Protecting Applications Using Trusted Execution Environments

Christian Priebe

Department of Computing
Imperial College London

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

October 2020

I dedicate this thesis to my parents Maren and Peter, my brothers Stefan and Thorsten, and my sister Kerstin.

DECLARATION OF ORIGINALITY

This thesis presents my work in the Department of Computing at Imperial College London during the time of my PhD studies.

Parts of the work were done in collaboration with other researchers. The chapters 1, 2, and 6 were exclusively composed and written by me. The chapters 3, 4, and 5 present collaborative work:

- **Chapter 3.** All work discussed in Chapter 3 is mine except for (i) the TEE host interface analyses in Section 3.2 which were done in collaboration with Shujie Cui and Joshua Lind; and (ii) a number of implementation aspects such as the initial integration of the musl c library and the Linux Kernel Library, the `futex` implementation, and in-enclave signal handling support, that were done in collaboration with other members of the LSDS research group.
- **Chapter 4.** The GLAMDRING partitioning framework is an extension of my MRes project on dynamic binary profiling and ILP-based partitioning for TEE applications [265]. My contributions are the overall framework design, the dynamic binary analysis and partitioning phase implementations, and the evaluation. GLAMDRING's static analysis component was implemented by Joshua Lind, and the code generation and hardening was developed in collaboration with Joshua Lind and Divya Muthukumaran.
- **Chapter 5.** ENCLAVEDB was developed in collaboration with Kapil Vaswani and Manuel Costa, Microsoft Research Cambridge. My main contribution is the logging and recovery protocol for providing confidentiality, integrity, and freshness of persisted data and its integration.

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

Christian Priebe

October 2020

COPYRIGHT

RELATED PUBLICATIONS

The following authored or co-authored publications have directly contributed to this thesis and are discussed in detail:

- *Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, Peter Pietzuch*. **SGX-LKL: Securing the Host OS Interface for Trusted Execution**. Currently under submission, 2020 [267].
- *Christian Priebe, Kapil Vaswani, Manuel Costa*. **EnclaveDB: A Secure Database using SGX**. In Proceedings of the IEEE Symposium on Security and Privacy (S&P), 2018 [266].
- *Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Ruediger Kapitza, Christof Fetzer, and Peter Pietzuch*. **Glamdring: Automatic Application Partitioning for Intel SGX**. In Proceedings of the 2017 USENIX Annual Technical Conference (ATC), 2017 [206].

The following authored or co-authored publications have not directly contributed but impacted this thesis in one way or another:

- *Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch*. **LibSEAL: Revealing Service Integrity Violations Using Trusted Execution**. In Proceedings of the 13th European Conference on Computer Systems (EuroSys), 2018 [17].
- *Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer*. **SCONE: Secure Linux Containers with Intel SGX**. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2016 [15].
- *Divya Muthukumaran, Dan O'Keeffe, Christian Priebe, David Eyers, Brian Shand, and Peter Pietzuch*. **FlowWatcher: Defending against Data Disclosure Vulnerabilities in Web Applications**. In Proceedings of the 22nd edition of the ACM SIGSAC Conference on Computer and Communications Security (CCS). ACM, 2015 [230].

ACKNOWLEDGEMENTS

First of all, I would like to thank Prof. Peter Pietzuch for his supervision, guidance, and encouragement throughout the last few years. Without his support, this thesis would not exist.

I also want to thank my friends and colleagues who accompanied and worked with me as part of the Large-Scale Distributed Systems/Large-Scale Data and Systems (LSDS) Group, the HiPEDS programme, and the Department of Computing: Alexandros, Andreas, April, Dan, Divya, Dom, Florian, George, Holger, Jana, Josh, Lukas, Luo, Marcel, Panos, Pierre-Louis, Pijika, Raul, Shujie, Simon, Victoria, and Will. Your company and support throughout the years made this time unforgettable for me. I would also like to thank Dr. Krysia Broda and Dr. Amani El-Kholy for always being there to help with any administrative question or problem I had.

Thank you to Kapil and Manuel and the systems and networking group at Microsoft Research Cambridge as well as the Cloud Security team at Google in Kirkland for the opportunities to intern with them and making the internships I did during my PhD inspirational and memorable experiences.

I also want to thank all the friends I made at Wood Lane and elsewhere in London that made moving to London so much easier and such an incredible experience: Antoine, Astrid, Dilan, Federico, Giorgia, Isaac, Inês, Kate, Marc, Nadesh, Paula, Sami, Sascha, Sergio, Stas, Stavros, Tina, and everyone else I have met in my time here.

Thank you Cheng, for your patience, your understanding, for keeping me sane, and for being there for me whenever I needed it.

Lastly, I would like to thank my parents, Maren and Peter, and my siblings, Stefan, Thorsten, and Kerstin for being the best family I could wish for and for always supporting and believing in me.

ABSTRACT

While cloud computing has been broadly adopted, companies that deal with sensitive data are still reluctant to do so due to privacy concerns or legal restrictions. Vulnerabilities in complex cloud infrastructures, resource sharing among tenants, and malicious insiders pose a real threat to the confidentiality and integrity of sensitive customer data. In recent years trusted execution environments (TEEs), hardware-enforced isolated regions that can protect code and data from the rest of the system, have become available as part of commodity CPUs. However, designing applications for the execution within TEEs requires careful consideration of the elevated threats that come with running in a fully untrusted environment. Interaction with the environment should be minimised, but some cooperation with the untrusted host is required, e.g. for disk and network I/O, via a *host interface*. Implementing this interface while maintaining the security of sensitive application code and data is a fundamental challenge.

This thesis addresses this challenge and discusses how TEEs can be leveraged to secure existing applications efficiently and effectively in untrusted environments. We explore this in the context of three systems that deal with the protection of TEE applications and their host interfaces:

SGX-LKL is a library operating system that can run full unmodified applications within TEEs with a *minimal general-purpose host interface*. By providing broad system support inside the TEE, the reliance on the untrusted host can be reduced to a minimal set of low-level operations that cannot be performed inside the enclave. SGX-LKL provides transparent protection of the host interface and for both disk and network I/O.

GLAMDRING is a framework for the semi-automated partitioning of TEE applications into an untrusted and a trusted compartment. Based on source-level annotations, it uses either dynamic or static code analysis to identify sensitive parts of an application. Taking into account the objectives of a *small TCB size* and *low host interface complexity*, it defines an *application-specific host interface* and generates partitioned application code.

ENCLAVEDB is a secure database using Intel SGX based on a partitioned in-memory database engine. The core of ENCLAVEDB is its logging and recovery protocol for transaction durability. For this, it relies on the database log managed and persisted by the untrusted database server. ENCLAVEDB *protects against advanced host interface attacks* and ensures the confidentiality, integrity, and freshness of sensitive data.

TABLE OF CONTENTS

List of Figures

LIST OF TABLES

List of Algorithms

List of Abbreviations

ABE	Attribute-Based Encryption
AEAD	Authenticated Encryption with Associated Data
AEX	Asynchronous Exit
AIK	Attestation Identity Key
ASLR	Address Space Layout Randomisation
AST	Abstract Syntax Tree
CAR	Cache-as-RAM
CAT	Cache Allocation Technology
CIP	Confidentiality and Integrity Protection
CP	Cloud Provider
CPT	Compartment Page Table
CRTM	Core Root of Trust for Measurements
DBMS	Database Management System
DOS	Denial-of-Service
EDL	Enclave Definition Language
EK	Endorsement Key
EPC	Enclave Page Cache
EPCM	Enclave Page Cache Map
EPID	Enhanced Privacy ID
FDE	Full Disk Encryption
FE	Functional Encryption
FHE	Fully Homomorphic Encryption
HID	Hardware Identifier
IAS	Intel Attestation Service
IBE	Identity-Based Encryption

ILP	Integer Linear Programming
IPC	Inter-Process Communication
ISA	Instruction Set Architecture
KMS	Key Management System
LCP	Launch Control Policy
LKL	Linux Kernel Library
LLC	Last Level Cache
LOC	Lines of Code
MAC	Message Authentication Code
MC	Memory Controller
ME	Management Engine
MEE	Memory Encryption Engine
MKTME	Multi-Key Total Memory Encryption
MLE	Measured Launch Environment
NVRAM	Non-Volatile Random Access Memory
OLTP	Online Transaction Processing
OPE	Order-Preserving Encryption
ORAM	Oblivious Random Access Memory
OS	Operating System
PAL	Piece of Application Logic
PCR	Platform Control Register
PDG	Program Dependence Graph
PFSL	Protected File System Library
PHE	Partial Homomorphic Encryption
PKI	Public Key Infrastructure
PLC	Protected Code Launcher
PRM	Processor-Reserved Memory
PS	Partition Specification

PUF	Physical Unclonable Function
ROP	Return-Oriented Programming
SECS	SGX Enclave Control Structure
SEV	Secure Encrypted Virtualisation
SGX	Software Guard Extensions
SLB	Secure Loader Block
SML	Stored Measurement Log
SMM	System Management Mode
SP	Service Provider
SRK	Storage Root Key
SSA	Stack Save Area
SVM	Secure Virtual Machine
TCB	Trusted Computing Base
TCS	Thread Control Structure
TE	Trusted Execution
TEE	Trusted Execution Environment
TLS	Thread-Local Storage
TOCTOU	Time-of-Check to Time-of-Use
TPM	Trusted Platform Module
TSM	Trusted Software Module
TSX	Transactional Memory Extensions
TXT	Trusted Execution Technology
VA	Version Array
VFS	Virtual File System
VMM	Virtual Machine Monitor
VPN	Virtual Private Network
XOM	Execute Only Memory

CHAPTER 1

INTRODUCTION

Cloud computing has long become an essential part of the internet and a large number of services are run on public cloud infrastructure. Still, organisations in industries such as insurance, finance, healthcare, and the public sector that deal with sensitive data are reluctant to move their services to the cloud [236]. The lack of insight and loss of control over their data and applications raises concerns regarding both confidentiality and integrity of application data and might prevent the use of public clouds due to legal or regulatory requirements. At the same time, organisations that do store and process sensitive data in the cloud experience frequent security incidents as a result of immature security practices, misconfigured servers, or insider threats [322].

There are a number of ways how both service providers and cloud providers try to adhere to strong privacy and security requirements. To protect the confidentiality of data, data is encrypted, both *at rest*, e.g. when stored on persistent disks, and *in transit*, e.g. when sent over the network. Nowadays, public cloud providers such as Amazon, Google, and Microsoft all provide capabilities to transparently encrypt stored data [9, 135, 224]. Even internal network traffic, e.g. between data centers, is encrypted to protect against external adversaries that are capable of intercepting or observing data in transit [10, 136, 223].

However, this is insufficient for services that also process data. Processing data typically requires data to be available in unencrypted form. Such *data in use* is stored in non-persistent memory such as RAM or CPU caches, or CPU registers at runtime. Application data in main memory is not only accessible by the process owning the data but also by higher-privileged components such as the operating system or hypervisor. A bug in one of these components might allow other tenants with whom the machine is shared or external attackers that gain access to the system to access sensitive data [144]. Malicious insiders who have access to the system are another potential threat [98]. While not as easy to access, data stored in CPU caches can potentially be exposed to an attacker even without privileged system access, through side-channel attacks such as Meltdown [207], Spectre [191], and Foreshadow [345].

Lastly, physical attacks such as bus snooping [275] or cold boot attacks [154] can also reveal sensitive data such as cryptographic keys residing in main memory.

1.1 Protecting Data in Use

Advanced cryptography can be used to protect data in use in some cases. *Homomorphic encryption* allows operations to be directly performed on encrypted data without revealing operands or results in plaintext. With *fully homomorphic encryption* (FHE) [125], arbitrary computations can be performed on ciphertext. However, FHE is impractical for most real world use cases due to its significant performance overhead [233]. Efficient *partially homomorphic* cryptographic systems that support a subset of operations on ciphertext exist [25, 102, 129, 249, 279] but are restricted in their use cases.

Cloud providers also try to provide strong isolation between tenants, most commonly enforced through the use of virtual machines (VMs) managed by low-level hypervisors. Containers are another way of isolating execution contexts. In contrast to VMs, container isolation is enforced by a shared underlying operating system rather than a hypervisor. Containers are generally considered to be more efficient in regard to both memory consumption and start up times, and simplify deployment as there is no OS to be managed. However, OS-based container isolation is also considered to be weaker than hypervisor-based VM isolation. Therefore, containers are commonly deployed on top of virtual machines. More recently, hybrid approaches have been proposed that aim to provide the resource efficiency and simplicity of containers, and the security guarantees of VMs. For example, *Kata Containers* [186] provide a container runtime that transparently deploys containers on top of a hypervisor. *Firecracker* [18] is a minimal hypervisor that provides lightweight *micro VMs* with start up times and memory footprints comparable to those of container runtimes.

Any Hypervisor- or container-based solution assumes trust in the cloud provider, not only to be non-malicious but also to keep the corresponding infrastructure secure and free of exploitable vulnerabilities. While it is reasonable to assume that cloud providers try their best to secure cloud platforms, tenants are reliant on the cloud provider to secure their infrastructure and to stay honest. However, as past incidents have shown vulnerabilities and misconfigurations of the complex cloud infrastructure as well as insider attacks are a real threat [98, 150, 322].

In addition to software-based isolation, a number of hardware-based solutions have been proposed in the past. One way to provide secure computation is by offloading computation to another trusted hardware component such as specialised FPGAs or secure coprocessors. Data is always encrypted while residing on the main system. For secure computations, it is transferred to the trusted hardware, decrypted, processed, and re-encrypted before it is transferred back. However, these solutions typically suffer from poor performance e.g. due to low bandwidth, low amounts of available memory, or low processing power. Furthermore, due to their specialised nature public cloud providers are less likely to make them available to customers.

In order to be broadly adopted, hardware-based solutions should be tightly integrated with commodity hardware. *Trusted Platform Modules* (TPMs) are secure cryptoprocessors available on most commodity systems, typically as dedicated chips or as part of integrated chipsets on the mainboard. TPMs provide a predefined set of cryptographic functions as specified in the TPM specifications [338, 339]. These include implementations of symmetric and asymmetric cryptographic algorithms and hash functions, and a random number generator. A fundamental feature of TPMs is the ability to measure a system's integrity. For this, TPMs compute cryptographic digests of software and software configurations and store them in TPM-internal registers. This can include measurements of the BIOS, the OS, and the OS configuration. These measurements are obtained before the software is run and cannot be reset at runtime. A party that wants to verify the system's integrity can remotely attest to it by requesting the registers' values from the TPM and comparing them to known-good measurements. They are signed by the TPM and therefore cannot be tampered with or forged without detection. Only after successful verification, the remote party transmits sensitive data for processing.

TPMs allow attestation of a system's state at a specific point in time but do not prevent changes to the system at a later time, in particular by previously unmeasured components. They rely on a trusted low-level *Core Root of Trust for Measurements* (CRTM) software component that performs a TPM measurement of the BIOS. The BIOS will measure the next component in the boot chain which will in turn create a measurement for subsequent components, ultimately forming a chain of measurements. For a remote party to verify the integrity of a system, it has to keep track of measurements of all components and all acceptable system states. With the complexity of commodity operating systems and OS configurations this can be challenging. CPU extensions such as Intel's Trusted Execution Technology (TXT) [142] rely themselves on TPMs but add a *late launch* feature. With this it is

possible to put the system in a clean state at runtime and then measure only components that are to be executed. However, TXT has also been shown to be vulnerable via higher-privileged system components [97, 103, 365].

While TPMs and TXT are aimed at securing the integrity of a system, more recently CPU manufacturers have introduced new extensions to protect the confidentiality of data in use, even in the presence of powerful attackers. AMD's *Secure Encrypted Virtualization* (SEV) [185] and Intel's *Multi-Key Total Memory Encryption* (MKTME) [178] technologies provide support for encrypted memory for individual virtual machines. Data is only decrypted within the CPU package itself and corresponding keys are generated and managed by the CPU or provided securely by the VM owner. Other system components such as hypervisors have no access to these encryption keys. Even with physical access to main memory or the system bus, an attacker would only be able to observe encrypted data exchanged between the CPU and main memory. Encryption and decryption is implemented efficiently with hardware acceleration. Both technologies are aimed at providing strong isolation of full virtual machines, not only from other VMs but also from an untrusted hypervisor and other system software. They provide no protection against an attacker that compromises the VM itself. The *Trusted Computing Base* (TCB) therefore includes full VMs including guest operating systems and all other code executed on the VM. Another problem with SEV and MKTME is their lack of integrity protection. As a result, attackers can bypass their protection mechanisms and potentially extract secrets, e.g. via memory replay attacks [159] or malicious modifications to page table mappings that are still under the control of the hypervisor [229].

1.2 Trusted Execution Environments

Trusted execution environment (TEE) technologies such as ARM *TrustZone* [14] and Intel *Software Guard Extensions* (SGX) [218] combine integrity and confidentiality protection. These technologies are available as instruction set extensions in recent commodity CPUs. Instead of protecting the whole system or full virtual machines they protect a subset of memory and execute code in full isolation from the rest of the system, as illustrated in Fig. 1.1. Data and code residing within a TEE such as an Intel SGX *enclave* cannot be accessed or tampered with by an external entity, including privileged software and adversaries with physical access to the machine and the ability to probe main memory or

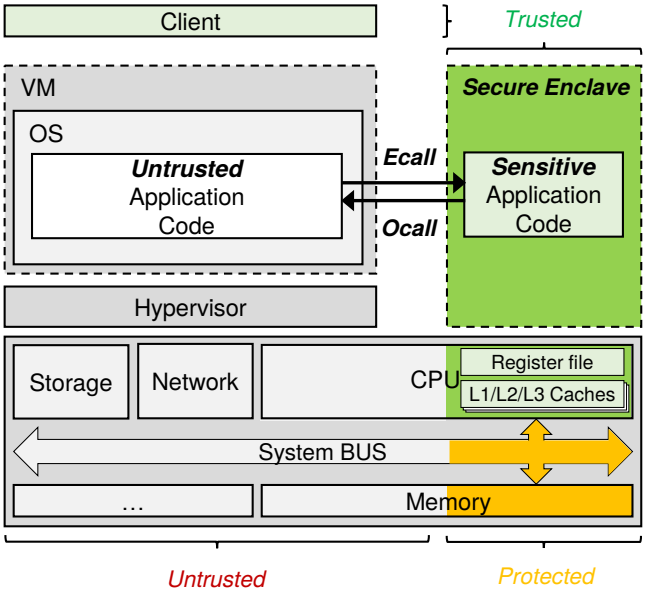


Fig. 1.1: Isolated execution with Intel SGX

snoop on the system bus. Therefore, TEEs can effectively combine both confidentiality and integrity protection, even in the face of a powerful adversary.

Arm TrustZone divides execution into two *worlds*, the *Normal World*, and the *Secure World*. For the execution of code in either world, the CPU explicitly switches to a corresponding mode. Code run in the Secure World is strictly higher privileged than code in the Normal World. It has full control over code and data residing in the Normal World. Data within the Secure World is inaccessible from the Normal World. Critical security-sensitive data can therefore be kept and processed in the Secure World by a small OS kernel. One use case is that of a security monitor which can detect anomalies in the Normal World.

In contrast to the TrustZone Secure World, SGX enclaves are unprivileged. Rather than allowing for a single isolated privileged component, Intel SGX supports the creation of multiple enclaves that are isolated both from each other and from the rest of the system. At the same time, due to being unprivileged, they can still be managed and controlled by the host system. This makes Intel SGX a good fit for cloud environments in which mutually untrusted tenants share resources provided by a potentially untrusted cloud provider. While still being a relatively new technology, a number of cloud providers have already introduced TEE-based confidential computing offerings as part of their

platforms [76] and TEE technologies such as Intel SGX have been the subject of an expanding body of research.

However, as enclaves are unprivileged, they are restricted in what they can do without cooperation from the host system. In particular, any privileged operation such as reading from or writing to disk or the network requires enclave code to interact with its host. For this, every enclave has an interface of entry and exit points corresponding to calls into (*ecalls*) and out of (*ocalls*) an enclave. A fundamental challenge this inherits is the design of a *host interface* that is expressive enough for potentially complex applications to run inside an enclave while not compromising the confidentiality and integrity of the enclave secrets. In particular, sensitive data must be protected from being exposed when data is passed to the untrusted host as part of *ocalls*. Similarly, any *ecalls* must not compromise the integrity of data and computations within an enclave.

Enclave applications can be developed using dedicated software development kits (SDKs) [114, 170, 239]. These applications consist of an untrusted and a trusted component. The trusted component operates on sensitive data and runs inside the enclave. There is an incentive to keep the amount of code running inside the enclave small as it contributes to the TCB, the sum of all components that are inherently trusted not to be malicious or exploitable. Vulnerabilities in the enclave code or Intel SGX itself could be exploited to reveal sensitive enclave data or to compromise its integrity. The untrusted component incorporates non-sensitive functionality. It also acts as an intermediary between its trusted counterpart inside an enclave and the rest of the system. The host interface between them has to be defined by the developer. SDKs help with some aspects of implementing the interface, e.g. with copying buffers provided as parameters out of or into the enclave, to make them accessible outside, or to prevent time-of-check to time-of-use (TOCTOU) attacks [2]. However, other aspects of the interface design and implementation are left to the developer and often require knowledge of application semantics. This includes the partitioning of the application into untrusted and trusted components and the understanding of what data is sensitive and therefore must be encrypted and decrypted as well as integrity-protected and -checked when transferred via the host interface.

1.3 Problem Statement

This is a fundamental problem for porting *existing* applications that have not been developed with the constraints of TEEs such as Intel SGX in mind. Operating systems and other components of the

systems they run on are assumed to be trustworthy. Therefore, privileged operations such as disk and network I/O are simply delegated to the OS via system calls. The redevelopment of applications under the specific constraints and requirements of TEEs requires not only time and effort but also expertise. While it can be an option for some applications, manually rewriting more complex applications is infeasible in many cases.

In this thesis, we focus on the secure execution of existing applications within TEEs and address accompanying challenges. In particular, we look at the following three challenges.

(1) Complex system support requirements. Many complex applications rely heavily on external system support, both from the OS and system libraries such as the C standard library. Besides access to hardware resources such as storage and the network, this includes features such as multithreading, signal support, dynamic library loading, and memory management. The basic assumption of a trusted host system which includes the OS and system libraries no longer holds in the context of TEEs.

In addition, TEEs such as Intel SGX neither allow direct hardware access nor system calls that would allow an application to access host OS services. Existing solutions [15, 23, 301, 341] have shown that it is possible to redirect system support requests of TEE applications to the host OS without requiring elaborate application changes. However, as the host OS is untrusted, such approaches expose potentially sensitive application state via a large host interface.

Instead, for applications that cannot be redesigned easily to reduce system support requirements of their trusted component, application exposure to the untrusted host should be *minimised*. To accomplish this, system support must be provided within the TEE, allowing for a *minimal general-purpose host interface* that only relies on the host OS when cooperation is unavoidable.

(2) Competing objectives for application-specific host interfaces. Relying on a general-purpose host interface might be the best choice for some applications, in particular applications that cannot be redesigned easily, e.g. due to their complexity or a monolithic design. However, there are other considerations for TEE application designs besides the *host interface complexity*, as illustrated in Fig. 1.2.

When running full applications inside TEEs, all application code runs with the same privileges and has access to sensitive data within the TEE, leading to a large *TCB size*. However, typically only a subset of application code requires access to sensitive data. As a large TCB size increases the risk of software

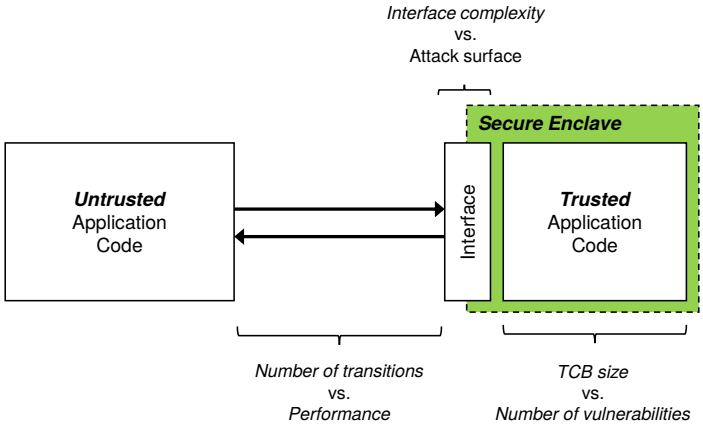


Fig. 1.2: Host interface trade-offs

vulnerabilities that might allow an attacker to access sensitive data [226, 297], it is desirable to restrict access to sensitive data to code that requires access. If possible, applications should therefore be partitioned into trusted and untrusted components. However, identifying security-sensitive code and data is non-trivial for complex existing applications. For example, some code paths and application outputs only indirectly depend on sensitive data.

Lastly, partitioned applications might require frequent interaction between untrusted and trusted code, leading to large numbers of TEE transitions. As we will discuss later, TEE transitions are expensive and can have a significant impact on application performance (see Section 2.4.2.5). Therefore, for performance-critical applications, the *number of TEE transitions* becomes an important partitioning objective.

Note that these are competing *partitioning objectives* and that there are a number of trade-offs between the different aspects of TEE host interfaces. These challenges demonstrate the complexity of redesigning existing applications for TEE execution and the design of appropriate *application-specific host interfaces*.

(3) Protecting the host interface. All applications have an interface that enables users to interact with them. For regular applications this can be a command line interface or a graphical user interface through which inputs are provided to the application. These inputs are validated, e.g. to check whether they are of the right type or within a expected range, and sanitised, e.g. by escaping untrusted input parameters. However, input validation and sanitisation alone are not enough in a scenario in which inputs are provided via an untrusted party or cooperation of the untrusted party is required to transmit

output data. The security of the host interface is fundamental for the security of both code that is executed, and data that is stored within a TEE. In particular, a host interface must

1. *protect the confidentiality of sensitive data*. Applications interact with clients and other trusted nodes remotely. Input and output data provided by another trusted party or sent to another party can contain sensitive data that must not be exposed to an adversary possibly in control of the untrusted environment such as the host the TEE runs on, or the network used for communication between the two parties. Similarly, any other interaction with the untrusted host, e.g. when transferring control to untrusted code, must not expose sensitive data;

2. *protect the integrity of sensitive code and data*. As data is transmitted through an untrusted environment, an adversary is able to tamper with input data by changing, omitting, or replacing it. Therefore the authenticity and integrity of input data provided must be verified before it is processed by TEE code. This is true for both confidential and non-confidential input data such as configuration options as they might enable an adversary to compromise trusted computations or trick TEE code into revealing secrets. Similarly, any output data transmitted to a remote party or temporarily persisted on untrusted storage, must be protected from tampering; and

3. *protect against advanced attacks*. TEEs address powerful attackers with complete control over the untrusted environment. This exposes the TEE itself to a number of advanced attacks. For example, an adversary might

   (a) compromise the integrity of TEE code or trick it into exposing sensitive data, e.g. through *Iago attacks* [62];

   (b) launch *freshness attacks* in which trusted code is provided genuine but out-of-date data, or in which genuine messages or inputs are provided to the enclave more than once without being instructed to do so; or

   (c) leverage *side channels* to passively observe access patterns or call sequences that can reveal sensitive application activities and with that potentially sensitive data.

1.4 Contributions

This thesis aims to address and present solutions to these challenges. We look at them in the context of three different systems that will be presented and discussed in detail in the remainder of the thesis.

1.4.1 SGX-LKL: Protecting Unmodified Binaries with a Minimal Host Interface

We describe **SGX-LKL**, a new TEE runtime system that executes unmodified Linux binaries inside Intel SGX enclaves while exposing a minimal, protected host interface. With SGX-LKL we explore how TEEs can be used to protect applications with *complex system support requirements*. In particular, SGX-LKL makes three contributions:

Minimal host interface. SGX-LKL is a library OS that runs alongside an application within the TEE. It provides extensive system support inside Intel SGX enclaves with minimal reliance on the host system. As a result, SGX-LKL has a *minimal general-purpose host interface*, only relying on the host for disk and network I/O operations that cannot be provided within the TEE.

Host interface protection. As part of SGX-LKL, we explore how to *transparently protect the host interface* of an application. SGX-LKL protects both the confidentiality and integrity of disk I/O and network I/O. For disk I/O, SGX-LKL provides transparent encryption and decryption of persistent data via full disk encryption. To protect the confidentiality, integrity, and freshness of network traffic, SGX-LKL uses a layer-3 Virtual Private Network (VPN).

Secure distributed application deployments. While TEEs provide attestation primitives, it is up to TEE applications to use these primitives to allow remote parties to remotely attest that they have been set up correctly and are protected by a genuine TEE. For this, SGX-LKL integrates transparent support for (i) *remote attestation*; (ii) the *secure deployment* of applications, including distributed deployments with multiple enclaves; and (iii) the secure *provisioning of enclave secrets* such as cryptographic keys and sensitive application arguments.

We demonstrate SGX-LKL's capability to run complex applications with minimal reliance on the host system and evaluate its performance on the JVM and a distributed TensorFlow setup as well as a set of microbenchmarks for network and disk I/O.

1.4.2 Glamdring: Automated Partitioning for Application-Specific Host Interfaces

We present GLAMDRING, a semi-automated framework for partitioning applications for the execution in TEEs. In contrast to SGX-LKL, GLAMDRING explores the other end of the spectrum of TEE system designs and aims for a *small TCB size*. It identifies security-sensitive parts of an application in its codebase, generates a partition specification, and generates the code for a partitioned application with an untrusted and trusted application component.

We explore two different methodologies for identifying security-sensitive application code and determining a partitioning, both based on an initial annotation of application code that indicates security-sensitive input and output data:

1. *Dynamic taint tracking*. First, we explore the possibility to determine security-sensitive application components dynamically. To do this, we present a Valgrind-based taint tracking tool that tracks the propagation of sensitive data throughout application code at runtime. In addition, it keeps track of memory allocations, performed system calls, and collects profiling data.

2. *Static analysis*. Second, we use static analysis to determine security-sensitive application components from annotated source code. We use *dataflow analysis* to identify code that might be exposed to sensitive data and should be treated as confidential, and *backward slicing* to identify code that can affect the integrity of sensitive data.

We discuss and compare benefits and shortcomings of both methodologies.

The collected data is used as input to an integer linear programming (ILP) solver together with an objective function that weighs different objectives such as a *small TCB size*, *low interface complexity*, and a *low number of transitions*. Based on this, the ILP solver generates a *partition specification* that splits the full application code into the corresponding untrusted and trusted components.

The specification is used by a code generator in order to automatically transform application code into an enclave library and its untrusted counterpart. Runtime checks and cryptographic transformations are added to the host interface to protect invariants, and the confidentiality and integrity of sensitive data, respectively.

1.4.3 EnclaveDB: A Secure Database with Advanced Host Interface Attack Protection

Lastly, we present EnclaveDB, a secure database that uses Intel SGX to protect the confidentiality, integrity, and freshness of data. EnclaveDB is based on the SQL Server Hekaton in-memory engine and is partitioned to achieve a small TCB. Besides sensitive tables, indexes, and metadata, only parts of the engine are placed inside an Intel SGX enclave.

With EnclaveDB, we focus on the risk of advanced host interface attacks and how to protect against them. The main contribution of EnclaveDB is its novel logging and recovery protocol which interacts with the host to provide durability for database data. The protocol

(i) protects the confidentiality and integrity of data. The protocol is used to persist data on disk in order to be able to recover it in case of a crash. EnclaveDB must rely on the host to store and retrieve data from disk. As the host is untrusted, the protocol protects data from tampering and ensures both its confidentiality and its integrity.

(ii) protects against advanced host interface attacks. Protecting the confidentiality and integrity of data exchanged with the host is not sufficient to protect against advanced host interface attacks. An adversary might replace data with genuine but out-of-date data or might selectively direct database requests to separate database instances. EnclaveDB's protocol integrates monotonic counters to prevent such *freshness and forking attacks*.

(iii) maintains high performance. EnclaveDB is based on a *high-performance* in-memory engine. The protocol is designed to provide confidentiality, integrity, and freshness protection of sensitive data while maintaining high performance. For this, it supports asynchronous append and truncation operations and minimises synchronisation between threads that process transactions.

With this protocol, EnclaveDB demonstrates how to augment the advantages of application partitioning through a customised host interface in order to improve both performance and security guarantees.

We show the protocol's efficiency by evaluating EnclaveDB using the TPC-C and TATP database benchmarks and a performance model to simulate overheads for large enclaves to show that EnclaveDB can achieve strong security guarantees with low overhead.

1.5 Outline

The remainder of this thesis is structured as follows:

Chapter 2 provides a detailed problem statement and an introduction to TEEs. It first discusses alternatives to TEEs, in particular computing on encrypted data using advanced cryptographic schemes and software-based secure execution. Next, we describe TPM-based Trusted Computing as a predecessor of modern TEE technologies and how systems use it to protect applications. This is followed by a survey of TEE technologies from academia and industry. Intel SGX is introduced in detail as the TEE technology the three systems presented in this thesis are based on. We then discuss TEE attacks and mitigations and the targeted threat model.

Chapter 3 presents SGX-LKL, a library OS based approach of running unmodified binaries within an Intel SGX enclave with a minimal host interface. We discuss host interfaces of existing TEE runtime systems and their limitations. We present an alternative minimal host interface derived from the requirements of complex applications and the constraints of Intel SGX. We then discuss how SGX-LKL implements system support on top of this minimal interface using LKL and other components such as user-level threading. We talk about how the remaining host interface is protected, and evaluate SGX-LKL's performance.

Chapter 4 discusses application-specific host interfaces and presents GLAMDRING, a framework for partitioning applications for the execution in TEEs with a minimal TCB. We look at two methodologies for determining security-sensitive code and data to partition an application into an untrusted and a trusted component: dynamic runtime taint tracking, and static code analysis. We then discuss how GLAMDRING generates code for the partitioned application, and evaluate the framework by applying it to a number of existing applications.

Chapter 5 presents ENCLAVEDB, a secure high-performant database based on Intel SGX, and addresses the protection of TEE applications against advanced host interface attacks. We first discuss how we partition the SQLServer-based in-memory engine Hekaton, the core of ENCLAVEDB, to reduce the TCB size and to keep the host interface small. The focus of this chapter then lies on the protection of the host interface. We present a logging and recovery protocol that (i) supports high-throughput transactions; (ii) guarantees transaction durability; (iii) protects the confidentiality

and integrity of user data; and (iv) prevents advanced attacks such as replay and forking attacks. Lastly, we evaluate ENCLAVEDB's performance.

Chapter 6 summarises and concludes the thesis.

CHAPTER 2

BACKGROUND

This chapter covers background and related work for this thesis. In particular, it discusses different ways of protecting computation on sensitive data in untrusted environments and introduces TEEs. The chapter is structured as follows:

Need for secure execution. Section 2.1 introduces the need for protecting data in use and program execution. It discusses the use case of executing applications that process sensitive data in public cloud environments and the different attack vectors that must be taken into account.

Sofware-based secure execution. Section 2.2 covers software-based secure execution and its shortcomings. First, it discusses how advanced cryptography can be used to compute on data while protecting its confidentiality in an untrusted environment. It also covers why cryptography alone is insufficient to protect complex applications. Second, it looks at related work on software-based systems that protect applications against untrusted OSes. In particular, it discusses systems that rely on a trusted hypervisor to protect the integrity and confidentiality of applications and their data.

Trusted Computing introduces the use of hardware modules to establish trust in a system. Section 2.3 covers how Trusted Platform Modules (TPMs) provide integrity measurement, sealed storage, and remote attestation functionalities. It also looks at Intel TXT and AMD SVM late launch features that extend TPMs, and discusses related work that provides secure execution environments based on these technologies.

Trusted execution environments. Section 2.4 introduces *trusted execution environments* (TEEs). It first gives an overview of existing TEE technologies from academia and industry. This is followed by a detailed discussion of Intel SGX, the TEE technology used by the three systems presented in this thesis. Lastly, we present related work on vulnerabilities and attacks against current TEEs and discuss potential mitigations.

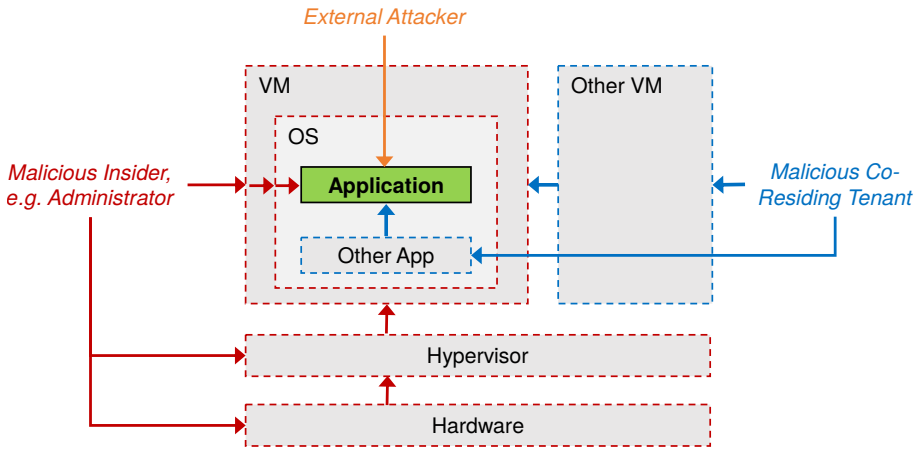


Fig. 2.1: Potential attack vectors of a cloud application

Threat model. Section 2.5 discusses the threat model of the three systems covered in the following chapters. It presents the assumed capabilities of the adversary and the types of attacks addressed. It also covers which threats are considered to be out of scope of this thesis.

2.1 The Need for Secure Execution

As discussed in Chapter 1, organisations that move application data off-premise, e.g. to public cloud infrastructure, can rely on encryption to protect *data at rest* and *in transit*. However, protecting *data in use*, e.g. when cloud applications process data, is more challenging. To be processed, data must typically be available as plaintext in non-persistent memory where it cannot only be accessed by the application itself but also by higher-privileged components including the OS and the hypervisor. A vulnerability in any of these components could be exploitable and allow an attacker to gain access to sensitive application data.

When applications are moved to a cloud infrastructure, additional attack vectors and potential adversaries must be taken into account, as illustrated by Fig. 2.1. These include:

- *external attackers* that exploit vulnerabilities in the application itself, the OS it runs on, or other parts of the cloud infrastructure which might allow them to gain access to a VM and sensitive data [144];

- *malicious co-residing tenants* that share resources with the target application such as the machine via a co-located VM, or even a single VM via a co-located process or container [278]; and
- *malicious insiders* that abuse privileged access rights [98]. These could be administrators of the application itself, e.g. a database administrator, of the VM, or of other cloud infrastructure. These might also include adversaries with physical access to cloud infrastructure.

This shows that there is a need for protecting the *confidentiality* of data in use in the same way data at rest and in transit are protected. However, protecting confidentiality alone might not be sufficient. An attacker might also try to compromise the *integrity* of application data, e.g. to remove, modify, or add data; or the integrity of application code, e.g. to hijack the control flow or to trick an application into exposing sensitive data. Therefore, in order to protect sensitive data, not only the data itself but also application code must be protected. The following sections present solutions for the secure execution of application in untrusted environments to allow for the protection of data in use.

2.2 Software-Based Secure Execution

In this section, we look at ways of providing secure execution using software. We first discuss how advanced cryptography can be used to operate on encrypted data, and then look at hypervisor-based solutions for protecting applications from an untrusted OS.

2.2.1 Computation on Encrypted Data

Cryptography plays a fundamental role in data protection. Data at rest and data in transit must be encrypted in order to protect their confidentiality while residing or travelling in untrusted environments. However, encrypting data limits how it can be processed. The naive client-side encryption of all data likely breaks most existing applications. *Functionally encryptable data* [271] is data that can be encrypted without affecting the functionality of an application that operates on it. This property can be leveraged to protect sensitive data through client-side modifications [157, 271] or by introducing a proxy [74] to transparently encrypt and decrypt sent and received data, respectively.

Naturally, such an approach can only protect the confidentiality of a subset of application data, requires complex key management, and is only applicable to some applications. In the following, we

instead discuss the capabilities of advanced encryption schemes to protect data in use as well as their practicality.

2.2.1.1 Advanced Encryption Schemes

Probabilistic encryption [130] provides strong security, with two equal plaintext values mapping to different ciphertexts with a high probability, and therefore provides security even in case of chosen plaintext attacks (*IND-CPA*). However, standard probabilistic encryption schemes make it difficult to operate on the encrypted data. In order to still allow computation over encrypted data, many different encryption schemes with distinct properties exist.

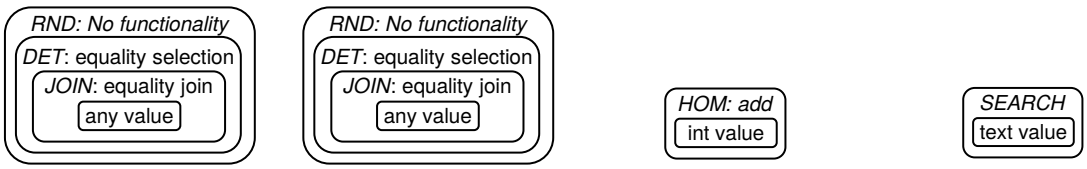
In contrast to probabilistic encryption, *deterministic encryption* provides weaker security guarantees as equal plaintext values are encrypted to the same ciphertext, but enables equality checks of two values without access to plaintext values. *Order-preserving encryption* (OPE) [5, 33] can be used when the order of plaintext values must be preserved in the corresponding ciphertext. It provides even weaker security guarantees than deterministic encryption but enables the evaluation of arbitrary range queries and other functions that rely on the order of data such as sorting or determining a minimum or maximum value. Searches on encrypted data are possible using *searchable encryption* schemes [35, 60, 133, 314].

Partially homomorphic encryption (PHE) allows for an operation to be applied to two ciphertext values resulting in a ciphertext of a plaintext value that is equal to the result of applying the operator to the plaintext values of the operands:

$$Enc(a) \bullet Enc(b) = Enc(c) \text{ if } a \bullet b = c \quad \text{with } Enc(x) \text{ denoting the ciphertext of } x$$

There are efficient PHE schemes for specific operations, such as modular multiplication [102, 279], modular addition [25, 249], or exclusive or [129].

One example of an application that makes use of these advanced encryption schemes is *CryptDB* [262]. CryptDB is a database system that provides full data confidentiality. It uses a number of different encryption schemes to allow for the execution of SQL queries over encrypted data. CryptDB leverages the fact that SQL queries are based upon a fixed set of primitive operators. By using both existing and new encryption schemes data can be encrypted so that unmodified DBMS can execute queries over



(a) Equality **(b)** Order **(c)** Addition **(d)** Search

Fig. 2.2: Layered encryption schemes for different data and computation types in CryptDB [262]

it without the need for decryption. These include probabilistic, deterministic, order-preserving, and searchable encryption. It also introduces two new encryption schemes for equality and range joins, respectively, and uses the PHE *Paillier cryptosystem* [249] to support the summation of encrypted values.

For each database column, CryptDB uses multiple layers of encryption, with the outermost encryption being the most secure, and the innermost encryption the least secure but most functional. For each data item, one or more so called *onions* exist, each consisting of one or more encryption layers, as can be seen in Fig. 2.2. Depending on the type of queries that are executed on the column, outer layers of encryption are removed by additional queries from a CryptDB proxy in order to reach an encryption layer that supports the required operations. The proxy is a trusted component in between clients and the database that holds decryption keys and that rewrites queries when necessary, e.g. to anonymise column and table names and to decrypt query results.

While CryptDB does support a wide variety of queries, it cannot support all computations. For example, it only supports 4 out of 22 analytics queries of the TPC-H benchmark [334] and has a significant overhead [342]. It also relies on a trusted proxy for key management and to rewrite database queries on-the-fly.

The advanced encryption schemes discussed so far allow specific computations to be performed over encrypted data while being efficient enough to be used in practical systems. *Fully homomorphic encryption* (FHE) [85, 125, 126] supports arbitrary operations on encrypted data but suffers from high computational overhead. While more recent FHE schemes [40, 41, 70, 72, 96, 108, 127] have improved FHE performance, they are still orders of magnitude slower than the equivalent computations on unencrypted data and therefore impractical for most use cases.

2.2.1.2 Garbled Circuits and Functional Encryption

Garbled circuits [166, 370] allow to encrypt arbitrary functions that operate on encrypted inputs, similar to FHE. In contrast to FHE, garbled circuits are based on symmetric encryption and provide better performance. However, each garbled circuit can only be evaluated once so that for each computation a new garbled circuit has to be created. Furthermore, the garbled circuit size is proportional to the size of the function leading to large communication overheads. Bugiel et al. [51] present a hybrid cloud model based on garbled circuits that tries to mitigate the former problem. It proposes to use a private cloud to pre-compute garbled circuits for known application queries. Once a client makes a request they are sent to the public cloud to be evaluated there. However, this approach still suffers from a large communication overhead between the private and the public cloud. Furthermore, it is not clear whether this model can be made more efficient than a model in which the same computations are performed unencrypted on the trusted private cloud in the first place. More recently, Goldwasser et al. [131] introduced *reusable garbled circuits* that can reduce the overhead due to generating new circuits for every computation.

Functional encryption (FE) [36, 283] allows authorised parties to compute a function on encrypted data and learning the result in cleartext without revealing the original data. In FE, data is encrypted with a public key PK which is associated with a master secret key. The master secret key is used to generate a secret key SK_f for a specific function f. Decrypting data $ENC_{PK}(x)$ with a secret key SK_f reveals the result of the function f applied to x.

$$Dec_{SK_f}(Enc_{PK}(x)) = f(x)$$

Identity-based encryption (IBE) [34, 295] is a subclass of FE in which a public key can be computed by any party based on a user's identity, defined e.g. as their user name or email address, and a set of general system parameters. These system parameters are generated together with a master key. A secret user key is generated using the master key. IBE can be seen as an alternative to standard *Public Key Infrastructure* (PKI). Rather than managing public keys, a central authority, the *Private Key Generator* generates a users' private key once they have authenticated themselves. In contrast to PKI, the PKG is trusted as it knows all private user keys. With IBE, data can be encrypted for a

particular user before the user authenticates themselves to the PKG for the first time. Once the user obtained their private key, it can be used to decrypt any ciphertext associated with their identity.

Attribute-based encryption (ABE) [26, 61, 140], another FE subclass, allows secret user keys and ciphertexts to be bound to a set of attributes. Only if all attributes of the ciphertext are matched by attributes of the user key, decryption is possible. ABE might be used for fine-grained access control to individual log entries during log auditing, or to subscription-based broadcasts [140].

Most FE schemes only support a subset of classes of functions such as boolean formulas. General FE schemes that can support any function have also been proposed [121, 131, 132, 138] in theory. However, no implementations exist as they rely on *program obfuscation* which has been shown to be impossible for general programs [20, 21, 120].

2.2.1.3 Discussion: Insufficiency of Cryptography

The previous sections have discussed encryption schemes that allow for the efficient computation on encrypted data for a variety of computation types. With fully homomorphic encryption and general functional encryption, schemes exist that support arbitrary operations on encrypted data without revealing the sensitive data they compute on but are currently impractical. However, even if they become practical in the future, they would be insufficient to fully protect confidentiality in arbitrary cloud applications. In their paper "*On the Impossibility of Cryptography Alone for Privacy-Preserving Cloud Computing*" [348], van Dijk and Juels analyse the feasibility of implementing different classes of cloud applications using cryptographic protocols. They show that, while FHE is able to implement applications with a single-client model, no cryptographic protocol can implement multi-client applications while preserving *functional privacy* of single users, i.e. the ability to control which computations may be performed on their private data. Similarly, practical functional encryption is limited to a subset of functions and to use cases in which the result of a computation on encrypted data is not sensitive itself.

So far, this section discussed the protection of data confidentiality but not the integrity of data and computations. Even with encrypted data, a malicious party in control of the untrusted environment might modify, replace, or remove data or return incorrect results. *Verifiable Computing* [124] uses cryptographic schemes to protect the integrity of outsourced computations and returned results. To do this, the untrusted worker has to provide a proof of correctness together with the result of a

computation. The client can then verify the correctness of the result based on the proof it receives. However, to become practical, the effort to set up the computation and the verification of the proof of the client side must be lower than the effort required for the actual computation which current verifiable computing approaches can not yet provide [253].

To summarise, cryptography alone is insufficient to enable sensitive data to be processed both efficiently and securely in untrusted environments. Instead, cryptography must be combined with or incorporated into other approaches such as software-based secure execution, Trusted Computing or trusted execution environments.

2.2.2 Protecting Against an Untrusted OS

In current cloud environments, applications rely on the trustworthiness of the underlying infrastructure, including the OS. However, commodity operating systems consist of millions of lines of code and are frequently reported to contain new security bugs that can be exploited to compromise the system and with that gain access to sensitive application data. In addition, curious administrators might have direct access to OSes on which applications are run. Ongoing research therefore explores how sensitive applications can be effectively protected from an untrusted operating system.

Some systems propose to reduce the size of the TCB by splitting the system into high-assurance and low-assurance partitions that execute sensitive and insensitive functionality of an application separately. Only the high-assurance part is included in the TCB and is smaller in size than the low-assurance part. *NGSCB* [55, 255] implements a small verified isolation kernel, similar to a virtual machine monitor (VMM), that allows multiple OS instances to run on top of it. This way a standard commodity OS can run alongside a small, trusted high-assurance operating system. Security-sensitive applications can be run inside the restricted high-assurance OS while untrusted or insensitive applications can run inside the low-assurance OS.

Proxos [325] follows a similar approach. It runs two virtual machines, an untrusted and a trusted instance, on top of a modified Xen hypervisor. The untrusted VM runs a commodity OS, while the trusted VM runs the private application, statically linked with the Proxos main library and a *private OS* library that implements functions to handle sensitive system calls. Application developers can configure their trust in the underlying operating system by partitioning the system call interface into insensitive and sensitive calls. While sensitive system calls are routed to the corresponding private OS

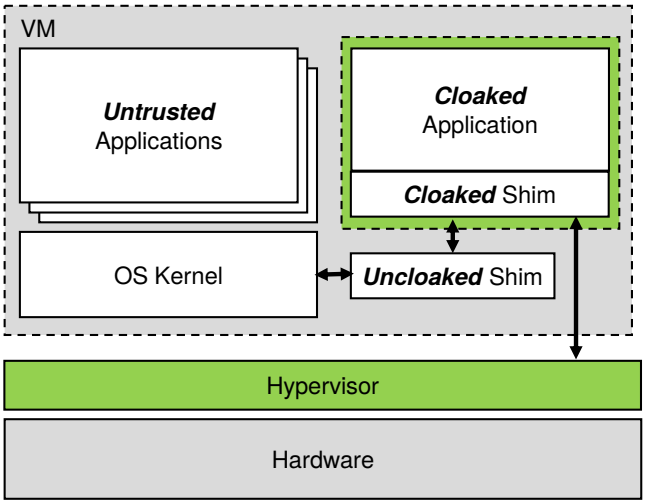


Fig. 2.3: Overshadow architecture [69]

functions, insensitive calls are routed to the commodity OS kernel via inter-VM remote procedure calls. Although these approaches successfully reduce the TCB size, they also incur big performance overheads on applications due to the need for context switches between VMs and expensive inter-VM communication.

Other research follows a more lightweight approach of isolating sensitive application by protecting application memory from unauthorised OS accesses. *Overshadow* [69] introduces a *multi-shadowing* technique that leverages memory virtualisation of a hypervisor to map guest VM physical addresses to different host physical addresses depending on the current execution context. This way, Overshadow can provide multiple views of memory for the untrusted OS and a protected application. Fig. 2.3 shows Overshadow's architecture. As can be seen in the figure, there are two isolation boundaries. One that isolates the guest VM from the VMM and one that isolates a *cloaked* application from the untrusted OS. Overshadow's VMM adds a *shim* layer into the address space of every protected application process which enables secure interactions with the OS by mediating events such as system calls in cooperation with the VMM. It also keeps track of the current execution context to distinguish between memory accesses that originate from the untrusted OS and those that are performed by the protected application. Overshadow introduces a concept called *memory cloaking* that ensures that for accesses of the protected application's address space by the OS the memory content is only returned in encrypted form. For this, it uses distinct shadow page tables for the protected application and the untrusted OS that map to different physical addresses on the host system for the address space

of the application. When the OS accesses an address in this memory region, Overshadow encrypts and integrity-protects the corresponding page before mapping the page into guest physical memory and making it accessible by the OS. This allows the OS to perform tasks such as paging without compromising confidentiality or integrity.

SP^3 [369] follows a similar approach and utilises a *page-frame replication* mechanism to provide the untrusted OS and other untrusted applications with an encrypted view of memory while providing a plaintext view to the protected application. *InkTag* [162], another hypervisor-based system, provides similar guarantees but extends the previous systems by introducing *paraverification*, which forces the untrusted OS to support the isolation of a protected application's address space. In addition, it adds *attribute-based access control* which allows protected applications to define access control policies for secure files, therefore enabling applications to store and access secure data over multiple sessions. In contrast to Overshadow and SP^3, it also ensures system crash recoverability.

Virtual Ghost [81] protects application memory by using compiler instrumentation on the host OS code in addition to hypervisor-based runtime checks. By using compiler instrumentation, Virtual Ghost ensures that the OS has no access to *ghost memory* which is similar to cloaked or shadowed memory of previous approaches, but cannot be read or modified by the OS at all. The address space of an application is divided into three partitions. The kernel space, the traditional user-space, and a protected ghost memory partition. The ghost memory partition is only accessible by the application and Virtual Ghost itself and is unmapped and mapped back into virtual memory on context switches between the OS and the protected application. In contrast to the previously discussed systems, applications can choose to only partially use ghost memory or not to use it at all.

These approaches protect applications against untrusted operating systems but rely on a trusted VMM to correctly enforce the protection and to not disclose sensitive application data to parties that have indirect or direct control over the VMM itself, such as administrators of a cloud service. One way to circumvent this, is to use a secure coprocessor such as the TPM to integrity measure the system up to the VMM and provide remote attestation functionalities that allow remote parties to verify the used software stack, including the VMM to bootstrap trust in the platform. We discuss this approach in the following section.

2.3 Trusted Computing

Trusted Computing [337] is a technology for enforcing a specific system behaviour through combined hardware and software mechanisms. It relies on a special cryptoprocessor, the *Trusted Platform Module* (TPM) [338], that provides key functionalities to support, enforce, and remotely attest to system behaviour.

2.3.1 The Trusted Platform Module (TPM)

The TPM is a cryptoprocessor available on most commodity systems, typically as a dedicated chip or integrated into a chipset. Virtual TPMs that are implemented as firmware [273] or as part of the hypervisor [63, 257] also exist. Every TPM has an *Endorsement Key* (EK) pair that is created, embedded, and signed by the TPM's manufacturer and whose private key never leaves the TPM. It is used to provide proof that a given TPM chip is genuine and can be used to generate a *Storage Root Key* (SRK). SRKs are not bound to the chip but to the platform owner and a new SRK can be generated when the owner of the platform changes. It can be used to protect application keys that are stored outside of the TPM. TPMs also provide volatile storage in which the values of *Platform Configuration Registers* (PCR) and Attestation Identity Keys (AIK), both used for system integrity measurement and reporting, are stored at runtime. TPMs also provide a random number generator, an SHA-1 hash engine and a module for RSA key generation. Version 2 TPMs [339] implement additional cryptographic algorithms and hash functions, and add features such as trusted counters but provide the same core functionalities as first generation TPMs.

Integrity measurement. One of the main use cases of TPMs is *integrity measurement*. TPMs can measure data or application code, generating *measured values* that fingerprint the measured entity, and *measurement digests*, SHA-1 hash values calculated over the measured values. Measured values are stored in a *Stored Measurement Log* (SML). When a new measurement is performed, the measurement value is appended to the SML and the digest is *extended*.

Measurement digests are stored on TPMs inside PCRs which are not directly accessible from outside the TPM and can only be extended but not otherwise modified. PCRs are stored in volatile memory and therefore are reset on a system reboot.

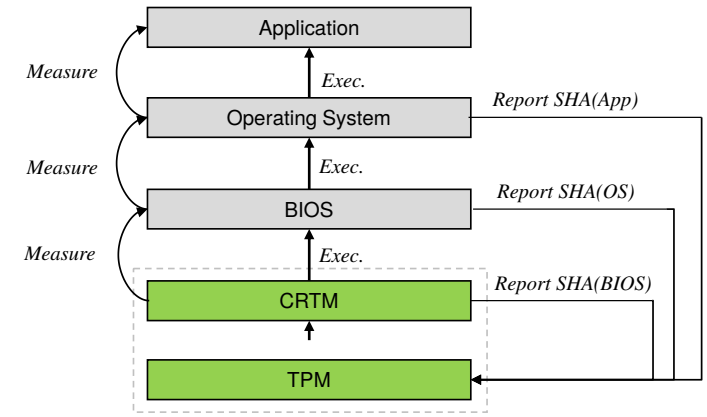


Fig. 2.4: Integrity measurements with TPMs

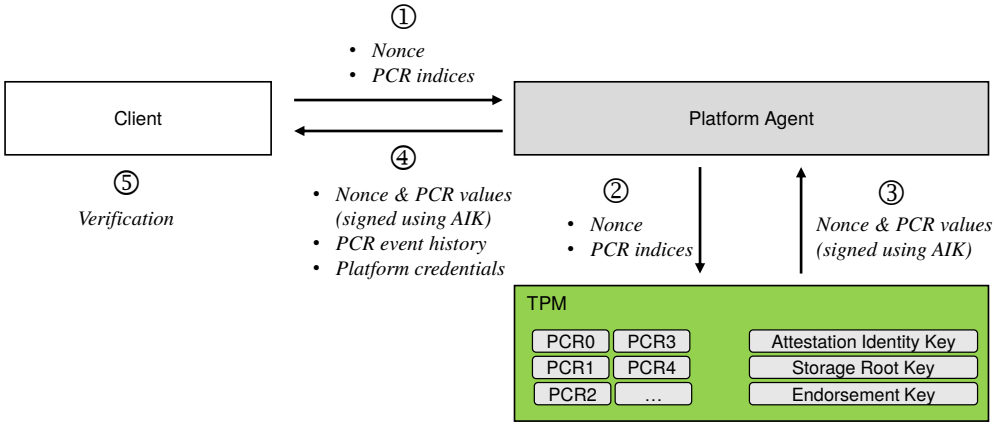


Fig. 2.5: Remote attestation with TPMs

Fig. 2.4 shows the process of measuring a system at runtime. The TPM constitutes a *root of trust for measurement* and together with a trusted BIOS boot code, also referred to as the *core root of trust measurement* (CRTM), can be used to build a *chain of trust* for the measurement of a system. The boot code measures the entity that comes next in the boot sequence so that a corresponding fingerprint is stored in a PCR. Once control has been passed on to the previously measured entity, it can measure its successor and extend the PCR. This way, integrity measurements can be performed for whole systems.

Remote attestation. With the integrity measurement capabilities of the TPM, remote parties can attest to the integrity of a TPM-based system. Fig. 2.5 shows how remote attestation is performed by a remote client. The remote party sends a request specifying the set of PCR values it wants to receive to a *Platform Agent* residing on the system that contains the TPM. The Platform Agent is a software

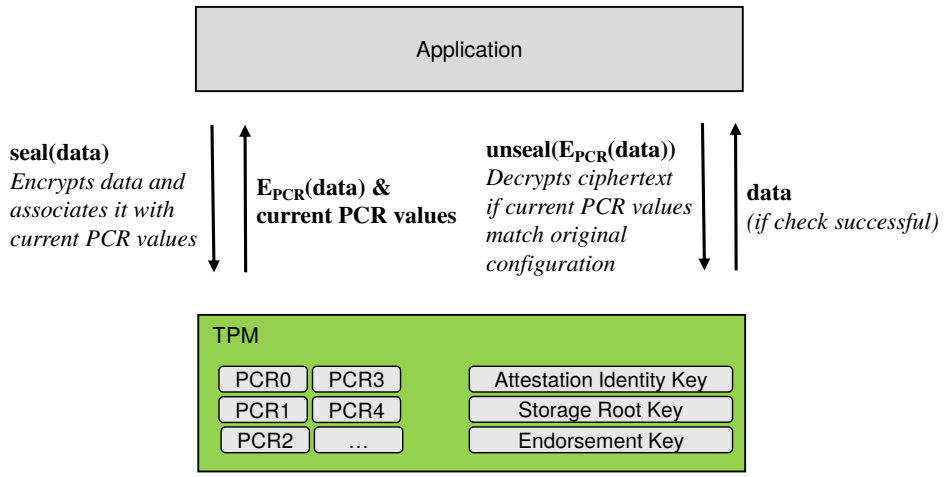


Fig. 2.6: Sealed storage with TPMs

component running on the target system and is not required to be trusted itself. It requests and receives the corresponding values from the TPM, signed by the TPM with an AIK, adds credentials vouching for the TPM itself, and sends this information back to the remote party. By comparing the received values to values generated from a known state, the remote party can verify the integrity of the TPM-based system.

Sealed storage. The *sealed storage* capabilities of TPMs, illustrated in Fig. 2.6, allow data to be bound to a specific platform configuration. For this, TPMs provide two instructions, a `seal` and an `unseal` instruction. The `seal` instruction takes PCR indices as well as the data that is to be sealed as inputs and produces a ciphertext that is associated with the PCR indices and the PCRs' values as output. The TPM only decrypts a ciphertext passed by the `unseal` instruction if the current PCR values match the PCR values at encryption time.

2.3.2 Late Launch

Integrity measurements provided by TPMs allow systems such as Terra [119] to implement *trusted virtual machine monitors*. By measuring the VMM and implementing a remote attestation protocol they allows remote parties to verify a system, including the VMM as well as individual virtual machines, to determine its trustworthiness.

However, due to the addition of more and more features, the sizes of commodity hypervisors have increased consistently over the years. Even when tenants are provided with an integrity measurement

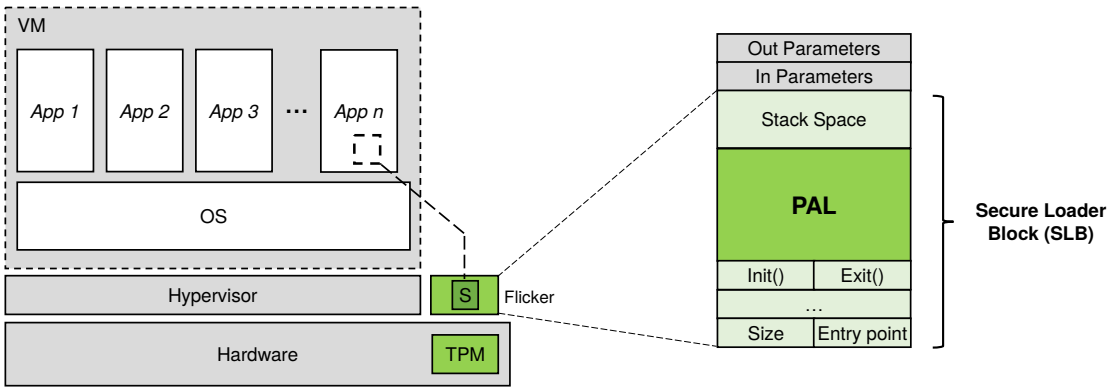


Fig. 2.7: Flicker architecture and simplified memory layout of the SLB, based on [214]

of the hardware and software stack up to the hypervisor, it is difficult to fully verify the trustworthiness of the platform.

Modern Intel and AMD CPUs implement *late launch* features to alleviate this. Intel's *Trusted Execution Technology* (TXT) [142] and AMD's *Secure Virtual Machine* (SVM) [11] processor extensions built on the capabilities of TPMs to guarantee the integrity of higher-level system components without requiring trust in underlying parts of the system. They introduce new instructions that put the CPU in a known good state, e.g. by resetting register values, reset a set of *dynamic PCRs* and start a *dynamic chain of trust*. The CPU itself performs the first measurement, the *Dynamic Root of Trust Measurement*, of a manufacturer-provided and -signed binary module. The CPU verifies the measurement and the module's signature. The module is responsible for two things: (i) it measures the next module, the *Measured Launch Environment* (MLE); and (ii) it verifies that the platform configuration adheres to a *Launch Control Policy* (LCP) that can be provided by the platform's owner. The LCP contains a whitelist of known-good MLEs and the CPU verifies the MLE's validity as part of the LCP check. If the check is successful, access to additional dynamic PCRs is enabled which allows the MLE to perform further measurements of following components, such as a trusted OS, before executing them. Measurements are stored in the TPM and the TPM's attestation capabilities can be used by a remote party to assess the integrity of the measured component. The component can then be executed in a trusted manner without requiring trust in lower-level system components such as a hypervisor.

2.3.3 Minimising the TCB with Trusted Computing

There are a number of systems that leverage the late launch feature to further reduce the size of the *trusted computing base* (TCB). *Flicker* [214] achieves a small TCB of just 250 lines, in addition to hardware and the application code itself by using the late launch feature. It implements a *Secure Loader Block* (SLB), a memory region whose address is passed as a parameter to the CPU instruction that activates the late launch feature. It measures the SLB, which can be up to 64 KB in size, and stores the measurement digest in one of the PCR registers of the TPM. Flicker uses this measurement to prove to a remote party that a particular initialisation code has been executed as part of a Flicker session. Finally, it jumps to a predefined entry point within the SLB and continues execution. Fig. 2.7 shows the memory layout of an SLB in Flicker. Applications can use Flicker to execute a *Piece of Application Logic* (PAL) securely. Applications use a Flicker module inside the untrusted host environment to initiate a Flicker session for the execution of a PAL. It writes both an uninitialised SLB containing its PAL as well as input parameters to corresponding entries in the `sysfs` file system. Flicker then starts a session, first initialising the SLB, then suspending the OS, and finally calling the late launch feature of the processor to bring the CPU into a secure state, measure the SLB and execute the PAL. When suspending the OS, Flicker saves the current kernel state in order to resume execution of the OS once PAL execution has finished. Before resuming, the exit code that is part of the SLB Core erases sensitive data from SLB memory. PALs can provide outputs by writing them to a special memory location near the end of the SLB. Flicker returns these output parameters to the application that started the Flicker session once normal OS execution is resumed.

Flicker provides a very small TCB which makes it easy for a remote party to verify its trustworthiness. On the other hand, it requires significant effort on the site of application developers that have to port secure parts of their applications to code that can run within the SLB that does not provide any software library support by default. Also, the frequent use of slow TPM operations has a significant impact on Flicker's performance.

A similar system that tries to mitigate Flicker's performance problems is TrustVisor [215]. TrustVisor is a special-purpose hypervisor with about 6,000 lines of code (LOC). It similarly enforces the isolation of PALs by using the late launch features of AMD and Intel processors, but uses so called software-based *μTPMs* for PAL integrity checking to circumvent the performance problems of using the hardware TPM. The root of trust for these μTPMs comes from TrustVisor itself that is integrity

measured by the hardware TPM. μTPM instances are only created during the PAL registration process when TrustVisor's memory protection is already active so that untrusted code cannot tamper with the μTPMs.

CloudVisor [374] does not target small PALs but instead tries to provide integrity and confidentiality protection for whole virtual machines in a multi-tenant cloud setting. It promotes the usage of *nested virtualisation* that is used to place a small security monitor (5,500 LOC) underneath a commodity VMM to protect hosted VMs. CloudVisor itself is bootstrapped using the late launch feature of Intel TXT and due to its relatively small size can be fully verified by remote parties. CloudVisor intercepts all control transfer events between the VMM and VMs hosted on the platform to transform VM events, if needed, to protect the VM before they are forwarded to the VMM. It also tracks the ownership of memory pages and page tables maintained by the VMM and intercepts updates to page tables. If the ownership of a physical page does not match the ownership of the page table, CloudVisor encrypts the content of the page to ensure that the memory of a protected VM can only be accessed in its unencrypted form by the VM itself. Similarly, all disk I/O operations from VMs to virtual disks trigger the encryption of written or the decryption of read data, respectively. To securely bootstrap VMs, tenants symmetrically encrypt VM images before sending it to a CloudVisor-based service once they have ensured that the platforms runs a known and trusted version of CloudVisor using remote attestation. In addition to the image, tenants send the used symmetric key encrypted with a public key belonging to the CloudVisor instance. CloudVisor then decrypts the image and ensures that the integrity of the VM is not compromised during the boot process.

While TPMs and the late launch CPU extensions are an important step in protecting the integrity of outsourced applications, they also have a number of shortcomings. Due to the low performance requirements prescribed by the TCG specification, TPM operations are relatively slow which prevents their usage in scenarios in which TPM operations are required frequently. Another problem is that TPMs cannot be used to provide runtime integrity guarantees. Remote attestation is typically performed before transferring control to the integrity-measured code but does not protect against any subsequent integrity violations. In addition, researchers have found vulnerabilities that allow an adversary to circumvent the protection of late launch features, e.g. via the System Management Mode (SMM) on Intel CPUs [97, 103, 365].

The use of secure coprocessors (SCPUs) [373] has been proposed to address the issue of missing runtime protection. SCPUs such as the IBM 4758 [311] or IBM 4765 [16] are tamper-resistant against physical attacks and can be used to outsource computation on sensitive data. They provide hardware implementations of cryptographic primitives for the efficient computation of cryptographic algorithms. Similar to the Endorsement Key of TPMs, SCPUs have a device-specific asymmetric key pair of which the private key never leaves the SCPU which can be used for secure communication with the SCPU. Secure databases such as *TrustedDB* [19] are an example of systems that make use of SCPUs to outsource computations on sensitive data and protect it from an otherwise untrusted system [37, 231, 309].

However, even modern secure coprocessors have limited processing power compared to general-purpose CPUs as well as limited amounts of secure memory and slower bus speeds. In practice, SCPUs are mostly deployed as part of Hardware Security Modules for the purpose of key generation, storage, and management, and to perform specific cryptographic operations.

2.4 Trusted Execution Environments

Trusted execution environments (TEEs) aim to overcome the shortcomings of the previously discussed technologies. They provide hardware-enforced isolated memory regions for the secure execution of software. As there is no generally agreed-upon definition of TEEs, we base our definition on that of Sabt et al. [282]. We define TEEs within the context of this thesis based on four core properties. TEEs

- are provided by a *main processor* rather than an additional hardware module such as a secure coprocessor;
- can guarantee the *authenticity* of TEE software, i.e. that code and static data initially loaded into the TEE is as expected by the software owner;
- protect the *integrity of TEE runtime state*. This includes state stored in TEE memory, register values, and CPU caches; and
- protect the *confidentiality of code and data at runtime*. After initialisation all TEE contents are no longer visible to untrusted components outside of the TEE. This includes data provisioned to the TEE from the outside, and data generated by the TEE software itself.

TEEs effectively combine features of other technologies, such as software authenticity and integrity protection provided by Trusted Computing technologies, and confidentiality protection of memory encryption technologies such as AMD SEV [185] and Intel MKTME [178].

In addition to the core properties above, TEEs often provide other capabilities. *Remote attestation* allows a remote party to attest that a TEE has been set up as expected. This includes *hardware attestation*, i.e. the platform provides a proof that the TEE is genuine, e.g. by signing the attestation message with a device-specific key. Other common features are *sealed storage* that allow a party to encrypt and integrity-protect data for storage with a TEE-specific key, *secure inter-TEE communication* for transmitting data between multiple TEE instances, and *trusted I/O* capabilities to establish secure communication channels between a TEE and I/O devices.

In the following, we will first discuss the history of trusted execution and give an overview of TEE technologies. We then present Intel SGX as the TEE technology the thesis is based on in detail. Finally, we discuss related work on TEE vulnerabilities, attacks that exploit such vulnerabilities, and possible mitigations.

2.4.1 TEE Technologies

While this thesis is based on Intel SGX as a TEE implementation, most insights and results apply to TEEs in general. This section looks at the history of TEEs and presents alternative TEE designs from both academia and industry.

The technologies presented here share some or all of the core properties defined above, but differ in other aspects such as the granularity at which they can protect software, their TCB and threat model, and additional features they provide. Table 2.1 shows an overview of the TEEs discussed in the following and their differences.

Besides a comparison of TEE features, the table also lists types of attacks each technology protects against. Depending on the technology's threat model this includes protection against attackers with physical access to a machine, integrity and freshness attacks, as well as different types of side-channel attacks: page-table based attacks, cache side-channel attacks, and attacks based on speculative execution. We discuss these types of attacks in more detail in Section 2.4.3.

Table 2.1: Comparison of different TEE implementations

	Arch./ Platform	Granularity	TCB (besides protected module)	Module location	# Compartments	Attestation	Protection against attacks					
							Physical	Integrity	Freshness	Page table	Cache	Spec.
Intel SGX [218]	x86/Intel	Software Module	CPU + architectural enclaves	PRM	Unlimited	Public/private key	✓	✓	✓	✗	✗	✗
XOM [205]	Custom	Process	CPU + hypervisor	DRAM	Unlimited	-	✓	✓	✓/✗	- *	✗	✗
AEGIS [321]	Custom	Software Module	CPU (+ security kernel)	DRAM	Unlimited	Public/private key/PUF	✓	✓	✓	✗	✗	✗
SP [201]	Custom	Software Module	CPU	DRAM	Unlimited	Device master key	✓	✓	✓	✗	✗	✗
ARM TrustZone [14]	ARM	-	CPU	On-chip SRAM	1	-	✓	✓	✓	✓	✗	✗
Bastion [58]	Custom	Software Module	CPU + hypervisor	DRAM	Unlimited	Public/private key	✓	✓	✓	✗	✗	✗
HyperWall [323]	Custom	Virtual Machine	CPU	DRAM	Unlimited	Public/private key	✗	✓	✓	✗	✗	✗
SecureME [71]	Custom	Process	CPU + hypervisor	DRAM	Unlimited	-	✓	✓	?	✗	✗	✗
IBM Secure-Blue++ [32]	Custom	Software Module	CPU	DRAM	Unlimited	Public/private key	✓	✓	✓	✗	✗	✗
Ascend [113], Phantom [212], GhostRider [208]	Custom	Algorithm	CPU	DRAM	1	Public/private key	✓	✗	✗	- *	✓	✓
Iso-X [106]	Custom	Software Module	CPU + RAM + system bus	DRAM	Unlimited	Public/private key	✗	✓	✓	✓	✗	✗
Sanctum [79]/ MI6 [38]	RISC-V	Software Module	CPU + security monitor + attest. enclave	DRAM	Unlimited	Public/private key	✗	✓	?	✓	✓	✓
OASIS [248]	Custom	Algorithm	CPU	Cache-as-RAM	1	PUF	✓	✓	✓	- *	✗	✗

* These systems do not support paging.

XOM [205] is an early proposal for an architecture that allows for the secure execution of processes. It introduces the concept of hardware-enforced *eXecute Only Memory* (XOM), memory that contains instructions that can only be executed but not modified at runtime. Processes run in isolated compartments provided by the CPU. Both external memory and the operating system are considered untrusted. Similar to Intel SGX CPUs, every XOM CPU has an associated public-private key pair. The private key is hold on-chip and the public key is used to encrypt application code that should be executed by the CPU. Every application is associated with a distinct symmetric session key, embedded in the application.

XOM relies on a special XOM hypervisor to implement a set of new instructions in software. Following an `enter_xom` instruction, the processor enters *XOM mode*. It uses its asymmetric private key to determine the active session key of the current compartment. This key is used to decrypt instructions when they are fetched from memory. Any data produced by the code is automatically tagged by the processor with a *XOM identifier* associated with the current compartment. All register values and cache lines have an associated tag identifying the compartment they belong to, or a special *null* tag if they do not belong to a protected compartment. Any access of tagged data from a different compartment results in an exception. The processor leaves XOM mode and clears the active session key on an explicit `exit_xom` instruction, or on a trap or interrupt. As external memory is considered untrusted, the session key is also used to encrypt and decrypt data on stores to and loads from external memory using designated `store_secure` and `load_secure` instructions. To prevent modifications to data in memory, `store_secure` computes a MAC and stores it with the data which is verified on a subsequent `load_secure`. XOM also provides instructions to re-tag data to support communication between processes and to sanitise data.

AEGIS [321] is another early proposal for a single-chip secure processor design similar to XOM. An AEGIS processor uses a public-private key pair or a physical unclonable function (PUF) [122] to identify itself. It implements on-chip data tagging and confidentiality and integrity protection for application state in external memory.

AEGIS partitions the virtual address space of a process into a protected and an unprotected region. The unprotected region allows for shared memory and communication with other secure processes as well as untrusted components. It is the responsibility of an application to partition its code and data accordingly. Besides `enter_aegis` and `exit_aegis` instructions that enable and disable

the processor's *trusted execution* (TE) mode, AEGIS also provides a `sign_msg` instruction. This can be used to produce a signature over a message and a hash of the TE region. The signature is generated using the processor's secret key. This allows TE code to send data, e.g. results of a computation over untrusted communication channels, and the receiving party to verify that the message is genuine.

In contrast to XOM's MAC-based integrity verification, AEGIS uses Merkle trees [221] to detect integrity violations of data loaded from external memory in order to protect against memory replay attacks that XOM cannot protect against.

AEGIS also proposes an optional additional *security kernel*, a trusted operating system component that is higher privileged than the rest of the OS. AEGIS computes a hash over the security kernel when its loaded at startup to allow users to verify its integrity which subsequently is protected by AEGIS itself. This allows AEGIS to implement functionality such as the signing of messages from TE code in software. However, this assumes trust in the security kernel and therefore increases the TCB size.

SP. Lee et al. [201] propose the *secret-protected* (SP) architecture. In contrast to XOM, AEGIS, and Intel SGX, it does not require factory-provided device secrets. Instead it aims to provide support for binding sensitive data to a user and across devices rather than to a specific device. SP supports *Concealed Execution Environments* (CEEs) similar to the TEEs provided by the previously discussed architectures. CEEs provide integrity protection for application code and protect confidentiality and integrity of application state in registers, caches, and external memory.

User secrets can only be accessed via a *Trusted Software Module* (TSM). The TSM runs in isolation from the rest of the system, including the OS. All user secrets are protected via a user-specific hierarchical key chain that contains encrypted keys associated with a user. The root of the chain, the *User Master Key*, can be used to decrypt and access specific user keys. Only TSMs have access to the master key at runtime. To provide a device with a master key, SP assumes a secure I/O channel through which a user provides the key or a passphrase from which the key can be determined. The TSM exposes a set of functions that rely on access to a user key to applications, e.g. to sign a message or to encrypt or decrypt data, without revealing the user secret itself. By using TSMs and secure I/O to provide users' master keys, SP avoids relying on a device-specific secret and instead allow secrets to be bound to users and to be shared across devices.

Arm TrustZone. *TrustZone* [14] is a security extension for Arm CPUs. TrustZone divides execution into two *worlds*, the *Normal World*, and the *Secure World*. For the execution of code in either world, the CPU explicitly switches to a corresponding mode. Physical memory is divided correspondingly. In addition, TrustZone implementations provide on-chip static RAM. In contrast to regular DRAM, this memory is not accessible from outside the chip itself and is therefore protected against physical attacks. It should be noted, that the size of this on-chip memory on currently available TrustZone implementations is limited and ranges from 72 KB to 512 KB [377].

Code run in the Secure World is strictly higher privileged than code in the Normal World. It has full control over code and data residing in the Normal World. Data within the Secure World is inaccessible from the Normal World. While it is possible to jump to arbitrary locations in the Normal World from the Secure World, transitions in the opposite direction are only possible via a special *Secure Monitor Call* instruction. Critical security-sensitive data can therefore be kept and processed in the Secure World by a small OS kernel. A special processor *Non-Secure* (NS) bit indicates the current security mode. The NS bit is transmitted with every transaction over the system bus. This allows both external memory and other peripherals to provide distinct address spaces for both worlds.

As a result of this design, code running in the Secure World is inherently trusted to be non-malicious as it has full control over the regular OS and other code and data running and residing in the Normal World. A common use case is that of a security monitor which can detect anomalies in the Normal World. Another difference to Intel SGX is that TrustZone has a single secure compartment. While it is possible to run multiple trusted applications alongside each other in the Secure World, they all run with high privileges and increase the TCB size. However, there have been many proposals to implement enclave-like TEEs on top of TrustZone [45, 48, 110, 147, 165, 182, 375].

For example, *vTZ* [165] adds support for virtualised *guest TEEs*. Together with the hypervisor, VMs still run in the Normal World but can make use of such guest TEEs to execute sensitive code in isolation. These guest TEEs run in a special VM in the Normal World that is protected by vTZ via a small monitor running in the Secure World. For this, vTZ interposes memory mappings and world switches attempted by VMs to emulate them by transferring control to the corresponding guest TEE and vice versa. *Komodo* [110] proposes the use of TrustZone to implement SGX-like TEEs in software to decouple SGX's core mechanisms such as memory encryption, address space isolation, and attestation from the hardware. Komodo implements these features via a formally verified security

module that runs in TrustZone's Secure World. One stated advantage of this approach is that new features can be added more easily as they can be implemented without hardware changes or microcode updates.

Bastion [58] extends previous proposals by introducing a hardware-protected trusted hypervisor. The launch and in-memory state of the hypervisor is protected by the CPU against both software and hardware attacks. The hypervisor is used to securely launch and manage secure isolated TSMs. Bastion introduces new hypercall-like instructions that allow TSMs to directly communicate with the trusted hypervisor instead of relying on an untrusted OS for hypercalls, e.g. for access to secure storage. Through the hypervisor, TSMs can be locally and remotely attested to. TSMs are confined to a single address space and a single thread. However, Bastion supports *trust domains* spanning multiple TSMs and has support for inter-TSM communication that allows passing messages via protected registers and memory. Both hypervisor and TSM memory is encrypted, integrity-, and freshness-protected via memory encryption and hashing engines in between the CPU's last-level cache and the main memory controller.

HyperWall [323] is an architecture that specifically aims to protect VMs against an untrusted hypervisor. It introduces virtualisation extensions that can protect guest memory from unauthorised access from a hypervisor as well as DMA. It does not protect against hardware attacks such as bus snooping and cold boot attacks. HyperWall allows hypervisors to manage and set up VMs but protect VMs at runtime according to security policies specified by VM owners. For this, page tables are left to be managed by the hypervisor, but HyperWall introduces separate *Confidentiality and Integrity Protection* (CIP) tables stored in processor-reserved memory. CIP tables contain entries for guest memory pages that indicate whether they should be accessible by the hypervisor or via DMA, e.g. for access to I/O buffers, both, or none. HyperWall extends the MMU to enforce these CIP constraints. As some of the previous designs, HyperWall processors have a unique secret key burnt in at manufacturing time used to support remote attestation of VMs and their CIP tables. An encryption key and a hash key are generated on every boot that are used to encrypt and protect register state of VMs on interrupts so that it can be stored by the hypervisor and later restored when a VM is resumed.

SecureME [71] is another secure hardware-software co-design consisting of a secure processor and a thin trusted hypervisor that provides isolated execution environments for processes. Similar to

Overshadow and SP^3 (see Section 2.2.2), SecureME implements memory cloaking. Application pages are encrypted and integrity-protected in memory and transparently decrypted by the CPU when accessed from an application context. An OS can access application pages, e.g. to implement paging, but only in encrypted form. SecureME implements this efficiently by temporarily disabling the processor's cryptographic engine responsible for encryption and integrity protection. Integrity protection is implemented using a Merkle tree over main memory contents of which the root node is stored on-chip. Applications and the OS are assigned a unique *hardware identifier* (HID) by the hypervisor. The hypervisor stores and restores HIDs on context switches and SecureME uses the HID to determine the current execution context to implement access control and to selectively disable or enable features of the cryptographic engine. SecureME requires instrumentation of OS code and introduces two new instructions for the initialisation and copying of application pages. It keeps track of copied pages in order to deal with subsequent integrity verification failures when accessing a copied page and trigger re-encryption and re-computation of related Merkle tree nodes. In addition, SecureME supports shared pages across HIDs to enable protected applications to communicate with each other and with the untrusted OS via system calls. It leaves it up the application to protect messages and system call arguments.

SecureBlue++ [32] is a secure processor architecture by IBM. Similar to Intel SGX, it allows for the isolated execution of parts of an application and assumes applications to consist of an untrusted and a trusted component. The untrusted component is responsible for setting up the protected region consisting of encrypted code and data. It also handles communication with the trusted component via buffers in untrusted memory. The CPU enters a special secure mode for the execution of trusted code via a dedicated enter instruction invoked by the untrusted code. The trusted region is encrypted via an application-specific encryption key, and requires an integrity tree to be provided by the application owner. Therefore, SecureBlue++ guarantees the confidentiality of all trusted code and data. The enter instruction takes the encryption key and the root hash of the integrity tree as arguments which allows the CPU to decrypt and integrity-check trusted code and data when they are loaded into the last-level cache. The arguments are in turn encrypted using a device-specific public key. Hardware interrupts cause SecureBlue++ to save confidential state such as register contents and the current encryption key and integrity tree root hash. A unique secure region identifier is stored in a new register and can be used by the OS to instruct the CPU to restore the secure execution context and re-enter secure mode after an interrupt has been handled. Besides protecting secure regions from external software

and hardware attacks, SecureBlue++ also provides protection against some application vulnerabilities such as stack and buffer overflows. For this, it divides secure regions into code and data regions and enforces them to be read only and non-executable, respectively.

Ascend, Phantom, and GhostRider. *Ascend* [113] is a secure processor architecture with built-in support for oblivious RAM to address the leakage of sensitive data via memory access patterns. Ascend ensures that the execution of any instruction results in the same amount of data-independent work, and hides timing patterns by accessing external memory at fixed time intervals and with a fixed I/O rate. Ascend implements Path ORAM [317] to hide spatial memory access patterns. The processor has a device-specific secret public/private key pair. In order to execute an algorithm securely on an Ascend-based platform, a remote user first sends user-provided inputs encrypted with the public key and, if required, the algorithm itself. It also specifies the number of cycles that should be spent on the computation. Ascend then initialises ORAM to store the algorithm, the user-provided input, and optionally server-side input. An untrusted server application can then instruct Ascend to compute the algorithm. To avoid leakage, Ascend itself does not reveal the completion of a computation but instead uses up the requested number of cycles. The encrypted program state is then sent back to the user. The user decrypts the program state to verify its completion. In contrast to previous architectures, Ascend provides stronger protection by making program execution oblivious. However, secure execution is restricted to applications that don't require any I/O.

Phantom [212] is another architecture that uses Path ORAM for oblivious program execution following the same approach and design goals as Ascend. Similar to Ascend, an application and user inputs are provided as an encrypted ORAM image. A loader sets up the ORAM, executes the program obliviously, and returns the resulting ORAM image after a fixed number of cycles.

GhostRider [208] extends Phantom to incorporate non-oblivious encrypted RAM (ERAM) for non-sensitive application data to speed up program execution. GhostRider provides a dedicated compiler and uses program analysis to make use of ERAM efficiently without leaking additional information, e.g. when an array is always accessed sequentially and in full.

Iso-X [106] is a secure processor architecture comparable to Intel SGX and SecureBlue++ that provides TEEs for software modules. For this, Iso-X extends the ISA with instructions to add pages to a protected compartment, to initialise, enter, and leave compartments, for attestation, and to resume

execution after interrupts. Iso-X allows compartment pages to reside anywhere in DRAM and does not require a dedicated fixed-size protected memory region such as the EPC of Intel SGX. Only a compartment table with metadata on all active compartments and a membership vector that indicates whether a physical page belongs to a compartment are stored in processor-reserved memory. Every compartment has an additional *Compartment Page Table* (CPT) containing virtual-to-physical page mappings for compartment pages. CPTs can only be accessed by hardware and are used to protect against malicious page table changes by system software. A *Compartment Metadata Page* within each compartment is used to store compartment-specific metadata such as a public key and a certificate generated by Iso-X during initialisation usable for secure communication and remote attestation. It is also used to store confidential CPU state during interrupt handling. In contrast to SGX, Iso-X does not consider physical attacks in its threat model and includes both DRAM and system buses in its TCB. As a consequence, Iso-X does not encrypt and integrity-protect main memory, and does not support OS paging.

Sanctum [79] provides side-channel resistant enclaves. It follows a similar TEE approach as Intel SGX, Iso-X and SecureBlue++ for the secure execution of software modules. Unlike SGX, Sanctum excludes physical attacks from its threat model, and relies on a trusted security monitor implemented in software. It runs at the highest privilege level and implements similar tasks to SGX's microcode, e.g. to provide primitives for untrusted software to interact with enclaves. Sanctum requires processor-reserved memory for both the security monitor and for enclave memory. To prevent page-table based side channels, it uses per-enclave page tables and page fault handlers that are isolated from system software. In contrast to SGX, enclaves can install custom interrupt handlers. Hardware exceptions that occur during the execution of enclave code are caught by the security monitor and forwarded to the enclave. To remove cache side channels, Sanctum clears per-core caches on enclave transitions and partitions the shared last-level cache using cache colouring [187] to isolate enclave cache lines. *Keystone* [188] is an open-source hardware project built on top of Sanctum that aims to allow for customisable TEEs, i.e. TEEs optimised for specific application needs.

MI6 [38] further extends the design of Sanctum to protect against a broader class of side-channel attacks while supporting speculative out-of-order execution. Like Sanctum, MI6 incorporates a security monitor, cache set partitioning, and per-enclave page tables. In contrast to Sanctum, MI6 also protects against cache coherence and DRAM controller bandwidth side channels. Furthermore, it

introduces a new `purge` instruction to flush all microarchitectural state used by the security monitor to implement secure context switches. This includes out-of-order buffers and queues as well as branch predictor structures to prevent side channels based on speculative execution.

OASIS [248] is an architecture design that provides trusted execution with minimal architectural changes. It restricts protected memory to CPU caches based on previous work [350] that provides isolated execution environments on commodity CPUs by using Cache-as-RAM (CAR) [211] and therefore can exclude external main memory and system buses from its TCB. OASIS uses a PUF to derive a device-specific secret key used as a root of trust during attestation and the deployment of protected modules. OASIS extends the ISA with instructions to set up platform keys derived from the PUF, and to set up and launch the secure execution of a software module. OASIS assumes that applications including both code and data fit into the CPU cache. Before execution, the CAR is cleared, all hardware interrupts are disabled, and the plaintext program code is loaded. Sensitive user inputs are provided in encrypted form and can be bound to a hash of the program code. OASIS verifies that the hash matches the current program and decrypt inputs via an `unbind` instruction. Correspondingly, a `bind` instruction can be used to encrypt output data and binding it to the hash of the program it originates from for verification by a user upon receiving the result.

The large number and the variety of TEE technologies presented in this section reflects the interest in TEEs both in academia and industry. The ongoing research on new CPU architectures and TEE extensions shows that existing commodity TEE technologies such as Intel SGX have limitations that should be addressed in future versions and newly developed TEE technologies. Newer TEE approaches such as Sanctum [79], MI6 [38], and OASIS [248] particularly focus on the prevention of side channels.

While acknowledging its limitations, in this thesis we choose Intel SGX as TEE technology for two main reasons: (i) it is available as part of commodity CPUs; and (ii) it is one of the most complete TEE technologies, offering capabilities such as remote attestation, sealing, paging, and support for multiple compartments, that are essential for complex TEE applications and the systems discussed in this thesis. The following section will present Intel SGX in detail.

2.4.2 Intel SGX

This section introduces Intel Software Guard Extensions (SGX). It discusses the internals of SGX, how SGX is used, and its implications, e.g. on the performance of isolated applications. As described in Chapter 1, Intel SGX is an instruction set extension first introduced in the Skylake generation of Intel x86 CPUs. It allows for the creation of isolated regions in memory, so called *enclaves* and for the secure execution of code within these enclaves. We focus on Intel SGX as it is available on commodity CPUs and its concept of protected but unprivileged enclaves matches the scenario of mutually distrusting cloud and service providers.

2.4.2.1 Isolation

The key concept of Intel SGX, as with other TEEs, is the isolation of code and data in memory and at runtime. With Intel SGX, the CPU can be in one of two modes, in *enclave mode* or in *normal mode*. When the CPU enters an enclave and with that transitions into enclave mode, it exclusively executes instructions from that enclave. Fetching instructions from outside the enclave will cause a general protection fault. Similarly, an enclave can only be entered from the outside via specific instructions at pre-defined entry points. Any interrupt causes an enclave exit and a switch to normal mode.

CPU state such as register values will be saved on exit and reset to a synthetic state in order to not reveal any enclave secrets.[1] Similarly, the CPU's L1 data cache is flushed on every exit.[2]

Enclave memory resides in an isolated, contiguous region of physical memory, the *Enclave Page Cache* (EPC). The EPC is shared by multiple enclaves, with every enclave corresponding to a contiguous region of virtual memory within its host process. The mapping from virtual addresses to physical addresses within the EPC is managed via the regular page tables under the control of the host. However, the EPC itself cannot directly be accessed by system software. SGX tracks EPC pages and their virtual-to-physical address mappings via per-page entries in the *Enclave Page Cache Map* (EPCM). The EPCM is used by the CPU for access control checks for EPC pages as part of address translation. Enclave code can only access EPC memory associated with the same enclave. In addition, enclave code can directly read and write unprotected memory within the virtual address

[1] On enclave exits initiated by enclave code, clearing register values is the responsibility of the enclave code.

[2] This behaviour has been introduced as a microcode update in response to a number of cache-based microarchitectural side-channel attacks such as Foreshadow [174].

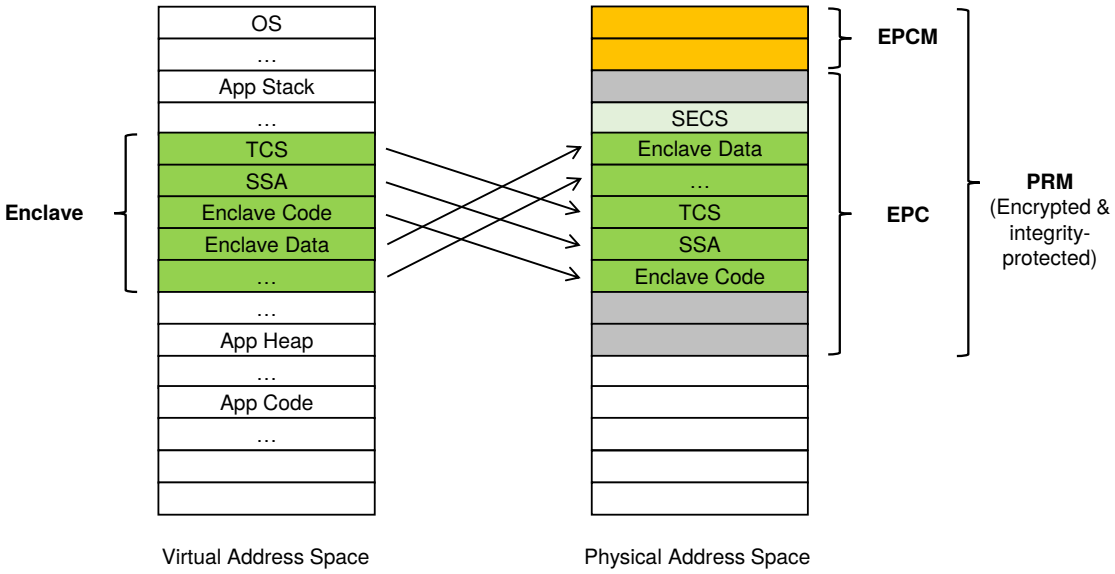
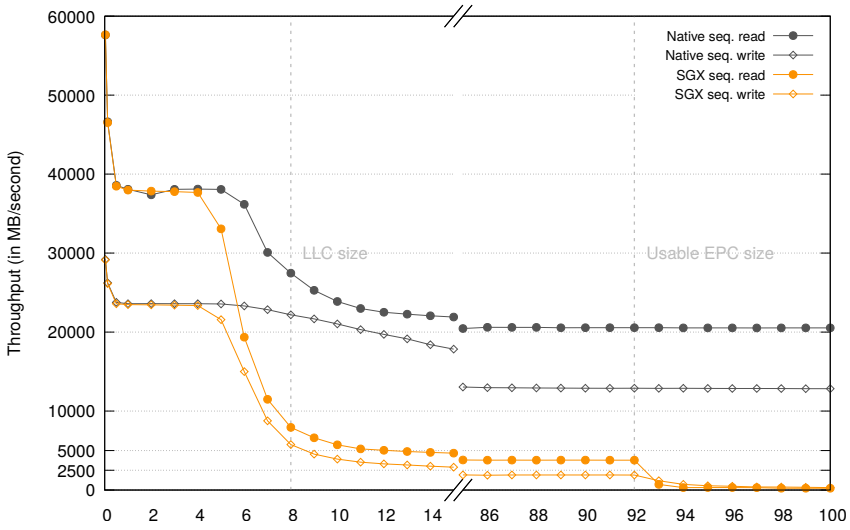


Fig. 2.8: Intel SGX EPC and EPCM in processor-reserved memory (PRM)

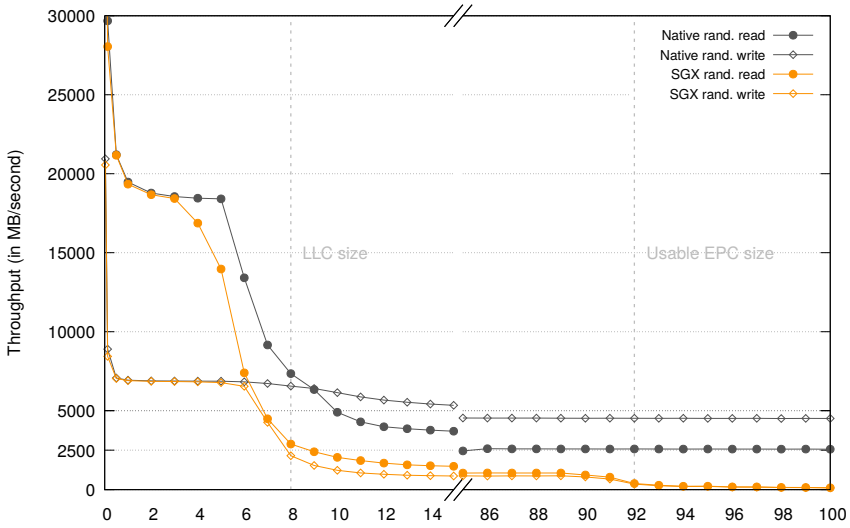
space of its host process. In contrast, untrusted code attempting to reading data from an enclave's address range results in abort semantics, i.e. all-one bytes are returned.

Both the EPCM and the EPC are assumed to be protected. However, the Intel SGX specification does not prescribe a specific EPC implementation. In practice, current SGX implementations use part of the system's DRAM as EPC memory as illustrated in Fig. 2.8. For this, a subset of DRAM is configured to serve as *processor-reserved memory* (PRM) which then holds EPC pages as well as the EPCM. As the DRAM in itself is not trusted in SGX's threat model, current CPUs incorporate a *Memory Encryption Engine* (MEE) as an extension of the memory controller (MC) which guarantees confidentiality, integrity, and freshness of EPC pages in DRAM.

All EPC memory in DRAM is encrypted and is only decrypted by the MEE when it is loaded into a CPU cache line. To protect integrity and freshness, the MEE computes *Message Authentication Codes* (MACs) over chunks of PRM memory of the size of a cache line (64 B), and generates nonces on every write. MAC tags are stored together with the data, while the corresponding nonces are stored in an *integrity tree*. All nodes of this tree except for the root node are themselves stored in the PRM and have corresponding MAC tags. Only the root is stored on-die and therefore can be trusted [149]. Any time a cache line is loaded from DRAM into the CPU cache, the MEE uses the MAC tag and the integrity tree to verify its integrity and its freshness, i.e. that the read data represents the most recently



(a) Sequential reads/writes



(b) Random reads/writes

Fig. 2.9: Effect of MEE and EPC paging on memory read and write throughput, measured with *bandwidth64* [312]

written version. Any violation prevents the MC from performing any further address translations, effectively halting the system.

The MEE has a significant impact on SGX performance. Fig. 2.9 shows the results of a single-core microbenchmark in which an in-memory buffer of varying sizes is repeatedly read from or written

Table 2.2: SGX version 1 (leaf) instructions [176]

Instruction	Leaf	Description
ENCLS	*Enclave creation*	
	ECREATE	Create an enclave.
	EADD	Add an EPC page to an enclave.
	EEXTEND	Extend EPC page measurement.
	EINIT	Initialize an enclave.
	Paging	
	ELDB	Load an EPC page in blocked state.
	ELDU	Load an EPC page in unblocked state.
	EPA	Add an EPC page to create a version array.
	EREMOVE	Remove an EPC page from an enclave.
	EBLOCK	Block an EPC page.
	ETRACK	Activate EBLOCK checks.
	EWB	Write back/invalidate an EPC page.
	Debugging	
	EDBGRD	Read data from a debug enclave by debugger.
	EDBGWR	Write data into a debug enclave by debugger.
ENCLU	*Enclave lifecycle*	
	EENTER	Enter an enclave
	EEXIT	Exit an enclave.
	ERESUME	Re-enter an enclave.
	Attestation & Sealing	
	EGETKEY	Create a cryptographic key
	EREPORT	Create a cryptographic report.

to.[3] For buffer sizes that fully fit into the CPU caches, SGX shows no overhead. As the buffer size approaches the size of the last-level cache (LLC) read and write throughput naturally drops but decreases significantly more for SGX compared to native reads and writes as data that is evicted from the LLC must be encrypted and integrity-protected by the MEE. Read throughput drops to around 4000 MB/s (4x overhead compared to native) for sequential reads and around 1000 MB/s (2.5x) for random reads. Write throughput drops to around 2000 Mb/s (6x) and around 870 Mb/s (5x) for sequential and random writes, respectively. As these results show, the design of current SGX implementations that use an MEE to be able to use parts of DRAM for the EPC can have a big impact on SGX performance. In practice, for most applications overhead due to the MEE is offset by the CPU caching and other bottlenecks such as I/O wait times.

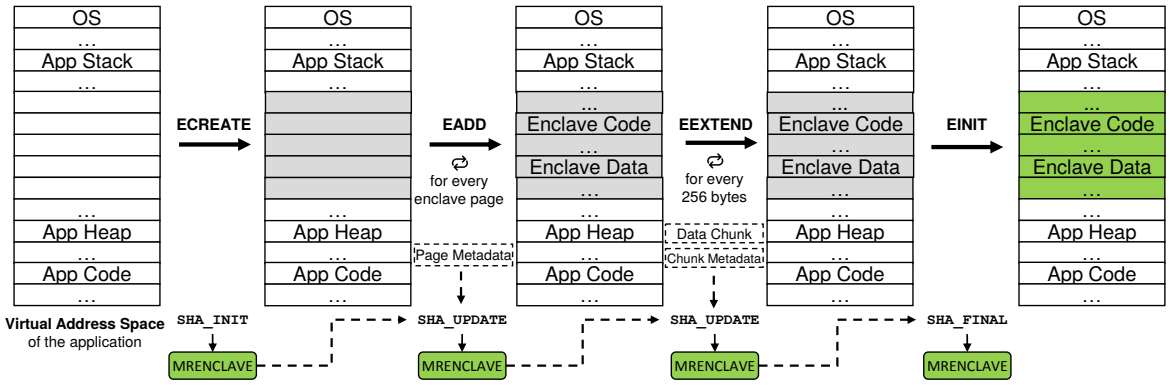


Fig. 2.10: Intel SGX enclave build process

2.4.2.2 Intel SGX Instructions

Table 2.2 shows an overview of all available SGX version 1 operations. All SGX operations are available via two instructions, a privileged supervisor instruction `ENCLS`, and an unprivileged user instruction `ENCLU`. The specific operation is specified as a *leaf* via the `RAX` register. For brevity, we will treat the combination of instruction and leaf as if they were individual instructions, e.g. as if there were explicit `ECREATE` or `EENTER` instructions. Privileged instructions are exposed to applications via an SGX device driver.

All work described in this thesis has been done in the context of SGX version 1. SGX version 2, available in more recent CPUs from the Gemini Lake generation onwards, extends the SGX instruction set. Most importantly, SGX version 2 adds instructions for dynamic memory management [219] allowing enclaves to dynamically grow and shrink at runtime, and extended virtualisation support. Most findings and conclusions in this thesis should be applicable to both SGX version 1 and version 2. SGX 2 will be explicitly referred to when relevant.

2.4.2.3 Enclave Setup

There are four main steps involved in setting up an enclave, illustrated in Fig. 2.10. Only after this build process is completed, the enclave can be entered to execute enclave code.

1. First, using the `ECREATE` instruction, an enclave is created. `ECREATE` copies an *SGX Enclave Control Structure* (SECS) page from a specified source address into the EPC. The SECS page

[3]The experiments were conducted using *bandwidth64* [312] on an Intel Xeon E3-1280 v5 CPU with an L3 cache size of 8 MB and an EPC size of 128 MB.

contains fields to specify the base address of the enclave within the virtual address space of its host process, the size of the enclave, and a number of other attributes. The SECS page stores enclave metadata but is not part of the enclave itself and cannot be accessed, neither by enclave code nor by untrusted code, after enclave creation.

Throughout the enclave build process, an enclave digest `MRENCLAVE` is created that identifies the code, data, and metadata of an enclave. This SHA256 hash is initialised as part of the `ECREATE` instruction. This measurement is later used during enclave launch, attestation, and sealing.

2. Using the `EADD` instruction, source pages that can contain enclave code or data are copied into the EPC. Every page added via `EADD` is associated with a specific enclave by specifying the corresponding SECS page. The virtual address of the page as well as its security attributes, such as read, write, and execute permissions, are stored in an EPCM entry. In addition, `EADD` will compute a hash over this metadata and extend the `MRENCLAVE` digest.

3. `EEXTEND` is used to measure the actual page contents and extend `MRENCLAVE`. `EEXTEND` measures 256 bytes of enclave page contents and the corresponding offset from the enclave base. In order to measure a whole 4KB enclave page added via `EADD`, `EEXTEND` is called 8 times for each 256-byte chunk within the page.

4. Lastly, `EINIT` finalises the `MRENCLAVE` measurement and the enclave build process. `EINIT` requires three arguments: (i) a reference to the SECS page of the targeted enclave; (ii) an enclave signature structure `SIGSTRUCT`; and (iii) an *EINIT token* structure `EINITTOKEN`.

   `SIGSTRUCT` is provided by the developer of an enclave. It contains a number of fields describing the enclave, most importantly the expected enclave measurement `ENCLAVEHASH`, an `ATTRIBUTES` bitset specifying enclave attributes that must be set, as well as a product id `ISVPRODID` and a security version number `ISVSVN`. `ISVPRODID` and `ISVSVN` are used to identify enclaves belonging to the same product and are used during enclave sealing (see Section 2.4.2.6). `SIGSTRUCT` also contains a signature over these fields, computed by the enclave developer, as well as the corresponding public key.

   `EINITTOKEN` is a token that proves that an enclave is permitted to launch. Only with a valid init token, `EINIT` will succeed and finalise the enclave build process. An init token can be obtained from a *launch enclave*. With SGX version 1, the launch enclave is one of a number of Intel-

provided system enclaves with special privileges. Based on a provided SIGSTRUCT structure and the launch enclave's *launch control policy*, the launch enclave decides whether or not an enclave should be permitted to launch, e.g. based on a whitelist of enclave measurements and signing authorities, and to generate a corresponding init token. With SGX 2, Intel introduced a *Flexible Launch Control* feature that allows platform providers to implement their own launch enclaves with custom policies.

When executing EINIT, the CPU performs a number of steps. It (i) verifies that SIGSTRUCT has been signed with the enclosed public key; (ii) check that the finalised MRENCLAVE hash matches the ENCLAVEHASH of SIGSTRUCT; (iii) checks that the enclave's attributes match the ones requested as part of SIGSTRUCT; (iv) verifies the validity of the init token and that it corresponds to the targeted enclave and the provided SIGSTRUCT; and (v) updates the enclave's SECS page to store the enclave's ISVPRODID, ISVSVN, MRENCLAVE, and MRSIGNER values, and to mark it as initialised. The MRSIGNER field is a digest of the signing authority's public key provided as part of SIGSTRUCT.

Once the enclave has been initialised via EINIT, it can be entered. At this point, enclave memory can no longer be modified from the outside. With SGX 1, only writeable pages can be modified by enclave code but changes to page metadata such as permissions and the removal or addition of pages are not possible. SGX 2 adds new instructions that allow these modifications. Any access to addresses within the enclave from untrusted code results in abort page semantics, i.e. read accesses will return all-one bytes, and write accesses are ignored. For debug purposes, enclaves can be marked as debug enclaves via the SECS ATTRIBUTES field. This allows enclave memory to be read and written to via the corresponding debug instructions EDBGRD and EDBGWR.

2.4.2.4 Paging

At the time of writing, CPUs with SGX have a maximum EPC size of 128 MB of which a subset is used for internal metadata, resulting in a usable amount of memory of approximately 92 MB. As with regular pages, Intel SGX leaves it up to the host system to manage page table entries for SGX pages. This is required in order to allow for EPC paging. EPC pages can be paged out to regular main memory, allowing one or more enclaves to be larger than the EPC itself.

As the EPC cannot directly be accessed by untrusted code, SGX provides a set of paging instructions. For an eviction, `EBLOCK` and `ETRACK` are used to block an EPC page and prevent new virtual to physical address mappings, and to remove old mappings for the page, respectively. With the `EWB` instruction, an enclave page can then be securely evicted. For this, the CPU encrypts the page and writes it to unprotected memory. It also assigns the evicted page a version number and computes a Message Authentication Code (MAC) over it which are both stored along with the page itself. SGX keeps track of versions of evicted pages in a special *Version Array* (VA) page in EPC memory.

An evicted page can be loaded back into EPC memory, e.g. after the CPU signals a page fault, using the `ELDU` or `ELDB` instruction. They load the evicted page back into EPC memory and leave it in an unblocked or blocked state, respectively. They also verify that the page contents match the MAC to detect whether a page has been tampered with, and that the page corresponds to the latest evicted version to prevent replay attacks. Any verification failure causes the CPU to halt.

As a result, EPC paging incurs a significant performance overhead. A page fault for an enclave page will result in (i) a transition from the CPU's enclave mode to normal mode; (ii) the eviction of a page currently residing in the EPC as described above; (iii) the loading of the accessed, currently evicted page back into EPC memory as described above; and (iv) transitioning back to enclave mode. The impact of this can be seen in Fig. 2.9 (see Section 2.4.2.1) which shows the memory throughput for repeated sequential and random memory reads and writes to a buffer in EPC memory. As soon as the buffer size exceeds the available 92 MB of EPC memory on currently available CPUs, throughput drops from ~3800 MB/s to ~225 MB/s and from ~1900 MB/s to ~220 MB/s for sequential reads and writes, respectively. For random accesses, throughput drops from ~1050 MB/s to ~120 MB/s and from ~870 MB/s to ~110 MB/s for reads and writes, respectively.

Futures versions of Intel SGX will have larger EPC sizes [303] and might improve EPC paging performance. In this thesis, we will evaluate performance of the presented systems both with SGX hardware and the current EPC limitations, and with simulations to approximate performance for larger EPC sizes in the future.

2.4.2.5 Enclave Lifecycle

Once an enclave has been created and initialised, an enclave can be entered with `EENTER` and enclave code can be executed. On executing `EENTER`, the CPU transitions into enclave mode and transfers

control to code within the enclave. The exact point of entry is specified by referencing a *Thread Control Structure* (TCS) page that is stored within the enclave itself. TCS pages are special enclave pages that once added to an enclave cannot be accessed.[4] During enclave creation, one or more `TCS` pages can be added to an enclave. Every TCS page can be used by at most one thread at once. Therefore, the number of `TCS` pages determines the maximum number of threads that can enter an enclave simultaneously.

A TCS page contains a number of fields. Most importantly, the `OENTRY` field of the `TCS` structure specifies an offset from the enclave's base to which control is transferred to on `EENTER`. Other fields include `OFSBASGX` and `OGSBASGX` to specify offsets into the enclave used to set the FS and GS segment base addresses, and the `OSSA` and `NSSA` fields. `OSSA` specifies the offset of the *Stack Save Area* (SSA) stack, and `NSSA` specifies the total number of available slots for SSA frames on this stack. The SSA is used by the CPU to save an enclave thread's state on interrupts before clearing the CPU state and transitioning back to normal mode. By requiring to specify a TCS page on `EENTER`, SGX enforces that an enclave can only be entered via predefined entry points.

There are two ways to exit an enclave. First, enclave code can explicitly exit an enclave by executing the `EEXIT` instruction. This is typically used when (i) the current thread or the enclave as a whole should terminate; or (ii) control should be (temporarily) transferred to code running outside of the enclave. `EEXIT` allows to specify a target address in untrusted memory to which control will be transferred to after exiting the enclave. With `EEXIT` it is the responsibility of enclave code to clean up CPU state. Register contents are not changed by SGX and can be used to pass arguments to the outside code. Consequently, registers containing sensitive data must be cleared by enclave code before executing `EEXIT`.

Secondly, hardware and software interrupts can cause so called *asynchronous exits* (AEX). Examples are page faults, segmentation faults, or illegal instruction exceptions caused by the execution of enclave instructions, or interrupts caused by peripherals or the host OS such as I/O or timer interrupts. Before the CPU exits the enclave, it saves potentially sensitive CPU state in the SSA before leaving the enclave. On an AEX, control is first transferred to the host OS to handle an interrupt. Subsequently, control is transferred to an *AEX handler*. The AEX handler is untrusted code and is specified for every thread when entering an enclave as an argument to `EENTER`.

[4] apart from the `FLAGS` field, accessible in debug enclaves

In order to resume execution inside the enclave, the AEX handler can use ERESUME. ERESUME restores the CPU state from the SSA and then continues execution from the point at which the AEX happened[5].

SGX allows for reentrancy, i.e. re-entering the enclave via EENTER after an asynchronous exit by allowing for multiple SSA frames to exist for one TCS. On an AEX, the CPU increments the CSSA field in a TCS page that keeps track of the number of SSA frames currently in use. If another AEX happens after re-entering an enclave via EENTER during a previous AEX, the CPU automatically stores CPU state in the next available SSA frame. Reentrancy is useful in a number of cases. For example, it allows for in-enclave signal handling where an enclave instruction causes an exception and an AEX. The enclave can be re-entered on the same TCS via EENTER in order to react to the signal and then leave the enclave again, before resuming regular enclave execution via ERESUME.

Entering and exiting an enclave and as a consequence AEX are expensive operations and frequent transitions can impact the performance of enclave applications. Leaving and re-entering an enclaves takes around 10,000 CPU cycles [244, 362].

2.4.2.6 Sealing

SGX allows for the *sealing* of enclave data. Sealing refers to the encryption and integrity protection of data that leaves an enclave temporarily to be read again at a later time by the same or another compatible enclave. For example, enclave code might want to persist sensitive data on an untrusted host between restarts. The EGETKEY instruction can be used to generate a cryptographic *sealing key*. It supports two types of sealing keys: (i) sealing keys that are bound to the MRENCLAVE enclave identity for data that should only be accessible by enclaves with exactly the same MRENCLAVE hash; and (ii) sealing keys that are bound to the signing authority identified by MRSIGNER, the product id ISVPRODID, and the security version ISVSVN. These keys can be regenerated by enclaves with the same MRSIGNER and ISVPRODID values, and a ISVSVN version equal or higher to the one requested. This allows updated enclave software to access data that has been sealed by an earlier version of the same software. It is up to the enclave software to use the key to encrypt and integrity-protect data.

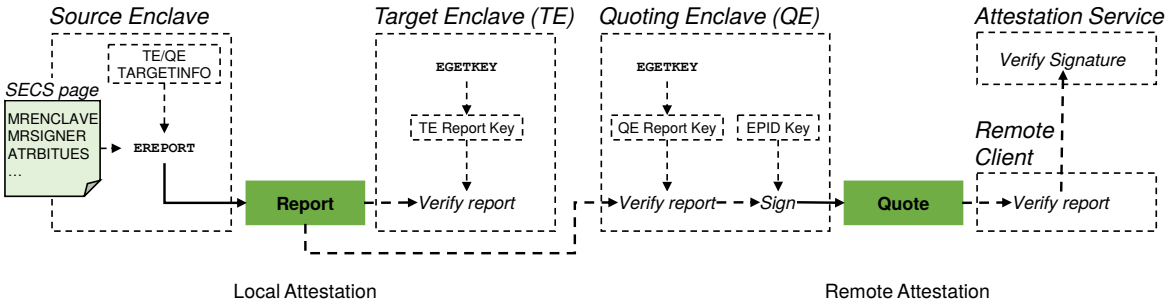


Fig. 2.11: Local and remote attestation with Intel SGX

2.4.2.7 Attestation

Similar to TPMs, Intel SGX allows for the attestation of an enclave in order to prove that an enclave has been initialised and configured correctly, and that it is a genuine hardware-protected Intel SGX enclave. Fig. 2.11 illustrates both local and remote attestation.

Local attestation

Local attestation allows one enclave (target enclave) to verify another enclave (source enclave) on the same platform. This involves a number of steps.

1. The source enclave obtains a `TARGETINFO` structure that contains information about the target enclave the report should be generated for. Most importantly it contains the enclave measurement `MRENCLAVE` of the target enclave and the target's enclaves attributes.

2. The source enclave uses the `EREPORT` instruction to generate a cryptographic report for the target enclave by providing the `TARGETINFO` structure as an argument. It can decide to include up to 64 byte of additional data as part of the report by providing a `REPORTDATA` argument.

   `EREPORT` will then (i) generate a report key for the target enclave; (ii) assemble a report of the source enclave based on its SECS page including the enclave's measurement, signing authority and attributes, and the `REPORTDATA`; and (iii) compute a cryptographic hash over the report and add it to the report structure.

3. The generated report is sent to the target enclave.

[5]The `RIP` register value, that holds the instruction pointer, itself is stored in the SSA. As the SSA is accessible from enclave code it is possible to modify it during an AEX and therefore modify where execution continues.

4. The target enclave uses `EGETKEY` to obtain its own report key, the same key the CPU used as part of `EREPORT` executed in the source enclave.

5. The target enclave recomputes a hash over the report with its report key and verifies that it matches the hash attached to the report. It then checks whether the report matches its expectations, e.g. whether the source enclave has a specific `MRENCLAVE` or `MRSIGNER` value.

Using this process, the target enclave can establish trust in the source enclave. A typical scenario is that of mutual attestation in which this process is repeated with the target and source enclave switching roles in order to establish trust in each other. The report data can be used to exchange public keys allowing the enclaves to subsequently establish a secure communication channel.

Remote attestation

Remote attestation is used when verifying an enclave on a different host. It involves a *quoting enclave*, another privileged system enclave. To create a report for a remote party, an enclave can obtain a *quote* from the quoting enclave. For this, it generates a report using the local attestation process described above with the quoting enclave as a target. The quoting enclave then verifies the report and replaces the cryptographic hash with a new signature. This signature is created using a special device-specific *Intel Enhanced Privacy ID* (EPID) key.

EPID [49] is a group signature scheme that allows a remote party to verify that the key owner is a member of a larger EPID group without being able to identify the specific group member. Every SGX-enabled CPU is assigned to an EPID group at manufacturing time. The EPID scheme allows Intel SGX CPUs to prove that they are genuine and to sign data without revealing the platform identity.[6]

The quote can then be sent to a remote party. In order to verify that the quote has been generated on a genuine Intel SGX platform, it can be sent to an *Intel Attestation Service* (IAS) [12] for verification. The IAS is an Intel-provided service for the verification of enclave quotes based on their EPID signatures.[7] The IAS responds with an attestation report that not only verifies the validity of the quote but also contains information on the state of the platform itself. Every report contains a CPU security version number field `CPUSVN` identifying the version of the CPU's microcode. If the microcode is

[6] Dall et al. [84] have shown that the EPID scheme is not fully anonymous and cannot uphold its unlinkability guarantee.

[7] With SGX 2 and Flexible Launch Control, Intel has also introduced *Third Party Attestation*, allowing third parties to provide their own attestation infrastructure [287].

out-of-date, the attestation report indicates this. Similarly, the attestation report contains information on platform configurations that are considered insecure. For example, following the Foreshadow [345] attack that relies on hyperthreading, the IAS reports that a platform configuration change is needed when an enclave runs on a platform with hyperthreading enabled. As a lot of attacks (discussed in Section 2.4.3) rely on running attack software on a co-located virtual core, verifying the IAS report before provisioning enclave secrets is vital for application security.

Depending on the IAS response and the enclave report itself, a remote party decides whether an enclave is sufficiently protected and trustworthy. As described previously, the report's `REPORTDATA` field can be used to exchange data specific to the enclave such as a public key for establishing a secure communication channel with the enclave.

2.4.2.8 Illegal Instructions

Besides the previously discussed restrictions such as not being able to fetch instructions from untrusted memory while in enclave mode and running in ring 3, there are a number of other restrictions for enclave code [176]. Most importantly, instructions that would result in a change of the privilege level such as `SYSCALL` and `INT` are not permitted within enclave code. Typically these are used to invoke operating system functionality. In case an enclave application requires host OS functionality, it therefore has to call out of the enclave instead. Any ring 3 I/O instructions such as `IN` or `OUT` are also considered illegal.

Other instructions that are commonly used by applications that are illegal within SGX enclaves and therefore require special considerations are `CPUID`, `RDTSC`, and `RDTSCP`. The latter two are no longer illegal in SGX 2. For application code that relies on these instructions, either the code has to be modified to replace these, e.g. by *ocalls* to perform them outside of the enclave, or the instructions have to be emulated by catching the resulting illegal instruction exception. Section 3 discusses this approach in more detail.

2.4.3 TEE Attacks and Mitigations

Intel SGX and other TEE technologies aim to protect the confidentiality and integrity of software and data even in the presence of a powerful adversary with physical access to the machine and full control over system software. However, there have been a number of attacks that have shown that

currently available TEE implementations such as Intel SGX are not free from vulnerabilities and do not protect against all attacks. In this section, we discuss these vulnerabilities, attacks, and potential mitigations.

2.4.3.1 Interface-Based Attacks

The host interface is the most obvious attack vector of an enclave as it is the only point of direct interaction. It is defined as the set of all supported *ecalls* and *ocalls* plus any other means of communication between untrusted code and an enclave, e.g. via shared buffers in untrusted memory. Sensitive data entering and leaving the enclave via the host interface can be observed, manipulated, or imitated by an adversary in control of the untrusted system. Therefore, any data transmitted between an enclave and another trusted party should be encrypted and integrity-protected. However, there are other types of interface-based attacks that are still possible.

Iago attacks are attacks by a malicious kernel in order to manipulate a trusted application's state or trick it into exposing sensitive application data. Checkoway and Shacham [62] introduce Iago attacks in the context of systems that allow to run application securely on an untrusted operating system, such as *Overshadow* (see Section 2.2.2). These systems prevent the untrusted OS from directly accessing application memory. However, system calls are still relayed to and performed by the untrusted OS. As Intel SGX enclaves are unprivileged, it is common for enclave applications to have a similar interface to access host OS functionality. By carefully choosing return values passed back to the enclave, an adversary can influence an enclave's behaviour. For example, Checkoway and Shacham show that carefully choosing a sequence of return values for the `getpid` system call allows a connection replay attack against an Apache server with `mod_ssl`. Whenever enclave code relies on return values or other arguments originating from the untrusted system, there is an attached risk that an incorrect value is returned to trigger unintended enclave behaviour. Note that in this case the enclave cannot rely on encryption and generic integrity protection as for input data coming from another trusted source. Instead, it has to employ specific integrity checks depending on the semantics of the individual call. We discuss these types of attacks in more detail in Chapter 3.

Freshness attacks. Another type of attack is one that violates the *freshness* of enclave data. Intel SGX itself does not provide any secure persistent storage. As discussed in Section 2.4.2.6, Intel SGX provides sealing capabilities to encrypt data and bind it to an enclave or a signing authority

before passing it to the untrusted host for temporary storage. When it is returned later on, only an enclave with the same signature or signing authority can decrypt and access the data. This prevents and adversary from tampering with the sealed data. However, in so called *rollback attacks* an attacker replaces genuine sealed data with an older version. As the data has been originally sealed by the same enclave, it is considered valid. Enclave code must take additional precautions to prevent such attacks, e.g. by associating each sealed data item with a nonce and keeping track of the nonce of the most recently stored version in-memory. That way, a rollback attack can be detected by checking the nonce when data is requested from the host.

Guaranteeing the freshness of data across enclaves or multiple runs of the same enclave is more difficult as metadata such as nonces cannot be kept securely in-memory across runs. A commonly proposed solution is to use a *trusted monotonic counter*, i.e. a counter that monotonically increases, and that can be trusted to not be tampered with by an adversary. One option is to use the trusted counter provided by the Intel *Management Engine* (ME) [177], an always-on subsystem within Intel processors intended for system management purposes. However, research has shown that access to the counter is slow and that the non-volatile memory backing the counter can wear out within days [318]. Other proposals use trusted clients [42] or a set of distributed enclaves [24, 213] to keep track of freshness metadata.

Note that freshness must also be protected for data from trusted sources. An adversary might launch *replay attacks* in which an adversary forwards a genuine message from a trusted source to the enclave more than once. In order to detect replay attacks, messages might include nonces and session keys are used for message encryption that are only valid during the lifetime of an enclave instance. This prevents replay attacks across multiple runs of the same enclave or *forking attacks* in which an enclave is instantiated multiple times and messages are selectively passed on to one of them or is replayed across enclaves.

We address the problem of freshness protection in the context of high-frequency database transactions in Chapter 5.

Passive attacks. The previously described attacks actively attempt to violate the integrity of enclave data and computations or to trick the enclave into exposing confidential data. However, an adversary might also be able to learn confidential information from observing enclave *ocalls* and their arguments. *SGX-Bleed* [202] is a subtle attack that exploits the use of insufficiently initialised data structures by

enclave code that are subsequently copied to untrusted memory as *ocall* arguments. More specifically, memory that previously contained confidential data might be reused to store a non-confidential data structure. Even if enclave code writes to all fields of the structure, some confidential memory might remain untouched if the structure contains buffers that are only partially overwritten or padding bytes that are left untouched. When the structure is subsequently used as an *ocall* argument, it is copied to untrusted memory and exposes the confidential data stored in uninitialised memory.

In other cases, information is implicitly leaked as part of an *ocall*. For example, the Intel SGX SDK provides a *Protected File System Library* (PFSL) [173] that can be used by enclave code to store files on the host file system and that automatically encrypt and integrity-protect file contents. However, metadata such as file names, file sizes, and directory structures, are directly exposed to the host OS and can be used to infer sensitive information about in-enclave operations. In-enclave file system implementations such as *SGX-FS* [52] address this problem by implementing file operations inside the enclave and only exposing a small host interface for reading and writing data stored on disk. However, the order and the size of these reads and writes still leak access patterns. Other in-enclave file system implementations such as *Obliviate* [6] specifically implement file system operations in a data oblivious manner to also hide access patterns. *BesFS* [302] is a library similar to PFSL that uses the host file system but that is provably secure against Iago attacks.

Other proposals extend the trusted computing base by including trusted devices. For example, *Pesos* [193] makes use of *kinetic disks* [293] that can be accessed via Ethernet and have built-in support for on-disk encryption and integrity protection as well as network encryption via TLS. Therefore, an enclave application can establish a direct communication channel with the disk without relying on the host for disk I/O. Similar approaches have been proposed for other device types such as bluetooth devices [258], video and audio outputs [161], and GPUs [353]. Weiser et al. [359] propose an architecture for generic trusted I/O that relies on a trusted hypervisor. The trust in the hypervisor is established via TPM integrity measurements. The hypervisor couples devices with corresponding device driver enclaves and ensures that all interaction with the device is done via the trusted device driver. An enclave application can attest to the correct deployment of the driver enclave and establish a secure communication channel to gain direct access to a device. Instead of a trusted hypervisor, *Aurora* [204] proposes to use a higher-privileged trusted SMM [172] module to provide enclaves with secure communication channels to I/O devices.

These approaches still require interaction with the host but reduce it to the forwarding of encrypted and integrity-protected messages between the enclave and a trusted device and with that reduce the host interface's complexity.

2.4.3.2 Side-Channel Attacks

While Intel has previously stated that SGX excludes side-channel attacks from its threat model, they are a serious threat for the confidentiality of enclave data. Side-channel attacks abuse side effects of CPU operations such as execution times or the state of CPU caches to infer confidential information. Traditionally side-channel attacks assume an unprivileged execution context. They assume limited control over the system and therefore have to take into account noise from the execution of other applications, the OS, and interrupts. In contrast, Intel SGX assumes that an attacker has full control of the system which makes side-channel attacks more viable as signal noise can be reduced significantly. Furthermore, having control over page table mappings, context switches, and interrupts enables new TEE-specific attacks [153, 368].

Page-table based attacks. As discussed in Section 2.4.2.4 Intel SGX leaves it up to the host system to manage virtual to physical address page table mappings. The CPU verifies during address translation that mappings for enclave pages are not tampered with. However, it allows EPC pages to be paged out to DRAM and for page faults to occur and to be handled by the untrusted host system. This creates a new page-table based side channel. By selectively paging out enclave pages, an adversary can observe memory access patterns of enclave code based on resulting page faults. Xu et al. demonstrate a practical attack that extras full text documents from enclave memory [368]. Other research [344, 355] shows that enclave page accesses can also be observed without inducing page faults by observing the access and dirty bits of page table entries.

Cache attacks. Cache-timing attacks [246, 256, 372] exploit that CPU caches are shared across isolation boundaries, i.e. across processes or virtual machines. Cache timing attacks exploit the fact that modern processors use *set-associative* caches. Each cache consists of a number of *cache sets* which hold a number of *cache lines*. Cache lines are fixed-size memory blocks, typically 64 bytes, loaded from main memory or a higher-level cache on a cache miss. Memory blocks are associated with a specific cache set based on their address, i.e. a given memory block is always cached in the same cache set. This associativity allows an attacker to e.g. forcibly evict a cache line from a target

process by filling the corresponding cache set via loads of its own memory associated with the same set.

Due to timing differences between CPU cache hits and cache misses an adversary can then infer whether some target memory has recently been accessed, e.g. by measuring the execution time of a specific victim process operation before and after evicting cache lines (*Evict+Time* [246]). Other examples of cache timing attacks include *Prime+Probe* [246, 256] in which the cache is primed with attacker's data and the time is measured to re-access the same data after victim code is run, and *Flush+Reload* [372] in which a specific cache line is evicted via the `clflush` instruction and an attacker checks whether the line is reloaded into the cache as part of a victim operation. These attacks have been shown to enable an attacker to extract confidential information such as AES keys, even across virtual machines in the cloud [180].

Some of these attacks rely on shared memory between attacker and victim, e.g. due the use of shared libraries across processes or memory deduplication across VMs. Intel SGX prevents these as EPC memory cannot be shared across enclave boundaries. However, other cache timing attacks have been demonstrated against SGX enclaves. Götzfried et al. [139] and Brasser et al. [43] demonstrate Prime+Probe attacks to extract AES and RSA keys, respectively, from enclaves by scheduling an attacker process on the same physical core via hyperthreading and using kernel-privilege CPU performance counters for precise timing. Hähnel et al. [153] and Moghimi et al. [227] demonstrate how having full control over the host system, including the scheduling of enclave threads, can be leveraged to create high-resolution side channels using Prime+Probe to extract enclave secrets in a single enclave run. *MemJam* [228] is an attack that exploits small timing differences in the execution of a set of instructions based on data dependencies between them. It has a higher resolution than previous attacks, allowing it to infer memory accesses on an intra-cache line level. The authors show that MemJam can extract cryptographic keys even with current constant-time crypto implementations and that it is equally applicable to SGX enclaves. Not a cache attack, but similar in nature, *Nemesis* [346] is another attack that demonstrates how microarchitectural instruction timings can be leaked by exploiting differences in interrupt latencies.

Speculative execution attacks. Speculative execution describes the CPU behaviour of executing instructions speculatively based on a prediction that they will be needed in the near future. Examples are out-of-order execution and branch prediction. The main goal of speculative execution is to improve

performance by executing multiple instructions in parallel. In the case of misprediction changes to the CPU state are rolled back. However, recently the *Meltdown* [207] and *Spectre* [191] attacks have shown that speculatively executed instructions have microarchitectural side effects that are not rolled back by the CPU. This includes memory loads into cache lines that can be inferred via regular cache timing channels. Following these attacks, many more speculative execution based attacks have been discovered [53, 54, 107, 181, 192, 291, 316, 345, 349].

Some of these attacks such as Spectre [65] have also been shown to be possible against Intel SGX enclaves. The *Foreshadow* attack has been demonstrated to extract cryptographic keys from the Intel-provided launch enclave and quoting enclave [345]. *Branch Shadowing* [203] and *BranchScope* [107] target the branch target buffer and directional branch predictor, respectively, to infer taken branches on branch instructions within enclaves. *ZombieLoad* [291] exploits side effects of faulting instructions on the *fill buffer*, a buffer between CPU caches, and demonstrate the attack's effectiveness on SGX enclave code.

As these attacks show, side-channel attacks are a serious threat for TEEs. With full control over the host OS, adversaries gain fine-grained control over enclave execution which further facilitates such attacks. *SGX-Step* [343] is an attack framework that allows an attacker to single-step over individual enclave instructions using high-frequency timing interrupts. *MicroScope* [308] introduces *microarchitectural replay attacks* that allow an attacker to replay individual enclave instructions and thus denoise side-channel attacks by inducing page faults. *STACCO* [367] is a differential analysis framework that allows an untrusted OS to infer SSL/TLS vulnerabilities in enclave code by collecting and analysing execution traces of enclave code based on observing page and cache line accesses.

Protecting against TEE side-channel attacks. Many proposals have been made to protect against TEE side-channel attacks such as those described in the previous sections. To protect against page-table based attacks, *SGX-LAPD* [118] uses large enclaves pages to reduce the resolution of the page-table based side channel. To ensure that the OS in fact uses large pages, SGX-LAPD deliberately induces page faults periodically and verifies that the page fault base address is aligned with the expected large page size. *InvisiPage* [3] and *CoSMIX* [245] implement in-enclave on-demand paging to hide page access patterns via externally visible page faults. CoSMIX additionally provides a

compiler that instruments memory accesses to use the in-demand paging. Shinde et al. [300] and Sinha et al. [307] present compilers for enclave applications that add dummy page accesses to make page accesses oblivious and hide real access patterns. *Dr.SGX* [44] is another compiler-based approach that randomises the memory location of data and performs periodic re-randomisation to hide access patterns.

Other systems propose the use of Intel's *Transactional Memory eXtensions* (TSX) to detect page-table and other interrupt based attacks. TSX allows to wrap groups of instructions into atomic transactions. If an interrupt occurs within a transaction or a cache line of the transaction memory is evicted, any changes are reverted and a user-specified handler is invoked. Chen et al. [68] propose to wrap security-critical enclave functions in TSX transactions and abort enclave execution when detecting abnormal TSX abort rates that indicate an active side channel attack. *T-SGX* [299], *Cloak* [145] and *Heisenberg* [319] automate the protection of wrapping sensitive code in TSX transactions via compiler instrumentation. *Déjà Vu* [67] uses TSX to protect a *reference clock* thread that runs concurrently with an application thread executing sensitive code. The protected reference clock allows for the precise measurement of execution time of the sensitive code path and thus to detect deviations that would indicate a potential attack.

It should be noted that evictions from the L1 cache caused by a hyperthread executed on the same core does not lead to TSX abortions [68]. Most cache-based side channels target the L1 and L2 caches and require co-scheduling with a victim thread on the same core. *HyperRace* [64] and *Varys* [242] therefore enforce the control of both hyperthreads on a physical core when executing security sensitive code. Both systems ask the host OS to co-schedule pairs of enclave threads on the same physical core and verify the co-location by establishing an inter-thread communication channel and measuring communication speeds. Following the disclosure of the Foreshadow attack [345], Intel SGX quotes include whether hyperthreading is enabled as part of the platform configuration and the IAS reports platforms with enabled hyperthreading as potentially insecure. A remote party can therefore attest to the risk of hyperthreading-based side-channel attacks of a platform as part of the remote attestation process. However, disabling hyperthreading can degrade performance significantly as the number of available logical cores is halved. Intel has also issued microcode updates in response to the disclosure of vulnerabilities. Whether a microcode update has been applied to the platform a specific enclave runs on can be verified as part of attestation (see Section 2.4.2.7).

To prevent attacks against the branch target buffer, *ZigZagger* [203] replaces branch instructions with a set of indirect jumps and conditional moves to obscure real branch targets. Hosseinzadeh et al. [164] extend this approach by adding randomisation of the trampoline code at runtime as the ZigZagger defence is insufficient if an adversary can single-step through enclave-code [343].

Another class of defences aims to make memory accesses oblivious. With oblivious RAM (O-RAM) [128] observable memory access patterns are independent of performed algorithms and the data they operate on and are indistinguishable from a random memory trace of the same length. Costa et al. present a new ORAM construction called *Pyramid ORAM* [78] that incorporates the constraints of current trusted processors, in particular the limited amount of private memory. *ZeroTrace* [286] implement a memory controller providing oblivious memory primitives for in-enclave array, set, list, and dictionary data structures. *POSUP* [160] is an SGX-based platform for oblivious searches and updates on large datasets. *Racoon* [274] and *CoSMIX* [245] are compilers for enclave applications to guarantee oblivious execution of developer-annotated sensitive code. *OBFUSCURO* [7] obfuscates full programs.

As the previous discussion shows, TEE side-channel attacks and mitigations form an active research area. It is likely that in the future new TEE side channels and attacks will be found and that corresponding mitigations are developed. When developing new TEE applications, it is important to be aware of existing side-channel attacks and apply corresponding mitigations and to consider the risk of potential yet unknown vulnerabilities. In the context of this thesis, we consider side-channel attacks to be orthogonal. We discuss the targeted threat model in more detail in Section 2.5.

2.4.3.3 Enclave Software Vulnerabilities

Intel SGX does not protect against vulnerabilities in enclave code. In fact, the powerful attacker model of Intel SGX can make attacks easier. For example, *AsyncShock* [356] exploits synchronisation bugs in multi-threaded enclave software and the fact that the host OS is in control of the scheduling of enclave threads. As SGX leaves it up to the host OS to manage page table entries, AsyncShock temporarily removes page permissions in order to trigger page faults resulting in asynchronous exits from enclave threads. This gives the OS fine-grained control over when enclave threads execute and allows it to deliberately trigger a synchronisation bug such as a race condition. The authors show how this can be used to hijack in-enclave control flow or bypass access control checks in application code.

Lee et al. [200] show how a memory corruption vulnerability in enclave code can allow an attacker to determine the memory layout of an application, find *return-oriented programming* (ROP) gadgets, and launch a ROP attack to leak enclave secrets. Biondo et al. [29] extend this work and present a ROP scheme that works without enclave crashes and despite the use of enclave memory layout randomisation [294]. Other attacks target specific enclave frameworks [202, 360]. For example, Weiser et al. present an attack against the RSA implementation of the Intel SGX SDK crypto library [360] that allows an attacker to extract cryptographic keys from an enclave in a single run.

Besides regular techniques to protect against software vulnerabilities such as using a memory safe programming language such as Rust and enabling defensive compiler instrumentations such as stack canaries, some recent work addresses the protection of enclave software in particular. *Moat* [305] and */Confidential* [306] are source-code based verifiers that can ensure at compile time that enclave code cannot leak enclave secrets unintentionally. *SGXBounds* [196] provides compiler-based instrumentation of memory accesses to perform bound checks at runtime in order to prevent unintended writes to or reads from untrusted memory.

Attacks on enclave software are facilitated by the fact that the initial enclave contents are visible to the outside before an enclave is initialised. Intel's SGX SDK includes a *Protected Code Launcher* (PLC) [175] module that can load additional enclave code at runtime from an encrypted enclave library and thus protect code confidentiality. *EnGarde* [238] and *SGXElide* [22] also protect the confidentiality of enclave code via a trusted loader and self-modification, respectively. *SGX-Shield* [294] uses a trusted in-enclave loader that implements *address space layout randomisation* (ASLR) [27] within the enclave to prevent attacks such as the previously described code-reuse attack by Lee et al.

2.4.3.4 Malicious Enclave Applications

Even though enclave code is unprivileged, enclaves have some desirable properties for running malicious software. In particular, they allow to hide computation and with that potentially malicious behaviour. While the focus of this section and the overall thesis is on using TEEs to protect applications from an untrusted system, it is valuable to look at how SGX enclaves could be abused and how abuse can be prevented. This is particularly relevant for the adoption in cloud environments as

providers might be reluctant to provide TEE support if it would hinder detection of malicious tenant behaviour.

Schwarz et al. have shown that SGX enclave code can be used to conceal cache timing attacks that would allow enclave code to extract secrets from other processes [290]. In another work [292], they demonstrate how enclave code can mount a code-reuse attack on its host process and impersonate it, bypassing typical ROP defences such as stack canaries and address space layout randomisation. With full control over its host process, it is possible to then exploit host vulnerabilities similar to regular malware. The *Rowhammer* attack [190] provides another attack vector. The attack causes bit flips in nearby DRAM rows by repeatedly accessing a DRAM address with high frequency. These bit flips can lead to privilege escalation or data corruption, e.g. to turn strong cryptographic keys into weak ones. SGX enclaves might be used to mount and conceal a Rowhammer attack on host memory [146] or on enclave memory as a *denial-of-service* (DOS) attack on the host [146, 183]. For this, an attacker can exploit the behaviour of the Intel SGX *Memory Encryption Engine* (MEE) (see Section 2.4.2.1) on detection of an integrity violation when reading memory contents into a cache line. By causing a bit flip in an EPC page, the MEE will detect a violation on a subsequent load and the memory controller will refuse any further loads, effectively halting the system.

There are a number of ways to protect a system against these types of malicious enclave behaviour. First, attacks that have noticeable effects on system components outside the enclave might be detectable by traditional malware detection systems. For example, a code reuse attack that cause the host process to behave maliciously can be detected by traditional anti-malware software [281]. *SGXJail* [361] sandboxes potentially malicious enclave code by running it in an isolated process separate from the non-enclave application code. Furthermore, it should be noted that the SGX design assumes that enclave code is vetted before it is allowed to launch on a system which is enforceable via launch control policies (see Section 2.4.2.3). However, there might be scenarios in which enclave code implements a trusted loader that loads additional enclave code after an enclave launches, e.g. to protect code confidentiality. *EnGarde* [238] provides an implementation of a loader that can enforce pre-defined security policies on enclave code that could vet enclave code and detect malicious code without the need to reveal it to a cloud provider.

2.5 Threat Model

In this section we discuss the threat model targeted by the three systems described in the following chapters. It is largely driven by the threat model of Intel SGX and similar TEEs.

We consider a strong adversary that has full control over the entire system outside of the TEE itself. They control the complete software stack, including SMM code, the BIOS, hypervisor, operating system, and other software running on the same system. This includes system administrators as well as co-located tenants. As required by the Intel SGX design, the untrusted system is responsible for creating and launching enclaves but might tamper with enclave code and data before enclave initialisation. It can enter enclaves via the pre-defined enclave entry points and might perform *ecalls* in any way permitted by enclave code. The adversary might remove, add, or otherwise modify any state stored outside of enclave memory. This includes state stored in memory and on disk, as well as state transmitted over the network.

Furthermore, we assume the adversary has physical access to the machine. Again, this includes access to main memory, i.e. DRAM, storage devices such as HDDs and SDDs, network cards, and the system bus. An adversary might physically alter any state stored on or transmitted via these components, read persistent state, or observe transmitted data, e.g. via bus snooping [275]. However, we assume that the adversary cannot breach the processor package and access or tamper with any internal processor state such as CPU register values as this would allow them to bypass any TEE boundary enforced by the CPU.

Both the *confidentiality* and *integrity* of sensitive enclave data must be guaranteed. Furthermore, an adversary should not be able to violate the integrity of enclave execution, e.g. to hijack the in-enclave control flow. SGX-LKL, presented in Chapter 3, also addresses the protection of enclave code confidentiality which an enclave owner might want to ensure for proprietary software and to protect intellectual property. We address rollback and replay attacks that violate *freshness* guarantees in Chapter 5.

We assume that enclave software is free of vulnerabilities. Bugs in enclave software might be exploitable to hijack control flow or to extract enclave data as described in Section 2.4.3. We also assume that the Intel-provided architectural enclaves, i.e. the launch, quoting, and provisioning

enclaves, as well as the CPU microcode are free of bugs. We discuss the partitioning of applications in order to reduce the risk of vulnerabilities in enclave software in Chapter 4.

Lastly, we consider side-channel attacks out-of-scope for this thesis. While they are an important aspect to consider, they are orthogonal to the work described in this thesis. Some attacks require hardware changes to be fully mitigated. For others, defence mechanisms exist as described in Section 2.4.3. These can also be used in conjunction with the systems discussed in the following chapters.

2.6 Summary

In this chapter, we provided background and presented related work that is helpful for the understanding of the following chapters. We first discussed the need for secure execution. While data at rest and in transit are commonly encrypted, cloud computing applications that process data typically operate on plaintext data. This leaves sensitive application data exposed to adversaries such as malicious administrators, tenants, or external attackers that gain access to a system. We discussed why homomorphic encryption schemes that enable computation on encrypted data are only practical for a limited set of use cases. We concluded sole encryption of data is therefore not sufficient, and other ways of securely processing sensitive data are needed.

On the basis of this conclusion, we then looked at existing approaches of protecting the execution of applications in untrusted environments. First, we covered software-based approaches that isolate applications from untrusted OSes. Typically, these approaches assume trust in a higher-privileged hypervisor and therefore do not address the risk of a potentially untrusted cloud provider or a compromised cloud infrastructure due to an external attack. We then discussed related work on Trusted Computing that introduces TPMs, hardware modules that allow to integrity check system software and provide remote attestation capabilities to establish trust in a full system stack. We covered how late launch features of commodity CPUs such as Intel TXT and AMD SVM make use of TPMs to provide isolated execution environments for individual applications and discussed their limitations.

We then introduced TEEs as an approach of secure execution that provides both confidentiality and integrity protection for applications in untrusted environments. We compared existing TEE technologies from academia and industry and then discussed Intel SGX in detail. We covered the

creation, management, and lifecycle of SGX enclaves, as well as SGX features such as paging, sealing, and attestation. This background on the capabilities and limitations of Intel SGX provides important context for the system designs discussed in the next chapters.

We concluded the chapter with a discussion of the threat model of TEEs in general and the systems discussed in this thesis in particular. We assume a strong adversary with full control over the entire system, including physical access. We will use this threat model as a basis and refine it if required, e.g. when we address advanced host interface attacks in Chapter 5.

CHAPTER 3

SGX-LKL: PROTECTING UNMODIFIED BINARIES WITH A MINIMAL HOST INTERFACE

This chapter presents SGX-LKL, a library OS that runs unmodified binaries within Intel SGX enclaves with a minimal host interface. We discuss host interfaces of existing TEE runtime systems and their limitations. We present an alternative minimal host interface derived from the requirements of complex applications and the constraints of Intel SGX. We then discuss how SGX-LKL implements system support on top of this minimal interface using the Linux Kernel Library (LKL), and how it protects the remaining host interface.

3.1 Introduction

As discussed in Chapter 2, migrating existing applications to TEEs such as Intel SGX enclaves is non-trivial. Code running inside enclaves is unprivileged and cannot directly access hardware such as network or storage devices. Similarly, other system functionality typically provided by system libraries or the OS are not available within an enclave by default. Instead it is required to cooperate with the untrusted host. New applications can be developed using SGX SDKs [114, 170, 239] while taking these constraints into account. As part of this, developers manually define an interface between untrusted code outside and trusted code inside the enclave. However, existing applications have been built without these constraints in mind. This is particularly problematic for complex applications such as language runtimes, e.g. the Java runtime including the Java Virtual Machine JVM, or data processing frameworks such as TensorFlow. Rewriting them manually is not feasible. At the same time, they require complex system support such as (i) full file system support and access to special file systems such as `/tmp` or `/dev`; (ii) flexible networking support for different protocols such as TCP and UPD; (iii) support for signals, e.g. the ability to register custom signal handlers; and (iv) support for dynamic library loading at runtime.

Existing TEE runtime systems such as Haven [23], SCONE [15], Graphene-SGX [341], and Panoply [301] have demonstrated the feasibility of executing some applications fully inside TEEs with acceptable performance overheads. When applications inside a TEE require external resources such as storage, the network or other OS functionality not provided within the enclave, they must rely on the untrusted host OS. TEE runtime systems therefore have a host interface. However, its security implications are poorly understood and handled [347]. As the discussion of interface-based attacks in Section 2.4.3 has shown, the host interface is a fundamental risk factor for TEEs.

The interface may (i) expose state from the TEE to the outside by accidentally leaking sensitive data in host calls [324]; (ii) expose state from the TEE to the outside explicitly by sharing state with the host when required to perform some functionality, e.g. file metadata when relying on the host OS file system; (iii) may be implemented incorrectly, allowing an attacker to compromise application integrity inside the TEE, e.g. via Iago attacks [62]; and (iv) act as a side channel where the existence or absence of a call reveals application state [354].

Existing TEE runtime systems vary in the types of interfaces that they expose to hosts, and the nature of the interface is a consequence of their design. As a result, current systems expose wide and difficult-to-protect host interfaces, which may compromise the security guarantees of TEEs.

We explore the opposite approach: we begin our design of a TEE runtime system with a desired secure host interface, and then decide on the necessary system support inside the TEE. We aim for a host interface that is (i) *minimal* — only functionality that cannot be provided inside the TEE should be part of it; and (ii) *secure* — all data that crosses the host interface must be encrypted and integrity-protected.

We describe **SGX-LKL**, a new TEE runtime system that executes complex unmodified Linux binaries inside Intel SGX enclaves while exposing a minimal, protected host interface.[1] To protect the host interface and the application, the design of SGX-LKL makes three main contributions:

(1) Minimal host interface (Section 3.3). SGX-LKL only requires a host interface with 7 calls, exposing only low-level functionality for block-level I/O for storage and packet-level I/O for networking. Higher-level POSIX functionality is implemented in SGX-LKL by porting a complete Linux-based library OS to an SGX enclave. SGX-LKL uses the Linux Kernel Library (LKL) [270] to

[1] SGX-LKL is available as open-source software: *https://github.com/lsds/sgx-lkl*.

obtain a mature POSIX implementation, including a virtual file systems layer and a TCP/IP network stack.

(2) Protected host interface (Section 3.4). SGX-LKL ensures that all I/O operations across the host interface are encrypted and integrity-protected transparently: (i) for file I/O, SGX-LKL uses an encrypted and integrity-protected Linux *ext4* root file system image stored outside of the SGX enclave, which is accessed using the *device mapper* subsystem [88] of the Linux kernel; and (ii) for network I/O, SGX-LKL creates a virtual private network (VPN) overlay that secures all network traffic. Layer-3 IP packets are encrypted by the in-kernel Wireguard [364] VPN implementation.

(3) Secure distributed application deployments (Section 3.5). SGX-LKL has built-in support for secure distributed deployments of applications. SGX-LKL integrates support for (i) remote attestation allowing a remote party to verify the integrity of SGX-LKL itself; (ii) the secure deployment of applications while protecting both the integrity and the confidentiality of application code and static application data; and (iii) the secure provisioning of enclave secrets such as cryptographic keys and application arguments and data.

The threat model targeted by SGX-LKL follows the threat model described in Section 2.5 with a focus on attacks against the host interface. An adversary might try to (i) compromise *confidentiality*: they may observe the parameters, frequencies and sequences of host calls; and (ii) compromise *integrity*: the adversary may modify the input or output parameters of host calls, repeatedly perform arbitrary host calls or interfere with their execution outside of the TEE, e.g. via Iago attacks [62].

Our experimental evaluation shows that SGX-LKL exposes a secure host interface and is able to run complex applications with a reasonable performance overhead. With simulated SGX enclaves (to ignore current SGX memory limitations), SGX-LKL trains common deep neural network models using TensorFlow [1] and runs JVM applications with low overhead.

The remainder of this chapter is structured as follows: Section 3.2 discusses existing proposals for TEE runtime systems and their host interfaces; Section 3.3 introduces the high-level design of SGX-LKL; Section 3.4 describes how SGX-LKL hardens the calls in the host interface; Section 3.5 describes how SGX-LKL supports secure deployment, runtime attestation and secret provisioning; in Section 3.6, we report our experimental evaluation results; Section 3.7 discusses related work; Section 3.8 addresses limitations of the approach; and Section 3.9 concludes this chapter.

Table 3.1: Comparison of TEE runtimes

	Panoply [301]	**Scone** [15]	**Graphene-SGX** [341]	**Haven** [23]	**SGX-LKL**
Host interface					
Type	Libc calls	System calls	Hyper calls	Hyper calls	Hyper calls
Size	Large	Large	Medium	Medium	Small
In-enclave functionality					
TCB size	Small	Small	Medium	Large	Medium
File system	✗	✗	✗	✓	✓
Network stack	✗	✗	✗	✓	✓
Threading & Synchronisation	✗	✓	✗	✗	✓
Protection					
Disk I/O	✗	✓	✗	✓	✓
Network I/O	✗	✓	✗	✗	✓
Application code	✗	✗	✗	✓	✓

3.2 Host Interfaces for Trusted Execution

In this section we discuss existing TEE runtime systems and their host interfaces. In particular, we analyse the security of existing TEE host interfaces and derive principles for the design of a secure host interface.

3.2.1 TEE Runtime Systems

Privileged operations such as direct access to storage and network devices must be performed outside of the enclave. This can be achieved either by partitioning applications into untrusted and trusted compartments and ensuring that privileged operations are performed by untrusted application code outside of the enclave, or by delegating these operations to the host OS. Furthermore, application code typically relies on system libraries such as the C standard library (libc), even for unprivileged operations such as string manipulation or mathematical operations. When enclave code requires these operations, they either have to be provided alongside application code within the enclave, or are provided outside and are accessed by enclave code via *ocalls*.

There are a number of TEE runtime systems that allow full applications to run in SGX enclaves. They achieve this by providing system support inside the enclave, and by outsourcing functionality to the host OS transparently to varying degrees. Table 3.1 provides a comparison of these runtimes and SGX-LKL.

Panoply [301] places the whole application inside the enclave but places the libc standard library outside of the enclave. This reduces the TCB size but also results in a large host interface as libc calls made by the application are replaced with corresponding *ocalls*. Since system calls to access OS functionality are typically accessed via corresponding libc wrapper functions, this also ensures that privileged operations are delegated to the host. The calls are replaced at the source level via compiler instrumentation. Panoply itself does not provide protection for disk or network I/O. Instead it requires application developers to modify their applications to protect sensitive data sent to and from the TEE application.

SCONE [15] is another TEE runtime system that, similar to Panoply, delegates a large set of system functionality to the host OS. In contrast to Panoply, SCONE provides a full libc standard library within the enclave and only delegates system calls to the host OS. It uses the host OS file system and network stack but provides user-level threading in-enclave. In also adds a shim layer to transparently protect network I/O via TLS, and disk I/O by encrypting and authenticating individual files.

Graphene-SGX [341] reduces the host interface size compared to Panoply and SCONE by placing a custom library OS [340] inside the enclave alongside the application. By doing so, most system calls can be handled within the enclave alongside the application, and only a subset of OS functionality is delegated to the host OS, e.g. for privileged operations. Graphene-SGX runs unmodified Linux binaries and provides transparent support for OS functionality such as process forking. However, Graphene still relies on file system and network stack implementations as well as the scheduler of the host OS for which it provides a *hyper call* like host interface. We refer to these calls as hyper calls as they do not directly map to host OS system calls. Instead, they provide a custom abstraction on top of host OS system calls. For example, Graphene-SGX has a `wake_thread` *ocall* implemented on top of the `tgkill` system call of the host OS. Graphene-SGX loads application binaries and shared libraries from the host file system but ensures their integrity when loading them using precomputed cryptographic hashes. In contrast to SCONE, it does not provide transparent network I/O or disk I/O protection.

Haven [23] is a Windows-based TEE runtime and is closest to SGX-LKL. Haven runs unmodified Windows binaries using the *Drawbridge* library OS [264]. With Drawbridge, Haven provides full file system and network stack implementations inside the enclave which reduces the size of its host interface. Application executables and data are stored on an encrypted disk image on the host. This

way, Haven can support application code confidentiality by loading the executable from the encrypted image only after the enclave has been initialised. It also supports volume-level disk integrity using a Merkle tree. However, Haven relies on the host OS for multithreading and synchronisation, does not provide transparent network encryption, and assumes the existence of a number of instructions not available in SGX 1 hardware, e.g. for virtual memory management.

3.2.2 Host Interface as Attack Surface

Next we analyse the security of the host interfaces of these TEE runtime systems.

Size of the host interface. The size of a TEE host interface has a major impact on its security. With the strong memory isolation that TEEs provide, *ocalls* and *ecalls* represent the main attack surface for TEE applications. Logically, any additional *ecall* and *ocall* increases the host interface's complexity, requires additional checks for confidentiality and/or integrity, and adds to the risk of bugs in particularly security-critical code. In addition, just the absence or presence of a call allows an adversary to infer information about an application's state and potentially reveals sensitive information. Therefore, minimising the size of a host interface is an important aspect of secure interface design. For application-specific host interfaces, the size of the host interface can be reduced by minimising the code requiring system functionality inside the enclave (see Chapter 4 and Chapter 5). Furthermore, the host interface is reduced to the needs of the specific application. In contrast, TEE runtime systems must provide support for a diverse set of applications and should be application-independent.

Note that the size of the interface does not only refer to the number of distinct calls but also the number of call arguments. For example, there is no difference in the security of an interface design that has distinct `read` and `write` host calls, and one that has a single call for I/O operations with an additional argument that indicates a read or write operation. In our security analysis, we therefore focus on the parameters of host calls as a measure of the attack surface area.

Types of host interface parameters. Besides the total number of host calls and parameters of a system, a host interface's security is also impacted by host call parameter types. We categorise parameters into five types, ordered by the difficulty of protecting them from easiest to hardest:

(i) *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()` call [276];

Table 3.2: Security breakdown of parameters in host interface calls for existing TEE runtime systems

TEE runtime system	Function	Number of host calls	Out parameters (impact confidentiality) More difficult to protect →					In parameters (impact integrity) More difficult to protect →				
			Variable size buffer	Address range	Pure identifier	Impure identifier	Semantic parameters	Variable size buffer	Address range	Pure identifier	Impure identifier	Semantic parameters
Panoply [301]	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 [341]	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 [23]	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	–	–
SGX-LKL	I/O	4	2	–	–	–	2	2	–	–	–	4
	Events	2	–	–	–	–	1	–	1	–	–	2
	Time	1	–	–	–	–	–	–	–	–	–	1
	Threading	–	–	–	–	–	–	–	–	–	–	–

(ii) *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()`;

(iii) *pure/impure identifiers* point to entities: pure identifiers, e.g. a file descriptor, only identify an entity; impure identifiers, e.g. a path name, additionally disclose information about the entity; and

(iv) *semantic parameters* refer to parameters with opaque semantics specific to the host call, such as `mode` and `flags` for file access operations.

Table 3.2 shows the total number of host calls for each TEE runtime system, categorised according to function (I/O, events, time, threading).[2] Panoply exposes a large host interface, with 302 host calls in total; Graphene-SGX and Haven require 38 and 24 host calls, respectively. We do not include a breakdown of SCONE's interface as its implementation is not public. As it forwards most system calls, its host interface can be assumed to be similar to that of Panoply.

We break down the parameters according to the above types, distinguishing between *out* parameters, which are passed to the host and may compromise confidentiality, and *in* parameters, which are passed into the TEE and may affect integrity. Return values of *ocalls* are considered to be the same as in parameters.

Confidentiality attacks. To learn sensitive information, an adversary may observe the host call parameters. The data disclosure depends on the type of *out* parameters:

(i) *Variable-sized buffers* may contain security-sensitive data. To ensure confidentiality, such data must be encrypted and padded to a fixed size. If the true size is exposed, an adversary may infer size-dependent secrets. For example, in an image classification application [195], an adversary may learn the classification by considering different result buffer lengths.

Table 3.2 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. Buffers are encrypted but their sizes are disclosed. Haven implements an in-enclave file system and writes data to the host disk as fixed-sized blocks; SCONE divides files into fixed-size chunks for authentication but reveals the number of chunks; Graphene-SGX and Panoply do not

[2]For Panoply and Graphene-SGX, the host calls are taken from the GitHub source code [141, 251]; for Haven, they are obtained from the paper [23].

provide transparent file encryption. In addition, SCONE, Panoply, and Graphene also reveal sizes of individual files.

(ii) *Address ranges* passed from the TEE to the host point to continuous regions of untrusted memory. For I/O operations, Panoply, Graphene-SGX, and Haven expose 17, 1 and 3 address ranges, respectively. An adversary may observe their usage pattern. For example, Panoply uses untrusted memory address ranges for communication between enclaves that isolate compartments of the same application. An adversary can observe the usage pattern to reveal application-specific control flows, e.g. secret-dependent inter-enclave calls.

(iii) *Pure/impure identifiers.* Panoply, SCONE, Graphene-SGX, and Haven use a large number of identifiers for I/O operations, event handling and threading. Similar to address ranges, passing identifiers to the host reveals usage patterns. For example, Cash et al. [56] show that an adversary can learn database queries by observing record accesses.

Generally, pure identifiers reveal less information than impure identifiers, e.g. file names, as pure identifiers can be chosen randomly. File metadata such as directory and file names, sizes, and attributes can reflect the current state of an application or directly reveal application secrets. Preventing information leakage by impure identifiers can only be done on a case-by-case basis, e.g. by replacing file names with hashes. The above systems do not provide any protection for impure identifiers.

(iv) *Semantic parameters* have a context-specific meaning and therefore cannot be encrypted transparently. Existing runtime systems rely on a large number of 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 of the current application state and enables attacks such as *AsyncShock* [356], which exploit host control of enclave threads.

Integrity attacks. To compromise integrity, an adversary may tamper with parameters. Integrity-protecting *in* parameters varies in difficulty according to the parameter type:

(i) *Variable-sized buffers* passed to the TEE must have their integrity protected. For I/O calls, Panoply, Graphene-SGX, and Haven use 10, 3 and 1 parameters with variable-sized buffers, respectively. Panoply and Graphene-SGX also use variable-sized buffers for inter-enclave communication.

This exposes two means of integrity attacks: an adversary may modify (i) the buffer contents to violate data integrity; and (ii) the buffer count to trigger an overflow. To ensure buffer integrity, the contents must be protected cryptographically, e.g. by an HMAC. The TEE must ensure the freshness of the HMAC, otherwise an adversary can swap two valid buffers or perform a roll-back attack. Panoply protects the integrity of the inter-enclave communication buffer 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 do not protect the integrity of network I/O for applications without built-in TLS support.

To prevent buffer overflow attacks, the TEE must check buffer lengths. All analysed runtime systems do this.

(ii) *Address ranges* passed into the TEE are difficult to integrity check. For I/O operations, Panoply and Graphene-SGX pass 2 address ranges each into the TEE. They check that the ranges are entirely inside or outside of TEE memory, preventing an adversary from hijacking an enclave's control flow [29, 200].

An adversary, however, may still manipulate addresses by reusing old ranges to roll back data, swapping ranges, or modifying ranges to corrupt data. All three runtime systems are potentially vulnerable to such attacks, and the only effective mitigation is to avoid relying on untrusted address ranges.

(iii) *Pure/impure identifiers* are commonly used as *in* parameters: 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.

Adversaries may pass invalid or manipulated identifiers: if two file descriptors are swapped, the enclave may access incorrect files. Such malicious activity can be detected: Graphene-SGX maintains per-file HMACs, which reveal wrong file descriptors. Similarly, incorrect network sockets can be detected by TLS.

(iv) *Semantic parameters* must have their integrity verified on a case-by-case basis. All three systems use many semantic *in* parameters for I/O, events, time and threading operations.

For semantic parameters with a fixed set of 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 establish semantic correctness though. Since Panoply, SCONE and Graphene-SGX rely on the host file system, they can only check for the plausibility of returned file metadata, such as `st_size` and `st_blocks`.

3.2.3 Designing a Secure Host Interface

While host interfaces of existing runtime systems as well as SDKs have also been shown to contain implementation flaws [347], the above issues related to the host interface are more fundamental. We conclude that it is non-trivial (and at worst impossible) to protect *out* parameters from leaking information and exposing access patterns and to verify the correctness of *in* parameters. Therefore, the first step in designing a secure host interface is to keep it narrow and minimise the number of parameters. Only functionality that cannot be provided within an enclave should be delegated to the host.

In addition, it is important to reduce the number and complexity of host call parameters. By building on low-level host calls instead of high-level POSIX abstractions the parameter types become simpler. For example, Panoply, SCONE, and Graphene-SGX rely on the host file system and thus must expose impure identifiers such as file names. A secure host interface should avoid delegating resource management to the host as much as possible.

Based on these principles, Table 3.2 shows the host calls and parameter types of SGX-LKL, our TEE runtime system. It avoids address range and pure/impure identifiers and 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.

3.3 Minimising the Host Interface

To achieve such a narrow interface, it is necessary to reduce the reliance on the host system. For a minimal host interface, required functionality that can be provided inside the enclave, should be implemented without host interaction. Therefore, SGX-LKL exposes only seven calls to the untrusted host, as shown in Table 3.3. These calls relate to functionality that cannot be provided inside an SGX enclave: disk and network I/O operations, event handling, and time.

Table 3.3: SGX-LKL host interface

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

3.3.1 SGX-LKL Host Interface

In the following, we describe each call in more detail and discuss their dependence on the host.

I/O operations. SGX-LKL uses 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. As SGX-LKL reads and writes fixed-sized disk blocks, `len` is always the same as the block size (4 KB) or a multiple thereof.

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. A `net_poll()` call passes an `eventmask` to the host with the network events that SGX-LKL 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.

SGX-LKL must 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 SGX-LKL directly or forwarded to the application if it has registered a corresponding signal handler.

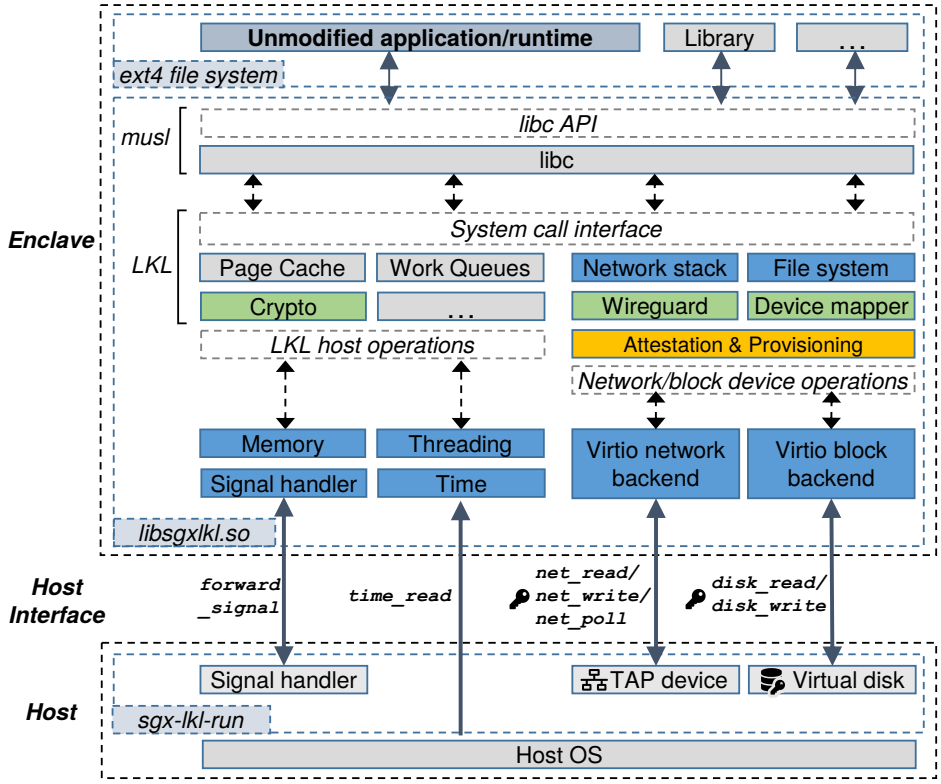


Fig. 3.1: SGX-LKL architecture

Time. SGX does not provide a trusted high-performance time source, and time must be provided by the host. The 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 SGX-LKL itself, e.g. to generate timer interrupts required by LKL (see Section 3.3.2).

3.3.2 In-Enclave OS Functionality

The host interface described above allows SGX-LKL to access low-level host resources, but Linux applications require higher-level POSIX abstractions. To bridge this gap, SGX-LKL provides the following OS functions inside the enclave: (i) file system implementations; (ii) a TCP/IP network stack; (iii) threading 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 [15, 341], SGX-LKL includes a libc implementation to support unmodified binaries that are dynamically linked against libc.

Fig. 3.1 shows the SGX-LKL architecture. Next we describe the OS functionality provided inside the enclave in detail. Components of SGX-LKL added to complement LKL and to implement the required LKL host operations are shown in blue. It also shows parts responsible for protecting the host interface (green, see Section 3.4), and those that add support for deployment, attestation, and secret provisioning (yellow, see Section 3.5).

An untrusted loader, *sgx-lkl-run*, creates the SGX enclave and loads the enclave library *libsgxlkl.so*, which runs alongside the application within the enclave. *libsgxlkl.so* includes a modified *musl* [105] C standard library that redirects system calls to an in-enclave library OS provided by the *Linux Kernel Library* (LKL) [270]. LKL is an architecture port of the Linux kernel to userspace. It enables SGX-LKL to make components such as the Linux kernel page cache, work queues, file system and network stack implementations, and crypto libraries available inside the enclave. As it is intended to run in userspace, LKL expects implementations of a set of *LKL host operations*, e.g. to create threads or to allocate memory. Therefore, *libsgxlkl.so* includes further components for memory management, user-level threading, signal handling and time.

3.3.2.1 File System Support

Existing TEE runtime systems [15, 301, 341] (see Section 3.2.1) forward POSIX file operations to a host file system implementation. However, this approach exposes security-sensitive metadata, such as file names, file sizes, file permissions, and directory structures to the host. Instead, SGX-LKL provides complete in-enclave file system implementations via the Linux *virtual file system* (VFS) layer [39]. The VFS layer only requires two host operations for block-level disk I/O (`disk_read()` and `disk_write()`, see Table 3.3).

Applications operate on files through file descriptors as usual, which are handled by the *ext4* file system implementation of LKL. LKL forwards block-level I/O requests to a *virtio block device* backend implemented by *libsgxlkl.so*, which issues `disk_read()` and `disk_write()` calls. On the host, the reads and writes are made to a single *ext4* disk image file. The image is mapped into memory by *sgx-lkl-run*. Since the disk image file has a fixed size the read/write operations can be implemented efficiently by memory-mapping the file and directly reading from and writing to the mapped region from within the enclave.

This approach has three advantages: (i) it maintains a small host interface with only 2 disk I/O calls; (ii) it ensures that individual file accesses are not visible to the host, which can only observe reads/writes to disk block offsets; and (iii) the in-enclave VFS implementation supports different file systems, such as the temporary in-memory file system `/tmp`, the `/proc` file system, and `/dev` for special device files, such as `/dev/{u}random`.

3.3.2.2 Networking

To provide a POSIX socket API, SGX-LKL uses LKL's network stack to process packets within the enclave. 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 SGX-LKL to support any transport protocol, e.g. TCP or UDP, without extra host calls; and (iii) it enables the use of Linux networking features such as packet encryption (see Section 3.4.2).

To send/receive network traffic, *sgx-lkl-run* sets up a layer-2 TAP device. SGX-LKL implements a corresponding *virtio network device* backend inside the enclave. To be notified about incoming and outgoing packets, the backend issues a `net_poll()` request. The return value indicates if the device is ready for reading or writing packets using `net_read()` and `net_write()`. `net_read` and `net_write` both use a shared buffer to exchange Ethernet frames processed and created by the in-enclave network stack.

One problem for network-intensive enclave applications is the high cost of enclave transitions. To avoid this cost, the *sgx-lkl-run* creates multiple network I/O threads that handle incoming and outgoing packets via a pair of shared queues, similar to asynchronous host call mechanisms in other systems [15, 244, 313, 331, 362]. A request queue is used to issue `net_poll()`, `net_read()`, and `net_write()` calls, which are handled by a host I/O thread that invokes a corresponding host system call on the TAP device. When the system call returns, the result value is returned through a return queue to the virtio network backend code inside the enclave.

3.3.2.3 Memory Management

SGX-LKL does not interact with the host for memory allocations and deallocations. SGX version 1 requires the enclave size to be fixed at initialisation time. SGX-LKL therefore pre-allocates enclave memory and provides low-level memory management primitives inside the enclave. When an enclave

is created, it initially contains *libsgxlkl.so*, SGX-specific memory pages such as the SSA and TCS structures, and an 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 SGX-LKL.

SGX-LKL 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. To support private file mappings, files are loaded into the enclave since SGX enclaves are bound to a linear address space. For the same reason, SGX-LKL only supports private mappings as writes cannot be carried through to the underlying file.

SGX-LKL must support changing page permissions, e.g. when loading executables and libraries. In addition, applications may modify permissions directly. For example, the JVM requires executable pages for just-in-time compilation and must be able to change permissions for guard pages during garbage collection. While SGX pages have their own permissions, SGX version 1 requires these permissions to be set on enclave creation and does not permit subsequent changes.

As a workaround, SGX-LKL implements an extra `mem_protect()` host call for SGX 1 execution. All enclave pages are created with full SGX page permissions and the actual permissions are set via the host-controlled page table permissions. Since relying on the host to manage page permissions is a security risk, SGX version 2 adds the ability to control page permissions from within the enclave.

3.3.2.4 Multithreading and Thread Management

While 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 creating, joining, and exiting threads as well as thread synchronisation requires host OS support, which poses a security threat [356].

Therefore, SGX-LKL implements user-level threading based on the *lthread* library [8] and provides synchronisation primitives inside the enclave. Fig. 3.2 illustrates SGX-LKL's threading architecture. A fixed number of host threads are assigned to enclave threads, which enter the enclave at startup and only leave when idle, or as part of asynchronous exits due to interrupts or exceptions. Each enclave

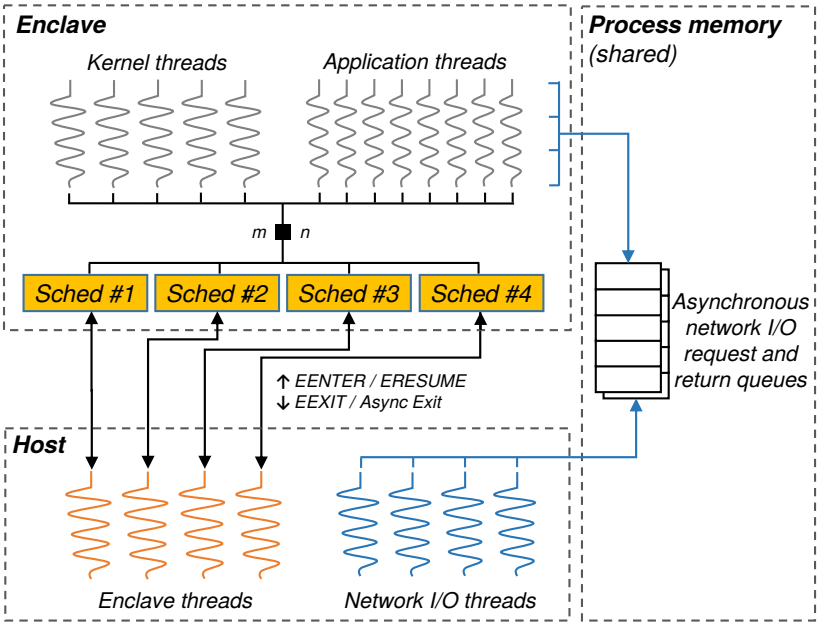


Fig. 3.2: SGX-LKL user-level threading

thread is controlled by a scheduler that together control the scheduling of in-enclave user-level *lthreads*. These are either application threads or LKL kernel threads that are created and managed via the standard *pthreads* interface. The m-to-n mapping of host and lthreads allow SGX-LKL to implement *futex* calls in-enclave to provide synchronisation primitives such as mutexes and semaphores.

3.3.2.5 Signal Support

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. SGX-LKL must forward these signals from the untrusted host to custom signal handlers inside the enclave.

During initialisation, *sgx-lkl-run* registers signal handlers for all catchable signals with the host. All signals are forwarded via `forward_signal()`. It also hides application-specific handlers. SGX-LKL 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 enclave, calls such as `sigaction()`, `sigsuspend()`, `sigtimedwait()` and `sigprocmask()` are supported without host interaction.

3.3.2.6 Timing

Applications, libraries and LKL frequently access time information. In addition, SGX-LKL reads the current time between context switches to reschedule blocked threads or to trigger timer interrupts. Current SGX implementations, however, do not offer a high-performance in-enclave time source, and SGX-LKL relies on time provided by the host.

Instead of issuing expensive individual host calls, SGX-LKL uses the *virtual dynamic shared object* (vDSO) [351] mechanism of the Linux kernel. The host 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 enclave 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 SGX-LKL thus depends on the frequency of vDSO updates. SGX version 2 does not have this limitation.

3.3.3 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 SGX-LKL must support unmodified binaries, illegal instructions cannot be replaced by corresponding host calls.

Instead, SGX-LKL catches the resulting `SIGILL` exception and emulates the instructions: `RDTSC` is executed outside the enclave, and the result is forwarded via the `forward_signal()` call; for `CPUID`, SGX-LKL caches all `CPUID` information during enclave setup. This eliminates the need for an extra host call and also hides the `CPUID` information requested by an application.

3.4 Protecting the Host Interface

We now describe how SGX-LKL protects its host calls (see Table 3.3) from attacks. In particular, we explain (i) how to ensure the confidentiality and integrity of data read and written via the disk

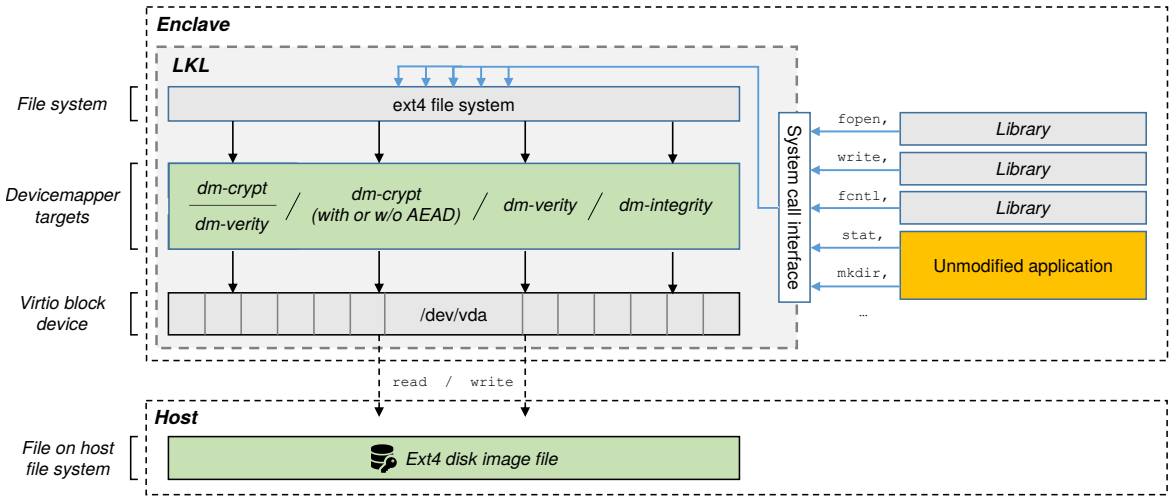


Fig. 3.3: Disk encryption and integrity protection in SGX-LKL via Linux' device mapper targets

I/O (Section 3.4.1) and network I/O (Section 3.4.2) host calls; and (ii) how event and time calls are prevented from compromising application integrity (Section 3.4.3).

3.4.1 Protecting Disk I/O Calls

Host disk blocks must have their confidentiality and integrity protected. For this, SGX-LKL makes use of incorporating the Linux kernel which provides mature full disk encryption. It uses the Linux *device mapper* subsystem [88], which maps lower-level virtual block devices, such as the one exposed by SGX-LKL's virtio backend, 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 device mapper. Data can be encrypted and integrity protected transparently before it reaches the underlying device.

SGX-LKL uses different device mapper targets as illustrated in Fig. 3.3: (i) *dm-crypt* [91] provides full-disk encryption using AES in XTS mode with the sector number as an initialisation vector; (ii) *dm-verity* [93] 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* [92] 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, SGX-LKL uses hardware acceleration through Intel's AES-NI instruction set extension. For this, we extended LKL to support x86-specific kernel modules that implement the required cryptographic primitives via Intel's AES-NI instruction set extension.

SGX-LKL combines the different targets to provide both confidentiality and integrity for block reads and writes, depending on the security requirements of the application. For example, for an in-memory key-value store such as *Memcached* [220], 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 sensitive application data is stored to disk, *dm-crypt* can be used with either of the integrity protection targets. SGX-LKL's use of LKL also allows it to support other device mapper based protection targets such as *dm-x* [57] 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 3.6.3.1.

By building on top of existing disk encryption technologies, SGX-LKL also simplifies the secure deployment as well as the provisioning of static data whose integrity or confidentiality must be protected. We discuss application deployment in more detail in Section 3.5.

3.4.2 Network Protection

SGX-LKL 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 [23, 301, 341] or use TLS [15]. However, some applications do not support TLS, e.g. TensorFlow, or rely on other transport protocols such as UDP-based streaming applications.

Instead, SGX-LKL exploits having a complete 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, SGX-LKL uses *Wireguard* [364], a layer 3 virtual private network (VPN) protocol, currently proposed for inclusion in the Linux kernel [210].

SGX-LKL 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 —

SGX-LKL binds keys to enclave identities and supports the secure provisioning of peers' keys (see Section 3.5).

3.4.3 Protecting Event and Time Calls

For the remaining calls in Table 3.3, SGX-LKL 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 or 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 3.4.2). A wrongly triggered `net_write` call might result in SGX-LKL overwriting existing buffer contents, potentially leading to modified or lost network packets. However, this can be detected and handled transparently by Wireguard's integrity protection and/or the higher-level transport protocol.

`forward_signal()`. SGX-LKL 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, SGX-LKL 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`). The latter is required for two reasons: (i) SGX stores the faulting instruction address in the SSA before an asynchronous exit and does not reveal the exact address to the host; and (ii) a malicious host could pass in an incorrect address in order to tamper with the behaviour of an in-enclave signal handler. SGX-LKL ensures that the signal handler receives a signal description with the correct address by loading the faulting instruction address from the SSA before calling the handler.

In addition, SGX-LKL can be configured to ignore user-controlled signals (e.g. `SIGINT` or `SIGUSR1`) to prevent a malicious host system from interacting with the application in unintended ways. For example, an application might register a `SIGUSR1` handler to interact with other trusted processes, e.g. to implement inter-process control flows. However, a malicious host could raise the signal at any time to trigger an action, not adhering to the expected control flow and potentially leading to unintended application behaviour.

`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, or allow application-specific replay attacks by resetting a clock to an older time. For `CLOCK_MONOTONIC_*` clock sources, SGX-LKL therefore checks for monotonicity and aborts if it detects a violation.

For other clock sources, verification is more difficult as they are not guaranteed to always go forward. Note that the official Intel SGX SDK provides *trusted time* for enclaves based on the Intel Management Engine (ME). Accesses, however, are slow and requests can be arbitrarily delayed by an adversary. Future TEE implementations may provide more practical trustworthy time sources.

3.5 Runtime Attestation and Secret Provisioning

SGX-LKL must execute securely in an untrusted and potentially malicious environment. For this, it must allow (i) parties to remotely attest that they execute a trustworthy version of SGX-LKL in a genuine SGX enclave; (ii) applications to be deployed securely, i.e. guaranteeing both the confidentiality and integrity of application code; and (iii) applications to be provisioned securely with secrets such as cryptographic keys, configurations, and sensitive application data.

The above requirements go beyond the attestation mechanisms of current SGX SDKs [170, 239] and TEE runtime systems. While current SDKs can attest a single enclave library, they assume that all application code is compiled into the library, and an enclave measurement therefore is sufficient to verify integrity. They also do not protect the confidentiality of the application code. Applications typically must implement their own mechanism for secret provisioning. This is cumbersome given the wide range of secrets that must be shared, including administrator credentials, certificates and encryption keys, which are provided as command line arguments, environment variables, or configuration and certificate files.

SGX-LKL addresses these issues as part of three phases: (i) *application provisioning*, (ii) *remote attestation* and (iii) *secret provisioning*.

Fig. 3.4 shows the deployment workflow, involving three parties: (i) a *service provider* (SP) that wants to deploy an application and has a trusted client. For a distributed application, this may involve multiple trusted peers; (ii) an untrusted host controlled by a *cloud provider* (CP) that provides

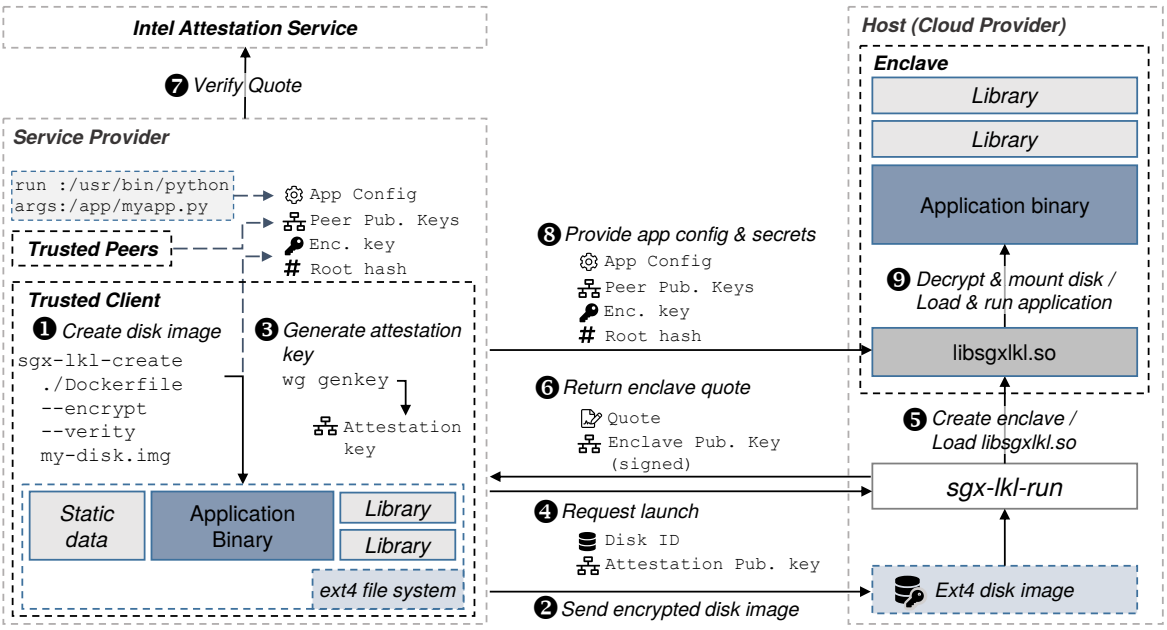


Fig. 3.4: Deployment workflow for SGX-LKL

enclaves; and (iii) the *Intel Attestation Service* (IAS), which allows the SP to verify an enclave measurement.

(1) Application provisioning. In the first phase, the SP provisions its application to the CP to be run inside an enclave. SGX-LKL supports this without revealing application code and static application data.

For this, in step ❶, the trusted client creates a disk image with the application binary and its dependencies. It can be created e.g. by exporting a Docker container image [94] — SGX-LKL provides an `sgx-lkl-create` tool to simplify the disk image creation process. For example, `sgx-lkl-create` can create an SGX-LKL ext4 disk image from a Dockerfile or an existing Docker image. It also incorporates *cryptsetup* [82] for disk encryption and integrity protection via *dm-crypt*, *dm-integrity*, and *dm-verity*. In addition to the encrypted disk image file, it outputs the disk encryption key and the root hash of the Merkle tree for *dm-verity* integrity protection. In step ❷, the disk image file is sent to the CP. Lastly, in preparation for attestation, the client generates a Wireguard asymmetric attestation key pair in step ❸.

(2) Remote attestation. In the second phase, an SP verifies that the application has been deployed without tampering and that it runs in a genuine enclave using remote attestation.

To prepare for remote attestation, in step ❹, the public key of the attestation key pair and the disk ID identifying the image is sent to the cloud host. The host then creates an enclave containing *libsgxlkl.so* (step ❺). *libsgxlkl.so* boots LKL, sets up the networking, and generates its own Wireguard key pair. *libsgxlkl.so* is set up to only accept a remote attestation request from a Wireguard peer that owns the attestation key pair, which guarantees that no other party can communicate with the enclave.

At this point, the enclave can be attested: *libsgxlkl.so* creates a report with a measurement of the enclave code. SGX-LKL includes the generated public key as report data bound to the report. The report is signed by the *quoting enclave*, and the resulting quote together with the enclave's public key is returned to the SP in step ❻. In step ❼, the SP sends the quote to the IAS, which returns a verification report, which is checked by the SP. If the attestation fails, e.g. because the enclave contents have been tampered with or the IAS verification fails, the SP aborts the deployment process.

(3) Secret provisioning. In the third and last phase, the SP establishes a secure communication channel with the enclave using the enclave public key, to provision application secrets.

For this, the SP sends the following information to the enclave in step ❽: (i) the disk encryption key; (ii) the root hash; (iii) public keys of other trusted peers; and (iv) configuration data, including the path to the executable, its application arguments, as well as environment variables if required. In step ❾, *libsgxlkl.so* mounts the disk and sets up the device mapper targets for decryption/integrity. It adds the new Wireguard peers, loads the application, and begins execution.

By providing public keys of other trusted peers, a service provider can ensure that only trusted clients can communicate with the enclave. It also allows an SP to securely set up a distributed application consisting of multiple enclaves. For this, it first creates and sets up individual enclaves as described above and then distributes the public keys of the involved SGX-LKL instances among them.

The deployment process above allows an SP to deploy an application securely and transparently, e.g. without requiring any changes to the application itself. It (i) protects application code confidentiality and integrity; (ii) attests that the enclave is genuine and Intel SGX is configured correctly; and (iii) sets up a VPN as a secure network between trusted application enclaves and other trusted nodes.

3.6 Evaluation

This section evaluates SGX-LKL using both application benchmarks and microbenchmarks. The goal of this section is to demonstrate (i) that SGX-LKL can run complex applications without modifications; (ii) that it allows for the secure deployment of both single-node and distributed applications; (iii) that it can do so with low overhead; and (iv) the impact of optimisations we have implemented to improve secure disk and network I/O performance.

In the following, we first describe the experimental setup. Then we demonstrate the deployment of a variety of complex applications with SGX-LKL and benchmark their performance. We conclude this section with a set of microbenchmarks for disk and network I/O.

3.6.1 Experimental Setup

We conduct all the experiments on SGX-enabled machines with Intel Xeon E3-1280 v5 4-core CPUs with 8 MB LLC, 64 GB RAM, and a 10-Gbps network interface. The CPU supports SGX version 1 and 128 MB of PRM of which approximately 92 MB are usable as part of the EPC. The machines run Ubuntu Linux 18.04, Linux Kernel 4.15, with the Intel SGX driver version 2.6.

We conduct experiments with SGX-LKL in two configurations, running (i) in SGX enclaves, and (ii) in a simulation mode. We use the simulation mode to estimate the overhead of SGX-LKL for future CPUs with larger EPC sizes. SGX-LKL runs the same code both in hardware and simulation mode and uses the same asynchronous host call mechanism. However, SGX-LKL in simulation mode does not suffer from memory access and paging overheads of current SGX implementations (see Section 2.4.2.1).

3.6.2 Applications

In this section, we demonstrate the ability of SGX-LKL to run complex applications. We show that SGX-LKL runs these applications (i) without requiring any changes to the application binaries; and (ii) with low overheads, with and without disk and network encryption and integrity protection. We show that SGX-LKL is capable of supporting a wide range of applications by evaluating it with a number of benchmarks and widely used language runtimes. We run the PARSEC benchmark suite [28], Python with TensorFlow [1], and the JVM using the Dacapo benchmark suite [31].

To ensure a fair comparison, we use Docker for native execution, and convert the Docker images to SGX-LKL disk images. This way, we guarantee that during both native and SGX-LKL execution the same application and library binaries are used. In all tests, we enable disk encryption and integrity protection using dm-crypt and dm-integrity, as well as network encryption using Wireguard.

Note that SGX-LKL uses cooperative user-level threading whereas the native execution relies on preemptive scheduling by the host kernel. While this can be an advantage for SGX-LKL, the main objective for our evaluation is to demonstrate that SGX-LKL's performance is competitive and does not introduce significant additional overhead. We believe that this can be shown across the two threading models, in particular as we exclusively run the benchmarked applications in which case the cost of preemptive scheduling should be small.

3.6.2.1 PARSEC

The PARSEC benchmark suite includes a set of multi-threaded applications from a number of application domains, including image processing (vips), data compression (dedup), and computer vision (bodytrack). It uses both POSIX threads and OpenMP [243] for multithreading. We evaluate it in different configurations: (i) with 1 and with 4 application threads to demonstrate SGX-LKL's efficient user-level threading; and (ii) with two different workloads, using the PARSEC-provided input data sets *simsmall* and *simlarge* which contain input data of different sizes. With this, we demonstrate the impact of EPC paging and the difference between SGX-LKL in simulation and hardware mode. We combine these two options and thus run the benchmark in four different configurations.

Table 3.4 shows the runtimes for each workload and benchmark when PARSEC is executed natively, with SGX-LKL in simulation mode, and with SGX-LKL in hardware mode.[3] It also shows the overhead of SGX-LKL relative to native execution. For the *simsmall* workload, SGX-LKL in simulation mode has runtimes between 0.61× and 1.22×, and between 0.88× and 1.67× compared to native execution for 1 and for 4 threads, respectively. In hardware mode, runtimes are between 0.67× and 2.82×, and between 0.93× and 3.37× those of native execution. SGX-LKL performs better than native in a number of tests due to efficient user-level threading and not requiring context

[3]The PARSEC benchmarks `facesim`, `ferret`, and `x264` are excluded as they are incompatible with Alpine Linux/ musl libc and fail to run, both natively and with SGX-LKL. As their key characteristics [28], e.g. the degree of data parallelism and their use of synchronisation primitives, are covered by other tests in the benchmark suite, we do not believe that their omission has an impact on the overall findings.

Table 3.4: Parsec runtimes (in seconds) and overheads for SGX-LKL in simulation and hardware compared to native execution

	black-scholes	bodytrack	canneal	dedup	fluidani-mate	freqmine	raytrace	stream-cluster	swaptions	vips
Workload: **simsmall** Threads: **1**										
Native	0.12s	0.70s	1.21s	0.58s	1.11s	0.19s	19.10s	0.61s	0.43s	0.33s
Simulation	0.07s (**0.61**x)	0.63s (**0.90**x)	1.36s (**1.12**x)	0.56s (**0.97**x)	1.11s (**1.00**x)	0.26s (**1.36**x)	23.37s (**1.22**x)	0.57s (**0.93**x)	0.40s (**0.93**x)	0.24s (**0.73**x)
Hardware	0.08s (**0.67**x)	0.99s (**1.42**x)	2.62s (**2.17**x)*	0.61s (**1.05**x)	1.41s (**1.27**x)*	0.54s (**2.82**x)*	34.65s (**1.81**x)*	0.58s (**0.94**x)	0.42s (**0.99**x)	0.30s (**0.92**x)
Workload: **simsmall** Threads: **4**										
Native	0.03s	0.19s	1.00s	0.20s	0.33s	0.16s	19.00s	0.17s	0.10s	0.20s
Simulation	0.03s (**0.88**x)	0.22s (**1.15**x)	1.17s (**1.17**x)	0.19s (**0.96**x)	0.35s (**1.05**x)	0.26s (**1.61**x)	22.89s (**1.20**x)	0.18s (**1.05**x)	0.10s (**1.04**x)	0.24s (**1.22**x)
Hardware	0.03s (**0.93**x)	0.32s (**1.68**x)	2.43s (**2.43**x)*	0.37s (**1.84**x)*	0.75s (**2.28**x)*	0.54s (**3.37**x)*	34.15s (**1.80**x)*	0.18s (**1.06**x)	0.11s (**1.12**x)	0.30s (**1.50**x)
Workload: **simlarge** Threads: **1**										
Native	1.26s	7.47s	6.59s	10.99s	7.10s	1.31s	25.62s	11.92s	6.68s	1.57s
Simulation	1.15s (**0.92**x)	7.73s (**1.03**x)	7.31s (**1.11**x)	9.71s (**0.88**x)	7.24s (**1.02**x)	1.62s (**1.23**x)	30.91s (**1.21**x)	11.90s (**1.00**x)	6.25s (**0.93**x)	1.88s (**1.20**x)
Hardware	1.28s (**1.02**x)	12.16s (**1.63**x)	76.82s (**11.66**x)*	13.74s (**1.25**x)*	13.26s (**1.87**x)*	3.75s (**2.86**x)*	39.70s (**1.55**x)*	13.65s (**1.15**x)	6.68s (**1.00**x)	2.45s (**1.56**x)
Workload: **simlarge** Threads: **4**										
Native	0.38s	2.04s	5.16s	3.40s	2.09s	1.28s	23.23s	3.05s	1.67s	1.56s
Simulation	0.39s (**1.03**x)	2.08s (**1.02**x)	5.76s (**1.12**x)	4.52s (**1.33**x)	2.17s (**1.04**x)	1.61s (**1.26**x)	24.76s (**1.07**x)	3.00s (**0.98**x)	1.64s (**0.98**x)	1.89s (**1.21**x)
Hardware	0.44s (**1.17**x)	3.32s (**1.63**x)	72.78s (**14.10**x)*	8.15s (**2.40**x)*	10.25s (**4.90**x)*	3.76s (**2.93**x)*	35.44s (**1.53**x)*	3.49s (**1.14**x)	1.71s (**1.03**x)	2.38s (**1.53**x)

* Test has working set size that causes EPC paging.

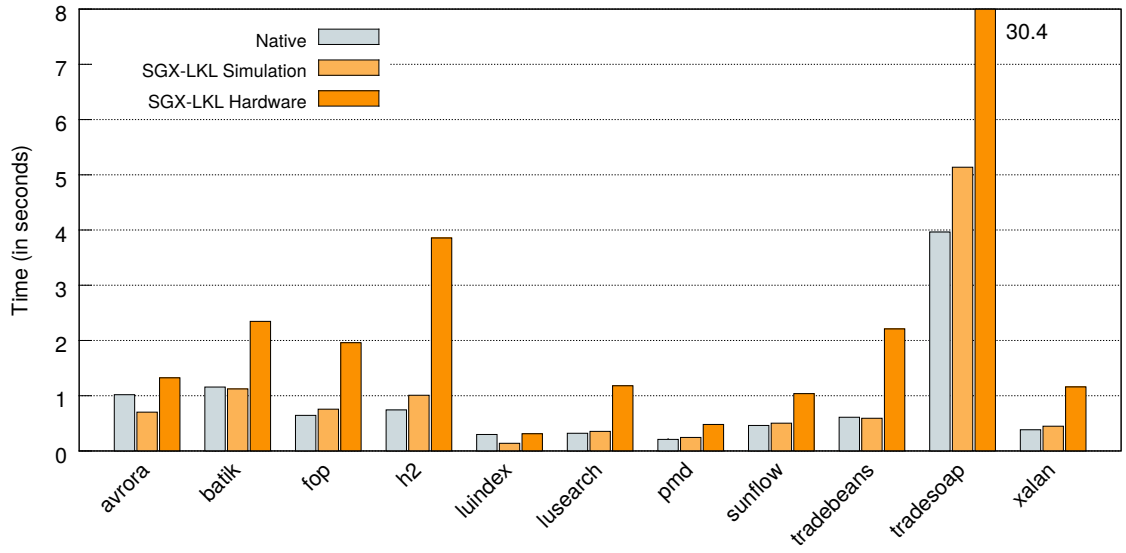


Fig. 3.5: Dacapo benchmark results

switches on system calls. SGX-LKL's use of cooperative scheduling also means that application threads are never interrupted if not required.

In hardware mode, tests that have a memory footprint that exceeds the EPC size and therefore cause EPC paging, show a significant overhead. For *simsmall*, SGX-LKL has runtimes of up to 3.37× the runtime of native execution (freqmine), while for tests without EPC paging, the largest relative overhead to native execution is 1.68× (bodytrack). With the *simlarge* workload, relative runtimes are between 1.53× (raytrace) and 14.10× (canneal) for tests requiring paging, and between 1.03× and 1.63× for tests without paging.

These results show that SGX-LKL scales with an increasing number of application threads with low overhead in both simulation and hardware mode. Performance with Intel SGX hardware decreases significantly when applications' memory footprints exceed the available EPC size. However, future EPC sizes will increase significantly [303] and allow low SGX-LKL overhead even for large working set sizes.

3.6.2.2 Java Virtual Machine

We evaluate SGX-LKL with an unmodified OpenJDK 8 JVM version 1.8.222 binary from the official Alpine Linux repository. The JVM is a language runtime with complex system support needs such as

dynamic library loading, custom signal handling, and access to special file systems such as `/proc`. To benchmark JVM performance, we use the Dacapo benchmark suite version 9.12 [31]. Dacapo is comprised of a diverse set of application benchmarks. These include an in-memory database benchmark (*h2*), Apache Lucene text indexing and search (*luindex* and *lusearch*), and an application server benchmark (*tradebeans*). We exclude the *eclipse*, *jython*, and *tomcat* tests as they assume fork support which SGX-LKL does not provide (see Section 3.8). Most of these benchmarks have working set sizes larger than the EPC size and require EPC paging. As the JVM has a large memory footprint itself, EPC paging is common for JVM applications and with the EPC limits of current CPUs.

Fig. 3.5 shows run times of the different Dacapo benchmarks when executed natively and with SGX-LKL in simulation and in hardware mode. In simulation mode, runtimes vary between 0.69× and 1.36× of native runtimes, and between 1.29× and 5.19× in hardware mode. As in the case of PARSEC, SGX-LKL performance in hardware mode largely depends on the working set size and memory access patterns.

With this benchmark, we demonstrated that SGX-LKL is capable of running a diverse set of complex JVM applications without requiring any changes to the application or the JVM itself. By supporting a widely adopted language runtime such as the JVM, SGX-LKL enables the secure execution of a large variety of existing applications that would require significant effort to port and protect as standalone TEE applications. We also demonstrated that SGX-LKL is able to run applications that heavily rely on system support with low overhead.

3.6.2.3 TensorFlow

By evaluating TensorFlow [1], a popular machine learning framework, we demonstrate SGX-LKL's ability to protect commonly sensitive workloads — both training data and trained models are considered sensitive in many use cases and should not be exposed. In addition, TensorFlow serves as another example of an application with complex system support requirements, e.g. thread-local storage and dynamic loading support. We also use TensorFlow's ability to distribute training and inference across a network of machines, to evaluate SGX-LKL's performance in a distributed setting.

We evaluate TensorFlow [1] training and inference using version 1.13.2 of TensorFlow and Python 3.6. We use the TensorFlow benchmark suite [329] with models representing different types of networks,

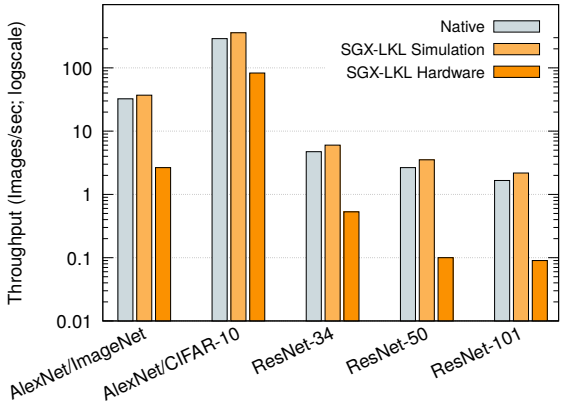


Fig. 3.6: Training throughput with TensorFlow

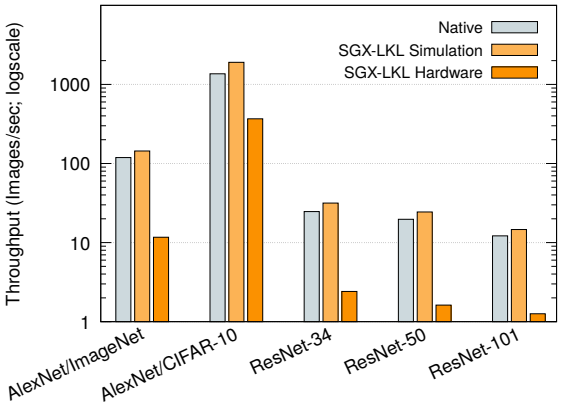


Fig. 3.7: Inference throughput with TensorFlow

both small (ResNet-34) and large (AlexNet, ResNet-101) as well as deep and low-dimensional (ResNet-50). The input datasets are CIFAR-10 [194] and a 230 MB subset of ImageNet [87, 328]. Both Python and TensorFlow have complex system requirements such as dynamic library loading — the TensorFlow benchmark depends on over 100 shared libraries, uses a large subset of the POSIX API, and requires support for thread-local storage.

In Fig. 3.6, we report the training throughput with SGX-LKL compared to native execution. SGX-LKL outperforms native execution in simulation mode: the training throughput of ResNet-34, ResNet-50 and ResNet-101 is higher than native by 28%, 33% and 32%, respectively. With ImageNet, the AlexNet outperforms the native system by 14.2% and 24.2% with CIFAR10 dataset. Fig. 3.7 shows the inference throughput. The results show the same trend as for training: the inference throughput is higher than in native execution by 22% for AlexNet with ImageNet, 42% for AlexNet with CIFAR10, 29% for ResNet-34, 26% for ResNet 50, and 20% for ResNet 101.

As these results demonstrate, SGX-LKL is capable of running TensorFlow training and inference without overhead in simulation mode. In fact, it achieves higher throughput than execution using Docker. As discussed in 3.6.2.1, this is likely due to SGX-LKL's use of user-level threading and cooperative scheduling.

However, TensorFlow has high memory requirements. To quantify this, we measure the working set sizes by counting the number of distinct pages that are paged into EPC memory over a 10 second interval.[4] The working set sizes range from 527 MB (ResNet-50/ImageNet) to 1495 MB

[4]Measured using the `sgxpagefaulttrace-bpf` tool available at https://github.com/cpriebe/sgxpagefaulttrace-bpf.

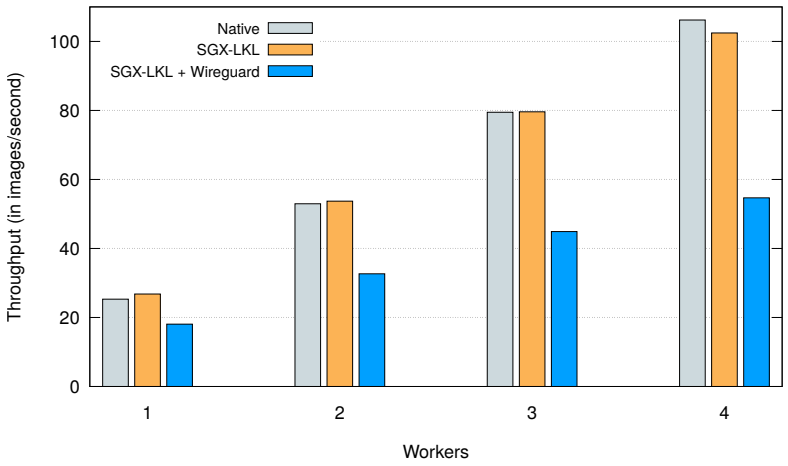


Fig. 3.8: Distributed TensorFlow training throughput for ImageNet/AlexNet with batch size 256 and a varying number of workers

(AlexNet/CIFAR10), and from 447 MB (ResNet-101/ImageNet) to 668 MB (AlexNet/CIFAR10) for training and inference, respectively. Note that TensorFlow accesses memory repeatedly so that pages are paged in and out of the EPC multiple times. Due to this, throughput in hardware modes degrades significantly. Training throughput is between 3.8% (Resnet-34) and 28.6% (AlexNet/CIFAR-10) of native throughput. Inference throughput ranges from 8.2% (Resnet-34) to 27% (AlexNet/CIFAR-10) of native throughput.

With this experiment, we again demonstrated SGX-LKL's ability to run complex applications and provide extensive system support. SGX-LKL is able to run TensorFlow training and inference throughput with no overhead in simulation mode. In addition, we showed the impact of large working set sizes on performance with SGX hardware, leading to a significant degradation in performance when working set sizes exceed the available EPC size.

3.6.2.4 Distributed TensorFlow

To evaluate performance for distributed applications, we also deploy TensorFlow with one parameter server and a varying number of workers, using a batch size of 256. With this experiment, we also demonstrate the benefit of SGX-LKL's transparent layer-3 network encryption. TensorFlow itself provides no TLS support, but its network traffic can be protected by SGX-LKL without modifications or configuration changes to TensorFlow.

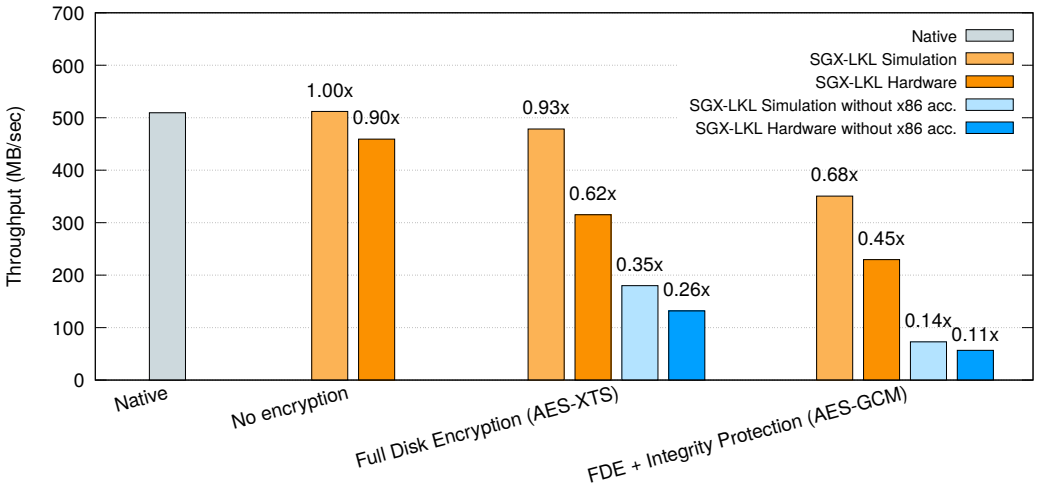


Fig. 3.9: Disk performance with encryption and integrity protection

Fig. 3.8 shows SGX-LKL's performance compared to native execution. We run SGX-LKL in simulation mode in two configurations: (i) without network encryption; and (ii) with network encryption. Without network encryption, SGX-LKL's throughput is comparable to native: with 4 workers, SGX-LKL's throughput is 3% slower. With the Wireguard VPN, the throughput decreases to 52% for 4 workers. As we explore in more detail in Section 3.6.3.2 the overhead is the result of encrypting and decrypting network packets.

With this experiment, we showed that SGX-LKL can protect complex distributed applications transparently with moderate overheads.

3.6.3 Microbenchmarks

In this section we perform microbenchmarks to evaluate SGX-LKL's overhead for disk and network I/O and their protection. The goal of these experiments is to (i) analyse in detail the origins of performance overheads in SGX-LKL; (ii) compare different configurations of SGX-LKL's protection mechanisms; and (iii) demonstrate the impact of optimisations we have made to improve I/O performance in SGX-LKL.

3.6.3.1 Disk I/O

We evaluate disk I/O performance by measuring the sequential read throughput when reading an uncached 1 GB file from a 256GB Samsung 850 Pro SSD. We consider three different configurations:

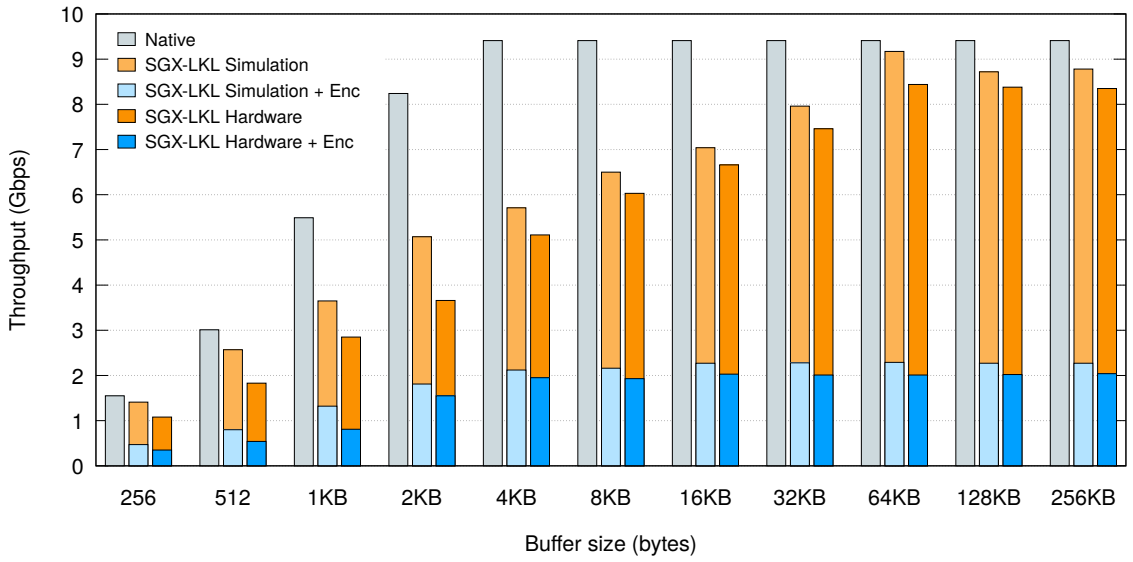


Fig. 3.10: Network I/O throughput measured with *IPerf3* for different buffer sizes

(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. All experiments are run in both simulation and hardware mode.

Fig. 3.9 shows that SGX-LKL achieves near native performance without encryption or integrity protection, fully saturating the bandwidth of the SSD of around 510 MB/sec. With FDE, the throughput decreases to 320MB/sec, around 62% of native throughput, in hardware mode. Enabling integrity protection further reduces throughput to around 230MB/sec or 45% of native throughput.

As described in Section 3.4.1 we extended LKL to support x86 kernel modules in order to make use of hardware acceleration, in particular for cryptographic operations. As shown, this optimisation improves encryption and integrity protection significantly. For FDE, throughput increases to 2.66× and 2.38× compared to the non-accelerated versions in simulation and in hardware mode, respectively, with hardware acceleration. For FDE with integrity protection, throughput increases to 4.86× and 4.09× the non-accelerated throughput in simulation and in hardware mode, respectively.

3.6.3.2 Network I/O

We evaluate network I/O performance by measuring network throughput for varying buffer sizes with *IPerf* version 3.1.3 [179]. We run SGX-LKL both in simulation and in hardware mode. In

addition, we measure SGX-LKL's network throughput with encryption, integrity protection, and authentication by enabling Wireguard. For all tests, we run an IPerf3 server with SGX-LKL inside an enclave and a native client on a separate machine, connected to the server host via a 10 Gbps Ethernet link.

As Fig. 3.10 shows, throughput increases with increasing buffer sizes for both the native execution and SGX-LKL. For 256 byte packets, SGX-LKL reaches about 0.7× and 0.9× of native throughput in hardware and simulation modes, respectively. For small buffers, the throughput is client-bound. Native execution saturates the full network bandwidth of about 9.41 Gbps with 4 KB buffers. In comparison, SGX-LKL reaches a throughput of 5.11 Gbps (0.54×) in hardware mode, and 5.71 Gbps (0.60×) in simulation mode.

In order to improve network performance, SGX-LKL supports both send and receive segmentation offloading. With this, larger buffers can be received and sent via the `net_read` and `net_send` host calls. Segmentation into smaller TCP segments and Ethernet packets is done on the host side, effectively reducing the number of required host calls. Therefore, SGX-LKL performs better for larger buffer sizes. For 64 KB buffers, SGX-LKL reaches a maximum throughput of around 8.38 Gbps (0.89×) and 8.78 Gbps (0.93×) for hardware and simulation mode, 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.

3.7 Related Work

There has been extensive research on system support for applications to make use of SGX capabilities. We already discussed other TEE runtime systems comparable to SGX-LKL in Section 3.2.1. In this section, we discuss other related work.

Data processing frameworks. Several other systems have been proposed that protect specific types of computations or applications rather than providing general-purpose protection. *Ryoan* [168] uses SGX to provide distributed sandboxes to run untrusted data-processing modules in order to allow mutually distrusting parties to process sensitive data. Ryoan modules are restricted to only process input once and must be stateless. They are confined by a Ryoan sandbox that (i) loads and validates

the module; (ii) provides system support via a custom libc implementation; and (iii) enforces privacy-preserving data flows between modules. The latter is enforced by attaching labels to module output data that identify the module owner. Data can have multiple labels if it passes through multiple modules. Ryoan enforces that a module cannot leak data with a different label, e.g. by persisting it on disk, or by sending it over the network to any party other than another Ryoan module or the trusted user. *Se-Lambda* [272], *Clemmys* [336], and *SecureV8/SecureDuk* [46] are similar systems that use SGX to protect small stateless modules in the context of *serverless computing* and *function-as-a-service* platforms. SGX enclaves have also been used to protect virtualised network functions in the context of middleboxes [80, 95, 134, 156, 261, 298, 335].

Another class of applications commonly deployed in the cloud to process sensitive data are data analytics frameworks. *VC3* [289] is a system that protects *Hadoop MapReduce* [86] jobs using SGX enclaves. Instead of running all of Hadoop inside a TEE, VC3 places only map and reduce functions inside enclaves to both reduce TCB size and overhead. It ensures that input and output data is encrypted and integrity-protected and guarantees that the distributed computation runs to completion without tampering. *Region self-integrity* of map and reduce tasks is enforced via a custom compiler. For this, memory accesses are instrumented to add range checks to ensure that all memory writes and reads are to or from in-enclave memory, respectively. Data can only enter and leave the enclave via specific framework interface functions that ensure its confidentiality and integrity. However, communication between the distributed enclaves can still reveal sensitive data to an adversary through analysis of access patterns, order of execution, and timings [90, 240]. M^2R [90] and similar systems [59, 240] therefore extend VC3's design to hide traffic patterns with additional shuffling and padding.

Opaque [378] is a secure distributed analytics framework built on top of Spark SQL that provides oblivious relational operators. Similar to previous systems it protects computation using SGX and hides network traffic patterns. In addition, it also makes memory accesses oblivious to prevent leakage via memory access patterns (see Section 2.4.3). Similarly, Ohrimenko et al. [241] and Shaon et al. [296] implement SGX-based frameworks for data-oblivious machine learning algorithms.

Lastly, the authors of *IRON* [112] and Felsen et al. [109] present SGX-based frameworks for functional encryption and secure and private function evaluation, respectively. By using TEEs, they overcome

the problem of program obfuscation (see Section 2.2.1.2) and improve performance over purely cryptographic implementations.

In contrast to these systems, SGX-LKL supports arbitrary applications without requiring application changes or recompilation.

TEE performance. As discussed in Section 2.4.2 Intel SGX can incur significant performance overheads due to the cost of enclave paging, enclave transitions, and memory encryption. To lower these costs, a number of systems have been proposed that provide in-enclave system support to mitigate SGX overheads.

FlexSC [313] first introduced the idea of exception-less asynchronous execution of system calls for improved system call performance. Instead of triggering a software interrupt by executing a `syscall` instruction to transfer execution to the kernel directly, a request is written to a designated system call page, and is asynchronously picked up and executed by an in-kernel system call. After completion, the result is written back to the system call page. This prevents costly userspace-kernel transitions and cache pollution. *SCONE* [15], *SGXKernel* [331], *HotCalls* [362], and *Eleos* [244] apply the same concept to *ocalls* to reduce the cost of enclave transitions. *Ocall* requests are written to a designated area in untrusted memory and picked up and performed by untrusted *ocall* threads running outside of the enclave. Eleos additionally uses Intel's *Cache Allocation Technology* (CAT) [171] to partition the CPU's last-level cache into enclave and *ecall* thread partitions in order to reduce enclave cache misses and the resulting MEE overhead (see Section 2.4.2.1).

Eleos and other systems also provide ways to mitigate overheads introduced by enclave paging (see Section 2.4.2.4). Eleos implements per-enclave user-level page tables and caches that allow for application-specific eviction policies. Enclave code can access the in-enclave page cache using *secure pointers* obtained via a custom `malloc` implementation. Memory accesses via these pointers can trigger page evictions and loads on page cache misses but avoid regular page faults that would trigger expensive asynchronous enclave exits and handling of the fault by the host OS. *CoSMIX* [245] implements a similar in-enclave *self-paging* mechanism but enforces its use via compiler instrumentation. *VAULT* [326] proposes hardware changes to use an optimised integrity verification structure to improve paging performance.

SGX-LKL employs similar mitigations, e.g. it performs host calls asynchronously and schedules other application threads while waiting for a host call to return. It also naturally reduces overhead by minimising reliance on the host OS and with that the number of enclave transitions. Other mitigations, e.g. to reduce EPC paging overheads, are orthogonal and could be used with SGX-LKL.

3.8 Limitations

In this section, we discuss limitations of SGX-LKL. In particular, we discuss SGX-LKL's missing support for multiprocess applications and the TCB size of applications that run on top of SGX-LKL.

Multiprocessing. In contrast to Graphene-SGX [341] and Panoply [301], SGX-LKL does not provide multiprocessing support. While there is no fundamental reason why SGX-LKL could not support multiple processes, currently available TEEs such as Intel SGX are not designed for multi-process applications. Intel SGX enclaves are bound to a single virtual address space. Multiple enclaves, residing in separate virtual address spaces, cannot share trusted enclave memory. Instead they have to rely on custom message passing mechanisms communicating via unprotected shared memory in their host processes. Common optimisations such as copy-on-write memory replication is not available in SGX enclaves. Forking involves the creation of a new enclave, local attestation of the enclave contents, establishing a secure connection, and the exchange of enclave secrets. While some multi-process applications can benefit from pre-forking, many rely on short-lived child processes and fast inter-process communication. Most applications are single-process or can be configured to use a single process and instead rely on multithreading for concurrency. For future TEEs with inherent multi-processing support or applications that strictly require support, e.g. databases relying on process forking for snapshotting, SGX-LKL could be extended similarly to Graphene-SGX or Panoply. For example, as done by Graphene-SGX, SGX-LKL could support forking by dynamically creating new enclaves for child processes and using shared untrusted memory to exchange encrypted messages to replicate process memory and to support inter-process communication.

TCB Size. The goal of SGX-LKL is to transparently protect complex data processing systems while exposing a minimal host interface. However, the choice of placing a complex library OS within the enclave to achieve this as well as the restrictions of Intel SGX as chosen TEE is based on trade-offs. The size of the TCB is an important security property. TEE runtime systems such as Panoply and

Table 3.5: Lines of code and binary sizes of *libsgxlkl.so* components

Component	Lines of Code (in thousands)[5]	Binary Size (in MiB)[6]
`LKL`	598*	5.28
`musl libc`	88	0.83
`libcryptsetup`	20	0.33
`libprotobuf-c`	13	0.06
`SGX-LKL runtime`	17	-
`libsgxlkl.so`	749*	6.98

* We approximate the lines of code for LKL by counting the lines of source files of all object files created during compilation. This does not include header files.

SCONE aim to reduce the TCB size by delegating work to the host OS but by doing so also expose a larger host interface.

Placing a comprehensive library OS inside the enclave naturally leads to a larger TCB. However, the configurability of Linux allows SGX-LKL to reduce its size significantly. Support for drivers as well as unneeded file systems and network protocols and other subsystems can be removed at compile time. Table 3.5 shows the estimated lines of code as well as the binary size of *libsgxlkl.so*'s components. The main components of *libsgxlkl.so* are LKL and the musl libc library. In addition, *libsgxlkl.so* includes: *libcryptsetup* which is used for the in-enclave configuration of device mapper targets for disk encryption and integrity protection; *libprotobuf-c* which provides *protocol buffer* support for SGX-LKL's remote control capabilities, including remote attestation and secret provisioning; and the SGX-LKL runtime which comprises the actual host interface and in-enclave system functionality not provided by LKL such as enclave memory management, threading, and signal handling.

While SGX-LKL's codebase is larger than systems such as SCONE [15] (187,000 lines) and Panoply [301] (10,000 lines) that delegate most of the low-level resource management to the host, SGX-LKL is similar in size to Graphene-SGX [341] (1.3M lines) while exposing a smaller host interface and providing full file system and network stacks inside the TEE. It is an order of magnitude smaller than Haven [23] (5.3M lines) which has a similar architecture but relies on a larger monolithic Windows-based library OS. In addition, with the Linux kernel, SGX-LKL relies on a mature and proven codebase. This reduces the risk of potential vulnerabilities. It is also actively maintained, making it simple to incorporate future fixes and features in SGX-LKL.

[5] As reported by `cloc`.
[6] As reported by `GNU size`.

Lastly, complex applications themselves have large codebases. The OpenJDK 8 and current TensorFlow codebases consist of about 500,000 and about 2,000,000 lines of code, respectively. We therefore believe that SGX-LKL's TCB size is a reasonable trade-off for its minimal host interface, in particular for large monolithic applications. Alternatively, application TCB sizes can be reduced through application partitioning, which we will explore in the next two chapters.

3.9 Summary

In this chapter we presented SGX-LKL, a TEE runtime system that is designed to support complex applications with a minimal host interface.

We first surveyed existing TEE runtime systems and analysed their host interfaces. We show that TEE host interfaces represent a substantial attack surface. A large and complex host interface inherently leaks information when TEE runtime systems rely on the untrusted host, e.g. by using the host's file system or network stack. Similarly, calls into the enclave pose a risk to the integrity of in-enclave data and computation. We therefore present a minimal host interface that takes the requirements of complex applications into account and only allows use of the host OS for functionality that cannot be provided inside the enclave.

We then presented SGX-LKL's implementation on top of this interface which executes unmodified Linux binaries within SGX enclaves by using the Linux kernel to provide POSIX abstractions. We discussed how SGX-LKL extends LKL to provide file system, networking, signal handling, threading, and timing functionality to applications.

We also showed how SGX-LKL protects the host interface. In particular, we discussed its use of Linux's device mapper subsystem to provide transparent encryption and integrity protection for disk I/O, and the use of Wireguard, a layer-3 VPN for transparent low-level network encryption and integrity protection.

We showed how SGX-LKL implements secure application deployments that maintain confidentiality and integrity of both application code and initial data. We presented the complete deployment workflow of SGX-LKL applications, including the creation of protected disk images, remote attestation, and the provisioning of runtime secrets such as disk encryption keys, public keys of Wireguard peers, and application arguments.

Finally, we evaluated SGX-LKL's ability to run a diverse set of applications and its performance. We show that SGX-LKL can run unmodified versions of the PARSEC application benchmark suite, the OpenJDK JVM, and TensorFlow, with low overheads when the working set size does not exceed the EPC size. For larger working sets, performance degrades significantly. However, this is a result of a limitation of current SGX implementations and will go away in the future with larger EPC sizes.

CHAPTER 4

GLAMDRING: AUTOMATED PARTITIONING FOR APPLICATION-SPECIFIC HOST INTERFACES

This chapter presents GLAMDRING, a semi-automated partitioning framework for Intel SGX. We discuss the benefits of application-specific host interfaces as well as objectives of application-specific host interface design. These include reducing TCB size, interface complexity, and performance overhead due to enclave transitions. We present a framework that partitions existing application codebases based on code annotations that identify security-sensitive data for the execution in TEEs, taking these objectives into account.

4.1 Introduction

With SGX-LKL, discussed in the previous chapter, we have explored one way of running existing applications in TEEs. SGX-LKL and other TEE runtime systems can run full unmodified applications inside enclaves to protect them from an adversary in control of the untrusted environment. Such an approach makes it easy to run complex applications in TEEs and can reduce the reliance on the host system. However, it also relies on a large TCB, increasing the risk of exploitable vulnerabilities within an enclave. The TCB not only includes the full application but also components to provide application-independent system support such as library OSes [23, 267, 341].

While executing full applications inside TEEs reduces the porting effort and simplifies deployment, it violates the *principle of least privilege* [284]. This states that any component should have the least privileges required to perform its functionality. When running complete applications inside an enclave, all application code executes at the same privilege level and has access to sensitive enclave data. However, as studies have shown there is a correlation between the TCB size and the number of vulnerabilities [226, 297]. Such vulnerabilities might be exploitable to compromise the confidentiality [169] or the integrity [73] of sensitive data. Therefore, to follow the principle of least privilege, only code that requires access to sensitive data should execute within an enclave

and applications should be partitioned accordingly. Partitioned applications consist of distinct compartments, trusted in-enclave code and data, and untrusted application code and data outside of the enclave.

SGX SDKs [114, 170, 239] provide support for the development of partitioned applications but are targeted at the development of new applications. They provide little support for the partitioning of existing applications, which can be challenging. Developers have to i) identify security-sensitive code, data structures and execution paths to be placed in the enclave; ii) refactor code into an enclave library and untrusted code; and iii) design a secure and efficient host interface for the two compartments to interact with each other.

While TEE runtime systems have a well-defined application-independent host interface, host interfaces for partitioned applications are *application-specific*. They implement function calls between untrusted and trusted application code outside and inside the enclave. Note that it is not sufficient to identify sensitive application code and place the code inside the enclave. The enclave relies on the outside code for some functionality and sensitive data must be exchanged with remote clients or passed to the host OS, e.g. for storage, securely. Therefore, the host interface must also be protected in order to guarantee the confidentiality and integrity of sensitive data that enters and leaves the enclave.

Furthermore, the *size of the TCB* is not the only concern for the partitioning of applications for TEE execution. In Chapter 3 we already discussed the role of the *host interface complexity* in regard to the security of a TEE application. Interface-based attacks (see Section 2.4.3.1) pose a substantial threat. Finally, the host interface can have a significant impact on application performance. On current SGX implementations, enclave transitions can take up to 10,000 CPU cycles (see Section 2.4.2.5) and cause additional performance overhead due to the inherent flushing of CPU caches. Therefore, the *number of enclave transitions* at runtime is another factor that must be taken into account.

However, reducing the TCB size, host interface complexity, and the number of transitions are competing *partitioning objectives*. For example, in a scenario in which a sensitive enclave function calls a non-sensitive helper function outside of the enclave, it can be beneficial to move the helper function into the enclave, thus slightly increasing the TCB size but reducing the interface complexity and the number of enclave transitions. This demonstrates how designing an application-specific host interface is non-trivial and that it poses a challenge for the manual partitioning of applications for TEE execution.

Instead, we argue for a principled approach for TEE application partitioning and application-specific host interface design on the basis of these objectives. We propose to use (i) *code analysis* to identify security-sensitive code and data, and to collect call graphs and other metrics; (ii) *code partitioning* to decide how to partition an application based on the analysis results and the discussed partitioning objectives; and (iii) *code generation* to transform application code into a trusted enclave library, untrusted code running outside of the enclave, and a hardened host interface for their interaction.

In this chapter, we present GLAMDRING, a semi-automated partitioning framework for TEE applications that implements this approach. It works in three phases:

(1) Code analysis. In the first phase, GLAMDRING employs code analysis to identify sensitive code and collects a number of metrics related to the partitioning objectives. For this, GLAMDRING relies on developer-provided code annotations. GLAMDRING supports two analysis approaches: *dynamic analysis* and *static analysis*. While dynamic analysis is done at runtime and therefore allows GLAMDRING to collect and take into account performance metrics, static analysis is workload-independent and is more conservative as it analyses the full codebase at once.

For dynamic analysis, the framework conducts *dynamic profiling* using Valgrind [235] while running a representative application workload. It obtains information about (i) which functions access sensitive data structures; (ii) the location at which sensitive data structures are allocated; (iii) which functions perform system calls; and (iv) the dynamic call sequence between functions.

For static analysis, GLAMDRING performs workload-independent (i) *dataflow analysis* [277] on annotated sensitive input variables to determine functions that access sensitive data or data derived from it; and (ii) *backward slicing* [358] on annotated sensitive output variables to determine functions that can affect the integrity of such outputs.

The dynamic and static code analysis produce annotated call trees and program dependence graphs, respectively, that encode the analysis results. Both approaches have benefits and shortcomings. We compare both approaches and discuss their trade-offs.

(2) Code partitioning. This phase solves an optimisation problem based on the analysis output and partitioning objectives, assigning a subset of all functions and data structures to the enclave. This exposes the trade-off between (i) the size of the TCB; (ii) the host interface complexity, i.e. the

number of enclave entry and exit points that must be protected against attacks; and (iii) the number of transitions in and out of the enclave, which impact performance.

GLAMDRING provides a *partitioning tool* that formulates this optimisation problem as an integer linear program (ILP) that incorporates the analysis results, and solves it with an ILP solver. The tool outputs a *partition specification* with (i) a list of (potentially duplicated) enclave functions; (ii) a list of functions that act as entry and exit points for the enclave; and (iii) all memory allocations that must be performed inside the enclave.

(3) Code generation and hardening. In this phase, GLAMDRING transforms the application code using a source-to-source compiler based on the LLVM/Clang compiler toolchain [75, 199]. It (i) splits up source files into enclave and outside code; (ii) generates code for the entry and exit points of the host interface; (iii) ensures that memory allocations for data structures are performed inside or outside of the enclave depending on the sensitivity of the data; and (iv) adds cryptographic operations and runtime checks to the host interface to protect it and to ensure invariants required for the soundness of the code analysis phase. The output of this phase is an untrusted binary and a trusted enclave library.

GLAMDRING's threat model follows the threat model described in Section 2.5. GLAMDRING aims to protect against a strong adversary in control of the untrusted system, including the whole software stack, with physical access to the machine. An adversary can observe all host interface interactions and might try to learn sensitive information based on enclave outputs. GLAMDRING must therefore protect the *confidentiality* of sensitive enclave data. Similarly, an adversary might use the host interface in unintended ways, e.g. by tampering with enclave input data, or calling into the enclave unexpectedly to impact the *integrity* of enclave code and data. GLAMDRING must protect the host interface against such attacks.

In contrast to TEE runtime systems such as SGX-LKL, GLAMDRING also aims to protect against a subset of enclave software vulnerabilities. By partitioning the application, GLAMDRING excludes software vulnerabilities in code that it places outside of the enclave from its TCB. However, we assume that code in the enclave library is free of vulnerabilities and can be trusted.

We implement a prototype of GLAMDRING and evaluate both its dynamic and its static analysis capabilities. We do this by applying it to the relational database management system (DBMS)

MySQL [232] and the key-value store *Memcached* [220]. We show that GLAMDRING can significantly reduce the TCB size by partitioning these applications and demonstrate its performance impact. Simulating larger enclaves, we estimate the overhead of MySQL partitioned by GLAMDRING to be between 11% and 23% compared to native execution. For Memcached, the throughput is reduced to about 27% of the native version when executed in an SGX enclave, comparable to other TEE-based versions.

The remainder of this chapter is structured as follows: Section 4.2 discusses objectives of TEE host interface designs and related work on alternatives to application-specific host interfaces; Section 4.3 introduces the high-level design of GLAMDRING; Section 4.4 covers the code analysis phase and discusses trade-offs between dynamic and static analysis for TEE application partitioning; Section 4.5 describes how GLAMDRING implements the code partitioning using ILP; Section 4.6 then discusses the code generation phase and how GLAMDRING hardens the interface; in Section 4.7, we report our experimental evaluation results; Section 4.8 presents related work; Section 4.9 discusses limitations; and Section 4.10 concludes this chapter.

4.2 TEE Host Interface Designs

This section provides background on TEE host interface designs and the benefits of application-specific host interfaces. We first look at objectives and trade-offs for TEE host interfaces. We then give an overview of the design space and compare alternative host interface designs in the context of these objectives.

4.2.1 Objectives

There are different ways of designing and securing TEE applications. With SGX-LKL, discussed in Chapter 3, we have presented one way of running TEE applications that focuses on a minimal host interface to reduce the attack surface of the secured application. However, reducing host interface complexity is just one potential objective. We identify three criteria that affect security or performance of TEE applications and impact the design of the host interface:

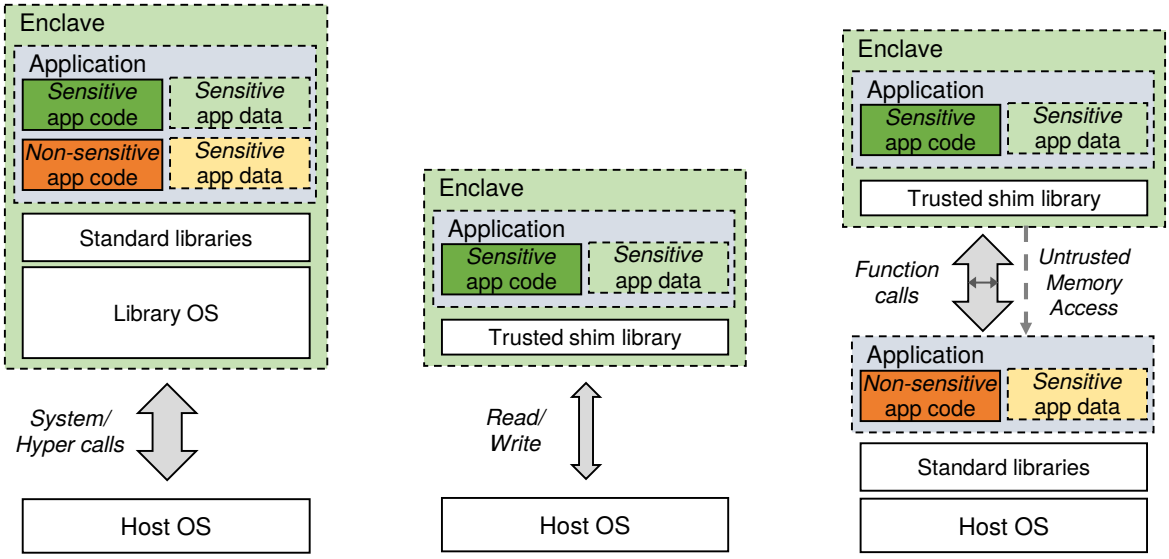
TCB size. An important factor is the TCB size. The TCB of TEE applications includes the TEE implementation, e.g. the Intel SGX hardware and microcode, as well as all code executed within the enclave. When application code contains vulnerabilities an adversary might be able to exploit them

to compromise confidentiality or integrity of the application. As there is a correlation between the amount of code and the likelihood of it containing vulnerabilities [226, 297], reducing the TCB size is an important factor in securing TEE applications. Following the principle of least privilege [284], only the parts of an application that require access to sensitive data should be executed within an enclave, with all other code executed outside.

Complexity of the host interface. The complexity of the host interface is another factor that impacts the security of enclave code and data. In the context of this chapter, we look at the host interface complexity in terms of the number of entry and exit points into and out of the enclave. As each of them interacts with untrusted code, they increase the *attack surface* of the enclave and may represent potential vulnerabilities — parameters passed out of the enclave must be sanitised correctly to ensure that the confidentiality of sensitive enclave data cannot be compromised; and parameters passed into the enclave must be verified to not compromise the integrity of enclave code and data. As we have discussed in Chapter 3, protecting the host interface from interface-based attacks such as Iago attacks [62] (see Section 2.4.3.1) can be difficult. It is therefore desirable to have a small interface between the enclave and the outside.

Number of enclave transitions. The third criteria is the number of enclave transitions at runtime. Each transition carries a performance overhead as the CPU switches between enclave and normal mode, resets and restores CPU state, and flushes CPU caches. To minimise performance overhead, the number of enclave transitions can be reduced, e.g. by pushing non-sensitive functionality into the enclave.

As the host interface forms the boundary between enclave and untrusted code, it is a central component of partitioned applications. When designing the host interface, the objectives above must be considered. However, optimising for these objectives can be difficult and requires application-specific knowledge. For example, in order to reduce the TCB size, application code can be moved out of the enclave. This requires knowledge about which application data is sensitive and which code accesses such data. Only code that does not require access to sensitive data can be moved. Similarly, reducing the number of enclave transitions, e.g. by moving a function that is frequently called across the enclave boundary, requires knowledge of the runtime behaviour of the application and executed code paths.



(a) Application-independent host interface

(b) Predefined host interface

(c) Application-specific host interface

Fig. 4.1: Design space for TEE host interfaces

In addition, these partitioning objectives are not orthogonal but compete with each other. Optimising for one objective can negatively affect another objective. Moving a function out of the enclave in order to reduce the TCB size increases host interface complexity as an additional *ocall* is introduced. It might also affect performance by requiring additional enclave transitions. GLAMDRING takes these trade-offs into account and allows framework users to prioritise one objective over another. We discuss this in more detail in Section 4.5.

4.2.2 Design space

Next we explore the design space for TEE host interface designs and discuss the trade-offs with respect to the above objectives as well as *development effort* and the *generality* of the approach. Both the development effort and generality are important factors for the adoption of a given approach to secure TEE applications.

Fig. 4.1 shows three design alternatives for protecting applications using enclaves with different types of host interfaces:

Application-independent host interface. As shown in Fig. 4.1a, the approach adopted by systems such as SGX-LKL (Chapter 3), Haven [23], SCONE [15] and Graphene-SGX [341] provides isolation

at a coarse granularity by executing a *complete* application inside an enclave. Both security-sensitive and insensitive application code and data reside within the enclave, increasing the TCB size.

The host interface supports a complete set of system or hyper calls that cannot be handled inside the enclave and are therefore delegated to the host OS, e.g. for I/O operations. The interface is *application-independent*, but its complexity (in terms of number of distinct entry and exit points) depends on the adopted system abstraction. The required generic system support within the enclave further adds to the TCB size.

This approach incurs low development effort, as it can execute unmodified applications, and is generic across applications. However, by placing both a complete application and complex system support code, e.g. library OSes, inside the enclave, it also leads to large TCBs.

Predefined host interface. Fig. 4.1b shows an approach in which applications must adhere to a *predefined* restricted host interface [288, 305, 306]. For example, VC3 [288] protects map/reduce jobs using enclaves and forces map/reduce tasks to interact with the untrusted environment only through a particular interface. Similarly, a number of other systems provide runtime support for small modules or functions, e.g. in the context of *serverless computing*, but require them to be stateless and adhere to a predefined interface [46, 168, 272, 336]. The enclave contains a small trusted shim library, resulting in a smaller TCB compared to the previous approach.

This approach results in a small host interface which simplifies protection. VC3's interface consists of only two calls, one to read encrypted key/value pairs and another to write them as the job output. It also makes it possible to add additional runtime checks that enforce security invariants [306], e.g. to prevent enclave code from directly accessing untrusted memory.

However, the security benefits of this approach are offset by its limited applicability. Given the predefined host interface, the approach can only be used with applications that interact with the untrusted environment in specific ways, such as map/reduce tasks in the case of VC3.

Application-specific host interface. With GLAMDRING, we explore another approach. 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 [47, 304, 310]. As shown in Fig. 4.1c, 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.

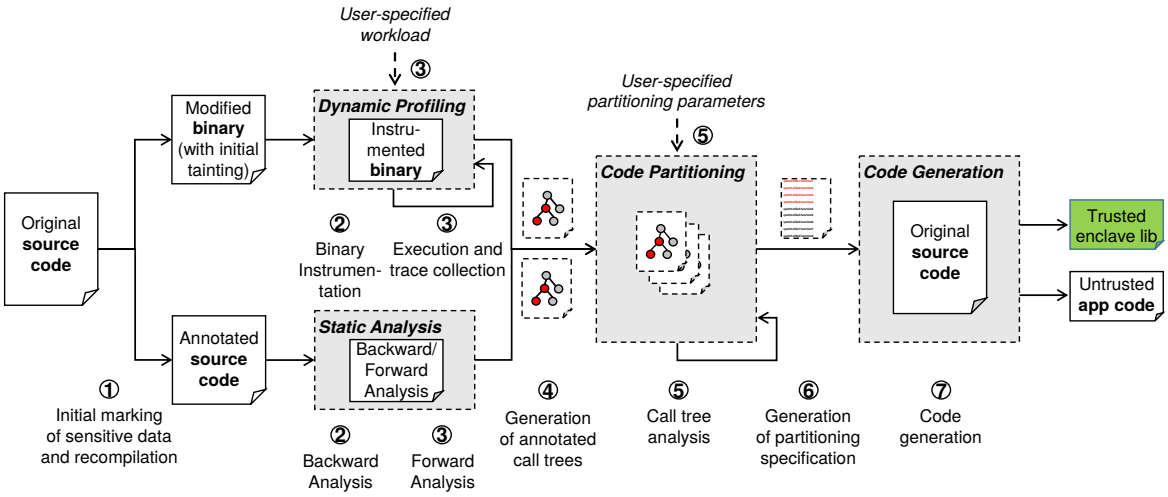


Fig. 4.2: GLAMDRING architecture with both design options: dynamic and static analysis

Past work has shown that partitioning for TEE execution can be done by hand for some applications [47, 259]. Instead, we want to explore the hypothesis that it is feasible to use *principled* techniques, such as code analysis, to partition applications for secure enclaves, and provide *security guarantees* about the enclave code and its interface to the untrusted environment.

With this approach, the host interface becomes *application-specific*: a set of *ecalls* and *ocalls* is required between trusted and untrusted application code. In contrast to a complete application-independent host interface, the host interface only consists of calls needed for the execution of a specific application. In addition, fewer *ocalls* might be required to invoke host OS functionality, as application code that is placed outside the enclave can issue system calls directly.

Since some application data now resides outside of the enclave, enclave code must be allowed to access untrusted memory. This means that it is no longer possible to prohibit all untrusted memory accesses, as with the predefined host interface [306]. Instead, it is important to give security guarantees that, despite the richer application-specific host interface, sensitive enclave data cannot be disclosed to the untrusted environment and that its integrity cannot be compromised.

4.3 GLAMDRING Design

In this section, we present GLAMDRING, a framework for protecting existing applications by executing security-sensitive code in an Intel SGX enclave. To achieve this, GLAMDRING partitions application code and data into a trusted enclave component and an untrusted component executed outside.

GLAMDRING targets the following requirements: (i) it must protect the *confidentiality* and *integrity* of input and output data; (ii) automate and/or reduce required modifications to application code; and (iii) must follow the partitioning objectives of a small TCB, low host interface complexity, and few enclave transitions outlined in Section 4.2.1.

To achieve this, GLAMDRING operates in three phases, as illustrated in Fig. 4.2.

(1) Code analysis. GLAMDRING must know which application data is sensitive. It requires the developer to provide initial information about the *sources* (inputs) and *sinks* (outputs) of security-sensitive data by annotating variables whose values must be protected. Based on these annotations, GLAMDRING performs either dynamic or static analysis to track data flows and program dependencies to identify a subset of code and data that is security-sensitive. GLAMDRING supports both, dynamic and static analysis, as both approaches have different strengths (Section 4.4).

(2) Code partitioning Next, GLAMDRING creates a *partition specification* (PS) that defines which parts of the code should be protected by the enclave. The PS enumerates the functions, memory allocations and global variables that must be part of the enclave library. It is created based on annotated call trees and program dependence graphs obtained from the code analysis. GLAMDRING implements partitioning as an ILP problem with an objective function that encodes the three partitioning objectives: (i) a small TCB size; (ii) low host interface complexity; and (iii) few enclave transitions. As these are competing objectives, GLAMDRING allows the objectives to be weighted by the framework user. By solving the ILP optimisation problem, GLAMDRING decides on a PS that is optimal in terms of the user-specified objectives (Section 4.5).

(3) Code generation. Finally, GLAMDRING uses a source-to-source compiler that, based on the PS, partitions the code into a trusted enclave library and untrusted code. It also hardens the host interface by adding runtime checks that enforce invariants on the program state and by adding cryptographic transformations that ensure the confidentiality and integrity of input and output data of host calls (Section 4.6).

In the following sections, we describe each phase in detail.

4.4 Code Analysis

The goal of the code analysis phase is to identify the subset of application code and data that is security-sensitive and therefore must be protected. However, the sensitivity of data depends on application semantics. GLAMDRING therefore requires developers to provide code annotations for the initial marking of security-sensitive inputs and outputs. Based on these annotations, the code analysis finds dependent code and data that must be equally treated as sensitive. For example, code that is conditionally executed based on the value of a sensitive input, but otherwise does not process the sensitive data, must still be treated as sensitive. This is because subsequent interaction with the untrusted host could allow an adversary to infer the sensitive input value. Similarly, data derived from sensitive data must be treated as sensitive as well.

As a consequence, GLAMDRING by default treats all code as untrusted and all data as non-sensitive. Only code and data that is marked as sensitive as part of the code analysis is guaranteed to be placed inside the enclave. This also means that the correctness of GLAMDRING's output is dependent on the completeness and correctness of the developer annotations.

The code analysis can be performed *dynamically* or *statically*. Dynamic analysis is done at runtime and can therefore be used to collect information such as call trees and other *performance*-based metrics. However, dynamic analysis results are workload-dependent, i.e. they depend on the application inputs at the time of the analysis. In contrast, static analysis is workload-independent and hence more conservative when determining code and data dependencies. Therefore, static analysis can be considered more *secure* as dynamic analysis because it can also cover unexpected application workloads and non-deterministic application behaviour. However, as static analysis is performed on source code, it does not consider runtime behaviour which is possible with dynamic analysis.

We implement both of these methodologies for the code analysis with GLAMDRING and discuss them in the following sections:

(1) Dynamic analysis. First, we present a Valgrind-based [235] *taint tracking* and *profiling* tool for dynamic analysis. It tracks the propagation of sensitive data throughout application code at runtime. By running the application with this tool under a representative workload and tracking the propagation of sensitive data, GLAMDRING can identify security-sensitive application code and data. The tool also keeps track of memory allocations, performed system calls, and collects profiling data.

It produces annotated call trees that capture the collected information and that serve as input for the code partitioning phase (Section 4.4.1).

(2) Static analysis. Second, we use static analysis to determine security-sensitive application components from developer-provided source code annotations. GLAMDRING uses *dataflow analysis* to identify code paths and data that depend on sensitive data and should be treated as security-sensitive itself. For sensitive outputs, it performs *backward slicing* to identify code and data that can affect the data's integrity. The static analysis produces annotated program dependence graphs used as an input of the code partitioning phase, similar to the annotated call trees resulting from the dynamic analysis (Section 4.4.2).

In Section 4.4.3 we compare and discuss the benefits and limitations of both approaches. We also discuss how dynamic and static analysis could be combined to improve application partitioning for TEEs.

4.4.1 Dynamic Analysis

Using *dynamic analysis* allows GLAMDRING to collect runtime information such as call sequences and the number of times functions have been executed. This information can later be used to decide on a partitioning that not only considers sensitive data flows but in addition takes into account *performance* as one of the partitioning goals. Dynamic analysis can also be used when dealing with applications that rely on dynamic control flow features such as function pointers or virtual method invocations that are difficult to handle using static analysis. To analyse how sensitive data propagates within the application, we use *dynamic taint tracking* [237] to identify application code and data that need protection.

We develop a *profiling tool* using the *Valgrind* dynamic binary rewriting framework [235]. Valgrind supports the instrumentation of arbitrary binaries to generate dynamic call graphs and monitor memory accesses. The profiling tool collects information on executed function calls, their accesses to sensitive data, and performed system calls. Due to SGX enclaves being unprivileged, system calls must be performed outside and therefore must be taken into account as part of the partitioning. To later enforce access restrictions on sensitive data, the tool also tracks the allocation and usage of data structures by recording their allocation and where they are accessed.

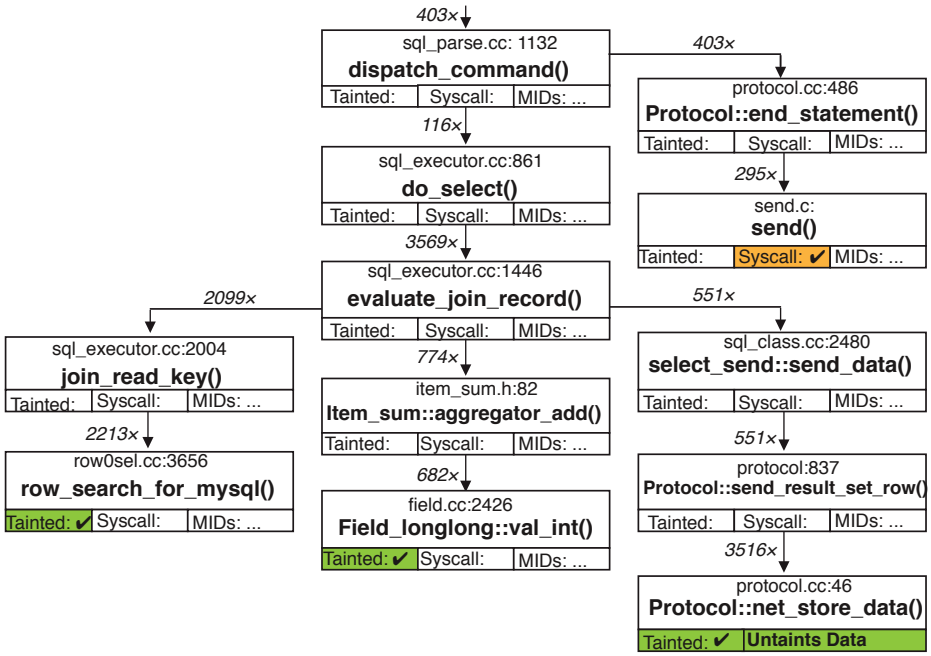


Fig. 4.3: Annotated call tree for MySQL, produced by GLAMDRING's profiling tool

The tool represents the collected information as per-thread *annotated call trees*. Fig. 4.3 shows an example of an annotated call tree for MySQL [232]. We use MySQL as an example application that benefits from GLAMDRING's dynamic analysis as it is a complex, high-performant and data-intensive application that uses function pointers and C++ virtual methods extensively. Each node in a tree represents an executed function and stores the following information: (i) the name and source code location of the function; (ii) the source code location that it was invoked from; (iii) a list of performed system calls; (iv) a list of allocated memory blocks; (v) a list of accessed memory blocks; and (vi) a flag specifying if the function accessed sensitive data.

For compactness, single nodes may represent multiple invocations if the called function and the location of the call are the same. In this case, the edge that connects the caller to the callee is annotated with the number of calls.

Profiling workload. Since we use a dynamic approach, the obtained partitioning depends on the application workload during the profiling phase. The profiling workload must therefore be representative of the application workloads encountered in production. While this can be challenging for some applications, it is possible for others. For example, for MySQL a comprehensive SQL query set can

be used for profiling. For the following discussion, we choose a workload based on the TPC-E [333] benchmark that results in good code path coverage for analytics queries.

To account for non-deterministic application behaviour, the profiling tool supports multiple runs with different workloads. All the obtained annotated call trees are combined in subsequent phases of the partitioning framework to obtain a consistent partitioning.

Taint tracking. To track the propagation of sensitive data, the profiling tool first requires a manual annotation of sensitive data. We employ Valgrind's *client request* facility to allow framework users to specify memory regions that contain sensitive data. For MySQL, we mark all buffer pool pages with user table data as sensitive data when they are read from disk.

The profiling tool then uses Valgrind's shadow memory and registers to perform byte-level taint tracking of the sensitive data. Each time the data is read from or written to memory or a CPU register, the corresponding instruction is instrumented. When sensitive data is copied or new data is derived from existing tainted data, the affected memory locations are tainted as well.

Similar to the initial tainting, memory is untainted via manual annotations to indicate that sensitive data has been sanitised, e.g. by encrypting it. Sanitised data is safe to be handled by outside code. For MySQL, we can untaint data in the function `Protocol::net_store_data` (see Fig. 4.3), which prepares a query result before transmission over the network.

Memory management. Data structures containing tainted data must be allocated inside the enclave to be only accessible by enclave code. The profiling tool therefore collects information about the lifecycle of such data structures by tracking memory allocations and deallocations. The profiling tool identifies memory blocks by intercepting calls to `malloc` and related `libc` functions. Each allocated memory block is assigned a unique 64-bit identifier, and each function accessing the block is recorded.

Some applications, such as MySQL, implement their own memory management. For example, MySQL's default *InnoDB* storage engine preallocates the memory for a buffer pool, which is then allocated at a finer granularity to store individual buffer pool pages. The memory for buffer pages is also reused. Since the partitioning tool is by default unaware of this fine-grained allocation and reuse, the entire buffer pool would be considered tainted if it contained a single tainted buffer page. To address this issue, the profiling tool provides additional Valgrind client requests to annotate `malloc`-

and `free`-like functions such as the ones used in the MySQL codebase. This allows the profiling tool to track memory used by each buffer pool page separately.

Once the application codebase has been annotated using Valgrind client requests in order to taint and untaint sensitive data and to instrument memory allocations and deallocations in case of custom memory management, it is recompiled. The recompiled binary is then run as usual using Valgrind with GLAMDRING's profiling tool. The tool then produces the annotated call trees that contain the results of the dynamic code analysis phase and that are used as input of the following partitioning phase.

4.4.2 Static Analysis

GLAMDRING also supports static code analysis. While dynamic analysis has the benefit of being able to analyse runtime behaviour and incorporate performance characteristics, the correctness of the analysis results depends on the choice of a representative workload. In contrast, static analysis can be applied to the full codebase of an application at once and is therefore workload-independent. It is more conservative in regard to decisions on code dependencies, e.g. when determining possible targets of dynamic control flow targets via function pointers. While this might result in false positives and cause additional functions to be included in the TCB, it also ensures that no dependencies are missed and sensitive code or data is wrongly placed outside of the enclave.

Similar to the dynamic analysis, GLAMDRING also requires source code annotations for the static analysis, as the sensitivity of data is application-dependent. 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 must be available for processing, i.e. in unencrypted form. Similarly, the annotation `sensitive-sink` indicates a variable at which security-sensitive data leaves the application.

Listing 4.1 shows examples of both annotations for the key-value store Memcached. We choose Memcached as an example for the static analysis as it is data-intensive and is written in C and thus does not rely on extensive use of dynamic control flow features common in C++ applications, e.g. via virtual methods. We consider the command submitted by the client, e.g. `get` or `set`, and the associated

```
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 4.1: Annotation for GLAMDRING's static analysis of Memcached's source code

key/value data to be security-sensitive. GLAMDRING assumes that this data is encrypted and signed by the trusted client when sent to the application.

Using GLAMDRING, Memcached requires two annotations. We add a `sensitive-source` annotation in line 1 to mark the content of the parameter `command` as security-sensitive. The parameter holds both the command as well as the associated data and should therefore be encrypted up to this point. Processing within this function requires the data to be decrypted so that the `process_command` function and other code processing the data must be treated as security-sensitive. The `sensitive-sink` annotation for the `buf` parameter of the `add_iov` function in line 12 specifies that the output buffer for the client response also contains security-sensitive data.

These annotated statements form an initial set S_A of *security-sensitive* statements. The static code analysis then extends S_A by (i) the set of all statements that are influenced by the ones in S_A, for *confidentiality*; and (ii) the set of all statements that influence the ones in S_A, for *integrity*. GLAMDRING's analysis uses a *program dependence graph* (PDG) [111] P in which vertices represent statements, and edges represent both data and control dependencies between statements. PDGs are effective representations of code and data dependencies for the use during program slicing [163, 247]. Using P, GLAMDRING finds the set of all security-sensitive statements as follows:

Static dataflow analysis. To determine statements that are relevant for data *confidentiality*, we use dataflow analysis. More specifically, 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.

GLAMDRING encrypts and signs data before statements in S_A that are annotated as a `sensitive-sink` using the shared AES-GCM key. This makes it unnecessary to perform dataflow analysis from these statements onwards.

Static backward slicing. To determine statements that can impact the *integrity* of security-sensitive data, GLAMDRING uses static backward slicing. Given S_A and P, backward slicing finds 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.

GLAMDRING adds decryption and checks the signature of security-sensitive data after statements with the `sensitive-source` annotation in S_A. Up to that point, the integrity of data cannot be tampered with without detection when the signature is checked. Thus, backward slicing does not need to be performed further from these statements. For this, GLAMDRING requires client-side encryption and signing of the data.

Finally, the set of all security-sensitive statements S_s is obtained by combining P_c and P_i. The analysis is done on the level of statements, but GLAMDRING partitions applications at a function granularity. We therefore consider function dependencies. A function f_1 depends on another function f_2 if there exists a statement within f_1 that depends on a statement within f_2. Similarly to the annotated call tress of the dynamic analysis phase, the resulting PDG subgraph can then be provided as an input to the following partitioning phase.

4.4.3 Discussion: Dynamic and Static Analysis

In the previous sections we have presented two approaches to implement the code analysis phase of GLAMDRING: dynamic and static analysis. Both have different strengths and limitations:

Analysis of runtime behaviour. Only dynamic analysis allows GLAMDRING to analyse runtime behaviour and therefore incorporate performance metrics in the decision making on the optimal application partitioning. GLAMDRING's partitioning tool collects information about the frequency of function calls which GLAMDRING can use to decide on a partitioning that reduces the number of enclave transitions. Static analysis is based on the application source code and does not consider

runtime behaviour. We discuss how GLAMDRING takes the analysis results into account in more detail in Section 4.5.

Workload dependence. As discussed, GLAMDRING's dynamic analysis relies on a user-defined workload and is therefore *workload-dependent*. The analysis results depend on application inputs. Furthermore, applications might behave non-deterministically. GLAMDRING mitigates this problem by supporting the merging of results from multiple profiling runs. However, it is still up to the framework user to use representative workloads which can be difficult to predict depending on the application. For example, in the case of MySQL, the optimiser will be restricted to plans consisting of steps observed during dynamic analysis. Similarly, requiring an operator not observed during analysis will result in the program to abort, as we will discuss in Section 4.6.2. In contrast, static analysis is *workload-independent*. It takes any possible execution into account. This makes it more conservative than dynamic analysis, i.e. it might determine more functions to be security-sensitive, but guarantees the coverage of the full codebase.

Integrity protection. GLAMDRING's dynamic analysis uses taint tracking to track the propagation of sensitive data from a *source* until it is untainted at a *sink*, e.g. after being encrypted. Analogously, GLAMDRING's static dataflow analysis tracks control/data dependencies to determine the flow of sensitive data. In both cases, the goal is to determine application code and other data structures that depend on the data initially annotated as security-sensitive. By protecting the dependent code and data by assigning them to the enclave, GLAMDRING can ensures the data's *confidentiality*.

However, GLAMDRING's static analysis allows for stronger *integrity* protection in contrast to its dynamic analysis approach. Using backward slicing [358], it can identify dependencies of data annotated as security-sensitive *sink*, including data and code that on its own is non-sensitive. While approaches for *dynamic program slicing* for dependencies of individual variables exist [4], full dynamic backward slicing in the context of GLAMDRING would require it to keep track of all program state and data flows throughout execution, which is not feasible.

This is in particular relevant for program inputs and *ecall* parameters that have not been annotated as security-sensitive but that can indirectly affect security-sensitive data. For annotated data, GLAMDRING guarantees integrity in both cases using authenticated encryption.

As discussed here, both dynamic and static analysis have strengths and shortcomings. While our prototype implementation (see Section 4.7) only supports the use of either dynamic or static analysis, a hybrid approach is possible. In such an approach, the conservative workload-independent static analysis results could be augmented by workload-specific performance metrics obtained using dynamic analysis. The combined results can then be used as input for the following partitioning phase which we describe in the next section.

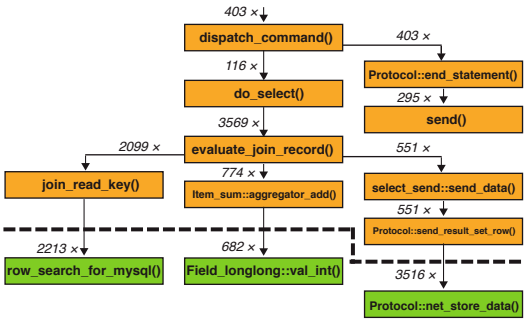
4.5 Code Partitioning

The goal of the *partitioning phase* is to produce a *partition specification* (PS) that can be used to partition the application's codebase. This is done based on the output of the code analysis phase. For the dynamic analysis, the output consists of a set of annotated call trees produced by the profiling tool. Similarly, GLAMDRING's static analysis produces an annotated PDG which encodes which functions are considered security-sensitive. Here, we describe the partitioning phase on the basis of dynamic analysis results as they contain additional performance metrics. However, the same methodology applies when using GLAMDRING's static analysis.

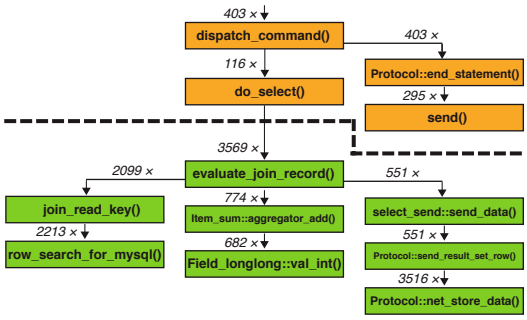
We implement a *partitioning tool* that, based on the code analysis results, finds a *valid* partitioning, i.e. a partitioning that does not violate the following SGX constraints: (i) all functions that access sensitive data or any field of a data structure with sensitive data must be assigned to the enclave; and (ii) all functions that perform system calls are executed outside of the enclave. Since many valid partitionings exist, the tool also considers the three objectives described in Section 4.2.1:

TCB size. To reduce the TCB size, ideally only functions that must be executed within the enclave due to accessing sensitive data would be assigned to the enclave partition, with all other functions executed outside. For example, in the call tree in Fig. 4.3, only the `val_int`, `net_store_data`, and `row_search_for_mysql` functions access sensitive data. Fig. 4.4a shows a valid partitioning that is optimised for a small TCB size and only assigns these functions to the enclave.

Interface complexity. Optimising for a small TCB size, however, can result in many entry and exit points for the enclave and thus increase the host interface's complexity. The partitioning from Fig. 4.4a has three unique entry points into the enclave that all interact with untrusted code. Fig. 4.4b shows a partitioning with only a single entry point into the enclave, but a larger TCB. This illustrates



(a) Smaller TCB size with more enclave transitions



(b) Larger TCB size with fewer enclave transitions

Fig. 4.4: Examples of two valid partitionings, with enclave functions shown at the bottom in green

the trade-off between a small TCB size and few entry/exit points: moving the entry point to the function `evaluate_join_record` reduces the entry points from 3 to 1, but adds 5 functions to the enclave.

Number of transitions. To reduce performance overhead due to frequent enclave transitions, additional functions, such as `evaluate_join_record` and all its children, may be added to the enclave library. For example, while the partitioning in Fig. 4.4a has 6,411 transitions for the TPC-E workload we use, the one in Fig. 4.4b only has 3,569 transitions. Moving the enclave entry point higher in the call hierarchy, however, enlarges the TCB.

Optimisation problem. The partitioning tool generates a partitioning that is both valid and takes the above objectives into account via user-specified weights. For this, it formulates the optimisation problem as an integer linear programming (ILP) problem and solves it using an ILP solver [151]. The transformation from the analysis output into an ILP problem is done as follows:

Each node of the annotated call tree, representing an invoked function, is represented as a binary decision variable $N_{f,i}$, where 1 denotes an assignment to the enclave and 0 to the outside. It is defined as:

$$N_{f,i} = \begin{cases} 0, & \text{if } (f,i) \text{ performed a system call} \\ 1, & \text{if } (f,i) \text{ accessed sensitive data} \\ \textit{undefined}, & \text{otherwise} \end{cases} \tag{4.1}$$

where f identifies the function, and i is an index to distinguish between nodes representing different invocations of the same function. Function nodes that represent invocations in which the function accesses sensitive data are set to 1; nodes that represent invocations with system calls are set to 0. The values of all other variables are determined by the solver.

Multiple nodes representing the same function can be assigned to both partitions. By doing so, a function is *duplicated*, i.e. it exists both inside the enclave and outside. Duplicating functions is useful to reduce transitions because both enclave and outside code can call their respective versions. This commonly happens for utility functions, e.g. for string manipulation, that are used at many locations in the codebase.

For every call tree edge, a variable $E_{(f,i),(g,j)}$ exists that determines if the function invocation represented by $N_{g,j}$ from another function $N_{f,i}$ requires an enclave transition. It has the value 0 if both connected nodes are assigned to the same partition, and 1 otherwise:

$$E_{(f,i),(g,j)} = \begin{cases} 0, & \text{if } N_{f,i} \text{ and } N_{g,j} \text{ are equal} \\ 1, & \text{otherwise} \end{cases} \tag{4.2}$$

To express the above relationship between the edge and node variables, we add corresponding constraints to the ILP problem. We also define additional constraints necessary for a valid partitioning, e.g. to ensure that the same invocation of one function by another always either represents the same kind of transition or no transition.

To construct an objective function, we require a number of additional variables. ${E^*}_{(f,i),(g,j)}$ exists for each edge variable $E_{(f,i),(g,j)}$ and represents a weighted edge. It can be used to record the total number of enclave transitions: it is 0 if an edge does not represent a transition, or the number of calls, otherwise:

$${E^*}_{(f,i),(g,j)} = c_{(f,i),(g,j)} E_{(f,i),(g,j)} \tag{4.3}$$

with $c_{(f,i),(g,j)}$ being the number of calls.

To calculate the number of unique entry and exit points, we create a variable T_f for each function f that is 1 if the function is the target of at least one transition into or out of the enclave, and 0 otherwise:

$$T_f = \begin{cases} 1, & \text{if at least one edge that targets } f \\ & \text{represents an enclave transition} \\ 0, & \text{otherwise} \end{cases} \tag{4.4}$$

Similarly, for determining the TCB size, we create a variable F_f that is 1 if at least one node corresponding to function f is assigned to the enclave, and 0 otherwise:

$$F_f = \begin{cases} 1, & \text{if at least one node representing } f \text{ is} \\ & \text{assigned to the enclave partition} \\ 0, & \text{otherwise} \end{cases} \tag{4.5}$$

For every function variable F_f, a weighted function variable ${F^*}_f$ exists, that takes into account the weight of a function, e.g. by setting it to the number of lines of code it consists of. ${F^*}_f$ is defined as follows:

$${F^*}_f = w_f F_f \tag{4.6}$$

with w_f being the function's LOC.

For the definition of the objective function, we define three sets: (i) B is the set of all weighted function variables F^*_f, representing the TCB size; (ii) U is the set of all target variables T_f, representing the number of unique entry and exit points; and (iii) V is the set of all weighted edge variables ${E^*}_{(f,i),(g,j)}$, representing the total number of transitions.

The partitioning tool then finds a solution that minimises the following objective function O:

$$O = \alpha \sum_{b \in B} b + \beta \sum_{u \in U} u + \gamma \sum_{v \in V} v \tag{4.7}$$

The parameters α, β and γ denote weights for the partitioning objectives: A high value for α puts more emphasis on a small TCB size; increasing β increases the impact of entry and exit points; and γ emphasises the objective of minimising the number of transitions.

Based on the assigned decision variables, the partitioning tool generates the PS for the code generation phase. It contains (i) the set of functions that are to be assigned to the enclave; (ii) the enclave entry and exit points that must be implemented via *ecalls* and *ocalls*; and (iii) the set of security-sensitive memory allocations determined during the code analysis phase.

4.6 Code Generation and Hardening

The last phase is the *code generation phase*. It produces a source-level partitioning of the application based on the PS. It also *hardens* the host interface to ensure confidentiality and integrity of sensitive data and to protect it against interface-based attacks (see Section 2.4.3.1). The result is a set of enclave and outside source files, along with an enclave specification to compile an enclave library using the Intel SGX SDK.

4.6.1 Code Transformation

The application code must be transformed based on the PS in order to (i) handle calls into and out of the enclave; (ii) handle memory allocations for sensitive and non-sensitive code in the generated enclave and non-enclave code; and (iii) protect the confidentiality and integrity of sensitive data through interface hardening.

For this, GLAMDRING provides a *code generator* that uses the LLVM/Clang compiler toolchain [75, 199] to rewrite preprocessed C/C++ application source code. It uses Clang to parse source code into an abstract syntax tree (AST), and then traverses the AST in multiple passes to analyse and modify the source code, as follows:

(i) Creating the enclave and outside code. For each source file, the code generator creates an enclave and an outside version, which contain a copy of the original preprocessed input file. From the enclave version, it removes all functions and methods that are not listed in the PS as being either enclave or duplicate functions. There are a few exceptions of code that cannot be removed by the

code generator, e.g. C++ constructors from class definitions or definitions of pure virtual functions in derived classes.

(ii) Generating *ecalls* and *ocalls*. Based on enclave transitions specified by the PS, the code generator creates a corresponding host interface. For calls from outside code to enclave functions, it creates a corresponding *ecall*; for enclave code calling outside functions a corresponding *ocall*. The interface is defined in the *enclave definition language* (EDL) used by the Intel SGX SDK [170].

Calls to C/C++ library functions are handled separately. A subset is supported by the Intel SDK inside the enclave and is handled in a polymorphic manner, so that the enclave and untrusted code call their respective versions. For unsupported library functions, the code generator creates *ocalls* to the corresponding library function linked to the outside code. However, these *ocalls* might violate the validity of the partitioning. If parameters to these *ocalls* are considered security-sensitive, manual modifications might be required to either encrypt and integrity-protect the data before the *ocall*, or to provide a corresponding in-enclave implementation.

(iii) Handling memory allocations. The PS includes a list of sensitive memory allocations that must use enclave memory. This is the default behaviour for the `malloc` implementation inside the enclave. For non-sensitive memory allocations, the code generator must replace `malloc` calls with a corresponding *ocall* to allocate memory outside of the enclave so that it can be accessed by both enclave and outside code.

4.6.2 Code Hardening

By partitioning the application, GLAMDRING creates an application-specific host interface. This interface presents an attack surface that must be hardened to ensure the confidentiality and integrity of sensitive application data. As part of the code generation phase, GLAMDRING therefore uses a number of techniques to protect the interface.

Encryption and integrity protection of sensitive data. GLAMDRING requires that security-sensitive data sent and received via the host interface to be encrypted and integrity-protected via a shared AES-GCM [100] key. It assumes that remote clients are modified to support AES-GCM encryption and integrity verification and that the shared key is established upon enclave creation. Similarly, sensitive data that leaves the enclave for interaction with untrusted application code and the host OS, e.g. for

storage, must be encrypted using AES-GCM and decrypted and integrity-checked upon retrieval. The code generator adds code at locations that have been annotated to (i) decrypt and authenticate security-sensitive data where data is tainted or marked as `sensitive-source`; and (ii) encrypt security-sensitive data where data is untainted or marked as `sensitive-sink`.

Runtime invariant checks. The security of the partitioning produced by GLAMDRING is based on a number of invariants established during the code analysis phase.

For the dynamic analysis, GLAMDRING assumes that the workload chosen to perform the dynamic analysis covers all code expected to be executed at runtime. Only functions executed during analysis can be assigned to the enclave. Functions that are not called are treated as if they would not exist. However, code that never executed during analysis, e.g. code of which the execution is conditional, could still end up in an enclave as part of partially analysed functions. This code could access sensitive data and potentially expose it to the outside. An adversary might be able to trigger its execution by choosing specific *ecall* parameter or *ocall* return values. To prevent such an attack, GLAMDRING records executed basic blocks during the profiling phase, and replaces all basic blocks not observed with an `abort`.

Similarly, GLAMDRING's static analysis infers invariants that must be uphold in order to guarantee the confidentiality and integrity of sensitive application data. 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, and places it inside the enclave (see Section 4.4.2). Therefore, an adversary is not able to affect integrity and confidentiality by tampering with host interface parameters or return values. However, static analysis infers invariants about the possible values of program variables in order to exclude infeasible program paths from analysis. The soundness of the static analysis therefore depends on these invariants. To ensure that they cannot be violated, GLAMDRING enforces the invariants assumed by the static analysis at runtime. For this, it extracts invariants from the analysis phase and adds them as runtime checks. GLAMDRING applies checks on both global variables and on parameters passed into and out of *ecalls* and *ocalls*.

Handling pointers. Both *function pointers* as well as *regular pointers* to data in memory might be under the control of an adversary, either because they are passed across the host interface, or because they are part of data structures in untrusted memory. Pointers are a risk to enclave security as they

could be exploited to hijack enclave control flow or to trick an enclave to expose sensitive data, e.g. by changing a pointer to enclave memory to one that points to untrusted memory. By swapping two pointers to enclave memory, an adversary could also impact the integrity of sensitive data.

To deal with potential tampering with *function pointers*, GLAMDRING must know every possible target of the function pointer that is passed to enclave code. It employs static function pointer analysis [376] to identify the possible target functions of function pointer arguments passed to *ecalls* and *ocalls*. Based on the set of possible targets for function pointers, the code generator amends the corresponding *ecalls* and *ocalls* to only accept function pointers to functions in the predefined set of possible targets.

For *regular pointers*, the static analysis infers for each pointer the subset of `malloc` calls that may allocate memory the pointer could reference. GLAMDRING distinguishes between two cases: (a) the analysis infers that a pointer may only point to *untrusted* memory. In this case, a runtime check upholds this and any other invariants on pointer aliasing; or (b) 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.

Enclave code vulnerabilities. The main goal of partitioning TEE applications is to reduce the size of its TCB and with that the risk of exploitable vulnerabilities. Bugs in untrusted application code cannot compromise the security of sensitive data as the data is protected by the enclave. However, enclave code may still contain vulnerabilities. Traditional vulnerability types such as buffer overflows [155, 198], data races [184], and memory leaks [197] are well-studied and mitigations, e.g. through the use of memory-safe languages, exist. While important, these are orthogonal to our work and can be used in conjunction. The same applies for TEE-specific attacks that exploit enclave software vulnerabilities and corresponding mitigations which we discussed in Section 2.4.3.3.

4.7 Evaluation

In this section, we evaluate GLAMDRING by applying it to two applications: (i) the MySQL database, using dynamic analysis; and (ii) the Memcached key-value store, using static analysis. We describe how we annotate their source code, and discuss the resulting partitionings and host interfaces. We also show how user-defined weights for the partitioning objectives affect the TCB sizes, host interface complexity, and performance of the generated partitioning. Finally, we evaluate the performance of the generated applications. We show that applications partitioned using GLAMDRING have a moderate performance overhead compared to executing the same applications natively and executing them using alternative TEE runtimes.

GLAMDRING implementation. We implement a prototype of GLAMDRING consisting of a set of tools that implement the previously described phases. For the dynamic code analysis phase, the taint tracking and profiling tool is implemented as a Valgrind tool and is based on the existing callgrind profiling tool [357]. For the static analysis phase, GLAMDRING uses the Frama-C Aluminium [83, 117] framework, with the "Impact Analysis" [115] and "Slicing" [116] plug-ins. The partitioning phase is implemented using the PuLP LP modeler [269] to model the ILP problem and uses Gurobi [151] as ILP solver. GLAMDRING's code generator is implemented on top of LLVM/Clang 3.9.

4.7.1 Dynamic Analysis

In this section, we evaluate GLAMDRING using its dynamic analysis profiling tool to partition MySQL for the execution of read-only analytics queries. We choose MySQL to demonstrate GLAMDRING's ability to partition complex applications, such as relational DBMS. We also use MySQL to demonstrate the impact of the user-specified weights for the partitioning objectives. In the following, we first discuss the partitioning of MySQL and then evaluate the performance of a set of partitionings generated by GLAMDRING. We choose a read-only analytics workload to demonstrate the dynamic analysis' workload dependence and its effects on the partitioning. It also simplifies code annotation, partitioning and interface hardening for our prototype implementation as no interaction is required with the host OS to persist updates. We discuss corresponding challenges and limitations in Section 4.9.

4.7.1.1 Partitioning

We first give a brief overview of MySQL's architecture and discuss the security objectives, including what data we aim to protect. We then perform the partitioning based on corresponding annotations, and present the partitioning results.

MySQL Architecture. MySQL is a popular relational DBMS. It executes queries as follows: each client connection is handled by a dedicated thread that is created by a *connection manager*. A SQL query submitted by a client is first parsed and checked by the *query parser* before it is rewritten by the *query optimiser* from a logical to a physical representation. The physical query is then executed by the *query executor*, which accesses data via a *storage engine*. MySQL supports multiple storage engines for the storage and retrieval of data. After being loaded into memory from disk, database tables managed by a given storage engine are represented as fixed-size *buffer pages*, stored in a pre-allocated *buffer pool*. The executor's query operator implementations operate on table data from the buffer pool pages. The query result is then constructed by the executor and returned to the client.

Partitioning. In this experiment, we aim to protect all user table data. We therefore mark all buffer pool pages with user table data as security-sensitive when they are loaded from disk. For this, we add the Valgrind client request to taint the data when it is read into the buffer pool in the MySQL function `buf_read_page_low`. While being processed, the data should be protected. Only when the query results are complete, the results can be encrypted and passed to the untrusted host. We therefore untaint data in the function `Protocol::net_store_data`, which prepares a query result before transmission over the network.

We then run GLAMDRING's profiling tool. We choose a workload based on the TPC-E benchmark [333], which simulates a typical *online transaction processing* (OLTP) scenario. Since we focus on querying existing sensitive data, we use 6 types of transactions that do not perform updates. We use the TPC workload generator to create a database for 1,000 customers with 300 days of trading.

4.7.1.2 Impact of Partitioning Objectives

First, we evaluate the impact of the user-specified weighting of partitioning objectives. Based on the code analysis results, we use the partitioning tool to generate 5 partitionings with different trade-offs in terms of the TCB size, the number of entry and exit points, and the number of transitions (see

Table 4.1: Properties of GLAMDRING MySQL partitionings (The total number of executed functions is 7579, corresponding to about 189K LOC)

Partitioning	Enclave size		Entry/Exit points	Transitions
	Functions	LOC (1000)		
P(−−|++|+)	3595	103	28	1632
P(−|+|∘)	2794	78	36	2620
P(∘|+|−)	2286	64	47	6878
P(+|+|−)	2209	60	61	9033
P(+|∘|−)	2111	57	78	10685

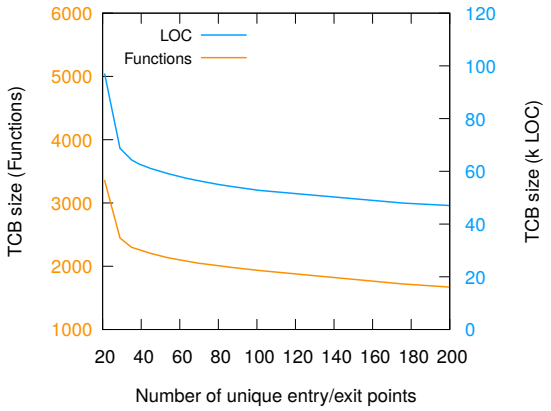


Fig. 4.5: Impact of enclave entry/exit point weight

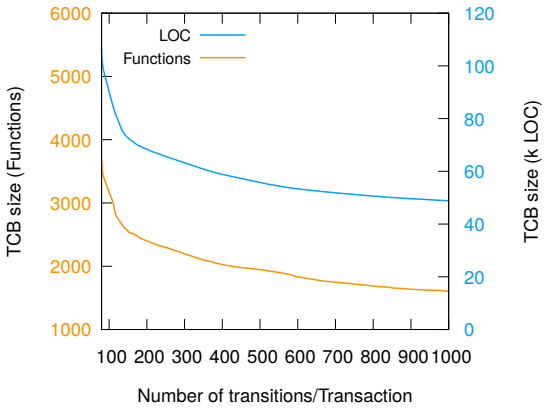


Fig. 4.6: Impact of enclave transitions weight

Table 4.1). They are represented by the parameters α, β and γ, as described in Section 4.5. P($\alpha|\beta|\gamma$) refers to the chosen partitioning weights, with $\{+, -, \circ\}$ indicating a high, low and default weight, respectively. As shown, there are clear trade-offs between the different objectives. P(−−|++|+) has the largest TCB but also the smallest host interface and the lowest number of transitions. In contrast, P(+|∘|−) has the smallest TCB but the most complex interface and the largest number of transitions.

Impact of enclave entry/exit points. To further illustrate the relationship between the objectives, we explore the space of generated partitionings for a wider range of weights. Fig. 4.5 shows the number of entry/exit points in relation to the TCB size, in terms of the number of enclave functions as well as the number of lines of code. This demonstrates the trade-off between optimising for TCB size and few entry/exit points – choosing a partitioning that emphasises the objective of a small host interface leads to a large TCB and vice versa. A small increase in the number of entry/exit points can have a big impact on the TCB: a partitioning with only 21 entry/exit points requires over 3,300 functions to

Table 4.2: Comparison of TCB sizes with alternative approaches (LOC for *Partitioned MySQL* is what is executed under the TPC-E benchmark)

Type	LOC	Functions
MySQL	1,700,000	-
Partitioned MySQL	189,000	7,579
Enclave	78,000	2,794
Untrusted	111,000	4,785
SQL Server + Haven [23]	>5,600,000	-
TrustedDB/SQLite [19, 315]	112,000	-
Cipherbase [13]	(Verilog) 6,000	-

be executed within the enclave, but increasing the host interface to 29 functions reduces the TCB by almost 900 functions (or 30K LOC).

Impact of enclave transitions. Fig. 4.6 shows the relationship between the TCB size and the number of transitions per executed transaction. It follows the same pattern of the relationship between the TCB size and the number of entry/exit points: a smaller number of transitions requires a larger TCB since more functions are assigned to the enclave for execution to continue without transitions. Going from a partitioning with 78 executed host calls per transaction to one with 133 transitions, reduces the TCB by 1200 functions (or 39K LOC).

TCB size. We evaluate the TCB size further by looking at one of the partitionings from Table 4.1 in detail. Table 4.2 shows the TCB size of the partitioning $\mathsf{P}(-|+|\circ)$ in comparison to that of other approaches. This partitioning places 42% of the executed LOC in the TCB, which is only 5% of the overall MySQL codebase. In total, 189K LOC are executed for the TPC-E-based workload. Out of this, only 78K LOC are placed inside the enclave and 111K LOC execute outside, with 6,200 LOC being duplicated and present in both parts. This represents an interface of 21 *ecall* and 15 *ocall* functions.

Note that the user table data we annotated as sensitive makes up a large part of data processed by MySQL and therefore a large fraction of all data is expected to be identified as sensitive. However, as these results show, the majority of code does not require direct access to unencrypted user table data. This demonstrates that even for data-intensive applications in which the majority of data is considered sensitive, a partitioning can result in significant reductions of the TCB size.

We also compare the TCB to other approaches. The TCB is significantly smaller than that of a library OS based approach. For example, Haven [23] uses a library OS [264] with 5.6 million LOC on its own

to run an unmodified version of SQL Server in an enclave. TrustedDB [19] has a TCB approximately the size of the SQLite [315] core library (112K LOC), which is similar to the partitioning presented here, including untrusted code. The FPGA implementation in Verilog used by Cipherbase [13] has a significantly smaller TCB of 6,000 LOC, but incurs the overhead of data movement to and from a secure FPGA, and required significant manual implementation effort.

It should be noted that this partitioning of MySQL is bound to the TPC-E workload and equivalent queries. While this restricts the use of the partitioning, it also allows for a small TCB by excluding code that is not needed for the given workload. This is in contrast to a partitioning based on static analysis which would lead to a larger TCB due to being workload-independent and therefore more conservative.

4.7.1.3 Performance

In this section we evaluate the performance of the different partitionings presented in Table 4.1.

Experimental setup. We deploy the partitioned versions of MySQL version 5.6 on a server with an Intel Xeon E5-4620 CPU (2.2 GHz) with 8 hyper-threaded cores and 64 GB of 1333 MHz DRAM. We use a default MySQL configuration with an increased buffer pool size to fit all data in memory, ensuring that performance is not disk-bound. The TPC-E client executes on a separate machine.

For this experiment, we emulate the overhead of SGX rather than running on SGX hardware for three reasons: (i) at the time of the development of GLAMDRING's dynamic analysis mode and when conducting this experiment, the official Intel SGX SDK was not yet available; (ii) EPC sizes of available Intel SGX CPUs are too small to fit the TPC-E workload in memory; and (iii) GLAMDRING's prototype is limited in its support for passing C++ object references across the host interface. While references themselves can be passed in and out, the prototype does not support methods of the same object to be called inside and outside of the enclave. This would require a duplication of each object's *virtual method table* (vtable) in order to support polymorphic method calls of the corresponding implementations inside or outside of the enclave and additional hardening to ensure that vtables cannot be tampered with by untrusted code.

Instead, we focus on evaluating the performance impact of different partitionings. To emulate the SGX overhead, we (i) add the cost of enclave transitions to all *ecalls* and *ocalls*; and (ii) add a TLB

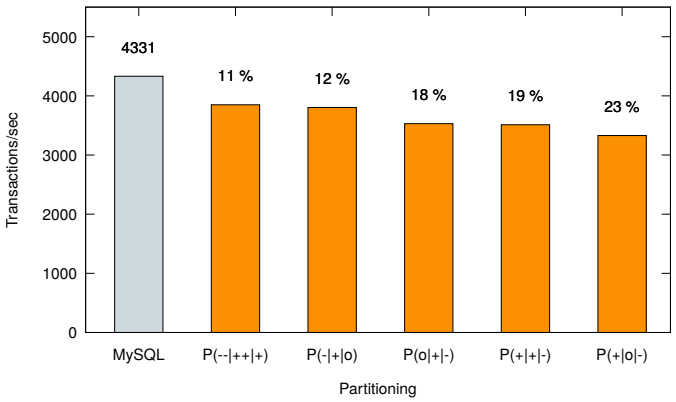


Fig. 4.7: Performance for different partitionings

flush to all *ecalls* and *ocalls* as SGX requires a TLB flush on all enclave transitions. Note that this is an approximation of the expected overhead. It does not take into account memory encryption and paging overheads. Chapter 5 will introduce a more advanced performance model for emulating SGX overheads of future CPUs with large EPC sizes.

Results. We evaluate the performance overhead using the TPC-E-based workload compared to an unmodified MySQL version. Our performance results omit the potential impact of the code hardening approaches (see Section 4.6.2) to focus on the trade-offs enabled by GLAMDRING. We perform measurements over 30 minutes after a 10 minute warm-up phase. In all cases, the CPU is fully utilised.

Fig. 4.7 shows how the transaction throughput varies across the partitionings due to their different numbers of enclave transitions. There is a clear trade-off between the TCB size and the performance: while $\mathsf{P}(--|++|+)$ has a large TCB size with over half of the executed functions, it also incurs the lowest overhead due to fewer enclave transitions; in comparison, $\mathsf{P}(\circ|+|-)$ has an overhead of 23% but only 2,111 (28%) executed functions in the TCB.

Note that the relationship between TCB size and performance is not linear: while the reduction of the TCB by more than 1,000 functions, between $\mathsf{P}(--|++|+)$ and $\mathsf{P}(\circ|+|-)$, increases overhead by only 7%, a further reduction by 175 functions, between $\mathsf{P}(\circ|+|-)$ and $\mathsf{P}(+|\circ|-)$, adds an additional overhead of 5%.

4.7.2 Static Analysis

Next, we evaluate GLAMDRING's static analysis. For this, we partition the key-value store Memcached. It is data-intensive and written in C, which makes it a good fit for the static analysis. As in the previous section, we first describe the partitioning and then evaluate its performance.

4.7.2.1 Partitioning

In order to partition Memcached, we identify security-sensitive sources and sinks within its codebase that require code annotations to guide GLAMDRING's static analysis (see Section 4.4.2).

Memcached Architecture. Memcached is a multi-threaded in-memory key-value store written in C. It uses the *libevent* [268] library to handle incoming connections and requests from remote clients. It implements both binary- and text-based protocols to receive commands to query and update in-memory key-value pairs, specify expiration times for cache invalidation, and to retrieve statistics. Incoming network packets containing Memcached requests are received by libevent's event loop and delegated to a Memcached worker thread for processing. Once processed, a response is sent to the client.

Partitioning. We aim to protect the integrity and confidentiality of all key-value pairs, preventing an attacker from reading or modifying the stored data. We also consider the command, e.g. `get` or `set`, to be sensitive. We assume the command along with the associated key-value data is encrypted and integrity-protected by the client. Based on this, we add a `sensitive-source` annotation for the `command` argument, which encapsulates both the command and data, of the `process_command` function, as previously discussed in Section 4.4. We also add a `sensitive-sink` annotation to the `add_iov` function.

We then run GLAMDRING to create a valid partitioning of Memcached version 1.4.25 with a focus on a small TCB. However, rather than opting for a minimal TCB partitioning, we choose a partitioning that places an additional non-sensitive function inside the enclave in order to reduce the number of enclave transitions.

TCB size. Table 4.3 shows a break down of the partitioned version of Memcached. GLAMDRING places 40% of LOC and 42% of functions inside the enclave, either because they are security-sensitive or because placing them inside reduces interface complexity and/or the number of enclave transitions

Table 4.3: TCB size, host interface, and enclave transitions for Memcached version 1.4.25 partitioned with GLAMDRING

	LOC	Functions	Enclave LOC	Enclave functions	Ecalls	Ocalls
Memcached	31,100	655	12,474 (**40%**)	273 (**42%**)	41	146
Memcached v1.4.25	13,800	247		215 (**87%**)		
libevent v1.4.14	17,300	408		57 (**14%**)		

at runtime. This includes: (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; and (iii) the hash functions over key/value pairs. Other code that is non-sensitive is placed outside of the enclave. This includes: (i) thread initialisation and registration functions; (ii) libevent functions for socket polling and network I/O; and (iii) signal handlers.

As these results show, GLAMDRING determines a large fraction of the Memcached core codebase (excluding dependencies) to be security-sensitive. 87% of its functions are assigned to the enclave. However, most of the libevent codebase, which makes up a significant part of the non-partitioned Memcached TCB, is non-sensitive. Only 14% of all libevent functions are placed inside the enclave while the majority is placed outside of the enclave.

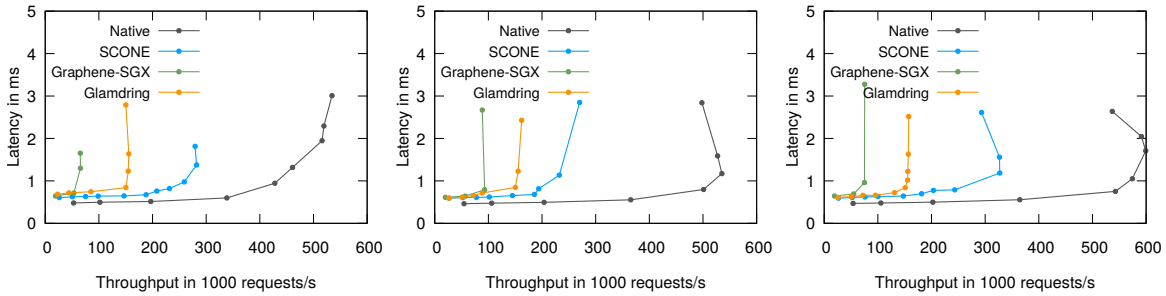
The Memcached host interface (see Table 4.3) has 41 *ecalls* and 146 *ocalls*. Out of these, 82 *ocalls* are to C library functions unavailable inside the enclave, and 64 *ocalls* are to application functions. However, most calls are related to initialisation code and other infrequently used functionality. The processing of common Memcached requests requires only few calls and transitions. For example, both `get` and `set` commands require 1 *ecall* and 2 *ocalls* per request on average.

4.7.2.2 Performance

Next, we evaluate the performance of the partitioned version of Memcached.

Experimental setup. All experiments are executed on a 4-core Intel Xeon E3-1280 v5 at 3.70 GHz with SGX version 1 support, 64 GB of RAM, running Ubuntu 14.04 LTS with Linux kernel 3.19 and the Intel SGX SDK 1.7. We disable hyperthreading for the experiments.

We evaluate Memcached with the YCSB benchmark [77]. Clients run on separate machines connected via a 10 Gigabit network link. We increase the number of clients until the server is saturated.



(a) Read-only workload **(b)** Write-only workload **(c)** Read/write workload

Fig. 4.8: Throughput versus latency for Memcached native, with SCONE, with Graphene-SGX, and with GLAMDRING

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.

Results. We measure the throughput and latency for Memcached partitioned by GLAMDRING. We compare it to a native execution of Memcached, as well as the execution with two alternative SGX-based approaches, SCONE [15] and Graphene-SGX [341]. We increase the request rate incrementally and consider three workloads: read-only, write-only and 50%/50% read/write.

Fig. 4.8 shows that all three variants exhibit consistent behaviour across the workloads. GLAMDRING reaches a maximum throughput of 160k requests/s; SCONE achieves between 270k and 330k requests/s; Graphene-SGX achieves between 65k and 95k requests/s; and the native execution of Memcached achieves around 530k and 600k requests/sec.

The reason for GLAMDRING's lower throughput compared to SCONE is that SCONE avoids all enclave transitions. By executing the whole application inside the enclave it trades off TCB size for performance (see Section 4.2.2). It also supports an asynchronous host call mechanism that allows it to make host calls for I/O without the cost of leaving and re-entering the enclave using a shared in-memory queue for host calls. Graphene-SGX executes both Memcached as well as a library OS inside the enclave and thus has an even larger TCB than SCONE. Compared to Graphene-SGX, GLAMDRING performs better than Graphene-SGX in all experiments, with a throughput between 1.75× and 2.5× of Graphene-SGX.

4.8 Related Work

We discussed alternatives to application partitioning for the execution in TEEs in Section 4.2.2 and in Chapter 3, which focused on TEE runtime systems for the trusted execution of complete applications. In this section, we discuss other related work on application partitioning. In particular, we discuss previous research on application partitioning for privilege separation and the partitioning of applications for the execution in TEEs.

Privilege separation. In order to reduce an application's attack surface, the principle of least privilege [284] demands that application components are run with the minimal privilege level required to perform their functionality. This confines exploits to the same or lower privilege levels but protects functionality and data with higher sensitivity. A number of systems exist that follow the principle to separate application components based on required privileges [30, 50, 123, 148, 209, 280, 366].

PrivTrans [50] splits an application into a small privileged monitor and an unprivileged slave component using static analysis. The monitor and the slave component are then executed in separate processes and leverage the OS's process isolation for privilege separation. However, PrivTrans only performs dataflow analysis and does not consider the integrity of sensitive data.

ProgramCutter [366] and Wedge [30] also split applications into least privilege components, but rely exclusively on dynamic runtime analysis to identify security-sensitive application parts. SeCage [209] combines static and dynamic analysis to partition applications based on user-annotated secrets. At runtime, the isolation is enforced using CPU virtualisation features. In contrast to these approaches, GLAMDRING does not rely on a trusted OS or hypervisor and takes the unique requirements of TEEs into account.

SOAAP [148] helps developers to reason about the potential compartmentalisation of applications based on source annotations and static analysis. However, unlike GLAMDRING, it does not support automated code partitioning but requires the developer to implement the partitioning based on the analysis results and potential requirements of the targeted isolation approach, e.g. process isolation or TEEs.

Rubinov et al. [280] propose a partitioning framework for the execution of Android applications in TEEs. It refactors the source code and adds a set of privileged instructions for the execution on Arm

TrustZone. However, it only supports typesafe Java applications and requires users to reimplement the security-sensitive functionality in C.

TEE applications. We previously discussed general frameworks for creating new TEE applications, e.g. in the context of serverless computing [46, 168, 272, 336], data analytics [59, 241, 289, 296, 378], and middleboxes [80, 95, 134, 156, 261, 298, 335] in Section 3.7.

Other work explores the manual partitioning of specific applications for the execution in TEEs. *SecureKeeper* [47] is a partitioned version of *ZooKeeper* [167], a distributed coordination service written in Java. The authors use a set of enclaves to perform TLS termination of client connections and to hold a shared encryption key used to encrypt all data managed by ZooKeeper. They also implement `increment` and `decrement` operations for stored counters. All other operations are performed on encrypted data within the remaining Java codebase that is considered untrusted and runs outside of enclaves.

EnclaveCache [66] is an SGX-based in-memory key-value store with a similar design. Per-tenant enclaves are responsible for establishing a secure connection with remote clients via TLS. They encrypt and decrypt incoming and outgoing data with a tenant-specific key before passing requests to an untrusted request handler to process the request on encrypted data.

ShieldStore [189] is another in-memory key-value store, that also stores sensitive key-value data in untrusted memory but adds encryption and integrity protection to protect the confidentiality, integrity, and freshness of data. *Pesos* [193] is a secure, durable key-value store that makes use of *kinetic disks* [293] that have built-in TLS support and therefore allow Pesos to persist data on disk without cooperation from the untrusted host OS. In contrast to SecureKeeper, EnclaveCache, and applications partitioned by GLAMDRING, ShieldStore and Pesos only partition data but not code. All application code is executed inside enclaves.

EnclaveDB [266], *StealthDB* [352], *ObliDB* [104], and *STANlite* [285] are partitioned databases. EnclaveDB, which we will discuss in detail in Chapter 5, only places part of an in-memory database engine as well as all sensitive table data inside the enclave. All other parts of the database are untrusted and run outside. StealthDB only implements individual operators inside the enclave. ObliDB and STANlite fully execute inside the enclave but store data on the untrusted host. We will discuss these systems in more detail in comparison to ENCLAVEDB in Section 5.8.

In contrast to applications partitioned with GLAMDRING, these applications were partitioned manually. While this allows for specific optimisations to reduce TCB size, perform additional interface hardening, or improve performance overhead, it also requires significant manual effort.

4.9 Limitations

In this section we discuss limitations of automatic application partitioning. We already covered specific trade-offs of both dynamic and static analysis in Section 4.4.3. However, some limitations apply to an automated partitioning approach as presented in this chapter in general.

Manual effort. GLAMDRING is a semi-automated partitioning approach. As the sensitivity of data depends on application semantics, it requires manual input from a developer to identify sensitive inputs and outputs. If sensitive data is not annotated accordingly, GLAMDRING cannot guarantee its protection. In contrast, an approach that executes the full application inside an enclave, e.g. SGX-LKL, treats all data as sensitive and can provide transparent confidentiality and integrity protection.

Partitioning constraints. As discussed in Section 4.2.1, there are different objectives for partitioned applications and their host interfaces: (i) a small TCB size; (ii) low host interface complexity; and (iii) few enclave transitions due to their performance impact. GLAMDRING is capable to optimise for these objectives. However, it is bound to the constraints of the existing codebase. While it can move functions in or out of the enclave in some cases, it cannot refactor code, e.g. to reduce dependencies between non-sensitive and sensitive code or to simplify interface hardening. In contrast, a manually partitioned application can be designed specifically taking the constraints of TEEs and the partitioning objectives into account.

Interface hardening and advanced attacks. Section 4.6.2 discussed how GLAMDRING protects sensitive data through encryption and hardens the host interface via runtime invariant checks. However, GLAMDRING might not be able to protect an application against all attacks. For example, freshness attacks such as rollback or replay attacks (see Section 2.4.3.1) are hard to protect against in an automated fashion. The correct way to protect against such attacks might depend on the use of the data. For example, the freshness of sensitive data sent over the network can be protected through the use of session keys and nonces. For data persisted on disk that must be accessible across enclave

restarts, more complex protection is needed, e.g. using trusted monotonic counters. Implementing such protection mechanisms correctly and efficiently therefore requires additional manual effort from developers.

These limitations show that GLAMDRING cannot fully solve all challenges of automated application partitioning and might require some additional effort. In these cases, GLAMDRING can be used in an iterative fashion: By applying GLAMDRING to a codebase, a developer can identify potential points for optimisation such as high-transition host calls or unneeded dependencies between non-sensitive and sensitive code. After refactoring the code accordingly, GLAMDRING can be rerun to achieve a more optimal partitioning, which can then be reviewed again.

We also address these issues in the following chapter, in which we discuss a manually partitioned application that can protect against advanced attacks efficiently while optimising for a small TCB with low performance overhead.

4.10 Summary

In this chapter we presented GLAMDRING, a semi-automated partitioning framework for TEE applications. Partitioning applications reduces the size of the TCB and the risk of vulnerabilities in enclave code that could be exploited to compromise the confidentiality or integrity of sensitive data. GLAMDRING helps developers to define an *application-specific* host interface and with that to partition an application into a trusted enclave library and untrusted code.

We presented how GLAMDRING partitions applications in three phases. In the *code analysis* phase, GLAMDRING analyses applications to identify security-sensitive code and data that must be protected. We discussed two approaches to implement GLAMDRING's code analysis phase: dynamic analysis and static analysis. For dynamic analysis, GLAMDRING provides a profiling tool that uses taint tracking to track the propagation of sensitive data at runtime, and to collect additional metrics such as number of function calls, memory allocations and deallocations, and performed system calls. For static analysis, GLAMDRING uses dataflow analysis and backward slicing to determine security-sensitive code that can affect the confidentiality or integrity of sensitive data. We covered the benefits and limitations of both approaches. In particular, we discussed how dynamic analysis allows to incorporate runtime

behaviour and performance metrics in the partitioning process, while static analysis allows for a conservative, workload-independent partitioning.

We discussed the different objectives for application-specific host interfaces: (i) a small TCB size, (ii) low interface complexity, and (iii) a small number of enclave transitions. We presented how the *code partitioning* phase uses the code analysis results to decide on a partitioning that takes these objectives into account by transforming them into an ILP program. We showed how an ILP solver and user-specified objectives are used to create an optimised partition specification.

Lastly, in the *code generation phase*, GLAMDRING performs a source-to-source translation to generate a trusted enclave library and untrusted code based on the partition specification. We also discussed how GLAMDRING adds code and interface hardening to protect sensitive data and the enclave library from attacks that could compromise the confidentiality and integrity of sensitive data.

We have implemented GLAMDRING using Valgrind and Frama-C for dynamic and static analysis, respectively, with Gurobi as ILP solver for the partitioning phase, and an LLVM-based code generator. We demonstrated how partitioning objectives compete with each other, and how the user-specified weights can impact TCB size, interface complexity, and performance. We also evaluated the performance of applications partitioned by GLAMDRING. For Memcached, GLAMDRING achieves a throughput of about $0.27\times$ of native execution, which is comparable to the overhead of Memcached with other TEE-based systems.

CHAPTER 5

ENCLAVEDB: A SECURE DATABASE WITH ADVANCED HOST INTERFACE ATTACK PROTECTION

This chapter presents ENCLAVEDB, a secure partitioned database using Intel SGX. ENCLAVEDB is based on the SQL Server [222] in-memory engine Hekaton [89]. We partition Hekaton to reduce the TCB size and to keep the host interface small.

In the context of ENCLAVEDB, we look at advanced host interface attacks such as rollback and forking attacks and how to protect against them. A core component of ENCLAVEDB is its logging and recovery protocol which interacts with the untrusted SQL Server host process for transaction durability. We design a protocol that (i) supports high-throughput transactions; (ii) guarantees transaction durability; (iii) protects the confidentiality and integrity of user data; and (iv) prevents advanced attacks such as rollback and forking attacks.

5.1 Introduction

In the past two chapters, we presented systems for running unmodified applications and porting existing applications for the execution in TEEs in an automated way. TEE runtime systems such as SGX-LKL have an application-agnostic host interface, which can be protected generically, e.g. via low-level disk and network encryption and integrity protection. GLAMDRING, presented in the previous chapter, automatically partitions applications resulting in an application-specific host interface. To protect the interface, the framework adds encryption and decryption of sensitive data on enclave transitions and runtime checks to preserve invariants inferred by the sensitive code analysis.

However, both these approaches have limitations. A common task of the host interface is to allow an application to share state with the untrusted host in a secure fashion, e.g. to store application data on disk that needs to be available across restarts. SGX-LKL benefits from an in-enclave page cache that

can delay writes to disk for efficiency, but sacrifices fault tolerance. If an enclave crashes before dirty pages are passed to the host OS for storage, the data is lost.

For applications such as databases that must be able to provide certain durability guarantees, this is insufficient. Applications can make use of direct I/O to bypass the in-enclave page cache. However, the low-level interface of SGX-LKL writes and reads page-sized blocks, which can be inefficient in scenarios in which small amounts of data are exchanged with high frequency, in particular when the data must be encrypted and integrity-protected.

Furthermore, SGX-LKL uses Linux device mapper targets that provide encryption and integrity protection but no freshness protection. While it can guarantee that data passed to the enclave as part of a read host call is genuine, i.e. has previously been written by the enclave, it cannot guarantee that it corresponds to the most recently written version. This is particularly problematic with an adversary that can arbitrarily stop enclave execution and present an old state of persistent data upon restart. Other device mapper targets such as *dm-x* [57] or runtime systems such as *Haven* [23] provide freshness protection via *Merkle trees* [221]. However, Merkle trees can have a significant performance overhead, and require additional trusted storage to guarantee freshness across restarts.

While GLAMDRING provides some host interface protection, automatically adding encryption, integrity and freshness protection to an application-specific interface is difficult. With an application-specific interface, the meaning of host calls and host call arguments vary and protection requires knowledge of application semantics. While it may be possible to automatically implement protection based on additional developer annotations, in many cases it must be implemented manually.

This is particularly true for the protection against advanced attacks such as freshness attacks. Freshness protection of sensitive data passed in and out of the enclave differs depending on the type of data. Data that can outlive the enclave and survive across enclave restarts, e.g. data persisted on disk, also requires freshness metadata such as monotonic counter values to be maintained across enclave restarts. In contrast, the freshness of ephemeral data, such as messages transmitted over the network, can be protected by storing freshness metadata in memory.

Another aspect of advanced host interface protection mechanisms is their impact on application performance. For example, trusted hardware monotonic counters that can be used for freshness protection are slow, and relying on a single counter can become a performance bottleneck. Therefore,

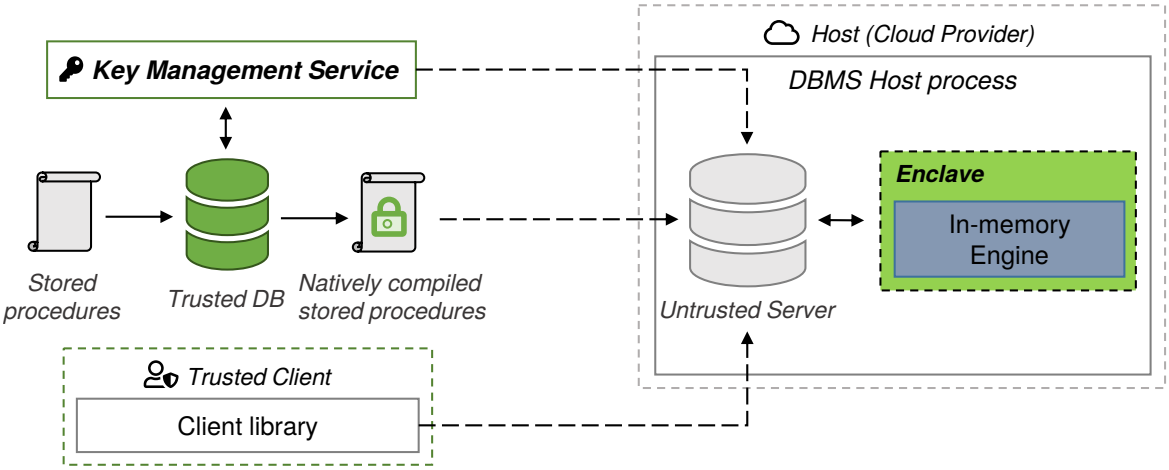


Fig. 5.1: Overview of EnclaveDB's architecture. EnclaveDB hosts sensitive data along with natively compiled queries and a query engine in an enclave.

efficient interface protection requires additional effort and must take application requirements into account.

In this chapter, we explore the challenges of protecting application-specific host interfaces in more detail. For this, we look at an example of an application with strong durability and high performance requirements, and describe how we protect its host interface, including against advanced freshness attacks.

We present EnclaveDB, a durable in-memory database that ensures confidentiality, integrity, and freshness for queries and data. EnclaveDB is based on Hekaton, SQL Server's in-memory database engine, and has a small TCB. It has a programming model similar to conventional relational databases — authorised users can create tables and indexes, and query the tables using stored procedures expressed in SQL. However, EnclaveDB also protects database state from adversaries such as external attackers, and malicious server or database administrators.

Fig. 5.1 shows an overview of its design. EnclaveDB hosts all *sensitive* data, i.e. tables, indexes, queries and other intermediate state, in enclave memory. While current SGX CPUs have limited EPC sizes, we assume that this is feasible as CPUs with large EPC sizes will be available in the near future [303].

Unlike a conventional database, EnclaveDB compiles queries on sensitive data to native code using an ahead-of-time compiler on a trusted client machine. Pre-compiled queries are signed, encrypted

and deployed to an enclave on the untrusted database server. Decoupling compilation from execution allows components such as the query parser, compiler and optimizer to be hosted outside of the enclave in a trusted client environment, thus reducing the TCB size. ENCLAVEDB clients execute pre-compiled queries by establishing a secure channel with the enclave and sending encrypted requests. The enclave authenticates requests, decrypts parameters, executes the pre-compiled query, encrypts query results, and sends the results back to the client.

In addition to confidentiality of data and queries, ENCLAVEDB also guarantees integrity and freshness of data. While in enclave memory, the integrity of tables and indexes is protected by SGX. Similarly, the integrity of queries is ensured by hosting queries and a transaction manager within the enclave. ENCLAVEDB employs a number of checks to detect and prevent both integrity and freshness violations during query processing and database recovery. This includes checks to detect invalid API usage and *Iago* attacks [62] caused by a malicious database server or operating system that violates its specification.

Central to ENCLAVEDB is an efficient logging and recovery protocol that ensures the confidentiality, integrity and freshness of the database log. The log is managed by the host process and is shared between the secure in-memory engine and the host process. It is required to guarantee durability and crash tolerance. To achieve high performance, the protocol supports concurrent appends and truncation of the log and requires minimal synchronisation between threads. At the same time, it ensures freshness of data recovered from the log after a restart of the database. ENCLAVEDB maintains transactional guarantees including durability, i.e. that committed transactions are guaranteed to be persisted permanently, even in case of a subsequent crash.

In this chapter, we present ENCLAVEDB in detail and make the following contributions:

Strong host interface protection. ENCLAVEDB guarantees *confidentiality, integrity and freshness* using a combination of encryption, native compilation and a novel scalable protocol for enforcing integrity and freshness of the database log as part of its host interface.

TEE-aware design with a small TCB. ENCLAVEDB has a small TCB — over 100× smaller than a conventional database server. We achieve this by designing ENCLAVEDB specifically for the execution in a TEE that allows us to exclude large components such as the query optimiser from the TCB.

Large enclave performance model. Scalability is an essential feature of databases. In order to fully evaluate database performance, it must be tested with large workloads. However, currently available Intel CPUs have small EPC sizes. We therefore introduce a performance model that allows us to approximate the cost of executing in larger enclaves available in the future when running EnclaveDB in simulation mode.

Based on this model, we evaluate the performance of EnclaveDB using industry-standard database benchmarks. Our evaluation shows that EnclaveDB delivers high performance (up to 31,000 transactions per second for TPC-C) with low overheads (at most 40% lower throughput compared to an insecure baseline).

The remainder of this chapter is structured as follows: Section 5.2 gives an overview of the Hekaton engine EnclaveDB is based on. It also discusses monotonic counters, used by EnclaveDB to achieve freshness protection. Section 5.3 provides an extended threat model that acknowledges the risk of advanced attacks on the host interface. Section 5.4 presents EnclaveDB's architecture. We then describe the protocol for checking integrity and ensuring freshness of checkpoints and the log in Section 5.5, followed by an informal proof in Section 5.6. Section 5.7 presents a detailed evaluation of EnclaveDB. We discuss related work in Section 5.8 and then conclude with Section 5.9.

5.2 Background

In this section, we provide background information on some building blocks of EnclaveDB. First, we give an overview of Hekaton, SQL Server's in-memory engine that is used by EnclaveDB. Then we discuss monotonic counters and introduce a monotonic counter service that EnclaveDB relies on as part of its freshness protection protocol.

5.2.1 Hekaton In-Memory Database Engine

Hekaton [89] is a SQL Server database engine optimised for OLTP workloads where data fits in memory. As machines with over 1 TB of memory are commonplace, datasets for many OLTP workloads can fit entirely in memory. Current SGX implementations have limited EPC sizes but are expected to provide significantly larger EPC sizes in future generations [303].

Hekaton allows users to host selected tables in memory and create one or more memory-resident indexes on the table. Hekaton tables are durable — the Hekaton engine logs transactions on memory-

resident tables to a persistent log that is shared with SQL Server. Periodically, Hekaton *checkpoints* the log by compressing log records into a more compact representation on disk. On failure, the database state can be recovered using these checkpoints and the tail of the log.

For efficiency, indexes and all other internal data structures are implemented in a lock-free way, and therefore allow for high degrees of concurrency. To further optimise performance, Hekaton supports a mode of execution in which table definitions and SQL queries over in-memory tables are compiled to machine code. Native compilation is restricted to queries where decisions typically made by the query interpreter at runtime can be made at compile time, e.g. queries where the data types of all columns and variables are known at compile time. These restrictions allow the Hekaton compiler to generate efficient code with optimised control flow and no runtime type checks.

With ENCLAVEDB, we show that the principles behind the design of a high performance in-memory database engine are aligned with security. Specifically, in-memory tables and indexes are ideal data structures for securely hosting and querying sensitive data in enclaves. In-memory tables eliminate the need for expensive software encryption and integrity checking otherwise required for disk-based tables. Query processing on in-memory data minimises the leakage of sensitive information and the number of transitions between the enclave and the host. Finally, native compilation allows query compilation and optimisation to be decoupled from query execution. This enables a mode of compilation where queries are compiled on a trusted database and deployed to an enclave on an untrusted server, significantly reducing the attack surface available to an adversary.

5.2.2 Monotonic Counters

In Chapter 2, we discussed freshness attacks such as rollback and replay attacks in which an adversary provides an enclave with genuine but out-of-date data, or replays genuine messages multiple times unexpectedly (see Section 2.4.3.1).

One option to prevent freshness attacks is the use of trusted monotonic counters. For example, by associating counter values with data passed to an untrusted host, e.g. using authenticated encryption, it is possible to verify its freshness at a later time by comparing the stored value with the current counter value. ENCLAVEDB's logging and recovery protocol, which we will describe in Section 5.5, relies on monotonic counters to enforce *state continuity*. State continuity ensures that an adversary cannot provide stale state without detection (i.e. *freshness*), and that the system will always be able to

make progress, even after crashing at arbitrary times, when not under active attack [320] (i.e. *liveness* and *crash tolerance*).

There are many ways of implementing monotonic counters. SGX provides access to trusted monotonic counters available via the Management Engine [217], which uses wear-limited NVRAM to store counters. However, our experiments confirm prior results [318] that accessing these counters is slow (~100 ms per counter update), and that they are not sufficient for the latency and throughput requirements of EnclaveDB.

In EnclaveDB, we use a dedicated monotonic counter service implemented using replicated enclaves, similar to *ROTE* [213]. The service stores counters in enclaves replicated across different fault domains and uses a consensus protocol to order operations on counters. This approach is more flexible and efficient than SGX counters, and can tolerate failures as long as a quorum of replicas is available. Without a quorum, in order to not compromise freshness EnclaveDB halts execution and becomes unavailable.

This monotonic counter service exposes an API with functions `CreateCounter`, `GetCounter`, `IncCounter`, and `SetCounter`. `CreateCounter` creates counters bound to the TCB of the enclave and the platform, `GetCounter` returns the current value of a counter. `IncCounter` atomically increments a counter and returns the previous value of the counter, and `SetCounter` sets a counter to a given value if it is higher than the counter's current value or fails otherwise.

This API mirrors that of hardware monotonic counters which can be incremented and accessed to retrieve their current value. In addition, `CreateCounter` allows to create any number of trusted monotonic counters. We rely on this for EnclaveDB which requires multiple monotonic counters for freshness protection as we will discuss in more detail in Section 5.5.1. `SetCounter` is used by EnclaveDB during recovery. Note that the same action could be performed by calling `IncCounter` multiple times. We use `SetCounter` for efficiency and brevity.

5.3 Extended Threat Model

With EnclaveDB, we extend the threat model of the previous chapters, described in Section 2.5. As before, we consider a strong adversary that controls the entire software stack on the database server, except the code inside enclaves. This represents threats from an untrusted server administrator, the

database administrator, and attackers who may compromise the operating system, the hypervisor or the database server.

The adversary can access *and tamper* with any server-side state in memory, on disk and over the network. This includes attacks that tamper with database files such as logs and checkpoints, e.g. overwriting, dropping, duplicating and/or reordering log records.

In addition, the adversary can mount replay attacks by arbitrarily shutting down the database and attempting to recover from a stale state. The adversary can attempt to fork the database, e.g. by running multiple replicas of the database instance on the same or different machines and sending requests from different clients to different instances. The adversary can also observe and arbitrarily change control flow, e.g. make an arbitrary sequence of calls to any of the pre-defined entry points in the enclave.

We consider denial-of-service and side-channel attacks (see Section 2.4.3) to be out of scope and mitigations to them as orthogonal. We assume that the code placed inside the enclave is correct and does not leak secrets intentionally. We also assume that all client-side components such as SQL clients and the key management service are trusted.

Even under this threat model, we wish to guarantee both confidentiality and *linearizability* [158]. In a linearizable database, transactions appear to execute sequentially in an order consistent with real time. Therefore, clients do not have to reason about concurrency or failures. In our context, a linearizable database frees the clients from having to reason about an active attacker. In addition to linearizability, we would also like to ensure *liveness* i.e. the database should always be able to recover from non-malicious unexpected shutdowns at any time. Note that liveness does not imply availability — an attacker can always prevent progress, e.g. by not allocating resources.

5.4 ENCLAVEDB Architecture

In this section, we describe ENCLAVEDB's architecture. We first give an overview of how ENCLAVEDB is partitioned into a trusted in-enclave database engine and an untrusted database server component that runs on the host. We then discuss query compilation, transaction processing, and key management.

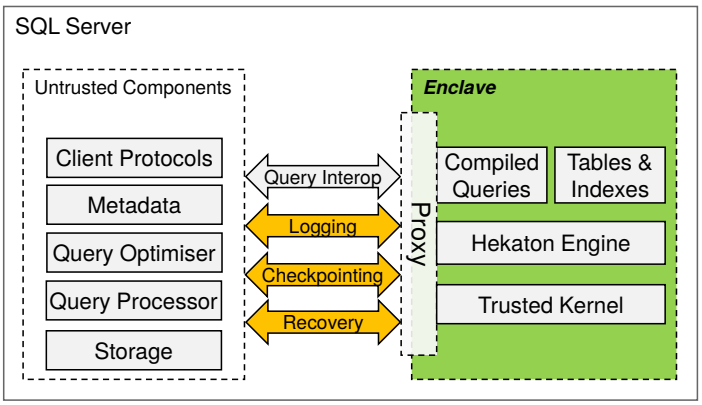


Fig. 5.2: Server-side components of ENCLAVEDB

5.4.1 Overview

An ENCLAVEDB service consists of an untrusted database server that hosts public data and an enclave that contains sensitive data. Fig. 5.2 shows the server-side components in ENCLAVEDB. The enclave hosts a modified Hekaton query processing engine, natively compiled stored procedures, and a trusted kernel, which provides a runtime environment for the database engine and security primitives such as attestation and sealing. The untrusted host process runs all other components of the database server, including a query compiler and processor for public data, and the log and storage managers. The untrusted server supports database administration tasks, e.g. maintenance operations such as backups, troubleshooting of server problems, or configuration of storage options. However, a database administrator does not get access to sensitive data. This is important since database administration is often outsourced, e.g. to a cloud provider or third parties.

The query processor on the untrusted database server supports generic queries on public data. In cases in which such data exists, it can be kept out of the enclave, e.g. as a performance optimisation. The query processor is also responsible for receiving requests to execute stored procedures on secret data and handing them over to the in-enclave Hekaton engine. ENCLAVEDB currently does not support queries that operate over both public and secret data; guaranteeing security in the presence of such queries is a challenging problem that we leave for future work. We now describe ENCLAVEDB's components in detail.

5.4.2 Trusted Kernel

ENCLAVEDB requires a number of typical OS services such as threading, memory management, and storage. Since we aim for a small TCB, we cannot rely on a full library OS such as SGX-LKL. Instead, we add a small *trusted kernel* that provides the Hekaton engine with the minimal set of services that it requires. The trusted kernel implements some enclave-specific services such as management of enclave memory and enclave threads, and delegates other services such as storage to the host OS. It also adds additional logic for ensuring confidentiality, integrity, and freshness, which we will discuss in Section 5.5.

The trusted kernel allocates a stack for each thread in enclave memory when a thread enters the enclave, and then transfers control to the application. From an application's perspective, ENCLAVEDB's threading model is best viewed as a *pool of enclave threads*. When the host calls into the enclave, the trusted kernel effectively 'suspends' the host thread and switches to an unused enclave thread. When the call completes (or an exception occurs), the trusted kernel reclaims the enclave thread and execution resumes on the host thread. Therefore, the size of the thread pool, which is fixed on enclave creation, determines the maximum degree of concurrency available to the application.

The trusted kernel also supports thread-local storage (TLS), which is used extensively by the Hekaton engine for efficient access to performance-critical data structures such as transaction read/write sets. Preserving TLS across calls would require the kernel to trust a host assigned thread identifier, thereby introducing a new attack vector. However, Hekaton already re-establishes TLS from the heap on every entry into the engine, except in the case of reentrancy, i.e. nested calls into the engine. As described in Section 5.5, Hekaton components such as the log use reentrancy for callbacks and assume that TLS is preserved on re-entrant calls. For this, we modified the Hekaton engine to save TLS state on the heap before enclave exits and restore TLS state on re-entrant calls.

5.4.3 Query Compilation and Loading

In a conventional database, the database server compiles, optimises and executes queries. Therefore, the entire query processing pipeline is part of the attack surface. ENCLAVEDB reduces the attack surface by relying on client-side, native query compilation. The client packages all pre-compiled queries (expressed as stored procedures) along with the query engine and the trusted kernel, and deploys the package into an enclave. This design offers strong security as the queries become part of

Table 5.1: Hekaton's transaction processing API

`Tx* TxAlloc()`
`bool TxExecute(Tx* tx, BYTE* name,BYTE** params, BYTE** ret)`
`bool TxPrepare(Tx* tx, TxPrepareCallback prepareCb)`
`void TxCommit(Tx* tx)`
`void TxAbort(Tx* tx)`

the enclave measurement. However, the design also implies that any change in schema, e.g. adding or removing queries, requires taking the database offline and redeploying the package.

Online schema changes can be supported using a trusted loader. For this, instead of packaging all queries along with the query engine and the trusted kernel, the initial deployment does not include the queries and attestation is performed without them. Instead the enclave code contains a trusted loader to add new stored procedures sent via a secure communication channel. To support multiple tenants, certificates of each tenant can be embedded in the the enclave library. Using these certificates the trusted loader can verify that a new stored procedure has been signed by all tenants and therefore has been agreed upon by them.

5.4.4 Transaction Processing

Hekaton uses a two-phase protocol for transaction processing (Table 5.1, see Section 5.5 for a more detailed diagram). When the host receives a request to execute a stored procedure, it first creates a new transaction using `TxAlloc` and assigns a *logical* start timestamp to the transaction using a monotonically increasing counter stored in memory. It then executes the stored procedure in the context of the transaction by calling `TxExecute` along with the name of the stored procedure, buffers containing parameter values, and buffers for storing return values. `TxExecute` loads the natively compiled binary and transfers control to a well-defined function corresponding to the stored procedure within the binary. The binary calls into the Hekaton engine to perform operations on Hekaton tables and update transaction state, e.g. read and write sets, or start timestamps.

After executing the stored procedure, the host prepares the transaction for committing by calling `TxPrepare`. The prepare phase validates the transaction by checking for conflicts, assigns a logical end timestamp to the transaction, waits for transactions that it depends on to commit, and logs the transaction's write set. Since logging involves expensive I/O operations, `TxPrepare` is asynchronous — it registers a callback and returns after initiating the log I/O. Once the I/O operations have completed,

the host calls `TxCommit` to commit the transaction, which releases any resources associated with the transaction, unblocks all dependent transactions, and writes return values to be sent to the client. However, if the prepare phase fails, e.g. due to conflicts, the host calls `TxAbort`.

This protocol is vulnerable to a number of attacks from a malicious host even if the Hekaton engine is hosted in an enclave. The host can pass arbitrary transaction handles (`Tx*`), tamper with the incoming request, e.g. by changing parameter or return values, and invoke the protocol functions out of order. To ensure integrity of client-server interactions, we make the following client-side modifications.

- We extend the query compiler to embed metadata such as the stored procedure name and the position and type of parameters in a dedicated section in the native binary. This metadata is used to validate incoming requests and to ensure that stored procedure calls refer to the embedded stored procedures and adhere to the right format, and thus cannot be tampered with.
- ENCLAVEDB clients connect to ENCLAVEDB by creating a secure channel with the enclave and establishing a shared session key $\mathcal{S}_{\mathcal{K}}$. During session creation, clients authenticate the enclave using a quote that contains the enclave's measurement. The enclave can authenticate clients using certificates or tokens issued by a trusted authority; our implementation uses certificates embedded in the ENCLAVEDB engine binary.
- ENCLAVEDB clients encrypt parameter values using $\mathcal{S}_{\mathcal{K}}$. Each parameter value is encrypted using authenticated encryption with the parameter position, type and a nonce as authentication data to prevent replay attacks.

We also make the following server-side modifications to protect the host interface:

- The transaction processing APIs verify that transaction objects passed as a parameter are allocated in enclave memory, and procedure names, parameters and return values are buffers in untrusted memory. Without these checks, an adversary could exploit the interface, e.g. by tricking ENCLAVEDB into writing sensitive data to untrusted memory, or unintentionally overwriting enclave memory.
- `TxExecute` authenticates all incoming requests. If the authentication succeeds, ENCLAVEDB loads the native binary and checks if the procedure name, parameter positions and types are consistent with metadata stored in the binary. If validation succeeds, ENCLAVEDB decrypts the

parameters, allocates buffers in enclave memory for return values, and forwards the request to the Hekaton engine.

- After a stored procedure has executed, ENCLAVEDB encrypts return values (or error messages) and writes them to buffers allocated by the host.
- The engine maintains additional state to ensure that the host does not attempt to commit a transaction if the prepare phase failed. Otherwise, a malicious host could deliberately cause ENCLAVEDB to ignore conflicts previously detected in the prepare phase and potentially corrupt the state of the database.

Observe that once a request has been validated, the stored procedure executes entirely within the enclave on tables hosted in the enclave. This prevents the host from tampering with query processing and reduces information leakage.

5.4.5 Key Management

ENCLAVEDB supports a much simpler model for key management compared to existing systems [13, 262], which require users to associate and manage encryption keys for each column containing sensitive data. In ENCLAVEDB, sensitive columns are hosted in enclave memory, and data in these columns is encrypted and integrity protected by SGX's Memory Encryption Engine while residing in memory. Users only need to create and manage a single database encryption key $\mathcal{D}_{\mathcal{K}}$, which is used to encrypt all persistent database state. Users provision the key to a trusted key management service (KMS), along with a policy that specifies the enclave, identified using the enclave's measurement, that the key can be provisioned to. When an ENCLAVEDB instance starts or resumes from a failure, it remotely attests with the KMS and receives $\mathcal{D}_{\mathcal{K}}$.

5.4.6 Host Interface Optimisations

In Chapter 4, we discussed trade-offs of partitioned applications in terms of TCB size, interface complexity, and the number of enclave transitions that impact both security and performance. We already discussed how ENCLAVEDB reduces its TCB size, e.g. by using precompiled queries and excluding the query processor from its TCB.

Compared to prior work that uses trusted hardware for evaluating individual expressions in a query [13, 19], ENCLAVEDB has a significantly smaller number of enclave transitions. Furthermore, the number

of transitions is fixed and independent of the data being processed. In addition, ENCLAVEDB only incurs the cost of encryption and decryption at transaction boundaries (parameters and return values) and for log records and checkpoints, which is more efficient than encrypting and decrypting individual values.

We make a number of further optimisations in order to reduce ENCLAVEDB's host interface complexity and the number of enclave transitions required during transaction processing.

- We refactored the Hekaton API to move state and logic that does not depend on secrets to the host. This includes state for tracking whether log records for a transaction have been committed to disk. Conversely, we refactored the host to move all secret-dependent functionality, such as support for comparing buffers of various data types and random number generation, to the Hekaton engine.
- Hekaton uses optimistic, multi-version concurrency control for serialising transactions and relies on a cooperative, non-blocking garbage collector to reclaim memory used by obsolete versions of records. In this mode, worker threads running the transaction workload also collect garbage they encounter after committing the transaction and sending results to the client. However, in ENCLAVEDB, this introduces additional transitions since control must transfer back and forth between the enclave and host. We optimised this design by performing garbage collection after the commit but before returning control to the host. This increases latency marginally but results in fewer transitions and higher throughput.
- SQL Server uses a cooperative thread scheduler, which expects all threads to periodically yield control. In ENCLAVEDB, every yield results in two transitions. We modified the Hekaton engine to reduce the frequency with which threads in enclave mode yield (by half).
- We use *prefetching* to reduce the number of transitions. Prefetching involves speculatively calling an enclave API as part of a previous enclave invocation, caching its results on the host and returning the cached result when the API is subsequently called without entering the enclave. This optimisation only applies to side-effect free APIs.

These optimisations resulted in over an order of magnitude reduction in the number of transitions (from an average of 110 to 8 transitions per transaction). This includes five *ecalls* with an additional callback to write the log record if the transaction performs any writes, and two *ocalls*. The number

Algorithm 5.1 Specification of the logging interface exposed by the host to Hekaton

```
1: L ← ∅
2: procedure LogAppend(tx, size, serializeCb, commitCb)
3:     buf ← alloc(size)
4:     serializeCb(tx, buf)
5:     L ← L ∪ buf
6:     startLogIO(tx, buf, commitCb)
7: procedure LogTruncate(buf)
8:     L ← L \ {b ∈ L | b < buf}
9: procedure GetLogIterator(start, end)
10:    return {b ∈ L | b ≥ start ∧ b ≤ end}
```

of calls to write log records decreases further in workloads with high concurrency in which case Hekaton uses group commits to consolidate log record writes from multiple transactions into a single write.

5.5 Logging and Recovery

The Hekaton engine makes in-memory tables durable by writing transactions on secret data to a persistent log managed by the host (i.e. SQL Server). Since the host is untrusted, EnclaveDB must ensure that a malicious host cannot observe or tamper with the contents of the log. In this section, we present a high-level specification of the logging and checkpointing APIs and then describe protocols to ensure confidentiality, integrity, and freshness.

As shown in Algorithm 5.1, the log can be abstracted as a stream of bytes and a set L that contains indexes in the stream where individual log records begin. The log supports operations for appending log records, truncating the log, and iterating over the log. The append operation allocates space in the stream for the log record and returns an index *buf* in the stream, which also serves as the address of the buffer used for reading or writing the log record. The append operation invokes a callback `serializeCb`, which writes the log record in the allocated buffer, and schedules another callback `commitCb`, which is invoked when the log record has been flushed to disk. The truncation operation deallocates all buffers preceding the given index. The iterator returns the indexes of all log records between any two indexes *start* and *end*.

The Hekaton engine uses the log as follows (Fig. 5.3). After a transaction has been validated (in `TxPrepare`), the engine serialises the transaction's write set into a log record by calling `LogAppend`. Each log record includes the start and end timestamp of the transaction. The host writes the log

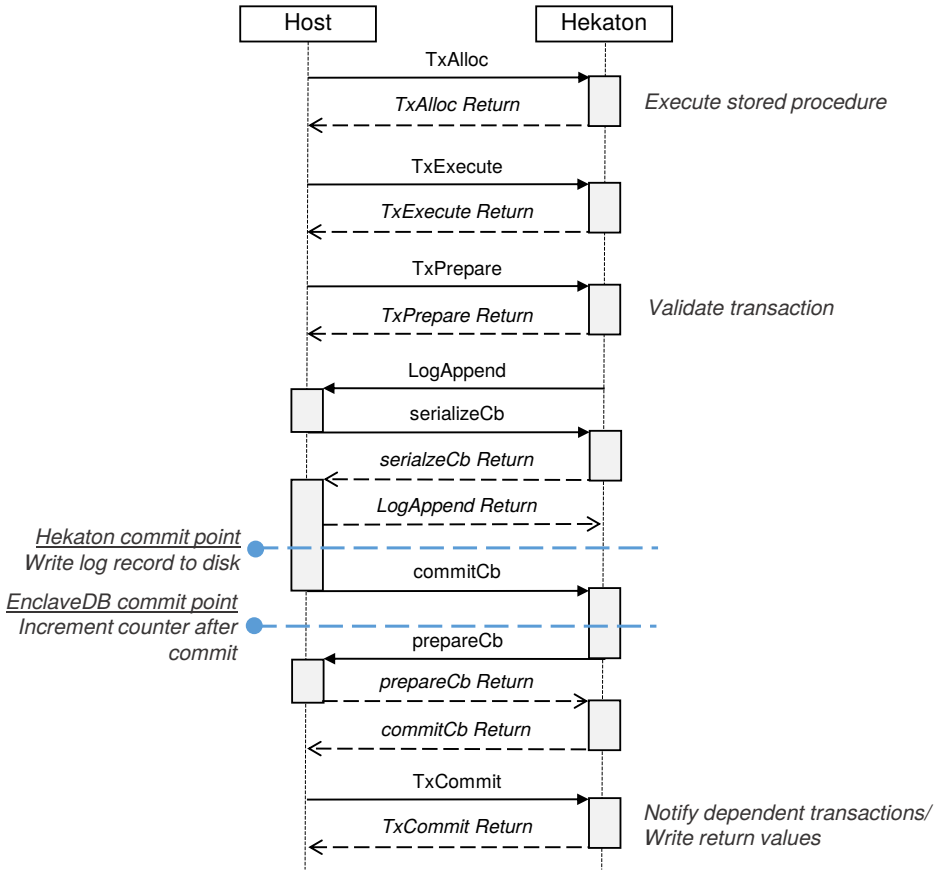


Fig. 5.3: Transaction commit protocol in ENCLAVEDB

record to disk and then invokes the commit callback. At this point, the transaction enters the commit state. During `TxCommit`, Hekaton unblocks dependent transactions and notifies the client submitting the transaction. Hekaton is designed to write multiple log records concurrently to avoid scaling bottlenecks with the tail of the log. This is possible because the serialisation order of transactions is determined solely by end timestamps and not by the ordering in the log. Also note that failures can occur at any point e.g. after a log record has been flushed to disk but before the client is notified or dependent transactions are unblocked. We refer to transactions for which Hekaton *may* have unblocked dependent transactions or notified the client as *visible transactions*, and transactions whose dependent transactions remain blocked and no notifications have been generated as *invisible transactions*.

To avoid unbounded growth of the log, Hekaton periodically creates checkpoints and then truncates the log (Algorithm 5.2). Each checkpoint is a pair of append-only files, a data file and a delta file. The data file contains all records that have been inserted or updated since the last checkpoint and the delta file contains all deleted records. Checkpoints are created using an in-memory cache of log

Algorithm 5.2 Hekaton operations for creating checkpoints and restoring the database after a failure

```
1: sys ← ∅
2: procedure CreateCheckpoint(start, end)
3:     {datafile, deltafile} ← SerializeLog(start, end)
4:     sys ← sys ∪ {datafile, deltafile}
5:     WriteFile(ROOT_FILE, sys)
6:     LogTruncate(end)
7: procedure RestoreDatabase(start, end)
8:     sys ← ReadFile(ROOT_FILE)
9:     for {data, delta} ∈ sys do
10:        RestoreCheckpoint(data, delta)
11:    ReplayLog(start, end)
```

records, which is updated during transaction commits. The names of data and delta files are saved in a special table called the *system table* (*sys*), which is persisted in a file called the *root file*. To avoid data loss, Hekaton truncates the log at a carefully selected index called the *truncation index*, which satisfies the invariant that all transactions committing after the truncation operation will be allocated indexes higher than the truncation index. During recovery (see `RestoreDatabase` in Algorithm 5.2), Hekaton retrieves the truncation index from the database master file and the list of checkpoint file pairs from the root file. It restores tables and indexes from checkpoints, and replays the tail of the log from the truncation index.

To protect the confidentiality of log records, we rely on authenticated encryption with associated data (AEAD). In the following, we write $Enc[k]\{ad\}(text)$ to represent the encryption of *text* using key k and authentication data ad. We write $Dec[k](enc)$ to represent authentication and decryption of ciphertext containing authentication data. Our implementation uses AES-GCM [216], a high-performance AEAD scheme.

5.5.1 Log Integrity

One way of ensuring integrity of the log and checkpoints is to use an integrity-protected encrypted file system as done in SGX-LKL. Files are encrypted with a key stored in enclave memory and integrity can be ensured using a Merkle tree [221]. However, maintaining a Merkle tree for highly concurrent and write-intensive workloads such as a database log can be expensive. A Merkle tree introduces contention because the log is an append-only structure with a large number of threads writing close to the tail of the file. These threads will update roughly the same set of nodes in the Merkle tree, and contend for locks protecting these nodes [101, 250]. The Merkle tree also introduces contention for

any monotonic counters used to protect the tree against replay attacks. Finally, if the Merkle tree is maintained on disk, a single log append can translate into multiple updates on disk.

With ENCLAVEDB, we propose a new and efficient protocol (Algorithm 5.3) for checking integrity and freshness of the log. The protocol is based on the following observations.

- Correctness of database recovery does not depend on the order of log records in the log. Instead, the ordering of transactions is determined by start and end timestamps embedded in the log records. Therefore, the log can be viewed as a *set* of log records instead of a raw file.
- Hekaton can ensure state continuity as long as all checkpoints and log records of all visible transactions that have not been truncated are read during recovery.

Our protocol uses monotonic counters (see Section 5.2.2) to track sets of log records and identify log records that must exist in the log on recovery. The protocol uses three counters: W tracks log records that have been written to the log; V tracks log records generated by visible transactions; and R tracks truncated log records. To ensure that the protocol does not introduce new synchronisation bottlenecks, we track these sets separately for each thread using per-thread monotonic counters. Therefore, W, V, and R are k-dimensional vectors, where k is the number of enclave threads, which is fixed on enclave creation.

The counters are updated as follows. Each thread t processing a transaction increments the counter W_t after transaction validation but *before* sending the log record to the host (Line 17). All log records are encrypted with $\mathcal{D}_{\mathcal{K}}$ using authenticated encryption, with authentication data consisting of the thread identifier t, the counter value W_t and the log record's index in the log (Line 18); the authentication data is stored in the log record's header. This ensures that each log record can be uniquely identified by the pair of attributes (t, w) embedded in the log record. The counter V_t is incremented during the commit callback for a log record generated by thread t *before* unblocking dependent transactions and notifying the client (Line 25). At any point in time, the difference between pairs of counters W_t and V_t represents log records of transactions that are not yet visible.

Tracking truncated log records is more challenging because at any given point, there are multiple log records being written to the log at indexes assigned by the host, and a malicious host can mount attacks by assigning indexes arbitrarily, e.g. in portions of the log that have already been truncated. In

Algorithm 5.3 Protocol for checking integrity of the log

```
1:  Monotonic counters
2:  ∀t ∈ Threads, W_t ← 0, V_t ← 0, R_t ← 0
3:  E ← 0
4:  Volatile enclave state
5:  ∀t ∈ Threads, l_t ← ∅, s_t^0 ← 0
6:  b_0 ← 0, p ← 1
7:
8:  procedure EnclaveLogAppend(tx, size)
9:      size ← size + HEADER_SZ
10:     LogAppend(tx, size, enclaveSerializeCb,
        enclaveCommitCb)
11:
12: procedure enclaveSerializeCb(tx, buf, size)
13:     t ← GetCurrentThreadId()
14:     assert(buf > b_p)
15:     tmp ← malloc(size − HEADER_SZ)
16:     serializeCb(tx, tmp)
17:     w ← IncCounter(W_t)
18:     buf ← Enc[D_K]{GetCounter(E), t, w}(
        tmp)
19:     l_t ← l_t ∪ buf
20:     free(tmp)
21:
22: procedure enclaveCommitCb(tx, enc)
23:     {e, t, c}, buf ← Dec[D_K](enc)
24:     assert(e == GetCounter(E) ∧
        c == GetCounter(V_t))
25:     IncCounter(V_t)
26:     commitCb(tx, buf)
27: procedure OnSerialisationPoint(buf)
28:     ∀t, s_t^p ← s_t^{p−1} + | {b ∈ l_t | b < buf} |
29:     ∀t, l_t ← l_t \ {b ∈ l_t | b < buf}
30:     b_p ← buf
31:     p ← p + 1
32:
    Require: (∃ i | i ≤ p ∧ b_i == buf)
33: procedure EnclaveLogTruncate(buf)
34:     j ← (i | i ≤ p ∧ b_i == buf)
35:     ∀t, SetCounter(R_t, s_t^j)
36:     LogTruncate(buf)
37:
38: procedure ReplayLog(start, end)
39:     ∀t, c_t ← GetCounter(R_t)
40:     ∀t, s_t^p ← GetCounter(R_t)
41:     I = GetLogIterator(start, end)
42:     for enc ∈ I do
43:         {e, t, c}, buf ← Dec[D_K](enc)
44:         assert(e == GetCounter(E))
45:         if GetCounter(R_t) ≤ c ≤
              GetCounter(V_t) then
46:             assert (c == c_t + 1)
47:             c_t ← c_t + 1
48:             l_t ← l_t ∪ buf
49:             ApplyLogRecord(buf)
50:     ∀t, assert(c_t == GetCounter(V_t))
51:     if ∃t | c_t ≠ GetCounter(W_t) then
52:         OnSerialisationPoint(end)
53:         CreateCheckpoint(start, end)
54:         IncCounter(E)
55:     ∀t, SetCounter(V_t, GetCounter(W_t))
```

order to detect malicious behaviour, we establish a *contract* between ENCLAVEDB and the host based on the notion of *serialisation points*. A serialisation point is defined as follows:

Definition 5.5.1 *Let t be a transaction and l be an index in the log. Let $log(t)$ represent the first index in the log where a log record for transaction t has been written. A serialisation point is a pair (t,l) such that all transactions committing after t should be written to the log after the index l. Formally, let $\leq$ represent the happens before relationship between transactions.*

$$serialisation\_point(t,l) \Rightarrow \forall t' \mid t \leq t', log(t') > l$$

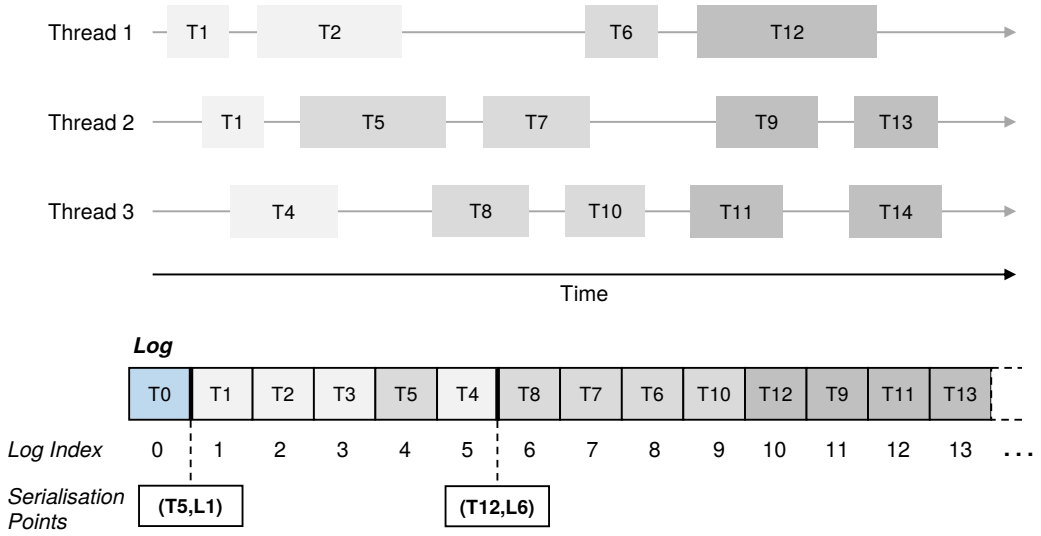


Fig. 5.4: Serialisation points during transaction processing

It follows that given a serialisation point (t,l), the log can be safely truncated at l once all transactions that have committed before t have been written to a checkpoint. From the perspective of integrity checking, serialisation points have the following implications.

- Given a serialisation point (t,l), it follows that once transaction t commits, a correct log implementation never returns an index l' such that $l' \leq l$ in any subsequent `LogAppend` call. A violation of this property indicates an attack.
- Once a serialisation point (t,l) is established, we can safely compute the expected number of log records that have written before the index l because no more log records should be written at any index $l' \leq l$.

Hekaton establishes serialisation points by grouping transactions based on end timestamps and *waiting* for all transactions in group g_{n-1} to commit (and hence be written to the log) before committing any transactions in g_{n+1}. Fig. 5.4 illustrates this mechanism using a sample execution. Transactions are colour-coded according to groups. $(T5, L1)$ is a valid serialisation point because all transactions committing after $T5$ are written to the log after $L1$.

Our protocol uses serialisation points for tracking truncated log records as follows. We maintain (in volatile enclave memory) the sequence of serialisation points b, a per-thread list of indexes l_t at which thread t has written log records since the most recent serialisation point, and a sequence s of sets, one for each serialisation point, where each element s_t^p is a number of log records written by thread t

before the serialisation point p. When a new log record is created, we add its index to the list l_t after checking that the host does not violate the serialisation point contract (line 14). We introduce a new operation `OnSerialisationPoint`, which is invoked by Hekaton when it establishes a serialisation point. For each thread t, this operation computes a summary s_t^p and removes all indexes preceding the serialisation point from l_t.

We also modified Hekaton to invoke `EnclaveLogTruncate` after creating a checkpoint, passing in a truncation index, which must be a previously declared serialisation point. This operation updates the vector clock R (line 35) to reflect the set of truncated log records using a previously computed summary before calling out to truncate the log.

During recovery (`ReplayLog`), ENCLAVEDB reads the counters R and scans the tail of the log. While scanning, we check that log records have not been tampered with and that the log records generated by each thread appear in order of their counter value with no gaps (Line 46). After scanning the tail, we check if *all* visible log records (tracked by V) have been read (Line 50), and report a freshness violation otherwise.

Note that the recovery protocol excludes log records generated by invisible transactions, i.e. log records with counter values greater than V_t. Excluding these log records is safe because, unlike Hekaton, the increment of the counter V_t (line 25) is the commit point in ENCLAVEDB — clients and dependent transactions are notified only after the increment.

However, simply excluding these log records would allow an adversary to mount a replay attack by withholding log records belonging to invisible transactions and adding them to the log in a later execution, thereby creating non-linearizable executions. We prevent these attacks by invalidating all such log records using an additional monotonic *epoch counter* E. This counter is included in each log record's authentication data, and incremented (line 54) when log records belonging to invisible transactions are detected in the log (line 51). The increment, coupled with the additional check that all log records read during recovery must belong to the current epoch (line 44) invalidates all invisible log records that the adversary may have withheld. We create a checkpoint and truncate the whole log before incrementing the epoch counter to ensure that visible transactions are not lost. We also update the counters V, setting them equal to W. Only after this, execution resumes and new transactions are accepted.

Algorithm 5.4 Protocol for checking integrity and freshness of checkpoints

```
1:  S ← 0 //Monotonic counter
2:  procedure EnclaveWriteFile(name, data)
3:      WriteFile(name, Enc[D_K]{GetCounter(S) + 1}(data))
4:      IncCounter(S)
5:
6:  procedure EnclaveReadFile(name)
7:      enc ← ReadFile(name, GetCounter(S))
8:      s, data ← Dec[D_K](enc)
9:      assert(s == GetCounter(S))
10:     WriteFile(name, Enc[D_K]{GetCounter(S) + 1}(data))
11:     IncCounter(S)
12:     WriteFile(name, Enc[D_K]{GetCounter(S) + 1}(data))
13:     IncCounter(S)
14:     return data
```

5.5.2 Checkpoint Integrity

As discussed, ENCLAVEDB periodically truncates the log after creating checkpoints. To avoid data loss, ENCLAVEDB must ensure that before log records are truncated from the log, they have been included in checkpoint files that are guaranteed to be read during recovery. Furthermore, ENCLAVEDB must also ensure that any tampering with the checkpoint files is detected.

ENCLAVEDB achieves these properties as follows: ENCLAVEDB maintains a cryptographic hash for each data and delta file. The hash is updated as blocks are added to the file. Once all writes to a checkpoint file have completed, the hash is saved in the system table along with the file name. ENCLAVEDB checks the integrity of all checkpoint file pairs read during recovery by comparing the hash of their contents with the hash in the root file.

ENCLAVEDB uses a state-continuity protocol based on Ariadne [318] to save and restore the system table within the root file while guaranteeing integrity, freshness and liveness. The protocol (shown in Algorithm 5.4) uses a monotonic counter S to track versions of the root file. The protocol binds the contents of each file with the counter value by adding the counter value plus 1 to the file and generating a keyed MAC. Then the file is written to disk and the counter is incremented. This protocol allows the adversary to obtain a file with a counter value one more than the current counter by introducing a failure after the file has been written but before the increment. However, all such versions are invalidated by rewriting two versions of the current root file (i.e. with enclosed counter value matching S) with counter values $S+1$ and $S+2$ before using the root file to reconstruct the system table (see [318] for a proof of correctness).

5.5.3 Forking Attacks

The protocol described above ensures that any database enclave recovers to the latest state. However, it permits the adversary to launch forking attacks by creating multiple enclaves with the same package on one or more servers, and directing different clients to different enclaves. ENCLAVEDB prevents forking attacks by ensuring that at any point in time, only one enclave (and therefore one database instance) is "active".

On creation, each ENCLAVEDB enclave generates a unique 128-bit identifier (ID). This ID is encrypted using the public key of the KMS and included in the enclave's quote. The KMS maintains a mapping from database instances to IDs and a *black list* of all IDs it has previously received in quotes. When it receives a quote containing a new ID, it adds the current ID to the blacklist and updates the ID associated with the database instance. Each database enclave also includes its ID (encrypted using the session key) in all communication with clients. ENCLAVEDB clients verify that they establish a session with or receive a response from the most recent incarnation of the database by validating the response with the KMS, and retry if validation fails.

5.6 Proving Continuity, Integrity, and Liveness

In this section, we present an informal proof that the logging protocol described above guarantees continuity, integrity, and liveness. Continuity and integrity are critical for establishing that ENCLAVEDB guarantees linearizability (proof beyond the scope of this chapter), and liveness ensures that ENCLAVEDB makes progress in the absence of an attacker.

Claim 5.6.1 ***Checkpoint continuity.*** *Once* `EnclWriteFile` *completes writing a root file, all log records included in checkpoint file pairs referenced in the root file are guaranteed to be read during any subsequent recovery.*

This follows from the freshness guarantees of Ariadne's protocol which prevents replay attacks, and integrity checks on the root file and checkpoint file pairs.

Claim 5.6.2 ***Continuity.*** *Log records generated by visible transactions are either contained in the log or included in a checkpoint file pair contained in the root file.*

Consider a visible transaction T which generates log record l. Let c be the counter value embedded in the log record, t be the identifier of the enclave thread generating the log record, and e be the epoch in which the log record is generated.

First, observe that each log record is uniquely identified by the pair (t,c). This follows from the fact that c is obtained from the tamper-proof, monotonic counter W_t, which is atomically incremented every time a log record is generated.

Now consider any recovery that occurs following a failure after the transaction T became visible. It follows that $c < V_t$. There are two possible cases we encounter during recovery.

- $R_t \leq c < V_t$. In this case, the log record l must be read from the log since the recovery protocol reads all records from R_t to V_t for each thread t. If the log record is missing or has been duplicated, either the assert at line 46 or 50 fails. If the log record has been tampered with, authentication fails. In either case, the database fails to recover.
- $c < R_t$. In this case, we show that the log record must belong to a checkpoint included in the root file. Observe that the counters R are updated *after* a checkpoint has been created and *before* the log is truncated. This implies that the counters always under-approximate the set of log records that have been included in checkpoints. Therefore, if $c < R_t$, then the log record must have been included in a checkpoint. This, together with checkpoint continuity (Claim 5.6.1) ensures that either the log record is included in a checkpoint read during recovery, or the database fails to recover.

Claim 5.6.3 ***Integrity.*** *Invisible transactions have no effect on database state.*

We show that log records generated by invisible transactions are ignored while reconstructing database state during recovery. Consider a log record l generated by an invisible transaction T. Let c and e be the counter and epoch values associated with l, and t be the enclave thread that generated the log record. By definition, the commit callback was not invoked for l. Since the counter V_t is incremented in the commit callback, it must be true that $W_t \geq c$ and $V_t < c$ until a failure occurs. We consider two cases.

- If no failure occurs, then the log record l is never used during recovery. Since the commit callback is never invoked for l, any transactions dependent on T continue to remain blocked, and the client issuing T is not sent a signed notification that the transaction was committed.
- If a failure occurs, then in the next recovery step, l is not used to reconstruct state since $c > V_t$. Since $V_t < W_t$, EnclaveDB creates a checkpoint that does not include l and increments the epoch counter before updating V_t and resuming transaction processing. Incrementing the epoch counter invalidates l.

Claim 5.6.4 ***Liveness.*** EnclaveDB *does not introduce new states in which execution terminates in the absence of an active attacker.*

We show that none of the assertions introduced by EnclaveDB fail in the absence of an active attacker.

- *Authentication failures.* If requests, responses, the log and checkpoints are not tampered with, and the database is not forked, none of the authentication checks fail.
- *Thread-level commit ordering failure* (line 46). Each thread in EnclaveDB generates a log record with counter value c only after a previous log record with counter value $c-1$ has been saved to storage. Therefore, in the absence of an active attacker, the recovery protocol should receive log records in a sequence that respects this ordering.
- *Mismatched epoch* (line 44). We prove this using induction. It is easy to see that this assertion cannot fail with $E = 0$. Assume this assertion does not fail during recovery when $E = i$. Consider any subsequent recovery which increments the epoch counter. In the absence of an active attacker, EnclaveDB initiates recovery with *start* and *end* indexes equal to the most recent truncation index and the tail of the log. This ensures that the entire log is truncated before the epoch counter is incremented. Since new log records are generated only after the counter is incremented and recovery completes, all subsequent log records will be generated with epoch counter $i+1$. Therefore, there will be no epoch mismatch when $E = i+1$.

5.7 Evaluation

In this section, we evaluate ENCLAVEDB. Due to the limited EPC size of 128 MB of current CPUs, we can only deploy small databases using SGX hardware. To evaluate performance for realistic database sizes, we use a performance model that simulates large enclaves and accounts for the main sources of overheads. In the following, we describe the performance model and present results from an evaluation using two standard database benchmarks.

5.7.1 Performance Model

To model SGX performance with larger enclaves, we assume that enclaves in future SGX CPUs will have the same performance as current CPUs except for (1) the additional cost of enclave transitions and (2) the additional cost incurred by last level cache (LLC) misses due to memory encryption and integrity checking while accessing the EPC. We model enclave transitions by introducing a system call to change protection of a pre-allocated page. This call flushes the TLB and adds a delay of ~10,000 cycles, which is approximately the cost of a transition measured on SGX hardware.

To model the cost of memory encryption, we considered the option of artificially reducing DRAM frequency and with that available memory bandwidth similar to Haven [23]. However, reducing frequency affects both enclave and non-enclave code, and does not accurately reflect the slowdown caused by memory encryption. Instead, we model the cost of memory encryption by using binary instrumentation to penalise all memory accesses generated by code within the enclave. Our instrumentation tool injects a fixed delay before every memory access within the enclave. We exclude accesses to the stack since stack memory is likely to be resident in the CPU caches. The amount of delay is obtained using a process of calibration on current SGX hardware.

We first measure the overhead of running a set of microbenchmarks using SGX enclaves. The delay is the lowest number of cycles such that running the same benchmarks after injecting the delay before every access on the same hardware *without enclaves* results in overhead higher than the overhead of running the application within the enclave. Our microbenchmarks consists of a simple key-value store [330] and a set of machine learning applications [241].

Using this calibration process, we find that our performance model with a delay of 10 cycles always over-approximates overheads of current SGX hardware. For example, SGX incurs an overhead of

37% for the key-value store whereas our performance model estimates the overhead to be 42%. To introduce this delay, we inject a single `pause` instruction, which delays the access by 10 cycles.

5.7.2 Benchmarks and Setup

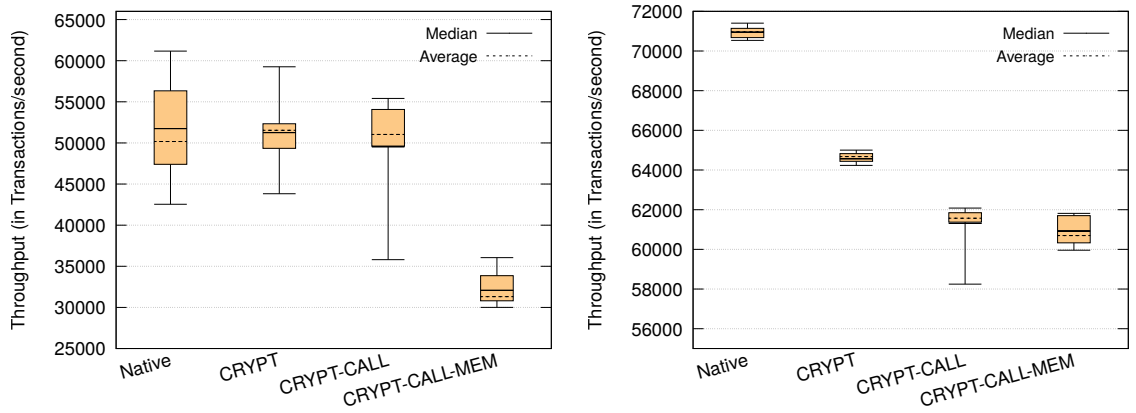
We evaluate the performance of EnclaveDB using two standard database benchmarks, TPC-C [332] and TATP [327]. The TPC-C benchmark represents the activity of a wholesale supplier that manages and sells products.

We use a database with 256 warehouses. This database has an in-memory size of 32 GB. The workload consists of a client driver running on two machines simulating 64 concurrent users each; each user executes five stored procedures in accordance with the TPC-C specification [332]. The in-memory size grows by ~6 GB/min during execution of this workload. The TATP benchmark simulates a typical location register database used by mobile carriers to store information about valid subscribers, including their mobile phone number and their current location. The database consists of 4 tables and 7 stored procedures. We create a database with 10 million subscribers. The client driver simulates 100 active subscribers querying and updating the database.

We optimised these benchmarks for use with an in-memory database by creating appropriately sized indexes to optimize query performance. The workload that we generate drives CPU utilisation close to 100% in the baseline configuration while leaving enough room for the checkpointing process to keep up with the rate of transaction commits. We run each benchmark 5 times for 20 minutes each and measure performance every minute.

We performed the experiments on Intel Xeon E7 servers running at 2.1 Ghz. The servers have 4 sockets with 8 cores each with hyperthreading disabled. Each CPU has an integrated memory controller with 4 memory channels attached to 8 32GB DDR4 DIMMs (2 DIMMs per channel), with a total capacity of 512 GB. The storage subsystem consists of 8 256 GB SSDs and is used for storing both the log and checkpoints. The servers run Windows Server 2016.

We evaluate each benchmark in four configurations: `BASE` is the configuration with an unmodified Hekaton engine running outside enclaves; `CRYPT` is a configuration with EnclaveDB running in simulated enclave mode; The model emulates enclaves within the application's address space by allocating a region of virtual memory and loading all enclave binaries in that part of the address space. In this configuration, all software security features such as log/checkpoint encryption and integrity



(a) TPC-C throughput

(b) TATP throughput

Fig. 5.5: TPC-C and TATP throughput of ENCLAVEDB in different configurations

checking are enabled; CRYPT-CALL is a configuration that adds the enclave transition cost to CRYPT; finally, CRYPT-CALL-MEM adds the cost of memory encryption to CRYPT-CALL. In all configurations except BASE, we simulate enclaves of size 192 GB. We configure the trusted kernel to use 128 threads. For each thread, we allocate a 64 KB stack. For both benchmarks, we consider all tables and stored procedures to be sensitive and host them in enclaves.

5.7.3 TPC-C

Fig. 5.5a shows the variation in TPC-C throughput for different configurations. In the baseline configuration, ENCLAVEDB achieves a mean throughput of 52,000 transactions per second (tps), which translates to 1.35 million new order transactions per minute (tpmC), with a peak of over 1.6 million tpmC. The throughput in both CRYPT and CRYPT-CALL configurations is statistically similar to the baseline. This suggests that the additional overheads of running the database with the trusted kernel, switching thread contexts, copying data in and out of the enclave, and encryption and integrity checking are negligible. We attribute this to our design which minimizes the number of context switches, amortizes the cost of encryption/decryption, and the efficient protocol for checking integrity of the log.

We also find that the variability in throughput is lower in the CRYPT configuration compared to baseline. This is due to two aspects of our design. First, virtual memory for the enclave is allocated at enclave creation and never returned to the host operating system. In contrast, the in-memory engine running outside the enclave periodically returns unused memory back to the SQL buffer pool, and

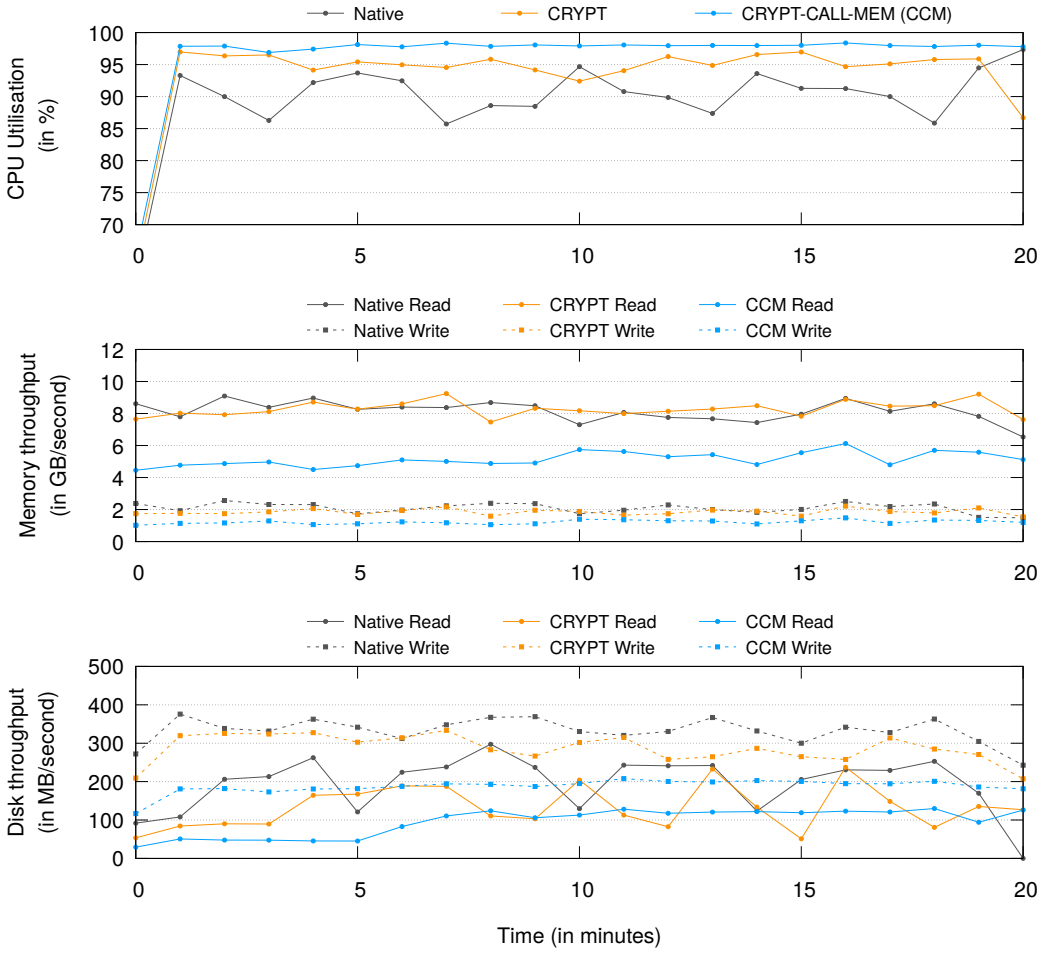


Fig. 5.6: Profiles comparing CPU, memory bandwidth and disk bandwidth utilisation for the TPC-C benchmark

must re-allocate memory when required. Secondly, the enclave thread scheduler yields control to the host less often compared to the baseline scheduler to reduce switching costs.

Finally, we observe that the CRYPT-CALL-MEM configuration, which models the cost of memory encryption, has a mean throughput of 31,000 tps, a drop of ~40% compared to baseline. Even with these overheads, throughput is over two orders of magnitude higher than prior work [23], which achieved a throughput of ~80 tpsE for the TPC-E benchmark on a 4-core machine using a similar performance model.

To further understand these overheads, we compare the configurations CRYPT and CRYPT-CALL-MEM with native execution across a number of other performance metrics. Fig. 5.6 shows the comparison over a 20-minute window with samples collected every 10 seconds and averaged every minute. All

configurations have high CPU utilisation (over 90%) on average, which suggests that CPU remains the main bottleneck. The periodic changes in CPU, memory, and disk bandwidth utilisation in BASE (and to a lesser extent in CRYPT) are caused by checkpointing. The similarity of memory and disk utilisation between BASE and CRYPT also confirms that the integrity checking protocol does not introduce new bottlenecks. We also observe that memory bandwidth utilisation is high (reaching 10 GBps), and memory reads dominate writes, whereas disk writes dominate reads. This is expected since most of the transaction processing occurs in-memory, whereas disk traffic is dominated by writes to the log. Finally, we observe that both memory and disk bandwidth utilisation in CRYPT-CALL-MEM configuration are significantly lower than baseline and CRYPT, caused by pause instructions which model memory encryption. Based on these observations, we conclude that if the future SGX hardware can deliver an effective memory read/write bandwidth of ~6 GBps and 2 GBps respectively with large enclaves, we can expect overheads of ~40%.

5.7.4 TATP

We deploy a scaled down version of the TATP benchmark (with 1000 subscribers) in ENCLAVEDB using SGX hardware. We were only able to run this benchmark for 40,000 transactions (4 client threads issuing 10,000 transactions each) before running out of enclave memory. For this scaled down workload, Native Hekaton execution achieves a peak throughput of 7,900 tps whereas ENCLAVEDB achieves a peak throughput of 7,700 tps, an overhead of 2.5%. This is, however, not a representative workload because of the small size of the database and short duration of the experiment.

Fig. 5.5b shows the throughput for the full TATP workload. Hekaton achieves a higher throughput of 71,000 tps with low variability compared to TPC-C because it is a predominantly read only workload (with 80% read transactions). The mean throughput reduces to ~65,000 tps after switching to the CRYPT configuration. Accounting for the costs of enclave transitions and memory encryption reduces the throughput further. We observe a mean throughput of 60,500 tps in the CRYPT-CALL-MEM configuration, an overhead of 15% relative to native execution.

As shown in Fig. 5.7, CPU remains the main bottleneck in the CRYPT-CALL-MEM configuration. Similar to TPC-C, memory reads dominate writes and disk writes dominate reads. However, both memory and disk bandwidth utilisation are lower compared to TPC-C, reflecting the predominantly read only nature of the workload. The memory read bandwidth for the CRYPT-CALL-MEM configura-

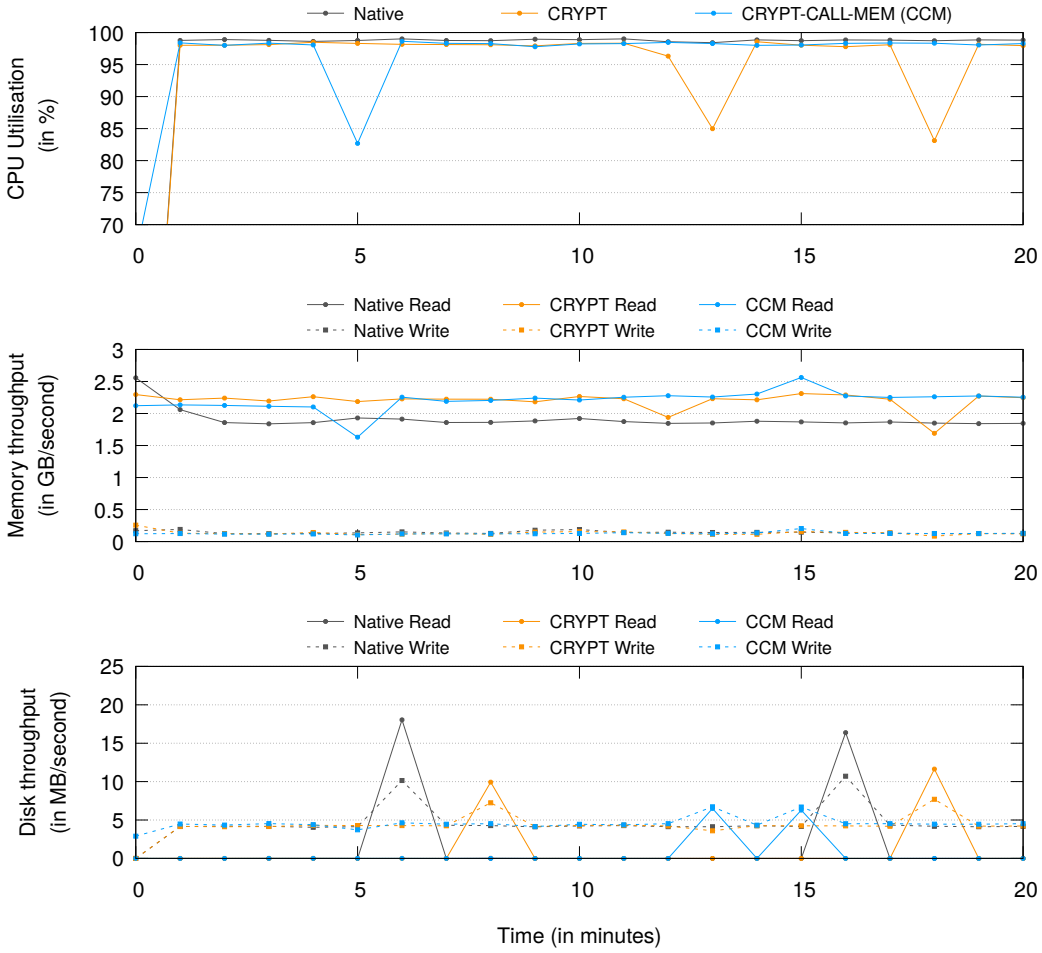


Fig. 5.7: Profiles comparing CPU, memory bandwidth and disk bandwidth utilisation for the TATP benchmark

tion is higher than baseline. This is because the additional memory traffic generated by EncloveDB (due to enclave transitions, encryption/decryption and integrity checking) dominates the reduction in utilisation caused by `pause` instructions. We also measured the end-to-end latency for this benchmark. We find an increase in average latency by 10%, 18% and 22% for CRYPT, CRYPT-CALL, and CRYPT-CALL-MEM configurations respectively over native execution.

Based on these experiments, we can conclude that EncloveDB achieves a desirable combination of strong security with confidentiality, integrity and freshness protection, and high performance.

5.7.5 TCB Size

We measure the size of the TCB for both benchmarks. The Hekaton engine is 300K LOC and the trusted kernel is 25K LOC. The queries and table definitions are 41K and 18K lines of auto-generated code in TPC-C and TATP respectively. In comparison, the SQL Server OLTP engine is 10M LOC and Windows is >100M LOC. Therefore, the TCB of ENCLAVEDB is over two orders of magnitude smaller than a conventional database. The main components that contribute to the TCB in ENCLAVEDB are checkpointing and recovery (~100K LOC) and the transaction manager (~50K LOC).

5.7.6 Enclave Transitions

The number of enclave transitions is an important indicator of overheads for applications using enclaves. We measured the number of transitions per transaction for both benchmarks. On average, TPC-C incurs 5 *ecalls* (4 for the commit protocol, and 1 call to serialize the log record). TPC-C also incurs 3 ocalls per transaction on average, a call to notify the host of the outcome of `TxPrepare`, a call to create a log record, and a number of other less frequent calls that occur periodically. TATP has a similar profile, with the difference that transitions due to logging are less frequent since 80% of the transactions are read only. In either case, the number of enclave transitions is independent of the database size in the enclave and the transaction logic (except in the case of read only transactions).

5.8 Related Work

Related work on advanced host interface attacks and mitigations has been previously discussed in Section 2.4.3.1. In this section, we explore related research on secure databases which falls into two broad categories: approaches based on homomorphic encryption and approaches using trusted hardware.

Homomorphic encryption. As discussed in Section 2.2.1.1, partially homomorphic encryption (PHE) schemes allow operations such as additions or multiplications to be efficiently applied to encrypted data without revealing plaintext operators or results. Systems such as CryptDB [262], Monomi [342], and Seabed [252] use PHE schemes to provide secure query processing while protecting the confidentiality

of data. This approach has also been adopted in several products [137, 225]. However, the types of queries supported by these systems are limited by the availability of corresponding encryption schemes or require augmentation, e.g. by offloading parts of the query execution to clients [152, 263, 342, 363]. Furthermore, these schemes have properties, e.g. order preservation, that provide weaker security than semantically secure encryption and can leak sensitive information [99, 143, 234].

Arx [260] introduces two new types of database indices for range and equality queries based on garbled circuits [371]. With these, it can support a similar set of queries as previous systems while using semantically secure encryption.

In contrast to these systems, EnclaveDB supports a broader set of queries, including arbitrary arithmetic, string manipulation, grouping and sorting, and uses strong probabilistic encryption. In addition, these systems solely focus on confidentiality, while EnclaveDB also provides integrity and freshness guarantees for both stored data and queries.

Trusted Hardware. Several database designs have been proposed that incorporate secure coprocessors [37, 231, 309] to process sensitive data securely. The data is stored in encrypted form on the host system and is only decrypted on the coprocessor as part of the query execution. However, currently available coprocessors are limited in terms of processing speed, storage, and bandwidth. TrustedDB [19] and Cipherbase [13] outsource computations on sensitive data to secure co-processors and FPGAs. These approaches require additional hardware and focus only on confidentiality, but do not guarantee the integrity or freshness of computation or data. They also suffer from high overheads due to the cost of data transfer over PCIe.

As discussed before, hosting the whole database service inside an enclave, e.g. using TEE runtimes such as SGX-LKL or Haven [23] results in a large trusted computing base (TCB) and increased performance overheads. In contrast to EnclaveDB, they also cannot provide protection from a malicious database administrator.

StealthDB [352], ObliDB [104], and STANlite [285] are other partitioned databases that use Intel SGX to protect stored data and query processing, developed after EnclaveDB. StealthDB aims for a minimal TCB by keeping all data outside of the enclave in encrypted form and only implementing individual operators inside the enclave. ObliDB and STANlite fully execute inside the enclave but store data in untrusted memory. Similar to EnclaveDB and StealthDB, ObliDB stores encrypted table

data on the untrusted host for durability but in addition implements oblivious operators to hide access patterns. STANlite is an in-memory database that implements a custom paging mechanism to store encrypted data in untrusted memory. In contrast to these databases, ENCLAVEDB provides both durability and full freshness protection of database data.

5.9 Summary

In this chapter, we presented ENCLAVEDB, a secure partitioned database with a small TCB and strong confidentiality, integrity, and freshness protection.

We presented ENCLAVEDB's architecture that is based on SQL Server's in-memory database engine Hekaton. We showed how such an in-memory engine is a good fit for the execution in TEEs as it keeps sensitive application state in memory at runtime which can be protected by the TEE. ENCLAVEDB achieves a small TCB by running only Hekaton inside the enclave and executing most of the database server outside. It further reduces the TCB size by using client-side native query compilation, which allows ENCLAVEDB to exclude components such as the query optimiser and query processor from its TCB.

ENCLAVEDB interacts with the host during transaction processing to execute stored procedures and to log transactions for durability. In contrast to SGX-LKL and GLAMDRING, presented in the previous two chapters, we extended the threat model to include freshness attacks such as replay and rollback attacks. We presented ENCLAVEDB's efficient logging and recovery protocol, which can support high transaction throughput while protecting confidentiality, integrity, and freshness of Hekaton log records stored as part of the database log on the untrusted host. We also presented an informal proof to show that the protocol achieves state continuity, integrity as well as liveness.

To evaluate ENCLAVEDB's performance, we introduced a performance model to emulate SGX overheads in order to estimate ENCLAVEDB's performance with large enclaves which will be available in the future. We evaluated ENCLAVEDB with two standard database benchmarks, TPC-C and TATP, and show that ENCLAVEDB has low overheads, with equal or less than 40% transaction throughput compared to native execution.

CHAPTER 6

CONCLUSION

TEEs have now become part of commodity CPUs and are made available in public cloud platforms. They allow tenants to make use of the typical benefits of cloud computing without giving up control over sensitive computation and data. However, TEE technologies such as Intel SGX also require a paradigm shift in application development and design.

TEEs enable small TCBs by isolating the execution of application code from the rest of the system. But this isolation requires TEE applications to either be self-contained or to cooperate with the host system. At the same time, TEEs are based on a threat model that assumes a powerful adversary with full control over the host. Existing software has not been developed taking into account a potentially malicious environment. This makes designing new or porting existing applications for the execution in TEEs and protecting their host interfaces a challenging and complex task.

In this thesis we covered some aspects of this task in the context of three systems. With SGX-LKL, we presented a TEE runtime system with a general-purpose host interface for securing untrusted binaries with a minimal host interface. With GLAMDRING, we discussed automated application partitioning, application-specific host interfaces and the relationship of competing partitioning objectives. With ENCLAVEDB, we presented a manually partitioned and optimised TEE application and discussed host interface protection against advanced attacks. In the remainder of this chapter, we summarise the thesis and its contributions and then conclude with a discussion of potential future work.

6.1 Thesis Summary

After giving an introduction and discussing the motivation for this thesis in Chapter 1, Chapter 2 provided background on the history of TEEs with technologies from both industry and academia, and compared them to alternatives such as homomorphic encryption and Trusted Computing. We described Intel SGX as one of the major TEEs in detail. We discussed the isolation provided by SGX enclaves, capabilities such as paging, sealing, and attestation, and its limitations. We also

analysed overheads incurred by SGX due to memory encryption, paging, and enclave transitions. Understanding these allows us to make informed design decisions for TEE systems such as those described in the later chapters.

We also covered TEE attacks and mitigations. TEEs such as Intel SGX assume a powerful adversary with complete control over the TEE environment. Being aware of the resulting risks such as side-channel attacks, enclave software vulnerabilities and interface-based attacks, as well as applicable mitigations is important when designing TEE systems and applications. In this thesis, we particularly focused on interface-based attacks as laid out in the threat model described at the end of the chapter. The *host interface* between the TEE and the untrusted host represents the main attack surface for TEE applications. Minimising and protecting the host interface is therefore vital for the security of sensitive application data.

In Chapter 3, we introduced SGX-LKL, a library OS based TEE runtime system with a *minimal general-purpose host interface*. With SGX-LKL, we addressed the question of how to design a minimal host interface while providing application-agnostic system support within an SGX enclave. The host interface only provides host calls for capabilities that cannot be provided within the TEE due to its inherent restrictions, e.g. enclave code being unprivileged and not having direct hardware access. This includes I/O operations, signal handling, and access to time.

In this chapter, we also started addressing the protection of the host interface. In particular, we discussed how a low-level interface simplifies the transparent protection of TEE applications. As part of SGX-LKL, we demonstrated how to provide block-level full disk encryption and integrity protection for disk I/O, and layer 3 encryption and integrity protection for network traffic.

Lastly, we looked at how to secure deployment, remote attestation, and secret provisioning. We discussed the deployment of distributed applications with SGX-LKL, based on (i) encrypted disk images containing application binaries and static data to protect their confidentiality and integrity; (ii) remote attestation to verify that SGX-LKL runs in genuine SGX enclaves and to establish secure communication channels, and (iii) a VPN between enclaves and other trusted nodes to provide initial secrets and to secure subsequent network communication between all nodes.

General-purpose host interfaces provided by TEE runtime systems such as SGX-LKL make it easy to protect complex applications by placing them completely inside a TEE. However, they also result in

large TCBs that include both application code and system support functionality. In contrast, Chapter 4 discussed *application-specific host interfaces* in the context of GLAMDRING, a semi-automated partitioning framework. By partitioning applications and following the principle of least privilege so that only sensitive parts of the application execute within the TEE, the TCB size can be reduced.

We also discussed the competitive relationship of the TCB size and the other partitioning objectives: low host interface complexity and few enclave transitions due to their performance overhead. Application-specific host interfaces allow developers to balance these trade-offs. For example, host interface complexity can be reduced by moving non-sensitive code into the TEE to remove a host call, for the cost of increasing the TCB size.

However, ensuring that balancing these trade-offs does not further compromise security, e.g. by moving sensitive code or data to the untrusted side, is non-trivial. We therefore presented how automated tooling can help with partitioning TEE applications while taking partitioning objectives and their trade-offs into account. As part of GLAMDRING, we discussed both dynamic and static analysis to track the flow of sensitive data and determine sensitive code that must be protected. They are also used to collect additional information such as performance metrics and call graphs that can facilitate partitioning decisions. We covered the benefits and drawbacks of both approaches, with dynamic analysis being capable of collecting runtime metrics but being workload-dependent, and static analysis being more conservative and workload-independent.

We discussed how these code analysis results can then be used to partition applications for TEE execution by transforming the optimisation problem of balancing the partitioning objectives into an ILP program. An ILP program can encode the analysis results, take additional weights for each objective into account, and can be solved automatically by an ILP solver. With GLAMDRING, we demonstrated how this can be used to generate a partition specification on which basis the application's codebase can be automatically transformed into a trusted TEE library and untrusted code.

We also covered the limitations of automated partitioning. Many optimisations to reduce TCB size, lower interface complexity, and improve performance, as well as extended interface hardening require knowledge of application semantics. In Chapter 5 we addressed this and discussed *protection against advanced interface-based attacks* for a manually partitioned high-performance database, ENCLAVEDB, with a small TCB.

We presented a TEE-aware design of EnclaveDB that excludes large components of a typical database server such as the query compiler and optimiser. The choice of an in-memory engine aligns well with the restrictions of current TEE technologies such as Intel SGX, as most processing can be done without host interaction. We also introduced a trusted kernel that provides system functionality inside the enclave if possible and only uses host calls to delegate functionality to the host OS if unavoidable, e.g. for storage, to achieve a small host interface.

We then presented a novel scalable logging and recovery protocol that protects the confidentiality, integrity, and freshness of durable database data that is persisted by the untrusted host. The protocol is responsible for logging committed database transactions to guarantee durability and linearizability, even in the face of failures. This is particularly relevant for TEEs since a malicious adversary in control of the host can induce failures on purpose. The protocol we presented in this chapter ensures that EnclaveDB can provide liveness in the face of non-malicious failures and prevent advanced attacks such as rollback attacks during recovery. For this, the protocol incorporates trusted monotonic counters to ensure freshness. We also demonstrated how to maintain scalability by using multiple counters concurrently and evaluated EnclaveDB's scalability using a custom performance model to approximate the cost of large SGX enclaves with memory-intensive workloads.

6.2 Future Work

While we have discussed a number of aspects covering the creation, deployment, and protection of TEE applications in this thesis, there are many challenges remaining in this field. In particular the introduction of Intel SGX has led to a large body of ongoing research on aspects such as TEE runtimes and libraries, performance optimisations, vulnerabilities, potential attacks, and attack mitigations which have been referenced throughout the thesis. In the following we will discuss potential future directions of research that relate to the work presented in the previous chapters.

Extended system support. With SGX-LKL we presented a solution for running existing unmodified binaries within TEEs. This is in particular useful for complex applications such as language runtimes that would be difficult to partition or rewrite for TEE execution. However, SGX-LKL has a number of limitations that reflect the limitations of current TEE technologies. For example, Intel SGX has no direct support for in-enclave signal handling requiring applications that rely on signals to expose them to the host system. Similarly, SGX provides no way of reliably determining time. Instead, enclave

code has to rely on the untrusted host which might rollback, slow down, or otherwise tamper with time information it provides to an application. Lastly, SGX enclaves are fundamentally bound to a single address space. Protecting multi-process applications or multiple applications that rely on inter-process communication cannot be support efficiently due to missing support for features such as protected shared memory and copy-on-write memory pages.

Some approaches exist to work around these problems, e.g. Roughtime [254] for secure clock synchronisation, or multi-process support based on custom IPC mechanisms in TEE runtime systems [301, 341]. However, complete mitigations will require additional support from TEE technologies themselves. A possible focus of future research could be on how to combine the generality of encrypted virtualisation approaches such as AMD SEV [185] and the strong isolation with confidentiality and integrity protection of TEEs such as Intel SGX, in order to offer extended system support. This could give applications more control over aspects such as signal handling, page table and memory management, and concurrency control. TEE runtime systems such as SGX-LKL will have to adapt accordingly.

Formal verification. With GLAMDRING and ENCLAVEDB we discussed two approaches of reducing the TCB size of TEE applications by partitioning them. The main benefit of a small TCB is the reduced risk of exploitable vulnerabilities within the TCB. However, a small TCB alone provides no guarantee for the correctness and quality of the contained application code. We have already discussed some mitigations against enclave software vulnerabilities in Chapter 2 such as the use of type-safe programming languages and enclave-specific approaches such as compiler instrumentation to add enclave bounds checking for memory reads and writes. A strong mitigation is offered by combining TCB reduction with formal verification. By formally verifying the trusted TEE code, it is possible to prove its correctness and security properties. While GLAMDRING uses information flow analysis to determine security-sensitive code and data in order to reduce the TCB, tools such as Moat [305] allow to formally verify that information flows within the TEE do not violate confidentiality properties. In the future, tools such as GLAMDRING and Moat could be combined to provide a comprehensive set of tooling around application partitioning and formal verification for TEEs.

Side-channel attacks and mitigations. In this thesis, we have provided an overview of TEE attacks and mitigations in Chapter 2 and presented solutions for protecting TEE applications from attackers. In particular, we addressed interface-based attacks including freshness attacks that we covered in

Chapter 5. However, we excluded side-channel attacks from our threat model. While side-channel attacks are not restricted to TEEs, side-channel attacks are of special concern for TEE applications for two reasons: (i) Simpler attacks such as privilege escalation attacks that are easier to mount than side-channel attacks against regular applications, do not allow an attacker to compromise TEE isolation. Side-channel attacks therefore are becoming more attractive; and (ii) the threat model of TEEs assume an attacker has full control over the host and with that is able to perform controlled-channel attacks [368] in which the adversary can use its control to reduce noise and with that simplify side-channel attacks.

There are multiple future research directions in the context of TEE side-channel attacks. With the recent discovery of a large number of side-channel attack vectors based on speculative execution and other CPU features that cause observable microarchitectural side effects, it will be necessary for future CPU generations to incorporate hardware mitigations. Academic CPU architecture proposals such as Sanctum [79] and MI6 [38] demonstrate how this could be achieved.

Furthermore, host calls made by a TEE application can become a potential attack vector. The amount and structure of data, access patterns, and timing patterns can all reveal potentially sensitive information to an observer. Minimising the interaction with the host via the host interface as done by SGX-LKL mitigates this problem. However, some host calls are necessary. Future work could explore making the host interface oblivious, e.g. by adding noise to disk and network I/O calls, or investigate secure I/O paths to trusted devices that can bypass an untrusted host kernel.

REFERENCES

[1] 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 *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 265–283, 2016.

[2] R. Abbott, J. Chin, Jed Donnelley, W. Konigsford, S. Tokubo, and D. Webb. Security Analysis and Enhancements of Computer Operating Systems. Technical Report, National Bureau of Standards, April 1974.

[3] Shaizeen Aga and Satish Narayanasamy. InvisiPage: Oblivious Demand Paging for Secure Enclaves. In *Proceedings of the 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019.

[4] Hiralal Agrawal and Joseph R. Horgan. Dynamic Program Slicing. In *Proceedings of the 11th ACM Conference on Programming Language Design and Implementation (PLDI)*, 246–256, 1990.

[5] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order Preserving Encryption for Numeric Data. In *Proceedings of the 30th ACM International Conference on Management of Data (SIGMOD)*, 563–574, 2004.

[6] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. Obliviate: A Data Oblivious File System for Intel SGX. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[7] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[8] Hasan Alayli. Lthread library. https://github.com/halayli/lthread, 2014. Last accessed 16th August 16 2019.

[9] Amazon. Encryption of Data at Rest. https://docs.aws.amazon.com/whitepapers/latest/efs-encrypted-file-systems/encryption-of-data-at-rest.html, 2018. Last accessed 10th March 2019.

[10] Amazon. Encryption of Data in Transit. https://docs.aws.amazon.com/whitepapers/latest/efs-encrypted-file-systems/encryption-of-data-in-transit.html, 2018. Last accessed 10th March 2019.

[11] AMD. *Secure Virtual Machine Architecture Reference Manual*, May 2005.

[12] 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 (HASP)*, 2013.

[13] Arvind Arasu, Spyros Blanas, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. Secure Database-as-a-service with Cipherbase. In *Proceedings of the 48th ACM International Conference on Management of Data (SIGMOD)*, 1033–1036, 2013.

[14] ARM. *ARM Security Technology - Building a Secure System using TrustZone Technology*. ARM Limited, 2009.

[15] 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 Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 689–703, 2016.

[16] T. W. Arnold, C. Buscaglia, F. Chan, V. Condorelli, J. Dayka, W. Santiago-Fernandez, N. Hadzic, M. D. Hocker, M. Jordan, T. E. Morris, and K. Werner. IBM 4765 Cryptographic Coprocessor. *IBM Journal of Research and Development*, 56(1.2):10:1–10:13, 2012.

[17] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. LibSEAL: Revealing Service Integrity Violations Using Trusted Execution. In *Proceedings of the 13th ACM European Conference on Computer Systems (EuroSys)*, 24:1–24:15, 2018.

[18] AWS. Firecracker. https://firecracker-microvm.github.io/. Last accessed 10th March 2019.

[19] Sumeet Bajaj and Radu Sion. TrustedDB: A Trusted Hardware Based Database with Privacy and Data Confidentiality. In *Proceedings of the 46th ACM International Conference on Management of Data (SIGMOD)*, 205–216, 2011.

[20] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (Im)Possibility of Obfuscating Programs. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 1–18, 2001.

[21] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (Im)Possibility of Obfuscating Programs. *Journal of the ACM (JACM)*, 59(2):6:1–6:48, May 2012.

[22] Erick Bauman, Huibo Wang, Mingwei Zhang, and Zhiqiang Lin. SGXElide: Enabling Enclave Code Secrecy via Self-modification. In *Proceedings of the 16th IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 75–86, 2018.

[23] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 267–283, 2014.

[24] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on Steroids: SGX-Based High Performance BFT. In *Proceedings of the 12th ACM European Conference on Computer Systems (EuroSys)*, 222–237, 2017.

[25] Josh Benaloh. Dense Probabilistic Encryption. In *Proceedings of the 1st Workshop on Selected Areas of Cryptography (SAC)*, 120–128, 1994.

[26] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-Policy Attribute-Based Encryption. In *Proceedings of the 28th IEEE Symposium on Security and Privacy (SP)*, 321–334, 2007.

[27] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Board Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium (USENIX Security)*, 8–8, 2003.

[28] 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 (PACT)*, 2008.

[29] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 1213–1227, August 2018.

[30] 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)*, 309–322, 2008.

[31] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, 169–190, 2006.

[32] Rick Boivie and Peter Williams. SecureBlue++: CPU Support for Secure Execution. Technical Report, IBM, May 2012.

[33] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-Preserving Symmetric Encryption. In *Proceedings of the 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, 224–241, 2009.

[34] Dan Boneh and Matthew K. Franklin. Identity-Based Encryption from the Weil Pairing. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 213–229, 2001.

[35] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public Key Encryption with Keyword Search. In *Proceedings of the 23th Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, 506–522, 2004.

[36] Dan Boneh, Amit Sahai, and Brent Waters. Functional Encryption: Definitions and Challenges. In *Proceedings of the 8th IACR Theory of Cryptography Conference (TCC)*, 253–273, 2011.

[37] Luc Bouganim and Philippe Pucheral. Chip-secured Data Access: Confidential Data on Untrusted Servers. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, 131–142, 2002.

[38] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. MI6: Secure Enclaves in a Speculative Out-of-Order Processor, 2018, `arXiv:1812.09822 [cs.CR]`.

[39] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[40] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully Homomorphic Encryption without Bootstrapping. Cryptology ePrint Archive, Report 2011/277, 2011. https://eprint.iacr.org/2011/277.

[41] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) Fully Homomorphic Encryption without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS)*, 309–325, 2012.

[42] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. Rollback and Forking Detection for Trusted Execution Environments Using Lightweight Collective Memory. In *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 157–168, 2017.

[43] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[44] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*, 12 2019.

[45] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. SANCTUARY: ARMing TrustZone with User-space Enclaves. In *Proceedings of the 26nd Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[46] Stefan Brenner and Rüdiger Kapitza. Trust More, Serverless. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR)*, 33–43, 2019.

[47] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. SecureKeeper: Confidential ZooKeeper Using Intel SGX. In *Proceedings of the 17th International Middleware Conference (Middleware)*, 2016.

[48] Stefan Brenner, David Goltzsche, and Rüdiger Kapitza. TrApps: Secure Compartments in the Evil Cloud. In *Proceedings of the 1st International Workshop on Security and Dependability of Multi-Domain Infrastructures (XDOM0)*, 5, 2017.

[49] Ernie Brickell and Jiangtao Li. Enhanced privacy ID from bilinear pairing for hardware authentication and attestation. In *Proceedings of the 2nd IEEE International Conference on Social Computing (SocialCom)*, 2010.

[50] David Brumley and Dawn Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th USENIX Security Symposium (USENIX Security)*, 5, 2004.

[51] Sven Bugiel, Stefan Nürnberger, Ahmad-Reza Sadeghi, and Thomas Schneider. Twin Clouds: Secure Cloud Computing with Low Latency. In *Proceedings of the 12th IFIP International Conference on Communications and Multimedia Security (CMS)*, 32–44, 2011.

[52] D. Burihabwa, P. Felber, H. Mercier, and V. Schiavoni. SGX-FS: Hardening a File System in User-Space with Intel SGX. In *Proceedings of the 10th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 67–72, 2018.

[53] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, and et al. Fallout: Leaking Data on Meltdown-resistant CPUs. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, 769–784, 2019.

[54] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, 249–266, 2019.

[55] Amy Carroll, Mario Juarez, Julia Polk, and Tony Leininger. Microsoft Palladium: A Business Overview. *Microsoft Content Security Business Unit*, 1–9, 2002.

[56] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-Abuse Attacks Against Searchable Encryption. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 668–679, 2015.

[57] 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 ACM Asia-Pacific Workshop on Systems (APSys)*, 16:1–16:7, 2017.

[58] David Champagne. *Scalable Security Architecture for Trusted Software*. PhD Thesis, Princeton University, Princeton, NJ, USA, 2010. AAI3410862.

[59] Swarup Chandra, Vishal Karande, Zhiqiang Lin, Latifur Khan, Murat Kantarcioglu, and Bhavani Thuraisingham. Securing data analytics on sgx with randomization. In *Proceedings of the 22nd European Symposium on Research in Computer Security (ESORICS)*, 2017.

[60] Yan-Cheng Chang and Michael Mitzenmacher. Privacy Preserving Keyword Searches on Remote Encrypted Data. In *Proceedings of the 3rd International Conference on Applied Cryptography and Network Security (ACNS)*, 442–455, 2005.

[61] Melissa Chase. Multi-authority Attribute Based Encryption. In *Proceedings of the 4th IACR Theory of Cryptography Conference (TCC)*, 515–534, 2007.

[62] Stephen Checkoway and Hovav Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 253–264, 2013.

[63] Chen Chen, Himanshu Raj, Stefan Saroiu, and Alec Wolman. cTPM: A Cloud TPM for Cross-device Trusted Applications. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 187–201, 2014.

[64] 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 *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP)*, 178–194, 2018.

[65] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SgxPectre: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *Proceedings of the 4th IEEE European Symposium on Security and Privacy (EuroSP)*, 2019.

[66] Lixia Chen, Jian Li, Ruhui Ma, Haibing Guan, and Hans-Arno Jacobsen. EnclaveCache: A Secure and Scalable Key-Value Cache in Multi-Tenant Clouds Using Intel SGX. In *Proceedings of the 20th International Middleware Conference (Middleware)*, 14–27, 2019.

[67] 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 12th ACM ASIA Conference on Computer and Communications Security (AsiaCCS)*, 2017.

[68] Sanchuan Chen, Fangfei Liu, Zeyu Mi, Yinqian Zhang, Ruby B. Lee, Haibo Chen, and XiaoFeng Wang. Leveraging Hardware Transactional Memory for Cache Side-Channel Defenses. In *Proceedings of the 13th ACM ASIA Conference on Computer and Communications Security (AsiaCCS)*, 601–608, 2018.

[69] 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 *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2–13, 2008.

[70] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Proceedings of the 23rd International Conference on the Theory and Application of Cryptology and Information Security (Asiacrypt)*, 409–437, 2017.

[71] Siddhartha Chhabra, Brian Rogers, Yan Solihin, and Milos Prvulovic. SecureME: A Hardware-software Approach to Full System Security. In *Proceedings of the 25th ACM International Conference on Supercomputing (ICS)*, 108–119, 2011.

[72] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Proceedings of the 22nd International Conference on the Theory and Application of Cryptology and Information Security (Asiacrypt)*, 3–33, 2016.

[73] Catalin Cimpanu. Hackers Modify Water Treatment Parameters by Accident. http://news.softpedia.com/news/hackers-modify-water-treatment-parameters-by-accident-502043.shtml, 2016. Last accessed 9th February 2020.

[74] Ciphercloud. http://www.ciphercloud.com. Last accessed 9th Feburary 2020.

[75] Clang. clang: a C language family frontend for LLVM. http://clang.llvm.org.

[76] Confidential Computing Consortium. https://confidentialcomputing.io/. Last accessed 10th March 2019.

[77] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 143–154, 2010.

[78] Manuel Costa, Lawrence Esswood, Olga Ohrimenko, Felix Schuster, and Sameer Wagh. The Pyramid Scheme: Oblivious RAM for Trusted Processors, 2017, `arXiv:1712.07882 [cs.CR]`.

[79] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, 2016.

[80] Michael Coughlin, Eric Keller, and Eric Wustrow. Trusted Click: Overcoming Security Issues of NFV in the Cloud. In *Proceedings of the 2nd ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFVSec)*, 2017.

[81] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 81–96, 2014.

[82] Cryptsetup. https://gitlab.com/cryptsetup/cryptsetup. Last accessed 16th August 2019.

[83] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A Software Analysis Perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods (SEFM)*, 233–247, 2012.

[84] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. CacheQuote: Efficiently Recovering Long-Term Secrets of SGX EPID via Cache Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 171–191, 2018.

[85] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Proceedings of the 32nd Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 643–662, 2012.

[86] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 10–10, 2004.

[87] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[88] Device Mapper. Linux Kernel device mapper framework. https://www.kernel.org/doc/Documentation/device-mapper. Last accessed 7th August 2019.

[89] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *Proceedings of the 48th ACM International Conference on Management of Data (SIGMOD)*, 1243–1254, 2013.

[90] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. M2R: Enabling Stronger Privacy in MapReduce Computation. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, 447–462, August 2015.

[91] DMCrypt. https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt. Last accessed 16th August 2019.

[92] DMIntegrity. https://gitlab.com/cryptsetup/cryptsetup/wikis/DMIntegrity. Last accessed 16th August 2019.

[93] DMVerity. https://gitlab.com/cryptsetup/cryptsetup/wikis/DMVerity. Last accessed 16th August 2019.

[94] Docker. Enterprise Container Platform for High-Velocity Innovation. https://www.docker.com. Last accessed 16th August 2019.

[95] Huayi Duan, Cong Wang, Xingliang Yuan, Yajin Zhou, Qian Wang, and Kui Ren. LightBox: Full-stack Protected Stateful Middlebox at Lightning Speed, 2017, `arXiv:1706.06261 [cs.CR]`.

[96] Léo Ducas and Daniele Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In *Proceedings of the 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, 617–640, 2015.

[97] Loıc Duflot, Daniel Etiemble, and Olivier Grumelard. Using CPU System Management Mode to Circumvent Operating System Security Functions. *CanSecWest*, 2006.

[98] A. J. Duncan, S. Creese, and M. Goldsmith. Insider Attacks in Cloud Computing. In *Proceedings of the 33rd IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 857–862, 6 2012.

[99] F. Betül Durak, Thomas M. DuBuisson, and David Cash. What Else is Revealed by Order-Revealing Encryption? In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, 1155–1166, 2016.

[100] Morris J. Dworkin. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Technical Report SP 800-38D, NIST, Gaithersburg, MD, USA, November 2007.

[101] Reouven Elbaz, David Champagne, Ruby B. Lee, Lionel Torres, Gilles Sassatelli, and Pierre Guillemin. TEC-Tree: A Low-Cost, Parallelizable Tree for Efficient Defense Against Memory Replay Attacks. In *Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 289–302, 2007.

[102] Taher ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *Proceedings of the 4th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 10–18, 1985.

[103] Shawn Embleton, Sherri Sparks, and Cliff Zou. SMM Rootkit: A New Breed of OS Independent Malware. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks (SecureComm)*, 2008.

[104] Saba Eskandarian and Matei Zaharia. ObliDB: Oblivious Query Processing for Secure Databases. *Proc. VLDB Endow.*, 13(2):169–183, October 2019.

[105] Rich Felker et al. musl libc. https://www.musl-libc.org. Last accessed 7th August 2019.

[106] Dmitry Evtyushkin, 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 IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[107] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 693–707, 2018.

[108] Junfeng Fan and Frederik Vercauteren. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144, 2012. https://eprint.iacr.org/2012/144.

[109] Susanne Felsen, Ágnes Kiss, Thomas Schneider, and Christian Weinert. Secure and Private Function Evaluation with Intel SGX. In *Proceedings of the 11th ACM Workshop on Cloud Computing Security (CCSW)*, 2019.

[110] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[111] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[112] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. IRON: Functional Encryption Using Intel SGX. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, 2017.

[113] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. A Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *Proceedings of the 7th ACM Workshop on Scalable Trusted Computing (STC)*, 2012.

[114] Fortanix. Enclave Development Platform. https://edp.fortanix.com/. Last accessed 10th March 2019.

[115] Frama-C Impact Analysis Plug-In. http://frama-c.com/impact.html. Last accessed 9th February 2020.

[116] Frama-C Slicing Plug-In. http://frama-c.com/slicing.html. Last accessed 9th February 2020.

[117] Frama-C Software Analyzers. http://frama-c.com/what_is.html. Last accessed 9th February 2020.

[118] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. Sgx-Lapd: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, volume 10453 of *Lecture Notes in Computer Science*, 357–380, 2017.

[119] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-based Platform for Trusted Computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 193–206, 2003.

[120] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate Indistinguishability Obfuscation and Functional Encryption for All Circuits. In *Proceedings of the 54th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 40–49, 2013.

[121] Sanjam Garg, Craig Gentry, Shai Halevi, Amit Sahai, and Brent Waters. Attribute-based Encryption for Circuits from Multilinear Maps. In *Proceedings of the 33rd Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 479–499, 2013.

[122] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon Physical Random Functions. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, 148–160, 2002.

[123] Dimitris Geneiatakis, Georgios Portokalidis, Vasileios P. Kemerlis, and Angelos D. Keromytis. Adaptive Defenses for Commodity Software through Virtual Application Partitioning. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, 133–144, 2012.

[124] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers. In *Proceedings of the 30th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 465–482, 2010.

[125] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD Thesis, Stanford University, 2009. crypto.stanford.edu/craig.

[126] Craig Gentry. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC)*, 169–178, 2009.

[127] Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the AES circuit. In *Proceedings of the 32nd Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 850–867, 2012. https://eprint.iacr.org/2012/099.

[128] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*, 182–194, 1987.

[129] Shafi Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC)*, 365–377, 1982.

[130] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.

[131] Shafi Goldwasser, Yael Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable Garbled Circuits and Succinct Functional Encryption. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC)*, 555–564, 2013.

[132] Shafi Goldwasser, Yael Tauman Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. How to run turing machines on encrypted data. In *Proceedings of the 33rd Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 536–553, 2013.

[133] Philippe Golle, Jessica Staddon, and Brent Waters. Secure Conjunctive Keyword Search Over Encrypted Data. In *Proceedings of the 2nd International Conference on Applied Cryptography and Network Security (ACNS)*, 31–45, 2004.

[134] D. Goltzsche, S. Rüsch, M. Nieke, S. Vaucher, N. Weichbrodt, V. Schiavoni, P. Aublin, P. Cosa, C. Fetzer, P. Felber, P. Pietzuch, and R. Kapitza. EndBox: Scalable Middlebox Functions Using Client-Side Trusted Execution. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 386–397, 2018.

[135] Google. Encryption at Rest in Google Cloud Platform. https://cloud.google.com/security/encryption-at-rest/default-encryption/, 2017. Last accessed 10th March 2019.

[136] Google. Encryption in Transit in Google Cloud. https://cloud.google.com/security/encryption-in-transit, 2017. Last accessed 10th March 2019.

[137] Google. Encrypted BigQuery client. https://github.com/google/encrypted-bigquery-client, 2018. Last accessed 9th February 2020.

[138] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Attribute-based Encryption for Circuits. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC)*, 545–554, 2013.

[139] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec)*, 2017.

[140] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based Encryption for Fine-grained Access Control of Encrypted Data. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, 89–98, 2006.

[141] Graphene-SGX Source Code. Graphene Library OS with Intel Registered SGX Support. https://github.com/oscarlab/graphene. Last accessed 7th August 2019.

[142] James Greene. Intel Trusted Execution Technology. *White Paper*, 2012.

[143] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-Abuse Attacks against Order-Revealing Encryption. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP)*, 2017.

[144] Nils Gruschka and Meiko Jensen. Attack Surfaces: A Taxonomy for Attacks on Cloud Services. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing (IEEE CLOUD)*, 276–279, 2010.

[145] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, 217–233, 2017.

[146] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP)*, 245–261, 2017.

[147] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 488–501, 2017.

[148] 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 *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 1016–1031, 2015.

[149] Shay Gueron. A Memory Encryption Engine Suitable for General Purpose Processors. Cryptology ePrint Archive, Report 2016/204, 2016. https://eprint.iacr.org/2016/204.

[150] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, and Vincentius a nd others Martin. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 1–14, 2014.

[151] Gurobi Optimization, Inc. *Gurobi Optimizer Reference Manual 6.0*, https://www.gurobi.com/documentation/6.0/refman/index.html, 2014.

[152] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing SQL Over Encrypted Data in the Database-Service-Provider Model. In *Proceedings of the 28th ACM International Conference on Management of Data (SIGMOD)*, 216–227, 2002.

[153] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-Resolution Side Channels for Untrusted Operating Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, July 2017.

[154] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest We Remember: Cold-Boot Attacks on Encryption Keys. *Communications of the ACM*, 52(5):91–98, 2009.

[155] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*, 49–64, 2013.

[156] Juhyeng Han, Seongmin Kim, Jaehyeong Ha, and Dongsu Han. SGX-Box: Enabling Visibility on Encrypted Traffic Using a Secure Middlebox Module. In *Proceedings of the 1st ACM Asia-Pacific Workshop on Networking (APNet)*, 99–105, 2017.

[157] Warren He, Devdatta Akhawe, Sumeet Jain, Elaine Shi, and Dawn Song. ShadowCrypt: Encrypted Web Applications for Everyone. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, 1028–1039, 2014.

[158] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[159] Felicitas Hetzelt and Robert Buhren. Security Analysis of Encrypted Virtual Machines. In *Proceedings of the 13th ACM International Conference on Virtual Execution Environments (VEE)*, 129–142, 2017.

[160] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila A Yavuz. Hardware-Supported ORAM in Effect: Practical Oblivious Search and Update on Very Large Dataset. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2019(1):172–191, 2019.

[161] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[162] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 265–278, 2013.

[163] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. In *Proceedings of the 9th ACM Conference on Programming Language Design and Implementation (PLDI)*, 35–46, 1988.

[164] Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, and Andrew Paverd. Mitigating Branch-Shadowing Attacks on Intel SGX Using Control Flow Randomization. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTex)*, 2018.

[165] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. vTZ: Virtualizing ARM TrustZone. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, 541–556, August 2017.

[166] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster Secure Two-party Computation Using Garbled Circuits. In *Proceedings of the 20th USENIX Security Symposium (USENIX Security)*, 2011.

[167] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*, 11, 2010.

[168] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[169] Identity Theft Resource Center. 2016 Breach List. http://www.idtheftcenter.org/images/breach/ITRCBreachReport_2016.pdf, 2016.

[170] Intel. Intel SGX SDK. https://software.intel.com/sgx/sdk, . Last accessed 10th March 2019.

[171] Intel. Improving Real-Time Performance by Utilizing Cache Allocation Technology. *White Paper*, 2015.

[172] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3C*. Intel Corporation, https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf, 9 2016.

[173] Intel. Overview of Intel Protected File System Library Using Software Guard Extensions. https://software.intel.com/en-us/articles/overview-of-intel-protected-file-system-library-using-software-guard-extensions, 12 2016. Last accessed 4th November 2019.

[174] Intel. INTEL-SA-00161. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html, 2018. Last accessed 10th August 2019.

[175] Intel. Intel Software Guard Extensions (SGX) Protected Code Loader (PCL) for Linux OS. https://github.com/intel/linux-sgx-pcl, 2018. Last accessed 16th November 2019.

[176] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3D*. Intel Corporation, https://software.intel.com/sites/default/files/managed/7c/f1/332831-sdm-vol-3d.pdf, 8 2019.

[177] Intel. Getting Started with Intel Active Management Technology (Intel AMT). https://software.intel.com/en-us/articles/getting-started-with-intel-active-management-technology-amt, 9 2019. Last accessed 4th November 2019.

[178] Intel. *Intel Architecture Memory Encryption Technologies Specification*. Intel, https://software.intel.com/sites/default/files/managed/a5/16/Multi-Key-Total-Memory-Encryption-Spec.pdf, April 2019.

[179] IPerf3. https://iperf.fr. Last accessed 7th August 2019.

[180] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (SP)*, 591–604, 2015.

[181] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, 621–637, August 2019.

[182] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. B. Kang. PrivateZone: Providing a Private Execution Environment Using ARM TrustZone. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 15(5):797–810, 2018.

[183] 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 (SysTex)*, 5, 2017.

[184] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 295–308, 2008.

[185] David Kaplan, Jeremy Powell, and Tom Woller. *AMD Memory Encryption*, https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, April 2016.

[186] Kata. Kata Containers. https://katacontainers.io. Last accessed 10th March 2019.

[187] Richard E Kessler and Mark D Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)*, 10(4):338–359, 1992.

[188] Keystone. https://keystone-enclave.org/. Last accessed 10th November 2019.

[189] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. ShieldStore: Shielded In-memory Key-value Storage with SGX. In *Proceedings of the 14th ACM European Conference on Computer Systems (EuroSys)*, 14:1–14:15, 2019.

[190] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA)*, 361–372, 2014.

[191] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)*, 1–19, 2019.

[192] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *Proceedings of the 12th USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[193] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. Pesos: Policy Enhanced Secure Object Store. In *Proceedings of the 13th ACM European Conference on Computer Systems (EuroSys)*, 2018.

[194] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The CIFAR-10 dataset. h t t p s : //www.cs.toronto.edu/~kriz/cifar.html. Last accessed 7th August 2019.

[195] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th Annual Conference on Advances in Neural Information Processing Systems (NIPS)*, 1097–1105, 2012.

[196] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the 12th ACM European Conference on Computer Systems (EuroSys)*, 2017.

[197] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 147–163, 2014.

[198] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium (USENIX Security)*, 2001.

[199] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 75, 2004.

[200] 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 *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, August 2017.

[201] Ruby B. Lee, Peter C. S. Kwan, John P. McGregor, Jeffrey Dwoskin, and Zhenghong Wang. Architecture for Protecting Critical Secrets in Microprocessors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, 2–13, 2005.

[202] Sangho Lee and Taesoo Kim. Leaking Uninitialized Secure Enclave Memory via Structure Padding, 2017, `arXiv:1710.09061 [cs.CR]`.

[203] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, August 2017.

[204] Hongliang Liang, Mingyu Li, Qiong Zhang, Yue Yu, Lin Jiang, and Yixiu Chen. Aurora: Providing Trusted System Services for Enclaves On an Untrusted System, 2018, `arXiv:1802.03530 [cs.CR]`.

[205] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 178–192, 2003.

[206] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. Glamdring: Automatic Application Partitioning for Intel SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, 285–298, 2017.

[207] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 973–990, 2018.

[208] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 87–101, 2015.

[209] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 1607–1619, 2015.

[210] LKML. WireGuard: Secure Network Tunnel. https://lkml.org/lkml/2019/3/22/95. Last accessed 16th August 2019.

[211] Yinghai Lu, Li-Ta Lo, Gregory R Watson, and Ronald G Minnich. CAR: Using Cache as RAM in LinuxBIOS. Technical Report, September 2006.

[212] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical Oblivious Computation in a Secure Processor. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

[213] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback Protection for Trusted Execution. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, 1289–1306, 2017.

[214] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proceedings of the 3rd ACM European Conference on Computer Systems (EuroSys)*, 315–328, 2008.

[215] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (SP)*, 143–158, 2010.

[216] David A. McGrew and John Viega. The Security and Performance of the Galois/Counter Mode (GCM) of Operation. In *Proceedings of the 5th International Conference on Cryptology in India (INDOCRYPT)*, 343–355, 2004.

[217] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[218] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[219] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel Software Guard Extensions (Intel SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of the 5th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 10:1–10:9, 2016.

[220] Memcached. What is Memcached? https://memcached.org. Last accessed 16th August 2019.

[221] Ralph Charles Merkle. *Secrecy, Authentication, and Public Key Systems*. PhD Thesis, Stanford University, 1979. AAI8001972.

[222] Microsoft. SQL Server. https://www.microsoft.com/en-in/sql-server. Last accessed 29th November 2019.

[223] Microsoft. Encryption of data in transit. https://docs.microsoft.com/en-us/azure/security/fundamentals/encryption-overview#encryption-of-data-in-transit, 2018. Last accessed 11th December 2019.

[224] Microsoft. Azure Data Encryption-at-Rest. https://docs.microsoft.com/en-us/azure/security/fundamentals/encryption-atrest, 10 2019. Last accessed 11th December 2019.

[225] Microsoft. Always Encrypted Database Engine. https://msdn.microsoft.com/en-us/library/mt163865.aspx, 2019. Last accessed 9th Feburary 2020.

[226] Subhas C. Misra and Virendra C. Bhavsar. Relationships Between Selected Software Measures and Latent Bug-density: Guidelines for Improving Quality. In *Proceedings of the 3rd International Conference on Computational Science and Its Applications (ICCSA)*, 724–732, 2003.

[227] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *Proceedings of the 19th Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2017.

[228] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations. *International Journal of Parallel Programming*, 47(4):538–570, August 2019.

[229] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. SEVered: Subverting AMD's Virtual Machine Encryption. In *Proceedings of the 11th European Workshop on Systems Security (EuroSec)*, 1:1–1:6, 2018.

[230] Divya Muthukumaran, Dan O'Keeffe, Christian Priebe, David Eyers, Brian Shand, and Peter Pietzuch. FlowWatcher: Defending Against Data Disclosure Vulnerabilities in Web Applications. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 603–615, 2015.

[231] Einar Mykletun and Gene Tsudik. Incorporating a Secure Coprocessor in the Database-as-a-Service Model. In *Proceedings of the 2005 Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA)*, 2005.

[232] MySQL. http://www.mysql.com/. Last accessed 9th February 2020.

[233] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can Homomorphic Encryption Be Practical? In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security (CCSW)*, 113–124, 2011.

[234] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference Attacks on Property-Preserving Encrypted Databases. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 644–655, 2015.

[235] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM Conference on Programming Language Design and Implementation (PLDI)*, 89–100, 2007.

[236] Netwrix. 2019 Netwrix Cloud Data Security Report. https://www.netwrix.com/2019cloudsecurityreport.html, 2019.

[237] James Newsome and Dawn Xiaodong Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS)*, 2007.

[238] H. Nguyen and V. Ganapathy. EnGarde: Mutually-Trusted Inspection of SGX Enclaves. In *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[239] OE. Open Enclave SDK. https://openenclave.io/sdk/. Last accessed 10th March 2019.

[240] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and Preventing Leakage in MapReduce. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 1570–1581, 2015.

[241] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, 619–636, 2016.

[242] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018.

[243] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.0, 2013.

[244] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the 12th ACM European Conference on Computer Systems (EuroSys)*, 238–253, 2017.

[245] Meni Orenbach, Yan Michalevsky, Christof Fetzer, and Mark Silberstein. CoSMIX: A Compiler-based System for Secure Memory Instrumentation and Execution in Enclaves. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 555–570, July 2019.

[246] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 1–20, 2006.

[247] Karl J. Ottenstein and Linda M. Ottenstein. The Program Dependence Graph in a Software Development Environment. In *Proceedings of the 1st ACM Software Engineering Symposium on Practical Software Development Environments (SESPSDE)*, 177–184, 1984.

[248] Emmanuel Owusu, Jorge Guajardo, Jonathan McCune, Jim Newsome, Adrian Perrig, and Amit Vasudevan. OASIS: On Achieving a Sanctuary for Integrity and Secrecy on Untrusted Platforms. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 13–24, 2013.

[249] Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Proceedings of the 18th Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, 223–238, 1999.

[250] HweeHwa Pang and Kian-Lee Tan. Authenticating Query Results in Edge Computing. In *Proceedings of the 20th IEEE International Conference on Data Engineering (ICDE)*, 2004.

[251] Panoply. Panoply: Low-TCB Linux Applications with SGX Enclaves. https://github.com/shwetasshinde24/Panoply. Last accessed 7th August 2019.

[252] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, and Abhishek Modi. Big Data Analytics over Encrypted Datasets with Seabed. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI))*, 2016.

[253] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (SP)*, 238–252, 2013.

[254] Christopher Patton. Roughtime: Securing Time with Digital Signatures. https://blog.cloudflare.com/roughtime. Last accessed 9th March 2020.

[255] Marcus Peinado, Yuqun Chen, Paul England, and John Manferdelli. NGSCB: A Trusted Open System. In *Proceedings of the 9th Australasian Conference on Information Security and Privacy (ACISP)*, 86–97, 2004.

[256] Colin Percival. Cache Missing for Fun and Profit, BSDCan, 2005.

[257] Ronald Perez, Reiner Sailer, Leendert van Doorn, et al. vTPM: Virtualizing the Trusted Platform Module. In *Proceedings of the 15th USENIX Security Symposium (USENIX Security)*, 305–320, 2006.

[258] Travis Peters, Reshma Lal, Srikanth Varadarajan, Pradeep Pappachan, and David Kotz. BASTION-SGX: Bluetooth and Architectural Support for Trusted I/O on SGX. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 3:1–3:9, 2018.

[259] Rafael Pires, Marcelo Pasin, Pascal Felber, and Christof Fetzer. Secure Content-Based Routing Using Intel Software Guard Extensions. In *Proceedings of the 17th International Middleware Conference (Middleware)*, 2016.

[260] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: A Strongly Encrypted Database System. *IACR Cryptology ePrint Archive*, 2016:591, 2016.

[261] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. SafeBricks: Shielding Network Functions in the Cloud. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2018.

[262] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 85–100, 2011.

[263] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan. Building Web Applications on Top of Encrypted Data Using Mylar. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[264] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the Library OS from the Top Down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 291–304, 2011.

[265] Christian Priebe. Semi-Automated Partitioning of Applications for the Execution on Trusted Hardware. Master's Thesis, Imperial College London, 2015.

[266] Christian Priebe, Kapil Vaswani, and Manuel Costa. EnclaveDB: A Secure Database Using SGX. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP)*, 264–278, 2018.

[267] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, and Peter Pietzuch. SGX-LKL: Securing the Host OS Interface for Trusted Execution, 2020, `arXiv:1908.11143 [cs.OS]`.

[268] Niels Provos and Nick Mathewson. libevent - An event notification library. http://libevent.org/, 2003.

[269] PuLP. Optimization with PuLP. https://coin-or.github.io/pulp/. Last accessed 1st May 2019.

[270] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. LKL: The Linux kernel library. In *Proceedings of the 9th Roedunet International Conference (RoEduNet)*, 328–333, 2010.

[271] Krishna P. N. Puttaswamy, Christopher Kruegel, and Ben Y. Zhao. Silverline: Toward Data Confidentiality in Storage-intensive Cloud Applications. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)*, 10:1–10:13, 2011.

[272] Weizhong Qiang, Zezhao Dong, and Hai Jin. Se-Lambda: Securing Privacy-Sensitive Serverless Applications Using SGX Enclave. In *Proceedings of the 14th International Conference on Security and Privacy in Communication Networks (SecureComm)*, 451–470, 2018.

[273] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. fTPM: A Software-Only Implementation of a TPM Chip. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, 841–856, August 2016.

[274] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-channels Through Obfuscated Execution. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, 2015.

[275] C Ravishanicar and James R Goodman. Cache Implementation for Multiple Microprocessors. In *Proceedings of the 26th IEEE Computer Society International Conference (COMCON)*, 1983.

[276] read manpage. http://man7.org/linux/man-pages/man2/read.2.html. Last accessed 14th August 2019.

[277] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL)*, 49–61, 1995.

[278] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, 199–212, 2009.

[279] Ronald L Rivest, Adi Shamir, and Len Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[280] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. Automated Partitioning of Android Applications for Trusted Execution Environments. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 923–934, 2016.

[281] Tamas Rudnai. Dispelling Myths Around SGX Malware. https://www.symantec.com/blogs/threat-intelligence/sgx-malware-explainer, 2019. Last accessed 28th October 2019.

[282] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted Execution Environment: What It is, and What It is Not. In *Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA, Volume 1*, 57–64, 2015.

[283] Amit Sahai and Brent Waters. Fuzzy Identity-Based Encryption. In *Proceedings of the 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, 457–473, 2005.

[284] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[285] V. Sartakov, N. Weichbrodt, S. Krieter, T. Leich, and R. Kapitza. STANlite – A Database Engine for Secure Data Processing at Rack-Scale Level. In *Proceedings of the 6th IEEE International Conference on Cloud Engineering (IC2E)*, 23–33, 2018.

[286] Sajin Sasy, Sergey Gorbunov, and Christopher Fletcher. ZeroTrace: Oblivious Memory Primitives from Intel SGX. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[287] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. Supporting Third Party Attestation for Intel SGX with Intel Data Center Attestation Primitives. *White Paper*, 2018.

[288] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy Data Analytics in the Cloud. Technical Report MSR-TR-2014-39, Microsoft Research, February 2014.

[289] 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 *Proceedings of the 36th IEEE Symposium on Security and Privacy (SP)*, 38–54, 2015.

[290] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Proceedings of the 14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, volume 10327 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2017.

[291] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, 753–768, 2019.

[292] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical Enclave Malware with Intel SGX. In *Proceedings of the 16th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2019.

[293] Seagate. *Seagate Kinetic HDD Product Manual*. Seagate, 10 2014.

[294] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[295] Adi Shamir. Identity-based Cryptosystems and Signature Schemes. In *Proceedings of the 4th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 47–53, 1985.

[296] Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. SGX-BigMatrix: A Practical Encrypted Data Analytic Framework With Trusted Processors. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, 2017.

[297] Vincent Y. Shen, Tze-Jie Yu, Stephen M. Thebaut, and Lorri R. Paulsen. Identifying Error-Prone Software An Empirical Study. *IEEE Trans. Softw. Eng.*, 11(4):317–324, April 1985.

[298] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. S-NFV: Securing NFV States by Using SGX. In *Proceedings of the 1st ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFVSec)*, 45–48, 2016.

[299] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[300] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the 11th ACM ASIA Conference on Computer and Communications Security (AsiaCCS)*, 2016.

[301] Shweta Shinde, DL Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-TCB Linux Applications with SGX Enclaves. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*, 12, 2017.

[302] Shweta Shinde, Shengyi Wang, Pinghai Yuan, Aquinas Hobor, Abhik Roychoudhury, and Prateek Saxena. BesFS: Mechanized Proof of an Iago-Safe Filesystem for Enclaves. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[303] Simon Johnson. Keynote: Scaling Towards Confidential Computing. *4th Workshop on System Software for Trusted Execution (SysTEX)*. https://systex.ibr.cs.tu-bs.de/systex19/slides/systex19-keynote-simon.pdf.

[304] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB Complexity for Security-Sensitive Applications: Three Case Studies. In *Proceedings of the 1st ACM European Conference on Computer Systems (EuroSys)*, 161–174, 2006.

[305] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. Moat: Verifying Confidentiality of Enclave Programs. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 2015.

[306] 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 *Proceedings of the 37th ACM Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[307] Rohit Sinha, Sriram Rajamani, and Sanjit A. Seshia. A Compiler and Verifier for Page Access Oblivious Computation. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE)*, 2017.

[308] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. MicroScope: Enabling Microarchitectural Replay Attacks. In *Proceedings of the 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019.

[309] S. W. Smith and D. Safford. Practical Server Privacy with Secure Coprocessors. *IBM Systems Journal*, 40(3):683–695, March 2001.

[310] Scott F. Smith and Mark Thober. Refactoring Programs to Secure Information Flows. In *Proceedings of the 1st ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, 75–84, 2006.

[311] Sean W Smith and Steve Weingart. Building a High-Performance, Programmable Secure Coprocessor. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 31(8):831–860, 1999.

[312] Zack Smith. Bandwidth: a memory bandwidth benchmark. https://zsmith.co/bandwidth.php, 2019. Last accessed 17th November 2019.

[313] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 33–46, 2010.

[314] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. Practical Techniques for Searches on Encrypted Data. In *Proceedings of the 21st IEEE Symposium on Security and Privacy (SP)*, 44–55, 2000.

[315] SQLite. How SQLite Is Tested. https://www.sqlite.org/testing.html. Last accessed 9th February 2020.

[316] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels, 2018, arXiv:1806.07480 [cs.OS].

[317] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 299–310, 2013.

[318] Raoul Strackx and Frank Piessens. Ariadne: A Minimal Approach to State Continuity. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, 875–892, 2016.

[319] Raoul Strackx and Frank Piessens. The Heisenberg Defense: Proactively Defending SGX Enclaves against Page-Table-Based Side-Channel Attacks, 2017, arXiv:1712.08519 [cs.CR].

[320] Raoul Strackx, Bart Jacobs, and Frank Piessens. ICE: A Passive, High-Speed, State-Continuity Scheme. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*, 106–115, 2014.

[321] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing. In *Proceedings of the 17th ACM International Conference on Supercomputing (ICS)*, 160–171, 2003.

[322] Symantec. 2019 Cloud Security Threat Report. https://www.symantec.com/security-center/cloud-security-threat-report, 2019.

[323] Jakub Szefer and Ruby Bei-Loh Lee. Architectural support for hypervisor-secure virtualization. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 437–449, 2012.

[324] Richard Ta-Min, Lionel Litty, and David Lie. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 279–292, 2006.

[325] Richard Ta-Min, Lionel Litty, and David Lie. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 279–292, 2006.

[326] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[327] TATP. Telecom Application Transaction Processing Benchmark. http://tatpbenchmark.sourceforge.net/. Last accessed 9th Feburary 2020.

[328] TensorFlow. Imagenet Dataset. http://download.tensorflow.org/example_images/flower_photos.tgz. Last accessed 7th August 2019.

[329] TensorFlow Benchmarks. https://github.com/tensorflow/benchmarks/blob/master/scripts/tf_cnn_benchmarks/benchmark_cnn.py. Last accessed 7th August 2019.

[330] Fredrik Teschke. KissDB. https://github.com/ftes/kissdb-sgx, 2017. Last accessed 12th October 2019.

[331] Hongliang Tian, Yong Zhang, Chunxiao Xing, and Shoumeng Yan. SGXKernel: A Library Operating System Optimized for Intel SGX. In *Proceedings of the 14th ACM International Conference on Computing Frontiers (CF)*, 35–44, 2017.

[332] TPC-C Homepage. http://www.tpc.org/tpcc/. Last accessed 12th October 2019.

[333] TPC-E Homepage. http://www.tpc.org/tpce/. Last accessed 12th October 2019.

[334] TPC-H Homepage. http://www.tpc.org/tpch/. Last accessed 12th October 2019.

[335] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. ShieldBox: Secure Middleboxes Using Shielded Execution. In *Proceedings of the 4th ACM Symposium on SDN Research (SOSR)*, 2:1–2:14, 2018.

[336] Bohdan Trach, Oleksii Oleksenko, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. Clemmys: Towards Secure Remote Execution in FaaS. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR)*, 44–54, 2019.

[337] Trusted Computing Group. http://www.trustedcomputinggroup.org. Last accessed 9th Feburary 2020.

[338] Trusted Computing Group. TPM Main Specification Version 1.2 rev. 116. http://www.trustedcomputinggroup.org/resources/tpm_main_specification, March 2011. Last accessed 10th March 2019.

[339] Trusted Computing Group. TPM 2.0 Library Specification. https://trustedcomputinggroup.org/resource/tpm-library-specification/, 2013. Last accessed 10th March 2019.

[340] 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 9th ACM European Conference on Computer Systems (EuroSys)*, 9:1–9:14, 2014.

[341] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, 2017.

[342] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB)*, 289–300, 2013.

[343] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution (SysTex)*, 2017.

[344] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, August 2017.

[345] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 991–1008, 2018.

[346] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, 2018.

[347] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, 1741–1758, 2019.

[348] Marten van Dijk and Ari Juels. On the Impossibility of Cryptography Alone for Privacy-Preserving Cloud Computing. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Security (HotSec)*, volume 10, 1–8, 2010.

[349] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)*, May 2019.

[350] Amit Vasudevan, Jonathan McCune, James Newsome, Adrian Perrig, and Leendert van Doorn. CARMA: A Hardware Tamper-resistant Isolated Execution Environment on Commodity x86 Platforms. In *Proceedings of the 7th ACM ASIA Conference on Computer and Communications Security (AsiaCCS)*, 48–49, 2012.

[351] vDSO manpage. http://man7.org/linux/man-pages/man7/vdso.7.html. Last accessed 7th August 2019.

[352] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. StealthDB: A Scalable Encrypted Database with Full SQL Query Support. *Proceedings on Privacy Enhancing Technologies*, 3:370–388, 2019.

[353] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted Execution Environments on GPUs. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 681–696, October 2018.

[354] Jinwen Wang, Yueqiang Cheng, Qi Li, and Yong Jiang. Interface-Based Side Channel Attack Against Intel SGX, 2018, `arXiv:1811.05378 [cs.CR]`.

[355] 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 24th ACM Conference on Computer and Communications Security (CCS)*, 2017.

[356] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX enclaves. In *Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS)*, 440–457, 2016.

[357] Josef Weidendorfer, Markus Kowarschik, and Carsten Trinitis. A Tool Suite for Simulation Based Analysis of Memory Access Behavior. In *Proceedings of the 4th International Conference on Computational Science (ICCS)*, 440–447, 2004.

[358] Mark Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE)*, 439–449, 1981.

[359] Samuel Weiser and Mario Werner. SGXIO: Generic Trusted I/O Path for Intel SGX. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 261–268, 2017.

[360] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single Trace Attack Against RSA Key Generation in Intel SGX SSL. In *Proceedings of the 13th ACM ASIA Conference on Computer and Communications Security (AsiaCCS)*, 2018.

[361] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. SGXJail: Defeating Enclave Malware via Confinement. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, September 2019.

[362] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 81–93, 2017.

[363] Peter Williams, Radu Sion, and Dennis Shasha. The Blind Stone Tablet: Outsourcing Durability to Untrusted Parties. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, 2009.

[364] WireGuard. https://www.wireguard.com. Last accessed 7th August 2019.

[365] Rafal Wojtczuk and Joanna Rutkowska. Attacking Intel TXT via SINIT Code Execution Hijacking. Technical Report, Invisible Things Lab, November 2011.

[366] Yongzheng Wu, Jun Sun, Yang Liu, and Jin Song Dong. Automatically Partition Software into Least Privilege Components Using Dynamic Data Dependency Analysis. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 323–333, 2013.

[367] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, 2017.

[368] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (SP)*, 640–656, 2015.

[369] Jisoo Yang and Kang G. Shin. Using Hypervisor to Provide Data Secrecy for User Applications on a Per-page Basis. In *Proceedings of the 4th ACM International Conference on Virtual Execution Environments (VEE)*, 71–80, 2008.

[370] Andrew C Yao. Protocols for Secure Computations. In *Proceedings of the 54th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 160–164, 1982.

[371] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Proceedings of the 27th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 162–167, 1986.

[372] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, 719–732, August 2014.

[373] Bennet Yee. Using Secure Coprocessors. Technical Report CMU-CS-94-149, Carnegie-Mellon University Pittsburgh PA Dept of Computer Science, May 1994.

[374] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 203–216, 2011.

[375] N. Zhang, K. Sun, W. Lou, and Y. T. Hou. CaSE: Cache-Assisted Secure Execution on ARM Processors. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (SP)*, 72–90, 2016.

[376] Wei Zhang and Yu Zhang. Lightweight Function Pointer Analysis. In Javier Lopez and Yongdong Wu, editors, *Proceedings of the 11th International Conference on Information Security Practice and Experience (ISPEC)*, 439–453, 2015.

[377] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. Minimal Kernel: An Operating System Architecture for TEE to Resist Board Level Physical Attacks. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 105–120, 2019.

[378] Wenting Zheng, Ankur Dave, Jethro Beekman, Raluca Ada Popa, Joseph Gonzalez, and Ion Stoica. Opaque: A Data Analytics Platform with Strong Security. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.