contracts can be used to ensure conflicts between settlement transactions in different states. Channels in a multi-hop payment can simply transition from *pre* to *post* in a single step.

For other blockchains, e.g., Bitcoin, Teechain must enforce transaction conflicts manually. Teechain constructs an intermediate *path settlement transaction* $\bar{\mathbf{tx}}$ that settles all channels in state *post* using a single blockchain transaction. $\bar{\mathbf{tx}}$ conflicts with individual settlement transactions in *pre* and *post* because it spends the same deposits. Teechain uses $\bar{\mathbf{tx}}$ to transition channels from state *pre* to *post* by moving to an intermediate state *inter* between the transition first. Channels in state *inter* can only settle using $\bar{\mathbf{tx}}$. If a party decides to settle a channel while it is in state *inter*, they settle all channels in the path. Therefore, during the transition from *pre* to *inter*, either the first channel settlement transaction $\mathbf{tx}$ written to the blockchain is in *pre*, in which case $\bar{\mathbf{tx}}$ cannot be written to the blockchain and all channels settle at *pre* by presenting $\mathbf{tx}$ as *PoPT*; or $\mathbf{tx}$ is $\bar{\mathbf{tx}}$, in which case all channels are settled in *post* using $\bar{\mathbf{tx}}$. The transition from *inter* to *post* is analogous.

Payment execution. Figure 5.3 shows the messages exchanged by the treasuries when A executes a multi-hop payment to C via B . Algorithm 3 shows the corresponding protocol steps. Teechain requires three network round trips to complete the payment: step ❶ locks the channel and ensures sufficient balances (Algorithm 3, line 1); step ❷ constructs $\bar{\mathbf{tx}}$ with all treasuries writing their channel balances and signatures (line 9); Teechain then updates the channel balances from *pre* (❸, line 17) to *inter* (❹, line 25) to *post* (❺, line 38) payment state; and finally, step ❻ unlocks the channels (line 46).

As multi-hop payments lock channels, this prevents concurrent payments. Teechain therefore dynami-

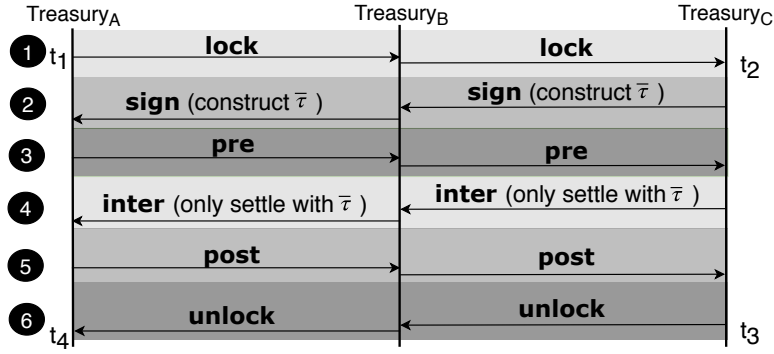


Figure 5.3: The protocol for multi-hop payments in Teechain.

cally constructs new channels for concurrent payments using unassociated deposits, as needed. This is feasible because Teechain can create channels and assign deposits with low latency. Teechain coalesces no longer needed payment channels by: (i) executing multi-hop payments in a cycle along the channels until they are at a neutral balance; and (ii) terminating the channels off-chain through deposit dissociation (see Section 5.4.2). We evaluate dynamic channel construction in section 5.6.3.

5.4.4 Payment protocol security

Teechain’s payment protocol (Algorithms 2 and 3) achieves balance security (section 5.2.4) under asynchronous blockchain access, i.e., parties can always receive their funds on the blockchain, regardless of blockchain access times or other parties’ actions. We sketch a proof below and defer to a technical report [LNE⁺19] for more details. We first show that Teechain achieves asynchronous blockchain access, and then prove balance security.

When Teechain writes to (`new_deposit`, `release_deposit`, `settle_channel`, `eject_multihop`) or reads from the blockchain (`approve_deposit`), the protocol makes no assumption about the duration of these operations. For example, when ejecting from a multi-hop payment prematurely (`eject_multihop`), Teechain uses the first settlement transaction written to the blockchain to determine the state at which all channels in a payment path are settled (Section 5.4.3). By considering all blockchain interactions on a case-by-case basis (Algorithms 2 and 3), we can see Teechain operates with asynchronous blockchain access.

Payment channel security. We now prove that Teechain achieves balance security using the Universal Composability (UC) framework [Can01]. Our definition of balance security (Section 5.2.4) under UC is similar to prior work [MMSK⁺17, MBKM17, DFH18]. We model committees as a single treasury executing the protocol.

Under UC, we consider a *real* world, in which parties run the Teechain protocol π_{Teechain} (algorithm 2),

and an *ideal* world, in which parties interact with an *ideal functionality*, $\mathcal{F}_{Teechain}$, a trusted third party that implements Teechain’s API (algorithm 2). Adversarial behavior is introduced in the ideal world by a simulator $\mathcal{S}$ with appropriate adversarial abilities (section 5.2.5).

To prove that Teechain achieves balance security, we show that (i) the real and ideal worlds are indistinguishable to an external observer $\mathcal{E}$. This implies that any attack violating balance security in the real world is also possible in the ideal one; and (ii) $\mathcal{F}_{Teechain}$ achieves balance security in the ideal world. This proves that $\pi_{Teechain}$ also achieves balance security.

We prove indistinguishability between the real and ideal worlds through a series of five *hybrid steps*, starting at the real world H_0 , and ending in the ideal world H_5 . In each step, a key element is changed and indistinguishability is proven. As commonly done [TZL⁺16, BJZ⁺17], in H_0 , the desired behavior of TEEs and the blockchain are replaced by two ideal functionalities, $\mathcal{F}_{TEE}$ and $\mathcal{F}_B$, respectively (defined in [PST17, PSS17]). In H_1 and H_2 , $\mathcal{S}$ simulates $\mathcal{F}_{TEE}$ and $\mathcal{F}_B$, respectively, and in H_3 and H_4 , incorrectly signed messages to $\mathcal{F}_{TEE}$ and $\mathcal{F}_B$, are dropped, to tolerate attacks on the signing schemes. Finally, in H_5 , we prove equivalence between $\pi_{Teechain}$ and $\mathcal{F}_{Teechain}$ to $\mathcal{E}$.

Next, we prove that $\mathcal{F}_{Teechain}$ achieves balance security by showing that a party can always eventually place transactions on the blockchain that grant it an amount equal to its perceived balance. This is done by ordering $\mathcal{F}_{Teechain}$ to create transactions that close all open channels, remove all unassociated deposits, and place them on the blockchain. Since Teechain does not make blockchain timing assumptions, denial-of-service attacks [HKZG15, MHG18]), do not violate balance security.

Multi-hop payment security. We show that the multi-hop protocol also maintains balance security. As shown in fig. 5.3, consider a payment from A to C via B of amount val at the following times: A begins step lock of the protocol at t_1 ; at $t_2 > t_1$, C begins step lock; at $t_3 > t_2$: C completes step unlock; and, at $t_4 > t_3$, A completes the protocol with unlock.

For A , the perceived balance for the channel is: before t_1 as if val was not paid; after t_4 , as if val was paid; between t_1 and t_4 either is acceptable. For C , the same as A but t_1 replaced with t_2 , and t_4 with t_3 . The perceived balance of the intermediate B is not affected. A considers the payment complete *iff* C considers it complete; funds are not lost or created.

We show that A and C can unilaterally reclaim their perceived balance. Note that single channel payments do not interfere with multi-hop payments, because all channels are locked (section 5.4.3). At any point, A and C can settle the channels in either the *pre-* or *post-*payment states, either with single settlement transactions or using $\bar{\tau}$ (see algorithm 3). For example, if a node is in state *lock*, the others are either in *unlock* or *lock* or in *lock* or *sign*. In all cases, if a node settles, the rest of the nodes can only

settle in the same state (pre- or post-payment), in accordance with balance security.

5.5 Treasury committees

As discussed in Section 5.3, Teechain employs TEE-heterogeneity in treasury committees to maintain balance security despite TEE failures and compromises. To achieve this, treasury committees: (i) employ threshold cryptography to jointly manage fund deposits together: Teechain requires *m-out-of-n* treasuries in a committee to agree before access to funds can be granted; and (ii) leverage force-freeze chain replication to maintain agreement between TEEs. This ensures consensus in a treasury committee.

In this section we describe how treasury committees employ force-freeze chain replication (Section 5.5.1). Next, we discuss treasury committee configurations and the impact on TEE fault tolerance (Section 5.5.2). Finally, we present persistent storage for committee members (Section 5.5.3).

5.5.1 Force-freeze chain replication

To maintain consensus among committee members, Teechain uses *force-freeze* chain replication, a new variant of chain replication [VRS04b]. The nodes form a chain, with the primary at the head, and the last backup at the tail. On an update, the primary propagates the update down the chain. Each node forwards the update to its backup, and waits for an acknowledgement before executing the update. When the primary receives an acknowledgement, the entire chain has updated. This provides strong consistency among the nodes.

Traditional chain replication [VRS04b] continues to execute state updates even after nodes have failed to update. Applying this naively to treasuries in a committee, would make Teechain vulnerable to roll-back and state forking attacks (see Section 5.3.3). Instead, in force-freeze chain replication (seen in Algorithm 4), if a node receives an update request (line 13) and it or its successor fails to update, the chain is frozen at its current state (line 21). All channels must now settle and release unused deposits. This ensures all treasuries in the committee maintain consistency.

Parties construct force-freeze replication chains using `assign_comm_chain` (line 1), which assigns a treasury to the end of the chain: a party provides the public key of the node at the tail of the chain. To secure state updates along the chain, nodes construct secure communication channels (lines 3 and 10).

To prevent malicious treasuries from executing denial-of-service attacks by freezing committee chains

Algorithm 4: Force-freeze chain replication (For brevity, we omit message encryption, authentication and freshness.)

<pre> 1 on assign_comm_chain(pub): 2 assert(pred = ∅) /* no chain */ 3 attest_and_auth_DH(pub) 4 pred ← pub /* set chain pred */ 5 send(addTail) to (pub) 6 wait_for(update, s) from (pub) 7 return ⊤ 8 on receive(addTail) from (pub): 9 assert(succ = ∅) /* current tail */ 10 attest_and_auth_DH(pub) 11 succ ← pub 12 send(update, curr_state) to (pub) </pre>	<pre> 13 on receive(update, s) from (pub): 14 assert(pred = pub) 15 if succ = ∅ then 16 update_state_to(s) 17 ack ← create_signed_ack() 18 else 19 ack ← send(update, s) to (succ) 20 if fail_or_invalid(ack) then 21 freeze() /* can't update */ 22 else 23 update_state_to(s) 24 ack ← sign_ack(ack) 25 send(ack) to pub </pre>
--	---

through forced failures, Teechain employs incentives for committee members: parties are assumed to be financially rational (Section 5.3.4), and committee members are paid fees for participation. If a committee member forces a freeze, it loses any participation fees that it has accumulated in that committee.

Unlike other replication protocols, e.g., Paxos [L⁺01] and PBFT [CL⁺99], force-freeze chain replication uses a chain communication topology and therefore does not require full network connectivity, which is impractical in peer-to-peer networks. Although other consensus protocols may enhance liveness, this comes at the cost of increased network communication. It also increases protocol complexity—a benefit of force-freeze chain replication is that it is simple to implement and reason about.

5.5.2 Treasury committee configurations

To ensure balance security (Section 5.2.4) despite TEE failures and compromises, Teechain uses treasury committees of size n for each deposit, and requires at least m treasuries in a committee to sign a blockchain transaction before that deposit can be spent. To violate balance security, an attacker must compromise at least m TEEs in a committee, or cause $(n - m) + 1$ TEEs to fail, e.g., crash or stop responding.

The values of m and n affect security: (i) 1-out-of-1 deposits provide no fault tolerance against TEE crash failures or compromises; (ii) 1-out-of- n treasury committees provide crash fault tolerance for TEEs but do not tolerate their compromises; and (iii) while larger values of m and n provide greater security, this affects the performance of payments in Teechain. We explore this performance trade-off in Section 5.6.

As deposits must be approved before association with payment channels (Section 5.3.2), parties can choose the values of m and n for their deposits and channels. For small deposits, a 1-out-of-1 committee

chain may be sufficient as there is little loss if a failure occurs. For medium deposits, 1-out-of- n may be desirable to tolerate crash failures. For large deposits, e.g., 2-out-of-3 committee chains are required to tolerate attacks. Larger committees, e.g., with more than five members, may only be required for high-value deposits.

We note that depending on the chosen values of m and n in committees, failures and compromises may result in *deadlock scenarios*, where it becomes impossible to gather sufficient signatures to settle a payment channel or release an unused deposit. This can occur, e.g., if a failure is forced in the middle of a replication chain that causes the committee to be partitioned into disjoint groups, where the size of these groups are $< m$. To avoid these scenarios, Teechain allows a force-frozen replication chain to be settled at either the pre-update or post-update state; the underlying blockchain determines the outcome as these states are conflicting and only one can be finalized on the blockchain. This is identical to the mechanism used to prematurely settle multi-hop payments in 5.4.3. We further note that additional mechanisms, such as *paralysis proofs* [ZDK⁺17], could also benefit possible deadlock scenarios.

To prevent an attacker from biasing the selection of committee members, parties select the committee treasuries themselves on deposit creation (`new_deposit`, Table 5.1). Selection criteria may include treasury reputation, trusted TEE vendors and implementations, blacklisted treasuries, and TEE heterogeneity. To avoid Sybil attacks [Dou02], Teechain can leverage several techniques, such as requiring treasuries to provide a proof-of-stake [BLMR14], operate in a permissioned setting [ABB⁺18], or use a reputation system.

Payment channels may contain multiple deposits, each with a separate treasury committee. These committees do not have to be updated atomically: for payments that span multiple deposits, the treasury committees must be identical, and thus the state updates can be batched. If a large payment spans deposits of multiple treasury committees, the payment is broken down into smaller payments, only affecting one deposit at a time. Having many deposits, each with distinct committee members, affects performance (see Section 5.6.1).

5.5.3 Treasury committees with secure persistent storage

In addition to treasury committees, Teechain also supports the optional use of secure persistent storage for crash fault tolerance. After a failure, a treasury can reload its state, settle channels and return deposits. To overcome roll-back attacks, state freshness must be guaranteed by the TEE hardware [AGJS13], e.g., through hardware monotonic counters [Int17b] (see Section 2.3.2).

Current Intel SGX implementations (Section 2.3.2) throttle accesses to hardware monotonic counters to tens of increments per second [SP16b, MAK⁺17], which limits performance. As a mitigation, Teechain batches transactions at the client side, similar to other payment networks [PD16] that merge payments from the same sender/recipient pairs. We demonstrate batching to overcome the performance limitations of Intel SGX hardware monotonic counters in Section 5.6. Current SGX implementations also limit the number of writes for hardware counters to 1 million [MAK⁺17]. For the majority of parties in Teechain, this should be high enough. When the limit is reached, Teechain forces treasuries to settle channels and return deposits.

5.6 Evaluation

To understand the performance impact of TEE-heterogeneity, we implement and evaluate Teechain. To measure the overheads of our approach, we perform all experiments with: (i) a 1-out-of-1 treasury committee that does not support TEE-heterogeneity or provide TEE fault tolerance (see Section 5.5.2); and (ii) m -out-of- n treasury committees of various sizes (Section 5.5). This allows us to measure the relative cost of employing TEE-heterogeneity to overcome TEE failures and compromises. In addition, we compare Teechain against the Lightning Network (LN) [PD16]: a state-of-the-art payment network that operates with synchronous blockchain access (see Section 5.2.3). This allows us identify the performance implications of designing a payment network that achieves asynchronous blockchain access (see Section 5.2.4).

To measure performance, we define throughput as the number of transactions sent per second, and latency as the time from when a payment is issued until an acknowledgement is received. We measure throughput and latencies without blockchain access times; measuring blockchain access times is orthogonal to our approach. When comparing Teechain to the Lightning Network, we use the Lightning Network Daemon (LND) [Lig17]. We further note that both Teechain and LN can optionally batch transactions at the client side, merging multiple payments into a single payment with increased latency.

Teechain Implementation. We implement Teechain using Intel SGX for the Bitcoin blockchain. We use the Linux Intel SGX SDK version 2.1 [Int17a] and a subset of Bitcoin core [The16a]. A release of our implementation is available at: <https://teechain.network>. Teechain consists of 20,000 lines of C/C++ code inside the TEE, and 65,000 lines of untrusted code. As the Linux SGX SDK does not support monotonic counters on all hardware [Int17a], we emulate them with a delay of 100 ms [SP16b, MAK⁺17]. Our implementation is hardened against side-channel attacks. Although TEE compromises are mitigated by treasury committees (Section 5.5), Teechain uses timing and memory-access

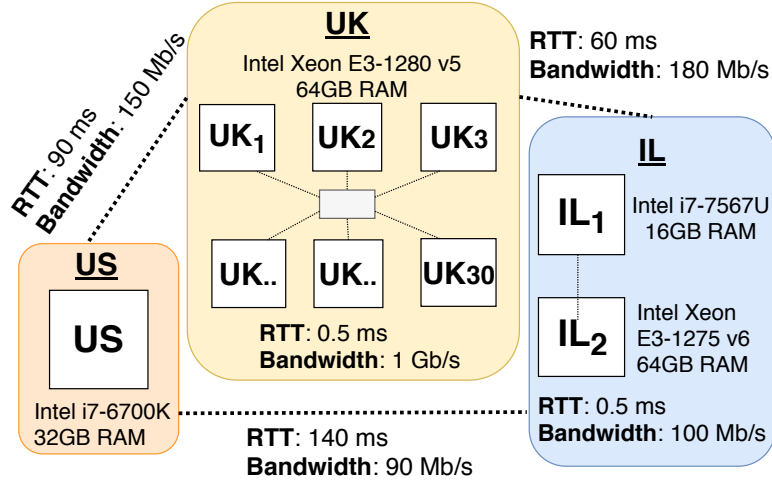


Figure 5.4: The machines, network and evaluation setup of our experiments.

side-channel resistant libraries for sensitive data: (i) secp256k1, a constant time and memory library for elliptic curve operations [The13]; (ii) a side-channel resistant implementation of Elliptic-Curve Diffie-Hellman [The16b]; and (iii) AES-GCM using AES-NI [The16b], immune to software side channels [Int17a].

We continue this section by exploring the performance implications of TEE-heterogeneity and asynchronous blockchain access on payment channels (Section 5.6.1), multi-hop payments (Section 5.6.2) and payment networks (Section 5.6.3)

5.6.1 Performance of payment channels

We want to answer three questions: (i) what is the performance overhead of employing TEE-heterogeneity for TEE fault tolerance? (ii) what is the performance overhead of supporting asynchronous blockchain access? and (iii) how well does transaction batching overcome the limitations of SGX hardware monotonic counters (see Section 5.5.3)?

We deploy Teechain on 33 SGX-capable machines in the UK, the US and Israel. Figure 5.4 shows the network topology and hardware set-up. We construct a payment channel between *US* and *UK*₁. To evaluate treasury outsourcing, *IL*₁ acts as a non-SGX client using *US* as a remote treasury.

In all experiments, treasury committees for different deposits have the same length, as the performance is bound by the slowest party. We vary n for m -out-of- n treasury committees. Note that m does not affect channel throughput because all n committee members must replicate the state regardless. When batching transactions, we batch for 100 ms before sending a transaction. Teechain requires one round-trip for a payment, while LN requires two [PD16]. Teechain can pipeline payments but LN only

Payment channel	Throughput (tx/sec)	Latency (ms)	[99th %]
<i>Lightning Network</i>	1,000	387	[420]
<i>Teechain</i>			
$n = 1$	130,311	86	[93]
$n = 2$ (IL)	34,115	292	[301]
$n = 3$ (IL, UK)	33,180	415	[432]
$n = 4$ (IL, US, UK)	33,178	672	[691]
$n = 1$ (batching)	150,311	191	[196]
$n = 3$ (batching)	135,331	516	[530]
$n = 3$ (outsourced)	33,178	483	[494]
<i>Persistent storage</i>			
$n = 1$	10	288	[294]
$n = 1$ (batching)	145,786	401	[408]

Table 5.2: The maximum performance of payment channels for both the Lightning Network and Teechain.

supports sequential transactions and must batch by default.

Table 5.2 shows the observed throughput and latency. LN achieves a maximum throughput of 1,000 tx/sec with a latency of 387 ms (99th percentile at 420 ms). With a treasury committee of $n=1$, i.e., no TEE fault tolerance, Teechain has two orders of magnitude higher throughput with a latency of 86 ms (no batching). With $n=2$ (i.e., an extra committee member in Israel), the throughput of Teechain is $34\times$ compared to LN, with similar latencies. Adding more members to each deposits treasury committee only increases latency, and throughput remains unchanged. Using persistent storage, performance is capped by the TEE hardware counters, resulting in 10 tx/sec, which can be overcome by transaction batching. Teechain achieves between $135\text{--}150\times$ better performance than LN when batching.

Table 5.3 shows the performance of different API calls. LN channel creation takes approx. 60 mins, as a transaction must be placed onto the blockchain and confirmation takes 6 Bitcoin blocks. Since Teechain decouples channels and deposits, channel creation takes only 2.8 secs; we assume deposits are already on the blockchain. Creation of an outsourced payment channel takes 4.3 secs, as the client (IL_1) must also verify the integrity of the outsourced treasury (US). Adding new members to a committee incurs similar latencies as channel creation. The latency for associating deposits depends on the committee length n , and dissociation is similar.

In summary, we can see how payment channel throughput is affected by TEE-heterogeneity. Without TEE-heterogeneity, i.e., a committee of $n=1$, Teechain offers $130\times$ the performance of LN. However, when TEE-heterogeneity is employed, payment channel throughput is reduced by $3\times$ and payment latency increases by $3\times\text{--}7\times$, for $n=2$ to $n=4$ committee members, respectively. However, despite this

overhead, Teechain with TEE-heterogeneity still offers $33\times$ better throughput than LN and similar latencies. In addition, we see that: (i) without transaction batching, persistent storage performs the worst due to hardware counters; (ii) with batching, Teechain achieves higher throughput for treasury committees and persistent storage hides the delay for counters.

5.6.2 Performance of multi-hop payments

Next we evaluate the performance of multi-hop payments and investigate: (i) what is the performance overhead of employing TEE-heterogeneity? (ii) what is the performance overhead of supporting asynchronous blockchain access? and (iii) how does latency increase with the number of hops in a payment path?

For our experiments, we limit the maximum number of hops in a payment path to 11, as longer payment paths are unlikely to be seen in practice. Recent work [SGNB19] studying LN shows that the average number of hops between two parties is approximately 3.

We construct the 11 payment channels, all of which are transatlantic in the topology from Figure 5.4. We send transactions along the path $UK \rightarrow US \rightarrow IL \rightarrow UK$. For UK and IL , we split the payment channels equally between the machines to spread load. All experiments use the same payment channels and treasury committees of the same length. Committee members are deployed in different failure domains.

We measure the latency of multi-hop payments. We vary the number of hops and the number of committee members per committee for each node. Figure 5.5 shows that LN scales linearly with the multi-hop chain length, taking 1 sec to complete a payment across 2 hops (2 channels) and 6 secs for 11 hops. Teechain also scales linearly but with a different slope: with $n=1$, the latency is about $2\times$ that of LN; using $n=3$, payments across 2 hops take 5 secs; payments across 11 hops take 26 secs. The $3\text{--}4\times$ overhead compared to LN is due to the extra network round trips for multi-hop payments.

To update all channels in a multi-hop payment consistently, both Teechain and LN do not pipeline payments. Therefore, throughput is $1/\text{latency}$. Teechain and LN batch transactions: throughput becomes the batch size divided by the latency to complete the payment. We compare the throughput for Teechain and LN, with each Teechain node using committee chains of $n=3$. Teechain batches 135,000 tx/sec; LN batches 1,000 tx/sec (see section 5.6.1). With this, the throughput of Teechain for 2 hops is 14,062 tx/sec, and for 11 hops is 3,649 tx/sec. For LN, throughput for 2 hops is 862 tx/sec, and 139 tx/sec for 11 hops. Teechain thus outperforms LN by $16\text{--}26\times$.

API operation	Latency (ms) [99th %]	
	Local	Outsourced
<i>Lightning Network</i>		
new_pay_channel	60 min.	[N/A]
<i>Teechain</i>		
new_pay_channel	2,810 [4,205]	4,322 [5,201]
assign_comm_chain	2,765 [3,910]	2,852 [3,993]
associate_deposit		
$n = 1$	101 [110]	
$n = 2$ (IL)	289 [297]	
$n = 3$ (IL & UK)	422 [429]	489 [514]
$n = 4$ (IL, US & UK)	677 [681]	
Persistent storage	302 [309]	

Table 5.3: The performance of API operations in the Lightning Network compared to Teechain.

In summary, we can see how multi-hop payments are affected by TEE-heterogeneity. Without TEE-heterogeneity, i.e., a committee of $n=1$, Teechain executes payments with $2\times$ the latency of LN. However, when TEE-heterogeneity is employed, latency increases to $4\times$ - $5\times$, for $n=2$ to $n=3$ committee members, respectively. This is because Teechain requires three network round trips to complete a payment, while LN requires only 1.5. Teechain must synchronize nodes off-chain with extra messages to support asynchronous blockchain access. In addition, Teechain is network-bound: chain replication increases latency. Despite the additional latency, Teechain with TEE-heterogeneity is able to batch transactions more efficiently, and thus can still offer higher payment throughputs than LN.

5.6.3 Performance of payment networks

We evaluate the performance of a complete Teechain payment network and investigate how its throughput is affected by (i) the use of TEE-heterogeneity; and (ii) the network topology.

We use 30 machines located in the UK (see Figure 5.4). As there exist no public micro-payment datasets, we use the transactions from the Bitcoin blockchain. We filter out transactions that we cannot replay, such as those that spend to/from multi-signature addresses. For transactions with multiple inputs and outputs, we choose only one. The resulting dataset has over 150 million payments from a source to a recipient address.

We construct two payment network topologies: (i) a *complete* graph, in which all node pairs have direct payment channels; and (ii) a *hub-and-spoke* topology (see Figure 5.6), in which the nodes are connected with 3 tiers of connectivity: tier 1 nodes have the highest connectivity and tier 3 nodes the lowest. We emulate wide-area network links by adding 100 ms latency between machines.

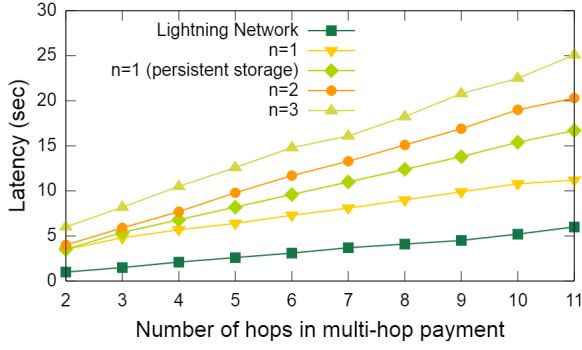


Figure 5.5: The performance of multi-hop payments.

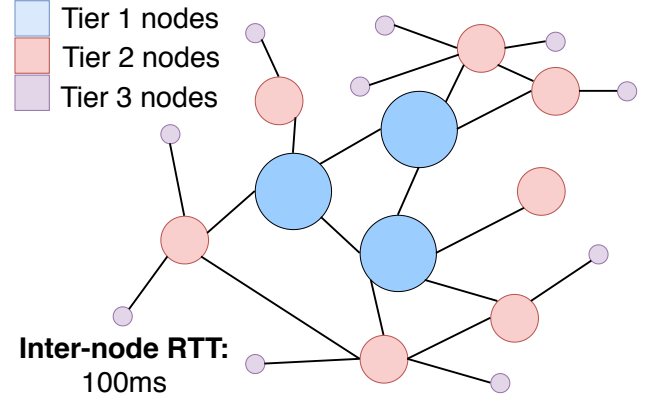


Figure 5.6: The hub-and-spoke network topology.

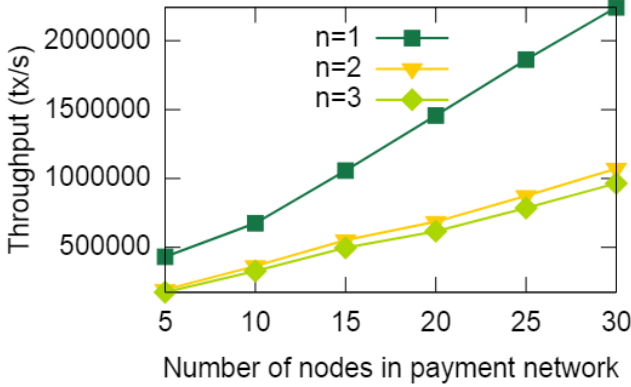


Figure 5.7: The throughput for the complete topology.

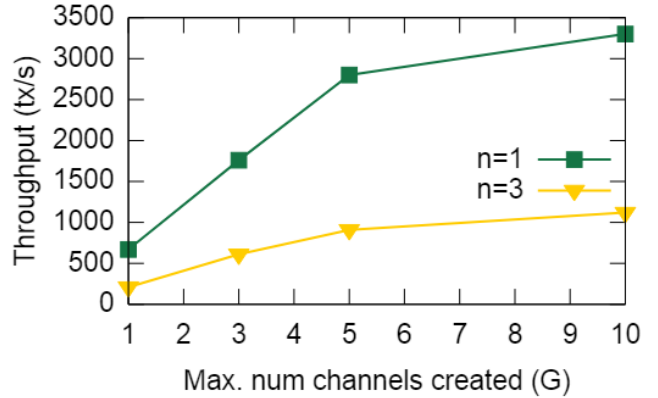


Figure 5.8: The throughput for the hub-and-spoke topology.

To execute payments, we assign Bitcoin addresses to the machines. For the complete graph, we randomly and evenly assign Bitcoin addresses; for the hub-and-spoke graph, we distribute the addresses in a skewed fashion, with larger nodes being assigned more addresses than smaller nodes (50% of addresses to tier 1 nodes, 35% to tier 2, and 15% to tier 3). For each graph deployment, we compare the throughput with different treasury committee lengths, for $n=1$ to $n=3$ committee members per deposit. We vary the number of nodes in the deployment from 5 to 30 machines.

Figure 5.7 shows the throughput for the complete graph topology. For all treasury committee lengths, throughput scales linearly with the node number. Without TEE-heterogeneity, i.e., committees of length $n = 1$, Teechain offers best performance (2.2 million tx/sec with 30 machines); with TEE-heterogeneity, i.e., committees with $n > 1$, throughput is limited (1 million tx/sec). There is little difference (9%) between $n = 2$ committee members and $n = 3$; throughput is bottlenecked by the time to replicate state using force-freeze chain replication.

Next we consider the hub-and-spoke graph topology. Multi-hop payments use the shortest path—if

there are multiple paths, only one is chosen. As multi-hop payments lock channels during execution, payments compete with one another. To overcome this, Teechain uses dynamic channel creation to allow concurrent payments between endpoints (see section 5.4.3).

Figure 5.8 shows how the throughput increases as intermediate nodes (i.e., tier 1 and 2) are permitted to create more dynamic channels. Without dynamic channels, i.e., $G = 1$, with $n = 3$ committee members, the network achieves around 210 tx/sec, with an average latency of 720 ms. With $G > 1$, throughput scales almost linearly with the number of channels, for both $n = 1$ and $n = 3$. We obtain diminishing returns as G increases further because tier 3 nodes become congested.

In summary, TEE-heterogeneity reduces payment throughput in the payment network. However, this can be overcome by constructing additional payment channels. Moreover, payment throughput is lower in the hub-and-spoke topology compared to the complete topology by several orders of magnitude. This is a result of locking channels for multi-hop payments: while dynamic channel creation alleviates contention, best performance requires high connectivity.

5.7 Related Work

We discuss work related to Teechain in detail below:

Payment channels and networks. There have been many different proposals made over the years for payment channels and networks. Unilateral payment channels were first proposed by Hearn and Spilman [HS15]. Duplex Micropayment Channels [DW15] subsequently proposed using time-locked transactions to prevent old channel states from being published, allowing bidirectional payment channels. The Lightning Network (LN) [PD16] proposed multi-hop payment support, requiring users to monitor the blockchain. Pisa [MBB⁺18a] built on LN to allow third parties to monitor the blockchain on behalf of other users. REVIVE [KG17] rebalances payment channels, but locks the funds during the rebalancing process. Sprites [MBKM17] allows funds to be added and removed to payment channels dynamically, but requires smart-contract support [Sza97]. State channels [DFH18, MBB⁺18b] offered a generalization of payment networks, but requires smart-contracts. Fulgor and Rayo [MMSK⁺17] proposed adding concurrency and privacy to existing payment channel networks.

In contrast to Teechain, all of these proposals assume synchronous blockchain access. To the best of our knowledge, Teechain is the first system to achieve asynchronous blockchain access.

Trusted hardware and blockchains. Prior work has also proposed electronic payment systems [SCW00] based on secure co-processors [DLP⁺01], smart cards [CCW96], and trusted hardware modules [BBC⁺94].

These works utilize dedicated hardware to enforce double-spending protection. However, these solutions do not integrate asynchronously with a blockchain and make weaker security assumptions, such as assuming no hardware compromises.

In addition, other work uses TEEs in conjunction with a blockchain: Microsoft’s Confidential Consortium Framework (CCF) [SCA⁺19] operates a permissioned blockchain using TEEs to enable high throughput and confidentiality for transactions. Unlike Teechain, CCF does not operate on top of an existing permissionless blockchain, but instead assumes a permissioned setting in which the identities of all members of the CCF consortium are known. TownCrier [ZCC⁺16] enables a secure data-feed for blockchain contracts; Tesseract [BJZ⁺17] is a secure multi-blockchain cryptocurrency exchange; Eki-den [CZK⁺18] offers a platform for privacy-preserving smart contracts; Obscuro [TLK⁺17] constructs a Bitcoin privacy mechanism; and Paralysis Proofs [ZDK⁺17] allows consensus reconfiguration with a blockchain. Apart from the different goals, Teechain uses a more refined security model: clients use a remote TEE to prevent fraud and a local TEE for availability.

5.8 Limitations

In this section, we briefly discuss the limitations of our approach:

Threshold and independent failures. Teechain employs TEE-heterogeneity to tolerate TEE failures and compromises (see Section 5.1). As such, it makes two assumptions: (i) it argues that by combining multiple different TEE implementations to secure critical functionality, it can better ensure that TEE failures and compromises are independent; and (ii) it defines an explicit threshold, or upper-bound, on the number of possible failures. However, these assumptions can be violated in several different ways. For example: (i) if a vulnerability is found that affects all Teechain TEEs, e.g., a flaw in the Teechain payment protocol (Section 5.4), failures will not be independent, and thus all TEEs can be compromised simultaneously. This will allow an attacker to compromise balance security; and (ii) Teechain relies on external mechanisms, such as proof-of-stake [BLMR14] to avoid Sybil attacks [Dou02] (see Section 5.5.2). If this mechanism is found to be broken, or vulnerable to attack, an attacker can bias the selection of committee members, and thus compromise more than the threshold number of TEEs in the committee. To provide strong security guarantees, Teechain must ensure that TEE failures and compromises are indeed independent and remain below the threshold. These are limitations of our approach.

Confidentiality compromises. Teechain demonstrates the practicality of employing TEE-heterogeneity to provide balance security. This ensures that funds cannot be lost or stolen by attackers, despite TEE

failures and compromises (Section 5.2.5). However, Teechain does not provide confidentiality or privacy to treasuries or parties in the payment network. This is because security-sensitive data, such as deposit balances, payment amounts and payment destinations, are replicated across all TEEs in the committee. If TEE compromise occurs, e.g., a treasury committee member is adversarial, security-sensitive information will be exposed. To overcome this, Teechain will need to be modified to employ threshold cryptography to protect sensitive data against TEE compromises. However, as Teechain does not currently support this, confidentiality and privacy is a limitation of our approach. We discuss how Teechain can be extended to provide this functionality in Section 6.2.

5.9 Summary

In this chapter, we proposed a solution to the third security challenge faced by TEE-based applications: tolerating TEE failures and compromises (Section 1.4.3). To achieve this, we leveraged TEE-heterogeneity: the protection of security-critical functionality in applications using multiple heterogeneous TEEs. We argue that: (i) rather than using a centralised TEE architecture, e.g., securing private keys with a single TEE instance, applications should adopt a decentralised architecture, where multiple TEE instances are employed; and (ii) instead of using homogeneous TEEs to secure applications, i.e., a set of identical TEE implementations, applications should employ heterogeneous TEEs. This provides protection against global TEE failures and compromises, where an entire class of TEEs might fail or be compromised simultaneously,

To leverage TEE-heterogeneity in applications, we make two contributions: (i) using threshold cryptography to distribute security-critical functionality from a single TEE to multiple heterogeneous TEEs in the application; and (ii) force-freeze chain replication to maintain consensus about the state of the application between different TEEs. We employed these contributions in Teechain, the first blockchain payment network that uses TEEs. Teechain offers secure and high performance cryptocurrency payments between parties, without assuming synchronous blockchain access. Teechain demonstrates that TEE-heterogeneity is a general and practical approach to securing applications against TEE failures and compromises; even in adversarial settings, such as blockchains and cryptocurrencies.

While we use Teechain to illustrate and implement our contributions, we note that the contributions made in this chapter are fundamentally application and TEE-agnostic: TEE-heterogeneity can be applied to a wide range of different applications and TEEs. Moreover, while the Teechain implementation targets the Bitcoin blockchain, the Teechain design and protocols presented here are also blockchain-agnostic. As such, Teechain can be employed on many different blockchains today.

6

Conclusion

We begin this chapter by providing a summary of the work presented in this thesis (Section 6.1). Next, we identify several directions for future work based on our contributions (Section 6.2). Finally, we present closing remarks (Section 6.3).

6.1 Thesis summary

TEEs offer powerful primitives for protecting the confidentiality and integrity of software applications. By leveraging hardware isolation in general purpose CPUs, TEEs enable new software deployment models today: only the CPU and the isolated code and data need to be trusted. All other software and hardware components can remain untrusted. This allows applications to execute with a small TCB. However, despite their strong security properties, TEEs have yet to see mainstream adoption. This is because TEEs pose fundamental security challenges that make it difficult for applications to leverage confidentiality and integrity guarantees. In this thesis, we explored the security challenges faced by TEE-based applications and offered novel solutions to overcome them.

We began the thesis by observing that *software is only as secure as the environment in which it executes*. This observation highlights the need to protect applications in untrusted and malicious environments. To overcome this, we introduced trusted computing and TEEs as potential solutions. We then identified three fundamental security challenges faced by applications that wish to employ TEEs for confidentiality and integrity: (1) as all code and data isolated by the TEE is trusted to be free from software bugs, applications must minimise the amount of trusted code and data to reduce the TCB; (2) as all components outside the TEE are untrusted, applications must secure TEE boundary interfaces between trusted and untrusted components, as these interfaces expose attack vectors; and (3) as TEEs do not provide availability guarantees, and current implementations are vulnerable to attacks, applications must tolerate TEE failures and compromises.

To overcome these challenges, we offered three solutions: (1) to minimise the TCB of applications, we proposed *automatic application partitioning for TEEs*: a principled approach for partitioning existing applications to be deployed with TEEs. Our approach allows applications to reduce the amount of code and data placed into the TEE, thus reducing the likelihood of application vulnerabilities; (2) to secure TEE boundary interfaces, we proposed a set of *TEE interface design principles*. Applications that follow these design principles can better protect against attacks on the TEE boundary interface; and (3) to tolerate TEE failures and compromises, we proposed applications employ *TEE-heterogeneity*: using multiple heterogeneous TEEs to protect security-critical functionality.

We presented each of our proposed solutions in detail, and evaluated our contributions:

Automatic application partitioning to minimise the TCB. First, to minimise the TCB of applications, we proposed automatic application partitioning as a way to reduce the amount of code and data placed into the TEE. Our approach offers three contributions: (i) using source code annotations and static program analysis to identify security-sensitive code and data in the application; (ii) partitioning the application at the function granularity to produce a secure TEE boundary interface; and (iii) using automatic source code generation to transform the application.

To illustrate these contributions, we presented Glamdring, the first automatic application partitioning framework for TEEs. Glamdring partitions existing C applications for Intel SGX enclaves. We presented the design and implementation of Glamdring, and offered a comprehensive performance and security evaluation. Glamdring demonstrates that automatic application partitioning for TEEs is not only feasible, but practical, as it can be successfully applied to several existing real-world applications with acceptable performance overheads.

Interface design principles to secure TEE boundary interfaces. Second, to secure TEE bound-

ary interfaces, we proposed three TEE interface design principles for protecting applications: minimality, protection and obliviousness. To illustrate these design principles, we designed a TEE boundary interface that secures communication between applications and an untrusted host OS; our host interface offers three contributions: (i) for minimality, we propose applications employ a library OS inside the TEE to reduce the amount of functionality required by the host OS; (ii) for protection, we propose transparently encrypting and integrity protecting all data that crosses the TEE boundary interface; and (iii) for obliviousness, we propose making host OS interface calls discrete and deterministic, to avoid leaking information about application execution.

To illustrate our contributions, we presented Citadel, the first TEE runtime system that offers a secure host OS interface for TEE-based applications. Citadel protects Linux binaries executing inside Intel SGX enclaves from boundary interface attacks. We presented the design and implementation of Citadel, and offered a practical performance evaluation using several real world applications. Citadel demonstrates that minimality, protection and obliviousness can efficiently protect applications against boundary interface attacks.

TEE-heterogeneity for TEE failures and compromises. Third, and finally, to tolerate TEE failures and compromises, we proposed applications employ TEE-heterogeneity. Our approach offers two contributions: (i) using threshold cryptography to distribute security-critical functionality from a single TEE to multiple heterogeneous TEEs in an application; and (ii) using force-freeze chain replication to maintain consensus about the state of the application between different TEEs.

To illustrate these contributions, we presented Teechain, the first application that employs TEE-heterogeneity to overcome TEE failures and compromises. Teechain leverages our contributions to protect TEEs in the context of blockchains and cryptocurrencies: environments that are particularly untrusted and adversarial. We presented the design and implementation of Teechain, and offered a detailed security and performance evaluation. Teechain demonstrates that TEE-heterogeneity is effective at achieving fault tolerance while offering reasonable performance overheads.

6.2 Future work

In this section, we extend our discussion by identifying several directions for future work:

Automatic application compartmentalisation for TEEs. Glamdring demonstrates how automatic application partitioning can minimise the TCB of applications secured with TEEs (Chapter 3). By placing security-sensitive components into the TEE, and leaving non-sensitive components outside

the TEE, Glamdring reduces the likelihood of application vulnerabilities. However, we observe that Glamdring only offers two *compartments* for privilege separation [BMHK08b], i.e., a compartment for sensitive components inside the TEE, and a compartment for non-sensitive components outside the TEE. As such, if an application contains two sensitive components, Glamdring will place these components into the same compartment, even if the components are unrelated. This allows an attacker to exploit a vulnerability in one component to compromise the security of another component.

To overcome this, we propose that Glamdring be modified or extended to support multiple separate compartments inside the TEE. These compartments can be isolated from each other, e.g., using compiler based techniques, and offer support for inter-compartment communication, e.g., by exposing well-defined compartment interfaces. By placing independent components, e.g., modules and libraries, into separate compartments within the TEE, this would: (i) better protect against attacks, e.g., secure components can be protected from components under adversarial control; and (ii) limit the damage an adversary could perform to the application, as a single bug in the code would only compromise one compartment.

Fine-grained automatic application partitioning for TEEs. Glamdring partitions application source code at the function granularity: all functions containing security-sensitive statements are deemed security sensitive and moved into the TEE (Chapter 3). While this helps to produce a natural TEE boundary interface, it: (i) unnecessarily bloats the TCB of the application, as large functions with many non-sensitive statements are placed into the TEE; and (ii) increases the size of the TEE boundary interface, as non-sensitive code must be able to read and write non-sensitive variables moved into the TEE.

To overcome this, we propose that Glamdring could be modified to partition applications at a finer-granularity, e.g., at the granularity of intermediate representation (IR) statements [CSR⁺17]. This would avoid unnecessarily bloating the TCB of the application, as less non-sensitive application code and data would be moved into the TEE; thus, reducing the likelihood of vulnerabilities existing in trusted TEE code. However, this approach would need to consider the trade-off between the size of the TCB and the application boundary interface: by partitioning code at a finer granularity, this could result in a larger TEE boundary interface, as more transitions between untrusted and trusted code could occur. As such, adopting this partitioning would require a systematic approach to dealing with this trade-off.

Trading off the application TCB for the boundary interface. While Glamdring partitions applications to reduce the size of the TCB (Chapter 3), it produces an application-specific boundary interface which is larger than that of other approaches. In contrast, the approach taken by Citadel

(Chapter 4) reduces the size of the boundary interface, while increasing the size of the TCB. A natural question can therefore be asked about whether or not these approaches and contributions could be combined to achieve a trade-off between the TCB and boundary interface of an application.

One way to achieve this would be to first partition an application using Glamdring and then take the security-sensitive part of the application and execute it inside Citadel. This approach is naive, however, as although it would prevent the partitioned code and data from leaking security-sensitive information through host OS calls, it would not protect application-specific calls between security-sensitive and non-sensitive code. As such, one could then ask whether or not the contributions offered by Citadel could be applied to these application-specific calls to better protect the partitioned application, e.g., whether or not some form of minimality could be achieved using EBR in Glamdring and if obliviousness could prevent side-channel information leakage.

Reducing the TCB of the library OS for TEEs. Citadel demonstrates how to protect applications against boundary interface attacks when communicating with an untrusted host OS (Chapter 4). However, one of the limitations currently faced by this work is that the TCB of the application is significantly increased. This is because an entire library OS is placed into the TEE: if a vulnerability exists in the library OS, it can be exploited to compromise the application.

To address this, we observe that not all applications executing inside a TEE will require every functionality provided by the library OS. For example, many applications may not require a TCP/IP network stack, or a complete file system implementation. As such, we ask whether or not the library OS inside the TEE could be reconfigured on a per-application basis, allowing developers to remove any functionality that is not required by the application. This would reduce the amount of code and data that needs to be trusted by the application and thus lower the corresponding risk of vulnerabilities.

Security testing TEE boundary interfaces. Citadel takes a conservative approach to protecting applications against TEE boundary interface attacks: it assumes that all interface calls can compromise application integrity and confidentiality. As such, Citadel protects every interface call, e.g., using encryption, integrity protection and oblivious execution (Chapter 5). However, this approach might not be ideal for small and self-contained applications, e.g., applications that do not expose large interfaces, or only perform in-memory calculations. This is because Citadel may significantly increase the performance overhead of these applications, e.g., due to encrypting/decrypting data, or executing interface calls in an oblivious manner.

Instead, we ask whether or not these applications could protect their boundary interfaces by simply security testing these interfaces, e.g., using fuzzing [SGA07, TDMK18] or symbolic execution [Kin76,

CS13a]. For example, developers could discover bugs in the application by: (i) testing the interface using a wide range of different inputs; and (ii) varying the order in which interface calls are executed. While this approach might not discover every possible vulnerability or protect against side-channel attacks, it may be sufficient to protect the application against the most common attacks; thus hardening the boundary interface and offering a sufficient level of security without having to incur heavy performance overheads.

Privacy and confidentiality for payment networks. Teechain demonstrates the practicality of employing TEEs to secure payments between distrusting parties (Chapter 5). However, one of the issues currently facing Teechain’s design, is the inability to provide privacy or confidentiality for payments and balances. For example, when parties in Teechain execute payments between each other, payment messages sent across the network expose the frequencies, timing and destination of payments. Moreover, as TEEs hold deposit amounts and balances for payment channels, TEE compromises can expose this information, thus leaking sensitive data.

To provide privacy and confidentiality in the payment network, we ask: (i) whether TEEs in Teechain can employ oblivious techniques, such as oblivious network communication, to conceal payment timing and destination information when sent across the network; and (ii) whether TEEs can employ secret sharing schemes or multiparty computation [Gol98] to help prevent the balances of parties from being exposed if a TEE is compromised. While this may appear trivial on the surface, we note that privacy and confidentiality is a difficult goal to achieve. This is because in order to route a payment across the payment network, the payment route implicitly exposes sensitive information, such as: (i) which parties have payment channels with other parties; and (ii) the capacity of those payment channels, i.e., the amount of funds deposited into those payment channels. As such, to achieve privacy and confidentiality, payment networks must be fundamentally redesigned.

6.3 Closing remarks

While there have been many different attempts to secure applications and protect sensitive data in untrusted environments, we believe that TEEs offer one of the most practical and general approaches for achieving this goal. However, as explored in this thesis, TEEs are by no means a panacea: they pose fundamental security challenges that make it difficult for applications to ensure confidentiality and integrity. As such, our work offers novel solutions to overcoming three of these challenges: (i) we propose using automatic application partitioning to minimise the TCB of applications for TEEs; (ii) we propose protecting applications against TEE boundary interface attacks by offering a set of interface

design principles; and (iii) we propose using TEE-heterogeneity to protect applications against TEE failures and compromises.

We hope that the contributions presented in this thesis will better enable software developers and researchers to overcome the challenges faced by TEEs. Moreover, we hope that the wider security community will continue to develop and advance trusted execution technology for the future of application and data security.

Bibliography

- [AAG⁺16] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. Posix abstractions in modern operating systems: The old, the new, and the missing. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 19. ACM, 2016.
- [AB01] Cyrille Artho and Armin Biere. Applying static analysis to large-scale, multi-threaded java programs. In *Proceedings 2001 Australian Software Engineering Conference*, pages 68–75. IEEE, 2001.
- [ABB⁺18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *EuroSys*, 2018.
- [ABC⁺12] Todd W Arnold, Carl Buscaglia, Felix Chan, Vincenzo Condorelli, John Dayka, William Santiago-Fernandez, Nihad Hadzic, Michael D Hocker, Michael Jordan, Thomas E Morris, et al. IBM 4765 cryptographic coprocessor. *IBM Journal of Research and Development*, 56(1.2):10–1, 2012.
- [ABC⁺16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, 2016. USENIX Association.

- [ACCSB09] Rafael V Aroca, Glaucio Caurin, and Sao Carlos-SP-Brasil. A real time operating systems (RTOS) comparison. In *Workshop de Sistemas Operacionais (WSO 09)*, 2009.
- [ACM17] Syed Taha Ali, Dylan Clarke, and Patrick McCorry. The Nuts and Bolts of Micropayments: a Survey. 2017.
- [AGJS13] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent 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*, volume 13. ACM New York, NY, USA, 2013.
- [AKSL18] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.
- [All70] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.
- [Ama19] Amazon. <https://www.amazon.com/>, 2019.
- [And01] Ross Anderson. Security Engineering: A Guide to Building Dependable Distributed Systems. John Wiley & Sons. Inc., Ny, 2001.
- [ARM09] ARM Ltd. Security technology: Building a secure system using TrustZone technology. *ARM Technical White Paper*, 2009.
- [ARM19] ARM Ltd. TrustZone. <https://www.arm.com/products/security-on-arm/trustzone>, 2019.
- [AS16] Sebastian Angel and Srinath Setty. Unobservable Communication Over Fully Untrusted Infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 551–569, 2016.
- [ASAP13] Niels Avonds, Raoul Strackx, Pieter Agten, and Frank Piessens. Salus: Non-hierarchical memory access rights to enforce the principle of least privilege. In *International Conference on Security and Privacy in Communication Systems*, pages 252–269. Springer, 2013.
- [ATG⁺16] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O’Keeffe, Mark L Stillwell, et al. SCONE: Secure Linux Containers with Intel SGX. In *OSDI*, 2016.

- [Baj02] Sundeep Bajikar. Trusted platform module (tpm) based security on notebook pcs-white paper. *Mobile Platforms Group Intel Corporation*, pages 1–20, 2002.
- [BBC⁺94] Jean-Paul Boly, Antoon Bosselaers, Ronald Cramer, Rolf Michelsen, Stig Mjølsnes, Frank Muller, Torben Pedersen, Birgit Pfitzmann, Peter De Rooij, Berry Schoenmakers, et al. The esprit project cafe—high security digital payment systems. In *European Symposium on Research in Computer Security*, pages 217–230. Springer, 1994.
- [BC02] Daniel P Bovet and Marco Cesati. Understanding the Linux Kernel. OReilly & Associates. Inc. October 2000, 2002.
- [BCD⁺17] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, Urs Müller, and Ahmad-Reza Sadeghi. DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization. *CoRR*, abs/1709.09917, 2017.
- [BCD⁺18] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The Guard’s Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *USENIX Security 2018*, pages 1213–1227. USENIX Association, 2018.
- [BCLK17] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. Roll-back and forking detection for trusted execution environments using lightweight collective memory. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 157–168. IEEE, 2017.
- [BDW17] Conrad Burchert, Christian Decker, and Roger Wattenhofer. Scalable Funding of Bitcoin Micropayment Channel Networks. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 361–377. Springer, 2017.
- [Bec14] B Beck. LibreSSL: An OpenSSL replacement. The first 30 days, and where we go from here. BSDCAN 2014, 2014.
- [BEM⁺17] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. Prochlo: Strong Privacy for Analytics in the Crowd. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 441–459. ACM, 2017.
- [BEMS⁺15] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. Tytan: tiny trust anchor for tiny devices. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.

- [BFM⁺08] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI C Specification Language, 2008.
- [BGNS06] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive*, 2006.
- [BHBN14] Swarup Bhunia, Michael S Hsiao, Mainak Banga, and Seetharam Narasimhan. Hardware trojan attacks: threat analysis and countermeasures. *Proceedings of the IEEE*, 102(8):1229–1247, 2014.
- [BJJ10] Sushil Bhardwaj, Leena Jain, and Sandeep Jain. Cloud computing: A study of infrastructure as a service (IAAS). *International Journal of engineering and information Technology*, 2(1):60–63, 2010.
- [BJZ⁺17] Iddo Bentov, Yan Ji, Fan Zhang, Yunqi Li, Xueyuan Zhao, Lorenz Breidenbach, Philip Daian, and Ari Juels. Tesseract: Real-time cryptocurrency exchange using trusted hardware. volume 2017, page 1153, 2017.
- [BKM⁺13] Michael Backes, Aniket Kate, Praveen Manoharan, Sebastian Meiser, and Esfandiar Mohammadi. AnoA: A Framework for Analyzing Anonymous Communication Protocols. In *2013 IEEE 26th Computer Security Foundations Symposium*, pages 163–178. IEEE, 2013.
- [BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.
- [BLMR14] Iddo Bentov, Charles Lee, Alex Mizrahi, and Meni Rosenfeld. Proof of activity: Extending bitcoin’s proof of work via proof of stake. ePrint Archive, Report 2014/452, 2014. <http://eprint.iacr.org/2014/452>.
- [blo18] blockchain.info. Average Confirmation Time. <https://blockchain.info/charts/avg-confirmation-time?timespan=all&daysAverageString=7>, 2018.
- [BMD⁺17a] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiaainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *USENIX 2017*. USENIX Association, 2017.

- [BMD⁺17b] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiaisen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: Sgx cache attacks are practical. page 33. Technical report, 2017.
- [BMHK08a] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI’08, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.
- [BMHK08b] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. USENIX Association, 2008.
- [BPH14] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI’14, pages 267–283, Berkeley, CA, USA, 2014. USENIX Association.
- [BPH15] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.
- [BS04] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, pages 57–72, 2004.
- [BWL⁺16] Stefan Brenner, Colin Wulf, Matthias Lorenz, Nico Weichbrodt, David Goltzsche, Christof Fetzter, Peter Pietzuch, and Rüdiger Kapitza. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Middleware*, 2016.
- [Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*, pages 136–145. IEEE, 2001.
- [CBC⁺18] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious Serializable Transactions in the Cloud. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 727–743, 2018.
- [CBZ⁺18] Shujie Cui, Sana Belguith, Ming Zhang, Muhammad Rizwan Asghar, and Giovanni Russello. Preserving Access Pattern Privacy in SGX-Assisted Encrypted Search. In *2018*

- 27th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9. IEEE, 2018.
- [CCW96] Eric K Clemons, David C Croson, and Bruce W Weber. Reengineering money: the mondex stored value card and beyond. volume 1, pages 5–31. Taylor & Francis, 1996.
- [CD16] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.
- [CDA14] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *ASPLOS*, 2014.
- [CEH⁺02] Mark Cox, Ralf Engelschall, Stephen Henson, Ben Laurie, et al. The OpenSSL Project. <https://www.openssl.org/>, 2002.
- [CGBM15] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An Anonymous Messaging System Handling Millions of Users. In *2015 IEEE Symposium on Security and Privacy*, pages 321–338. IEEE, 2015.
- [CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private Information Retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50. IEEE, 1995.
- [CGL⁺08] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *ASPLOS*, 2008.
- [CGPR15] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-Abuse Attacks Against Searchable Encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 668–679. ACM, 2015.
- [Cha88] David Chaum. The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. *Journal of cryptology*, 1(1):65–75, 1988.
- [CHB⁺15] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *NDSS*, 2015.

- [CJK⁺17] Anrin Chakraborti, Bhushan Jain, Jan Kasiak, Tao Zhang, Donald Porter, and Radu Sion. dm-x: Protecting volume-level integrity for cloud volumes and local block devices. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, page 16. ACM, 2017.
- [CL⁺99] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.
- [CL10] David Champagne and Ruby B Lee. Scalable architectural support for trusted software. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.
- [Cla16] Clang: a C language family frontend for LLVM. <http://clang.llvm.org>, 2016.
- [CLD16] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, 2016.
- [CMAM12] Alvin Cheung, Samuel Madden, Owen Arden, and Andrew C. Myers. Automatic Partitioning of Database Applications. *Proc. VLDB Endow.*, 2012.
- [CS13a] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.
- [CS13b] Stephen Checkoway and Hovav Shacham. Iago attacks: why the system call api is a bad untrusted rpc interface. In *ASPLOS*, volume 13, pages 253–264, 2013.
- [CS13c] Stephen Checkoway and Hovav Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’13*, pages 253–264, New York, NY, USA, 2013. ACM.
- [CSR⁺17] Franck Cassez, Anthony M Sloane, Matthew Roberts, Matthew Pigram, Pongsak Suvanpong, and Pablo Gonzalez de Aledo. Skink: Static analysis of programs in llvm intermediate representation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 380–384. Springer, 2017.
- [CST⁺10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010.
- [CWC⁺18] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in Hyperspace: Closing Hyper-

- Threading Side Channels on SGX with Contrived Data Races. In *SP 2018*, pages 178–194. IEEE, 2018.
- [CZK⁺18] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution. 2018.
- [CZRZ17] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjà vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 7–18. ACM, 2017.
- [CZWW17] Yue Chen, Yulong Zhang, Zhi Wang, and Tao Wei. Downgrade attack on trustzone. *arXiv preprint arXiv:1707.05082*, 2017.
- [Des94] Yvo G Desmedt. Threshold cryptography. *European Transactions on Telecommunications*, 5(4):449–458, 1994.
- [Dev19] Linux Kernel device mapper framework. <https://www.kernel.org/doc/Documentation/device-mapper>, 2019.
- [DFH18] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. General state channel networks. In *Proceedings of 2018 SIGSAC Conference on Computer and Communications Security*, pages 949–966. ACM, 2018.
- [DLP⁺01] Joan G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert Van Doorn, and Sean W. Smith. Building the IBM 4758 secure coprocessor. *Computer*, 34(10):57–66, 2001.
- [DM04] Christos Douligeris and Aikaterini Mitrokotsa. DDoS Attacks and Defense Mechanisms: Classification and State-of-the-art. *Comput. Netw.*, 2004.
- [DMC19] DMCrypt. <https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt>, 2019.
- [DMI19] DMIntegrity. <https://gitlab.com/cryptsetup/cryptsetup/wikis/DMIntegrity>, 2019.
- [DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. Technical report, Naval Research Lab Washington DC, 2004.
- [DMV19] DMVerity. <https://gitlab.com/cryptsetup/cryptsetup/wikis/DMVerity>, 2019.

- [Dou02] John R Douceur. The sybil attack. In *International workshop on peer-to-peer systems*, pages 251–260. Springer, 2002.
- [Dry16] Thaddeus Dryja. Unlinkable outsourced channel monitoring. https://youtu.be/Gzg_u9gHc5Q?t=2875, 2016.
- [DvDF⁺16] Srinivas Devadas, Marten van Dijk, Christopher W Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM. In *Theory of Cryptography Conference*, pages 145–174. Springer, 2016.
- [DW15] Christian Decker and Roger Wattenhofer. A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels. In *Stabilization, Safety, and Security of Distributed Systems - 17th International Symposium*, 2015.
- [Dwa16] DwarfPool. Why DwarfPool mines mostly empty blocks and only few ones with transactions. https://www.reddit.com/r/ethereum/comments/57c1yn/why_dwarfpool_mines_mostly_empty_blocks_and_only/, 2016.
- [Dwo07] Morris Dworkin. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Technical report, National Institute of Standards and Technology (NIST), 2007.
- [Eba19] Ebay. <https://www.ebay.com/>, 2019.
- [EEO⁺14] Dmitry Evtvyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. Iso-x: A flexible architecture for hardware-managed isolated execution. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 190–202. IEEE Computer Society, 2014.
- [Ern03] Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27. New Mexico State University Portland, OR, 2003.
- [EZ17] Saba Eskandarian and Matei Zaharia. ObliDB: Oblivious Query Processing Using Hardware Enclaves. *arXiv preprint arXiv:1710.00458*, 2017.
- [FGJ⁺09] Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andrew Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, and Ion Stoica. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):2009, 2009.

- [Fit04] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.
- [FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *Trans. Program. Lang. Syst.*, 1987.
- [Fra16a] Frama-C Software Analyzers. http://frama-c.com/what_is.html, 2016.
- [Fra16b] Frama-C Impact analysis plug-in. <http://frama-c.com/impact.html>, 2016.
- [Fra16c] Frama-C Slicing plug-in. <http://frama-c.com/slicing.html>, 2016.
- [Fre17] Free Software Foundation, Inc. Gcov - Using the GNU Compiler Collection (GCC). <https://gcc.gnu.org/onlinedocs/gcc/Gcov.html#Gcov>, 2017.
- [Gem14] Gemalto NV. No One is Immune to Breaches as 183 Million Accounts Compromised in Q3 2014. <https://safenet.gemalto.com/news/2014/q3-data-breaches-compromise-183-million-customer-accounts>, 2014.
- [GESM17] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *EUROSEC 2017*, pages 2:1–2:6. ACM, 2017.
- [GH12] John V Guttag and James J Horning. *Larch: Languages and Tools for Formal Specification*. Springer Science & Business Media, 2012.
- [GKM16] Pritam M Gharat, Uday P Khedker, and Alan Mycroft. Flow-and Context-Sensitive Points-To Analysis Using Generalized Points-To Graphs. In *SAS*, 2016.
- [GM84a] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.
- [GM84b] Shafi Goldwasser and Silvio Micali. Probabilistic Encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.
- [GM09] Eimear Gallery and Chris J Mitchell. Trusted computing: Security and applications. *Cryptologia*, 33(3):217–245, 2009.
- [GMSR⁺19] Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. Sok: Off the chain transactions. ePrint Archive, Report 2019/360, 2019. <https://eprint.iacr.org/2019/360>.
- [GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

- [Gol98] Oded Goldreich. Secure multi-party computation. *Manuscript. Preliminary version*, 78, 1998.
- [GPC⁺03] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 193–206. ACM, 2003.
- [GPKK12] Dimitris Geneiatakis, Georgios Portokalidis, Vasileios P. Kemerlis, and Angelos D. Keromytis. Adaptive Defenses for Commodity Software Through Virtual Application Partitioning. In *CCS*, 2012.
- [Gra09] David Grawrock. *Dynamics of a Trusted Platform: A building block approach*. Intel Press, 2009.
- [Gra16] GrammaTech, Inc. CodeSurfer. <https://www.grammatech.com/products/codesurfer>, 2016.
- [Gra19] Graphene Library OS with Intel Registered SGX Support. <https://github.com/oscarlab/graphene>, 2019.
- [GWA⁺15] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean Application Compartmentalization with SOAAP. In *CCS*, 2015.
- [Här02] Hermann Härtig. Security architectures revisited. In *ACM SIGOPS European Workshop*, pages 16–23, 2002.
- [HCP17] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 299–312, 2017.
- [Hex18] Hex-Five Security. Multizone: The first trusted execution environment for risc-v. <https://hex-five.com/>, 2018.
- [HKD⁺13] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *ASPLOS*, 2013.
- [HKZG15] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on Bitcoin’s peer-to-peer network. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 129–144, 2015.

- [HL11] Ben Hardekopf and Calvin Lin. Flow-sensitive Pointer Analysis for Millions of Lines of Code. In *CGO*, 2011.
- [HRB88] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. In *PLDI*, 1988.
- [HS15] Mike Hearn and Jeremy Spilman. Rapidly-adjusted micropayments to a pre-determined party. <https://en.bitcoin.it/wiki/Contract>, 2015.
- [HVCT10] Nicholas Hopper, Eugene Y Vasserman, and Eric Chan-Tin. How Much Anonymity Does Network Latency Leak? *ACM Transactions on Information and System Security (TISSEC)*, 13(2):13, 2010.
- [HZX⁺18] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. *ACM Transactions on Computer Systems (TOCS)*, 35(4):13, 2018.
- [Ide16] Identity Theft Resource Center. 2016 Breach List. http://www.idtheftcenter.org/images/breach/ITRCBreachReport_2016.pdf, 2016.
- [IL07] Cynthia E Irvine and Karl Levitt. Trusted hardware: Can it be trustworthy? In *Proceedings of the 44th annual Design Automation Conference*, pages 1–4. ACM, 2007.
- [Ima19] Imagenet. http://download.tensorflow.org/example_images/flower_photos.tgz, 2019.
- [Ins14] Ponemon Institute. 2014 Fifth Annual Cost of Cyber Crime Study: Global. 2014.
- [Int14] Intel Corp. Software Guard Extensions Programming Reference, Ref. 329298-002US. <https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf>, 2014.
- [Int16a] Intel Corp. Intel Software Guard Extensions (Intel SGX) SDK. <https://software.intel.com/sgx-sdk>, 2016.
- [Int16b] Intel Corp. Intel Software Guard Extensions Remote Attestation End-to-End Example. <https://software.intel.com/en-us/articles/intel-software-guard-extensions-remote-attestation-end-to-end-example>, 2016.
- [Int17a] Intel Corp. Intel SGX SDK for Linux. https://download.01.org/intel-sgx/linux-1.8/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.8_Open_Source.pdf, 2017.

- [Int17b] Intel Corp. `sgx_create_monotonic_counter`. <https://software.intel.com/en-us/node/709160>, 2017.
- [Int18] Intel Corp. Resources and response to side channel l1 terminal fault. <https://www.intel.com/content/www/us/en/architecture-and-technology/l1tf.html>, 2018.
- [Int19] Intel Corp. Resources and Response to Side Channel L1TF. <https://www.intel.com/content/www/us/en/architecture-and-technology/l1tf.html>, 2019.
- [IPe19] IPe3. <https://iperf.fr>, 2019.
- [JLLK17] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. Sgx-bomb: Locking down the processor via rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, page 5. ACM, 2017.
- [Joh16] Johnson, Simon et al. Intel Software Guard Extensions: EPID Provisioning and Attestation Services. <https://software.intel.com/en-us/blogs/2016/03/09/intel-sgx-epid-provisioning-and-attestation-services>, 2016.
- [Jor15] Jordan Pearson. WikiLeaks Is Now a Target In the Massive Spam Attack on Bitcoin. https://motherboard.vice.com/en_us/article/ezvw7z/wikileaks-is-now-a-target-in-the-massive-spam-attack-on-bitcoin, 2015.
- [Jos17] Joseph Young. Analyst: Suspicious Bitcoin Mempool Activity, Transaction Fees Spike to 16. <https://cointelegraph.com/news/analyst-suspicious-bitcoin-mempool-activity-transaction-fees-spike-to-16>, 2017.
- [JP 17] JP Buntinx. F2Pool Allegedly Prevented Users From Investing in Status ICO. <https://themerple.com/f2pool-allegedly-prevented-users-from-investing-in-status-ico/>, 2017.
- [KB07] Boris Köpf and David Basin. An information-theoretic model for adaptive side-channel attacks. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 286–296. ACM, 2007.
- [KG17] Rami Khalil and Arthur Gervais. Revive: Rebalancing Off-Blockchain Payment Networks. volume 200, page 400, 2017.
- [KGH83] Ehud Karnin, Jonathan Greene, and Martin Hellman. On secret sharing systems. *IEEE Transactions on Information Theory*, 29(1):35–41, 1983.

- [Kin76] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [KL88] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information processing letters*, 29(3):155–163, 1988.
- [KLR⁺06] Dirk Kuhlmann, Rainer Landfermann, H Ramasamy, Matthias Schunter, Gianluca Ramunno, and Davide Vernizzi. An open trusted computing architecture—secure virtual machines enabling user-defined policy enforcement. *Web Page: http://www.opentc.net/activities/otc_HighLevelOverview/OTC_Architecture_High_level_overview.pdf*, 2006.
- [KNH19] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The CIFAR-10 dataset. <https://www.cs.toronto.edu/~kriz/cifar.html>, 2019.
- [Knu98] Donald Ervin Knuth. *The art of computer programming, , Volume III, 2nd Edition*. Addison-Wesley, 1998.
- [KPMR12] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 189–204, 2012.
- [KPW16] David Kaplan, Jeremy Powell, and Tom Woller. AMD Memory Encryption. 2016.
- [Kra03] Hugo Krawczyk. Sigma: The ‘sign-and-mac’ approach to authenticated diffie-hellman and its use in the ike protocols. In *Annual International Cryptology Conference*, pages 400–425. Springer, 2003.
- [KSH17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM*, 60(6):84–90, 2017.
- [KSP05] Klaus Kursawe, Dries Schellekens, and Bart Preneel. Analyzing trusted platform communication. In *ECRYPT Workshop, CRASH-CRyptographic Advances in Secure Hardware*, page 8, 2005.
- [L⁺01] Leslie Lamport et al. Paxos made simple. volume 32, pages 18–25, 2001.
- [LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.
- [Lan92] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

- [LCC⁺02] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th international conference on Supercomputing*, pages 84–95. ACM, 2002.
- [LEPS16] Joshua Lind, Ittay Eyal, Peter Pietzuch, and Emin Gün Sirer. Teechan: Payment channels using trusted execution environments. 2016.
- [Lig17] Lightning Network community. Lightning network daemon. <https://github.com/lightningnetwork/lnd>, 2017.
- [Lin14] Linaro. Open portable trusted execution environment. <https://www.op-tee.org/>, 2014.
- [LJC⁺17] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. A survey on the security of blockchain systems. *Future Generation Computer Systems*, 2017.
- [LJJ⁺17] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in Darkness: Return-Oriented Programming Against Secure Enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 523–539, 2017.
- [LKM19] WireGuard: Secure Network Tunnel. <https://lkm1.org/lkm1/2019/3/22/95>, 2019.
- [LL17] Iuon-Chang Lin and Tzu-Chun Liao. A survey of blockchain security issues and challenges. *IJ Network Security*, 19(5):653–659, 2017.
- [LNE⁺19] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. Teechain: a secure payment network with asynchronous blockchain access. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 63–79. ACM, 2019.
- [LSD19] LSDS. spectre-attack-sgx. <https://github.com/lsds/spectre-attack-sgx>, 2019.
- [lth19] lthread. <https://github.com/halayli/lthread>, 2019.
- [LTM⁺00] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *Acm Sigplan Notices*, 35(11):168–177, 2000.
- [LWZY19] Liang Liu, Rujia Wang, Youtao Zhang, and Jun Yang. H-ORAM: A Cacheable ORAM Interface for Efficient I/O Accesses. In *Proceedings of the 56th Annual Design Automation Conference 2019*, page 33. ACM, 2019.

- [LYBB14] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Pearson Education, 2014.
- [LZC⁺15] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *CCS*, 2015.
- [MAB⁺13] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *Hasp@ isca*, 10(1), 2013.
- [MAK⁺17] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback Protection for Trusted Execution. Cryptology ePrint Archive, Report 2017/048, 2017. <http://eprint.iacr.org/2017/048>.
- [Mar08] Andrew Martin. The ten-page introduction to trusted computing. 2008.
- [MB03] Subhas C Misra and Virendra C Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality. In *International Conference on Computational Science and Its Applications*, pages 724–732. Springer, 2003.
- [MBB⁺18a] Patrick McCorry, Surya Bakshi, Iddo Bentov, Andrew Miller, and Sarah Meiklejohn. Pisa: Arbitration outsourcing for state channels. volume 2018, page 582, 2018.
- [MBB⁺18b] Patrick McCorry, Chris Buckland, Surya Bakshi, Karl Wüst, and Andrew Miller. You sank my battleship! a case study to evaluate state channels as a scaling solution for cryptocurrencies. 2018.
- [MBKM17] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. Sprites: Payment channels that go faster than lightning. 2017.
- [MD05] Steven J Murdoch and George Danezis. Low-Cost Traffic Analysis of Tor. In *2005 IEEE Symposium on Security and Privacy (S&P’05)*, pages 183–195. IEEE, 2005.
- [Mem19] Memcached. What is Memcached? <https://memcached.org/>, 2019.
- [MHG18] Yuval Marcus, Ethan Heilman, and Sharon Goldberg. Low-resource eclipse attacks on ethereum’s peer-to-peer network. volume 2018, page 236, 2018.

- [MIE17a] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 69–90. Springer, 2017.
- [MIE17b] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *CHES 2017*, pages 69–90. Springer, 2017.
- [Mit05] Chris Mitchell. *Trusted computing*, volume 6. Iet, 2005.
- [MKL⁺02] Dejan S Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-peer computing, 2002.
- [MLQ⁺10] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *S&P*, 2010.
- [MMSK⁺17] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and privacy with payment-channel networks, 2017.
- [MPB10] Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Conqueror: tamper-proof code execution on legacy systems. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–40. Springer, 2010.
- [MPC⁺18] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An Efficient Oblivious Search Index. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 279–296. IEEE, 2018.
- [MPP⁺08] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Eurosys*, 2008.
- [Mus19] Musl. <https://www.musl-libc.orgs>, 2019.
- [Nak08] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. <http://www.bitcoin.org/bitcoin.pdf>, 2008.
- [Net] NetApp. What Is a Lift and Shift Cloud Migration? <https://cloud.netapp.com/blog/what-is-a-lift-and-shift-cloud-migration>.
- [NZSS17] Zhenyu Ning, Fengwei Zhang, Weisong Shi, and Weidong Shi. Position paper: Challenges towards securing hardware-assisted execution environments. In *Proceedings of the Hardware and Architectural Support for Security and Privacy*, page 6. ACM, 2017.

- [OMA⁺18] Dan O’Keeffe, Divya Muthukumaran, Pierre-Louis Aublin, Florian Kelbert, Christian Priebe, Josh Lind, Huanzhou Zhu, and Peter Pietzuch. Spectre attack against sgx enclave, 2018.
- [OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19(5):177–184, April 1984.
- [Ope18] Open Enclave SDK Community. Open enclave sdk. <https://openenclave.io/sdk/>, 2018.
- [Osc17] Oscarlab. Graphene-SGX. <https://github.com/oscarlab/graphene>, 2017.
- [OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Cryptographers’ Track at the RSA Conference*, pages 1–20. Springer, 2006.
- [OTK⁺18a] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting sgx enclaves from practical side-channel attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 227–240, 2018.
- [OTK⁺18b] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *USENIX ATC 2018*, pages 227–240. USENIX Association, 2018.
- [Pan19] Panoply Source Code. <https://github.com/shwetasshinde24/Panoply>, 2019.
- [PB17] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts. 2017.
- [PBWH⁺11a] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen Hunt. Rethinking the Library OS from the Top Down. In *ASPLOS*, 2011.
- [PBWH⁺11b] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. Rethinking the Library OS From the Top Down. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 291–304. ACM, 2011.
- [PCEM04] Marcus Peinado, Yuqun Chen, Paul England, and John Manferdelli. Ngsch: A trusted open system. In *Australasian Conference on Information Security and Privacy*, pages 86–97. Springer, 2004.
- [PD16] Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network: Scalable off-chain instant payments. Technical Report (draft 0.5.9.1). <https://lightning.network>, 2016.

- [Pea02] Siani Pearson. Trusted computing platforms, the next security solution. *HP Labs*, 2002.
- [PEC05] Marcus Peinado, Paul England, and Yuqun Chen. An overview of ngsch. *Trust. Comput.*, 6:115, 2005.
- [PGT10] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. Lkl: The linux kernel library. In *9th RoEduNet IEEE International Conference*, pages 328–333. IEEE, 2010.
- [Pla11] Global Platform. Global platform device technology tee system architecture. *Public Review Draft*, 2011.
- [PM03] Niels Provos and Nick Mathewson. libevent - An event notification library. <http://libevent.org/>, 2003.
- [Pon13] L Ponemon. Cost of data breach study: Global analysis. *Ponemon Institute sponsored by Symantec*, 2013.
- [PPFF16] Rafael Pires, Marcelo Pasin, Pascal Felber, and Christof Fetzer. Secure Content-Based Routing Using Intel Software Guard Extensions. In *Middleware*. ACM, 2016.
- [PPY18] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Cacheshuffle: A family of oblivious shuffles. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.
- [pro19] GNU project. The GNU C Library (glibc). <https://www.gnu.org/software/libc/>, 2019.
- [PSS17] Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 643–673. Springer, 2017.
- [PST17] Rafael Pass, Elaine Shi, and Florian Tramer. Formal abstractions for attested execution secure processors. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 260–289. Springer, 2017.
- [Ram94] G. Ramalingam. The Undecidability of Aliasing. *TOPLAS*, 1994.
- [Rea19] Read(2) Linux Programmer’s Manual. <http://man7.org/linux/man-pages/man2/read.2.html>, 2019.
- [RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*, 1995.

- [RLT15] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-Channels Through Obfuscated Execution. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 431–446, 2015.
- [Rub16] Rubinov, Konstantin and Rosculete, Lucia and Mitra, Tulika and Roychoudhury, Abhik. Automated Partitioning of Android Applications for Trusted Execution Environments. In *ICSE*, 2016.
- [SAB15] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/Big-DataSE/ISPA*, volume 1, pages 57–64. IEEE, 2015.
- [Sar01] Luis Francisco Gumarú Sarmenta. *Volunteer computing*. PhD thesis, Massachusetts Institute of Technology, 2001.
- [SC16] Sukhpal Singh and Inderveer Chana. A survey on resource scheduling in cloud computing: Issues and challenges. *Journal of grid computing*, 14(2):217–264, 2016.
- [SCA⁺19] Alex Shamis, Amaury Chamayou, Christine Avanesians, Christoph M. Wintersteiger, Edward Ashton, Felix Schuster, Cédric Fournet, Julien Maffre, Kartik Nayak, Mark Russinovich, Matthew Kerner, Miguel Castro, Thomas Moscibroda, Olga Vrousseau, Roy Schwartz, Sid Krishna, Sylvan Clebsch, and Olya Ohrimenko. Ccf: A framework for building confidential verifiable replicated services. Technical Report MSR-TR-2019-16, Microsoft, April 2019.
- [SCF⁺15] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *SECP*, 2015.
- [SCG⁺14] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 459–474. IEEE, 2014.
- [SCL⁺16] Rohit Sinha, Manuel Costa, Akash Lal, Nuno P Lopes, Sriram Rajamani, Sanjit A Seshia, and Kapil Vaswani. A Design and Verification Methodology for Secure Isolated Regions. In *PLDI*, 2016.
- [SCW00] Susan Stepney, David Cooper, and Jim Woodcock. An electronic purse: Specification, refinement and proof. Oxford University, 2000.

- [SEC18] SECBIT. How the winner got Fomo3D prize — A Detailed Explanation. <https://medium.com/coinmonks/how-the-winner-got-fomo3d-prize-a-detailed-explanation-b30a69b7813f>, 2018.
- [SGA07] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.
- [SGF18] Sajin Sasy, Sergey Gorbunov, and Christopher Fletcher. ZeroTrace: Oblivious Memory Primitives From Intel SGX. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.
- [SGG18] Abraham Silberschatz, Greg Gagne, and Peter B Galvin. *Operating system concepts*. Wiley, 2018.
- [SGM⁺16] Karin Sumongkayothin, Steven Gordon, Atsuko Miyaji, Chunhua Su, and Komwut Wipusitwarakun. Recursive M-ORAM: A Matrix ORAM for Clients With Constrained Storage Space. In *International Conference on Applications and Techniques in Information Security*, pages 130–141. Springer, 2016.
- [SGNB19] István András Seres, László Gulyás, Dániel A Nagy, and Péter Burcsi. Topological analysis of bitcoin’s lightning network. 2019.
- [SGR09] Nuno Santos, Krishna P Gummadi, and Rodrigo Rodrigues. Towards trusted cloud computing. *HotCloud*, 9(9):3, 2009.
- [SGRE04] Emin Gün Sirer, Sharad Goel, Mark Robson, and Dogan Engin. Eluding Carnivores: File Sharing With Strong Anonymity. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, page 19. ACM, 2004.
- [SGZ16] Edi Shmueli, Sergey Goffman, and Yoram Zahavi. Helping Protect Software Distribution with PSWD. In *SYSTOR*, 2016.
- [Sha79] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
- [Shi16] Shift Devices AG. Digital Bitbox. <https://github.com/digitalbitbox/mcu>, 2016.
- [SLM⁺19] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. *arXiv preprint arXiv:1905.05726*, 2019.
- [SLTTS17] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *NDSS*, 2017.

- [SOD05] G Edward Suh, Charles W O'Donnell, and Srinivas Devadas. Aegis: A single-chip secure processor. *Information Security Technical Report*, 10(2):63–73, 2005.
- [Sof16] Softpedia. Hackers Modify Water Treatment Parameters by Accident. <http://news.softpedia.com/news/hackers-modify-water-treatment-parameters-by-accident-502043.shtml>, 2016.
- [SOSD05] G Edward Suh, Charles W O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and implementation of the aegis single-chip secure processor using physical random functions. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 25–36. IEEE Computer Society, 2005.
- [SP12] Raoul Strackx and Frank Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS 2012)*, pages 2–13. ACM Press, 2012.
- [SP16a] Raoul Strackx and Frank Piessens. Ariadne: A Minimal Approach to State Continuity. In *USENIX Security*, 2016.
- [SP16b] Raoul Strackx and Frank Piessens. Ariadne: A Minimal Approach to State Continuity. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 875–892, Austin, TX, 2016. USENIX Association.
- [SPHH06] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing tcb complexity for security-sensitive applications: Three case studies. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 161–174. ACM, 2006.
- [SRSV15] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. Moat: Verifying Confidentiality of Enclave Programs. In *CCS*, 2015.
- [SS75] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.
- [SS03] Ahmad-Reza Sadeghi and Christian Stübke. Taming “trusted platforms” by operating system design. In *International Workshop on Information Security Applications*, pages 286–302. Springer, 2003.
- [SSP05] Ahmad-Reza Sadeghi, Christian Stubble, and Norbert Pohlmann. European multilateral secure computing base, 2005.

- [ST06a] Scott F Smith and Mark Thober. Refactoring programs to secure information flows. In *Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 75–84. ACM, 2006.
- [ST06b] Scott F. Smith and Mark Thober. Refactoring Programs to Secure Information Flows. In *PLAS*, 2006.
- [Sto96] Neil R Storey. *Safety critical computer systems*. Addison-Wesley Longman Publishing Co., Inc., 1996.
- [sup17] superfreek. BTC Spam attack. 200,000 unconfirmed transactions halts bitcoin. <https://steemit.com/cryptocurrency/@superfreak/btc-spam-attack-200-000-unconfirmed-transactions-halts-bitcoin.>, 2017.
- [SvDS⁺13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS’13, Berlin, Germany, November 4-8, 2013*, pages 299–310. ACM, 2013.
- [SYM⁺14] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. Protecting Users by Confining JavaScript with COWL. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI’14*, pages 131–146. USENIX Association, 2014.
- [SYTP85] Vincent Yun Shen, Tze-Jie Yu, Stephen M. Thebaut, and Lorri R. Paulsen. Identifying error-prone software—an empirical study. *IEEE Transactions on Software Engineering*, (4):317–324, 1985.
- [Sza97] Nick Szabo. The idea of smart contracts. volume 6, 1997.
- [TAB⁺14] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, page 9. ACM, 2014.
- [TDMK18] Ari Takanen, Jared D Demott, Charles Miller, and Atte Kettunen. *Fuzzing for software security testing and quality assurance*. Artech House, 2018.
- [Ten19] TensorFlow Benchmarks. <https://github.com/tensorflow/benchmarks>, 2019.

- [TGL⁺17] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A Distributed Metadata-Private Messaging System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 423–440. ACM, 2017.
- [The13] The Bitcoin Community. libsecp256k1. <https://github.com/bitcoin-core/secp256k1>, 2013.
- [The16a] The Bitcoin Community. Bitcoin Core version 0.13.1 released. <https://bitcoin.org/en/release/v0.13.1>, 2016.
- [The16b] The Linux-SGX community. Intel(R) Software Guard Extensions for Linux OS. <https://github.com/intel/linux-sgx>, 2016.
- [The17a] The Bitcoin community. M-of-N Multisig, Multisig Output. <https://bitcoin.org/en/glossary/multisig>, 2017.
- [The17b] The Ethereum community. Ethereum White Paper. <https://github.com/ethereum/wiki/wiki/White-Paper>, 2017.
- [The17c] The Raiden Network community. The raiden network. <https://raiden.network/>, 2017.
- [TLK⁺17] Muoi Tran, Loi Luu, Min Suk Kang, Iddo Bentov, and Prateek Saxena. Obscuro: A bitcoin mixer using trusted execution environments. volume 2017, page 974, 2017.
- [TMLL06] Richard Ta-Min, Lionel Litty, and David Lie. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *OSDI*, 2006.
- [TPV17] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, 2017.
- [TZL⁺16] Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge. Cryptology ePrint Archive, Report 2016/635, 2016. <http://eprint.iacr.org/2016/635>.
- [Uni19] Princeton University. PARSEC. <https://parsec.cs.princeton.edu/>, 2019.
- [UNR⁺05] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

- [VBMW⁺18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Fore-shadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018.
- [VDHLZZ15] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 137–152. ACM, 2015.
- [vDS19] vDSO manpage. <http://man7.org/linux/man-pages/man7/vdso.7.html>, 2019.
- [Ver06] TCG Mobile Trusted Module Specification Version. 0.9 (draft). *TCG*, September, 12, 2006.
- [VRS04a] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, 2004.
- [VRS04b] Robbert Van Renesse and Fred B Schneider. Chain Replication for Supporting High Throughput and Availability. In *6th Symposium on Operating Systems Design and Implementation*, volume 4, pages 91–104, 2004.
- [Vuk15] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-Work vs. BFT replication. In *International Workshop on Open Problems in Network Security*, pages 112–125. Springer, 2015.
- [WCGFJ12] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in Numbers: Making Strong Anonymity Scale. In *Presented as Part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 179–182, 2012.
- [WCP⁺17] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2421–2434. ACM, 2017.
- [WCS15] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 850–861. ACM, 2015.

- [Wei81] Mark Weiser. Program Slicing. In *ICSE*, 1981.
- [Wir19] WireGuard. <https://www.wireguard.com/>, 2019.
- [WKPK16] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *ESORICS*, 2016.
- [WSLD13] Yongzheng Wu, Jun Sun, Yang Liu, and Jin Song Dong. Automatically Partition Software into Least Privilege Components Using Dynamic Data Dependency Analysis. In *ASE*, 2013.
- [WST12] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A Parallel Oblivious File System. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 977–988. ACM, 2012.
- [XCP15] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P*, 2015.
- [YS08] Jisoo Yang and Kang G. Shin. Using Hypervisor to Provide Data Secrecy for User Applications on a Per-page Basis. In *VEE*, 2008.
- [ZCC⁺16] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 270–282. ACM, 2016.
- [ZCCZ11] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *SOSP*, 2011.
- [ZDK⁺17] Fan Zhang, Philip Daian, Gabriel Kaptchuk, Iddo Bentov, Ian Miers, and Ari Juels. Paralysis proofs: Secure dynamic access structures for cryptocurrencies and more. 2017.
- [ZZ15] Wei Zhang and Yu Zhang. *Lightweight Function Pointer Analysis*, page 439–453. Springer International Publishing, 2015.