Flexible Provisioning of Attested Trusted Hardware Resources

João Tiago Alves Pino
Master in Information Security
Department of Computer Science
Faculty of Sciences of the University of Porto
2025

Flexible Provisioning of Attested Trusted Hardware Resources

João Tiago Alves Pino

Dissertation carried out as part of the Master in Information Security

Department of Computer Science

2025

Supervisor
Bernardo Luís Fernandes Portela,
Assistant Professor,
Faculty of Sciences of the University of Port

" I am putting myself to the fullest possible use, which is all I think that any conscious entity can ever hope to do "

HAL 9000, written by Arthur C. Clarke and Stanley Kubrick

Acknowledgements

It is with immense gratitude that I acknowledge the many individuals who supported me throughout this Masters program and the completion of this thesis.

I would like to start by thanking my supervisor, Professor Bernardo Portela, for his constant guidance and support, my deepest appreciation goes to him. His knowledge and expertise were invaluable, and I am especially thankful for his consistent push for greater methodical rigor and careful execution in my research and writing. This commitment to precision fundamentally improved the quality of my work and has instilled a critical discipline I will carry forward in my career.

To my family, particularly my parents, I owe everything. Their encouragement, and unwavering belief in my abilities over all these years provided the stable foundation necessary to pursue and complete this challenge. This achievement is as much theirs as it is mine.

Finally, I thank my friends, colleagues, and everyone in xSTF for their support, discussions, and the much-needed moments of distraction that kept me balanced.

UNIVERSIDADE DO PORTO

Abstract

Faculdade de Ciências da Universidade do Porto
Departamento de Ciência de Computadores

MSc. Information Security

Flexible Provisioning of Attested Trusted Hardware Resources

by João PINO

Protecting data-in-use in untrusted cloud environments is a major challenge, and hardware-based Trusted Execution Environments (TEEs) are the leading solution. This research focuses on Intel Trust Domain Extensions (TDX), which secures virtual machines, into Trust Domains (TDs), through hardware isolation. This work studies the remote attestation process that allows tenants to verify a TD's integrity and confidentiality.

Our findings show that while attestation reliably checks a TD's initial boot state, current cloud implementations lack stronger mechanisms for runtime application attestation. To address this, we proposed using a micro-kernel as the base for the guest OS within the TD, providing a minimal Trusted Computing Base (TCB). We also implemented a leader-replica provisioning architecture to study scalability and fault-tolerance in TEE deployments.

In conclusion, combining TDX's hardware isolation with a micro-kernel-based approach enables stronger runtime trust and scalable confidential computing.

Keywords: Trusted Execution Environments, Confidential Computing, Trusted Hardware

UNIVERSIDADE DO PORTO

Resumo

Faculdade de Ciências da Universidade do Porto
Departamento de Ciência de Computadores

Mestrado Segurança Informatica

Provisionamento flexível de recursos de hardware confiáveis e certificados

por João PINO

Proteger dados em uso em ambientes de nuvem não confiáveis é um grande desafio, e os Ambientes de Execução Confiáveis (TEEs) baseados em hardware são a principal solução. Esta pesquisa tem como foco o "Trust Domain Extensions" (TDX) da Intel, que protegem máquinas virtuais em "Trusted Domains" (TDs) através do isolamento do hardware. Esta pesquisa estuda o processo de atestação remota que permite que os utilizadores verifiquem a integridade e a confidencialidade de um TD.

Descobrimos que, embora a atestação verifique de forma confiável o estado inicial de inicialização de um TD, as implementações de nuvem atuais carecem de mecanismos mais robustos para a atestação de aplicações em tempo de execução. Para resolver isso, propusemos o uso de um microkernel como base para o sistema operacional dentro do TD, fornecendo uma forte base de computação confiável mínima. Também implementamos uma arquitetura de provisionamento líder-réplica para estudar a escalabilidade e a tolerância a falhas em implementações de TEE.

Concluindo, a combinação do isolamento de hardware do TDX com uma abordagem baseada em microkernel permite maior confiança no tempo de execução e computação confidencial escalável.

Palavras-chave: Ambientes de Execução Confiáveis, Computação Confidencial, Hardware Confiável

Contents

List of Figures

List of Tables

Glossary

AES-256	Advanced Encryption Standard with a 256-bit key
CA	Certificate Authority
CPU	Central Processing Unit
ECDSA	Elliptic Curve Digital Signature Algorithm
GCP	Google Cloud Platform
HRoT	Hardware Root of Trust
HSM	Hardware Security Module
LTS	Long Term Support
MAC	Message Authentication Code
MSc	Master of Science
OS	Operating System
PCE	Provisioning Certification Enclave
PCK	Provisioning Certification Key
PCR	Platform Configuration Register
PCS	Provisioning Certification Service
RAM	Random Access Memory
RTMR	Runtime-Extendable measurement registers
SEV	Secure Encrypted Virtualization
SGX	Software Guard Extensions
SMC	Secure Multi-party Computation / Secure Monitor Call
SNP	Secure Nested Paging

TCB	Trusted Computing Base
TCP	Transmission Control Protocol
TD	Trust Domain
TDQE	TD Quoting Enclave
TDVMCS	Trust Domain Virtual Machine Control Structure
TDX	Intel Trust Domain Extensions
TEE	Trusted Execution Environment
TLS	Transport Layer Security
TME-MK	Total Memory Encryption - Multi-Key
TPM	Trusted Platform Module
VM	Virtual Machine
VMCS	Virtual Machine Control Structure
VMM	Virtual Machine Manager

Chapter 1

Introduction

Data has three states that it can be in, it is either in rest when it is stored on a device or a persistent storage, it can be in transit when we are transmitting the data over to other location typically using the internet and lastly it can be in use when it is actively being processed by a system for example computing or executing an operation. While data-at-rest and data-in-transit can be reliably protected with mature cryptographic tools, protecting data-in-use remains a persistent challenge.

When it's resting, we have standard encryption such as AES-256 to prevent unauthorized access even if the storage is stolen and role-based access control. When data moves between devices or across networks, we also utilize standard encryption, such as TLS/SSL. But there is still the necessity of protecting the data while it is in use by applications [1].

Trusted Hardware devices are used in Cloud-based applications to securely process sensitive data in remote, untrusted servers [2]. These devices offer a secure execution environment isolated from other processes running on the same machine. Inside these environments, data can be processed without resorting to expensive cryptographic primitives, which introduce an overhead that can negatively impact the performance of the application.

Privacy and security are critical issues for everyone, especially when multiple parties need to compute on sensitive data without revealing their inputs to each other. The most famous case is the millionaires problem [3]. Proposed by Andrew Yao, a cryptographic experiment that shows even in a simple comparison problem, there are fundamental privacy challenges that need to be addressed. Two parties both want to determine who is wealthier without revealing their actual worth to each other.

The main problem is how two parties can participate in a computation of a function of their sensitive inputs without disclosing anything but the final output of the function.

The solution that Andrew Yao came with was to make that comparison between two values encoded as a boolean circuit, this concept garbled circuits where the circuit is encrypted in a way that the function can be evaluated without revealing what inputs it took, combined with techniques such as Oblivious Transfer, became the idea for the base for what is called Secure Multi party computation a paradigm that allows parties to compute together on private data without revealing their own sensitive inputs. Other solutions, such as Fully Homomorphic Encryption also, allow for the secure computation of sensitive data that belongs to separate parties.

Multi-tenant cloud environments are where multiple parties have access to the same machine on the hardware, but they all have their own isolated space. Cloud solutions are much more cost-effective than having one machine per user, the same goes for scalability and resource optimization. When sharing the same infrastructure for all tenants, this introduces a risk that is shared to all the users [4], that is even, though the cloud provider aims to isolate each tenant's data, the environment where that data is being used is still the same for all users. This means that if the hardware gets compromised or a user gains more privileges than they should, it creates a potential attack surface that would impact all users of that machine.

Some threats that are possible in these environments are:

1. **Noisy neighbor:**

   One tenant's malicious or wrong configuration of the workload could hinder performance or compromise the security of other tenants that reside within the same physical machine [5]. Exploiting shared resources such as caches or memory buses to extract sensitive information, some examples are cache-timing or speculative execution attacks

2. **Insider threats:**

   Privileged administrators or employees of the cloud provider that have physical/privileged access to the machines on premises, even with tight access control rules, compromised employee can misuse their access to inspect tenant data, bypassing logical isolation [6] [7].

3. **Data isolation and separation:**

Logical separation of data in a tenant's space is not enough when using virtual machines per user, if the hypervisor or the host OS where the VMs are is compromised, it may allow attackers to escape isolation and access other tenants' data [8].

These risks show why relying solely on SMC or software-based isolation is insufficient in practice. While SMC can protect privacy in theory, its inefficiency clashes with the scale of modern cloud workloads.

Secure Multi-party Computation provides a solution with strong theoretical privacy guarantees, but a practical deployment is often met by trust issues in cloud environments, and is hindered by performance overheads. In a multi-tenant infrastructure, where a client must rely on the cloud provider and purely cryptographic protocols cannot eliminate the concern about a compromised infrastructure.

Confidential Computing addresses these limitations by leveraging Trusted Execution environments, which isolate sensitive computations from the rest of the system with hardware-enforced isolation, therefore minimizing the Trusted Computing Base. TEEs, such as Intel SGX, Intel TDX, and AMD SEV, allow for secure memory areas that protect code and data from untrusted hosts. However, ensuring the integrity of virtual machines running in the cloud remains a challenge. Our approach is to develop a secure environment that allows for the secure computation of private data.

1.1 Objectives

The goal of this work is to design and implement a provisioning architecture for a multi-enclave system. In this architecture, a provisioning manager (leader) process will instantiate and manage replicas enclaves dynamically, while being the center point for client attestation. We will explore how the manager process can provision enclaves on demand, eventually on other machines, while still being able to transitively attest them for the client. With this solution, client trust is established once with the manager process, abstracting the multi-server architecture from the client.

As such, the goals established for this work were to:

- Analyse and compare competing Trusted Hardware devices and their applications in the market and academia
- Study and experiment with the SDK of a chosen device, particularly its remote attestation protocols

- Design and implement the provisioning manager and client protocols, enabling client-server and inter-enclave attestation
- Evaluate the application performance in comparison with the state-of-the-art, in particular single-server solutions

The novelty this research brings is:

- Development of a novel multi-server protocol for Trusted Hardware scenarios
- Better understanding of performance in a multi-enclave distributed solution

1.2 Report Outline

This work is exposed and organized in the following chapters, and has the following content:

1. Background: Here we will explain what the technologies are that exist in the Trusted hardware world, and what Trusted Execution Environments are.
2. Attestation and Trusted Execution: This is the chapter where we will explain in great detail TDX and its attestation. We will also explain what we did to gain a better understanding of the attestation and the discrepancies we found in documentation.
3. Scalable and replicate TE: This is where we explain the design and implementation of our provisioning manager protocol.
4. Conclusion: Finally, we will conclude our work with the key insights we developed and highlight what possible future work can be done starting from this point.

Chapter 2

Background and Related Work

Securing computations using only software-based solutions is a challenge and alone is not sufficient to guarantee a safe space for parties to perform sensitive operations. Since a compromised component of the machine (OS/hypervisor) can seriously compromise security guarantees provided at the software level. This creates a need to shift security from software to hardware.

The main goal here is to utilize trusted hardware so that we can rely on the cloud to execute arbitrary code in a trustworthy manner. This moves the assumption of a secure cloud software stack to the hardware, where we create a chain of trust in the hardware itself, which is designed to resist physical and software attacks.

A Trusted Computing Base (TCB) represents the minimal set of hardware, software, and firmware components that are critical to the security of a system. In other words, it encompasses everything that must be trusted to ensure the desired security properties. Traditionally, in cloud computing environments, the TCB includes a large portion of the system: from the physical hardware, firmware, hypervisor, and operating system, to applications and libraries. This broad TCB increases the attack surface and makes security verification and auditing more difficult. The goal of modern trusted hardware solutions is precisely to drastically reduce the TCB, narrowing the trust to well-defined hardware modules with auditable designs and implementations, such as TEEs or dedicated cryptographic modules. Instead of having to trust the entire cloud software stack, we only need to trust specific components, designed to withstand both physical and logical attacks, making security more robust and verifiable.

Leveraging specialized hardware to enhance security is not a new concept, trusted hardware has a wide range of devices that are designed to create a secure environment

for sensitive operations, such as cryptographic key generation and storage, and to protect against physical and software tampering. Devices like this typically create a root of trust that is more resilient to software attacks. This root of trust is inherently trusted and can enforce security policies regardless of the trust state of the host software. There are several trusted hardware technologies that have been developed and well-established as mature technologies, such as Trusted Platform Modules (TPMs), Hardware Security Modules (HSMs), and smart cards.

- Trusted Platform Modules (TPMs): They are a standardized crypto-processors integrated into a device's motherboard. Their primary function is to secure hardware by storing cryptographic keys, platform configuration information for boot integrity checks (Secure Boot), and performing cryptographic operations. However, a TPM is a passive device it can store secrets and report on the platform's state but cannot execute general-purpose application code in a protected environment. It protects keys at rest and measures the platform's integrity, but it does not protect application data in use.

- Hardware Security Modules (HSMs): Are dedicated, tamper-resistant external appliances designed for high-speed cryptographic operations and secure key management. They are the standard for protecting cryptographic infrastructure, such as in Certificate Authorities (CAs). While extremely secure, HSMs are specialized, expensive, and not designed for general-purpose computation. One cannot run an arbitrary workload, such as a machine learning model, inside an HSM. Their purpose is to be a cryptographic co-processor, not a secure execution host.

- Smart Cards: Portable devices with a secure microcontroller and limited memory, primarily used for authentication and storing small amounts of data like digital certificates. Their computational capabilities are extremely restricted, making them entirely unsuitable for the high-performance, server-side computations typical in a multi-tenant cloud environment.

To address the fact that infrastructure cannot be fully trusted in multi-tenant and cloud environments in modern computing systems, sensitive applications (such as healthcare, finance, data processing, and machine learning) often need to run on third-party hardware (cloud vendors), where the OS and other privileged software could be corrupt or

compromised. Traditional security mechanisms for data at rest and in transit that we previously talked about are insufficient to protect data while it's being processed.

To bridge this gap in protecting data-in-use, the development of hardware-based approaches that can isolate sensitive computations from the rest of the system began. In this context, TEEs have emerged as the current solution that does not rely on cryptographic primitives that introduce overhead.

There are a set of technologies that aim to extend this environment functionality in a way that we are able to create a fully isolated space where we can execute arbitrary code. These technologies go further beyond just protecting sensitive data, they can enable entire workloads, applications, or even virtual machines to run in a secure environment that is isolated from the infrastructure. This gives strong guarantees of confidentiality and integrity on an array of operations in cloud computing scenarios.

Having established the limitations of traditional trusted hardware and the need for stronger guarantees in cloud environments, we now turn to Trusted Execution Environments (TEEs). In the following sections, we define what a TEE is and explore the technologies that enable the creation of these secure and isolated execution spaces, which are essential for achieving robust confidentiality and integrity in modern computing scenarios.

2.1 Trusted Execution Environments

A Trusted Execution Environment (TEE) is an isolated and secure environment inside a computer's main processor. To allow this, TEEs are fundamentally built upon two interconnected concepts: confidentiality and integrity. Its main purpose is to protect data and code from being accessed or modified by any other software running on the device, including the main operating system (OS) and hypervisor if the environment is inside a virtual machine [9]. To execute sensitive operations with strong guarantees of confidentiality and integrity, TEEs leverage a hardware-enforced, isolated execution environment within the processor.

Most TEEs introduce the concept of a normal and secure world, as shown in Figure 2.1, a separation that leads to isolation. This is fundamental to how a TEE can provide the strong isolation needed between sensitive data and the rest of the system.

The normal world or untrusted world is what runs the operating system and regular applications. This has full access to hardware resources under normal conditions, but it's

considered untrusted since it can be compromised by a threat actor exploiting vulnerabilities in the system through malware. The secure world or trusted world is the protected execution environment that the TEE creates, the code and data that reside inside are isolated from the normal world.

This isolation is typically achieved through a combination of hardware-enforced memory protections, special CPU instructions, and a context switch controller that is managed by a secure monitor.

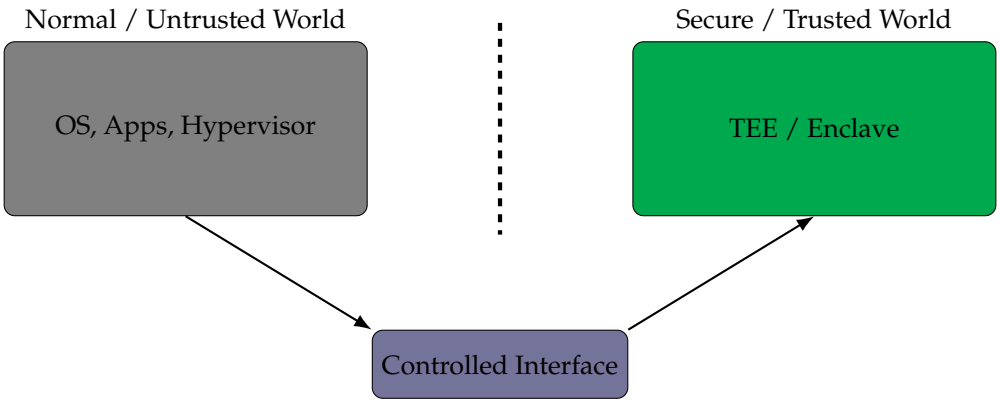

FIGURE 2.1: Separation between Normal (untrusted) and Secure (trusted) worlds in a TEE.

By having this unique separation, TEEs differ from other trusted hardware implementations. What further distinguishes TEEs is the capability of having a configurable environment. They allow users to define what code should be executed and what is being attested within the environment.

A strong characteristic of TEEs is the presence of a trusted source of randomness within the secure world. This capability enables the implementation and use of strong cryptographic systems, similar to Hardware Security Modules (HSMs) and Trusted Platform Modules (TPMs), to protect information transmitted outside the TEE.

In trusted environments, hardware isolation and attestation operate together to maintain security. Hardware isolation establishes the protected execution space, serving as the foundation that enforces confidentiality by shielding operations from external access. Attestation, in turn, provides verifiable proof that this isolated environment is authentic and trustworthy, assuring external parties that the protections are not only present but actively enforced.

Confidentiality in the context of a TEE ensures that both the code and data running inside the enclave are shielded from unauthorized access. Even if the host operating system

or hypervisor is compromised, the sensitive information processed within the TEE remains protected. This guarantee is achieved through hardware-enforced isolation, memory encryption, and strict access controls that allow only the enclave's trusted code to interact with its internal state. In effect, the TEE functions as a secure digital vault, enabling sensitive computations to be carried out without the risk of external observation or leakage.

As for Integrity in the same context, it refers to the assurance that both the code and data inside the enclave remain unaltered and trustworthy. It guarantees that the software executing within the TEE is the genuine, uncompromised version, and that any data it handles has not been tampered with. To uphold this property, TEEs employ mechanisms such as cryptographic hashing and digital signatures to validate the authenticity of loaded code and verify the consistency of protected data. These safeguards prevent adversaries from injecting malicious code or manipulating data, ensuring that the TEE continues to behave as intended.

2.1.1 Hardware isolation

A foundation for TEE is dedicated hardware resources, having physical hardware dedicated to computing the sensitive data or storing it. These are the fundamental mechanisms that manufacturers deploy to achieve this hardware isolation

- Processor modes:

  Processor modes such as ARM TrustZone or Intel SGX give distinct operating modes, normal and secure. This separation between normal and secure worlds minimizes the TCB trusted computing base, which leads to a smaller attack surface and an easier verification of integrity. The normal world is where the standard OS and most of the user apps run, it has access to most of the system resources and has a large trusted computing base.

  The secure world is a highly privileged and isolated environment where the processor operates under a new instruction set that is forced by hardware. By moving sensitive operations to the secure world, the TCB is reduced only to either the enclaves or trusted code running inside that world.

- Memory protection and isolation (Separate Memory Regions):

The hardware allocates different regions in the memory for the TEE, either physical memory or virtual memory. The TEE is responsible for ensuring that the region of memory of an enclave's secure world is reserved exclusively for that world.

Hardware components like Memory Protection Units, which are hardware components typically inside a CPU, that translate virtual memory addresses generated by software into physical addresses in main memory, can be configured to prevent unauthorized access from the normal world to the secure world.

If any attempt is made to read or write an operation from the normal world to a TEE memory address, the hardware will throw an exception fault and halt the operation.

TEE can also encrypt it at run time, not only isolate it. They encrypt the contents of the Secure world in the memory, even when it is in use in the RAM, at runtime, with special hardware keys stored inside the CPU. This helps prevent attacks where the threat actor tries to read data directly from memory (cold boot attacks) and Bus snooping.

- Access control/restricted interfaces:

  Secure World Entry and Exit Points Most TEEs have communication between the secure world and the normal world through controlled API calls. One example is Intel SGX enclave calls, Ecalls/Ocalls, while communicating with an enclave, only those calls are accepted, if any other program tries to do it gets blocked. The normal world cannot just read or enter the secure world, if it does, it has to be by the controlled API calls.

  Some processors maintain a set of dedicated registers for each world, so even in multi-tenant environments, this ensures that different secure worlds cannot access each other's data. CPU can track secure world ownership of memory pages and enforce separation, usually by a unique identifier.

- Root of Trust and Secure Boot:

  Hardware Root of Trust is a small piece of immutable code that is inside the ROM and is the very first code to execute whenever the device boots. This first boot or the secure boot is the foundation of the trust chain/root - the HRoT verifies the cryptographic signature for the next stage in the boot-loader, and each stage onward verifies the integrity and authenticity of the next component.

As illustrated in Figure 2.2, every stage must successfully validate the next one before execution proceeds. This enforces integrity and authenticity. This chain goes all the way to the OS or the application loaded into the TEE. If at any point any of these steps fails to confirm the cryptographic signature, the system will fail to boot, this means a signature was found to be tampered with / corrupted. This prevents unauthorized code from running in the TEE. Ensuring that only the correct and intended code can enter the secure part.

Without RoT and Secure Boot, a TEE could start in an untrusted state, making its guarantees meaningless.

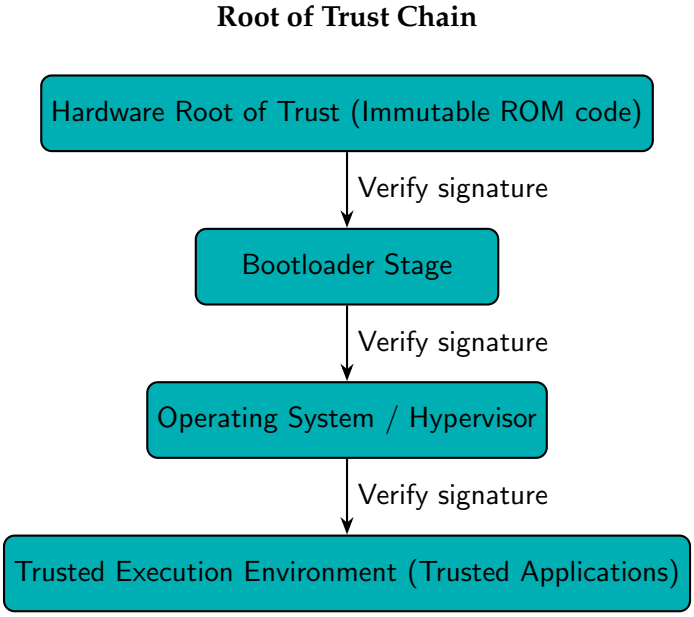

FIGURE 2.2: Secure Boot Chain of Trust rooted in the Hardware Root of Trust (HRoT).

All of these components help enforce access control, since the TEE has a higher privilege level than the rest of the system.

Bringing these mechanisms together, TEEs provide a unique way to execute code in isolated environments. Processor modes and memory protection establish the fundamental separation between secure and normal worlds, ensuring that sensitive operations and data remain isolated from the rest of the system. Access control and restricted interfaces further reinforce this isolation by strongly regulating how and when the secure world can be accessed, allowing only well-defined interactions. Finally, the root of trust and secure boot provide strong guarantees of authenticity and integrity for the entire execution environment, validating both the system and the code running within it—either at load time

or dynamically during execution. Together, these mechanisms enable TEEs to deliver robust security assurances, making it possible to trust the confidentiality and integrity of computations even in potentially untrusted infrastructures.

2.1.2 Threat Model

A clear definition of what the threat model is for a TEE is necessary, as it sets the ground rules for what it can and cannot prevent. The hardware isolation mechanisms described above, processor modes, memory protection, access control, and root of trust, collectively define the security boundaries that TEEs can enforce.

TEEs are designed to defend against powerful adversary models, in this case is an attacker who controls the host machine/environment (OS/hypervisor). In more detail, they provide protection against a compromised or malicious OS in cases where even the system kernel of the OS or virtual machine monitor is controlled by an attacker, the TEE is able to isolate sensitive data. This point is critical in multi-tenant cloud infrastructures. Memory snooping is protected by encryption engines that cover memory regions used by the secure environments. Cold-boot attacks typically involve reading or extracting secrets by reading the contents of RAM. Kernel-level malware or any other software that inspects secure world data without the proper permissions and page ownership is blocked, and the operation is halted.

However, while the mechanisms we described before provide strong guarantees of confidentiality and integrity, they are not a magic bullet. Certain classes of attacks, such as side-channel exploits, vulnerabilities in the secure world's own code, or physical tampering with the processor, remain outside the scope of what hardware isolation alone can prevent. Vulnerabilities in the code inside the environment. Rollback and replay attacks. Denial-of-Service. So understanding both the strengths and the limitations of these mechanisms is essential for accurately defining the threat model and for deploying TEEs as part of a broader security strategy.

Clearly defining what the threat model is for TEEs helps the end users choose and understand how their data is safe from attacks and assists developers of this type of environment to program secure worlds carefully and follow secure coding practices, since a TEE is meant to be more of an additional layer of security and not the only solution.

With the threat model of TEE defined, it also shows that there are some limitations and challenges in developing these environments. Particularly, the attacks that TEE does not

protect are still threats in real-world applications. Even when TEEs achieve the level of security intended, maintaining confidentiality and integrity guarantees at the hardware level, the complexity of creating and programming secure worlds and enclaves, scalability in cloud environments, and secure key management inevitably appear as obstacles.

2.1.3 Attestation

In a TEE, attestation is a critical process that allows the other party to cryptographically verify the authenticity and integrity of the environment running on a untrusted host, be it a local or remote party.

Local attestation is when both parties/environments are on the same machine and need to prove to each other that they are legitimate. On the other hand, remote attestation is when a remote entity, a cloud client, or an external party verifies the environment of the host machine where it has access.

This solves a core problem while dealing with sensitive data in an application on the cloud. While the TEE ensures the hardware isolation, this step ensures to the party that owns the data that the TEE is genuine, it is running under the correct configurations with unmodified software, the hardware and firmware are up-to-date with all the necessary patches to prevent known vulnerabilities, and finally that it is running on a trusted platform verified by the hardware signatures. In essence, attestation in a TEE allows a remote or local party to obtain cryptographic proof about the identity, integrity, and security of the execution environment before entrusting it with sensitive data or operations. This enables the secure establishment of trust between parties, even in potentially untrusted or multi-tenant infrastructures.

For this process, we have two main participants and one optional:

- **Prover (or Attester):**

  This is the entity that needs to prove its trustworthiness to the other party. It does this by generating evidence about its own identity and state. This evidence is cryptographically tied to the machine and is unique.

- **Verifier (or Challenger):**

  This is the remote or local entity that wants to trust the attester, but in order to do so, it must first send a challenge to the other party and validate the evidence it received.

- **The Attestation Service (or Endorser):**

This service is a trusted third party, usually ran by the hardware manufacturer, and it acts as the Certificate Authority for the hardware. Its job is to vouch for the attester hardware and verify that the signature in the evidence comes from a valid and non-revoked processor. This service is optional, but still needed in some cases, and its absence may change the trust model of the system.

When attestation is requested, the TEE generates cryptographically signed evidence about its state—including measurements of its software, firmware, and hardware. This evidence, signed with a unique device key, can be verified by an attestation service or directly by the client. The attestation service, typically provided by the hardware vendor, confirms the authenticity of the device and its keys, acting as a root of trust. This process allows the verifier to trust the TEE and its platform without needing to trust the host machine itself.

As is illustrated in Figure 2.3, a generalized flow for the attestation process (not tied to any TEE in particular)

1. The verifier or challenger wants to check if the TEE it's communicating with is trustworthy, it will send a challenge to the attester/TEE owner, often with a nonce to ensure freshness that prevents replay attacks of an old but still valid attestation.

2. The TEE will then collect the measurements that we talked about, cryptographic hashes of the code/data it's loaded, hardware and firmware versions, and configurations of the machine. The TEE will then put all of these measurements into an attestation report or quote. The CPU will sign this quote using the unique attestation key that is provided by the vendor.

3. Verifier checks for freshness by matching the nonce, authenticity by validating vendor and CPU signatures, and integrity by comparing the measurements to precomputed/stored known values.

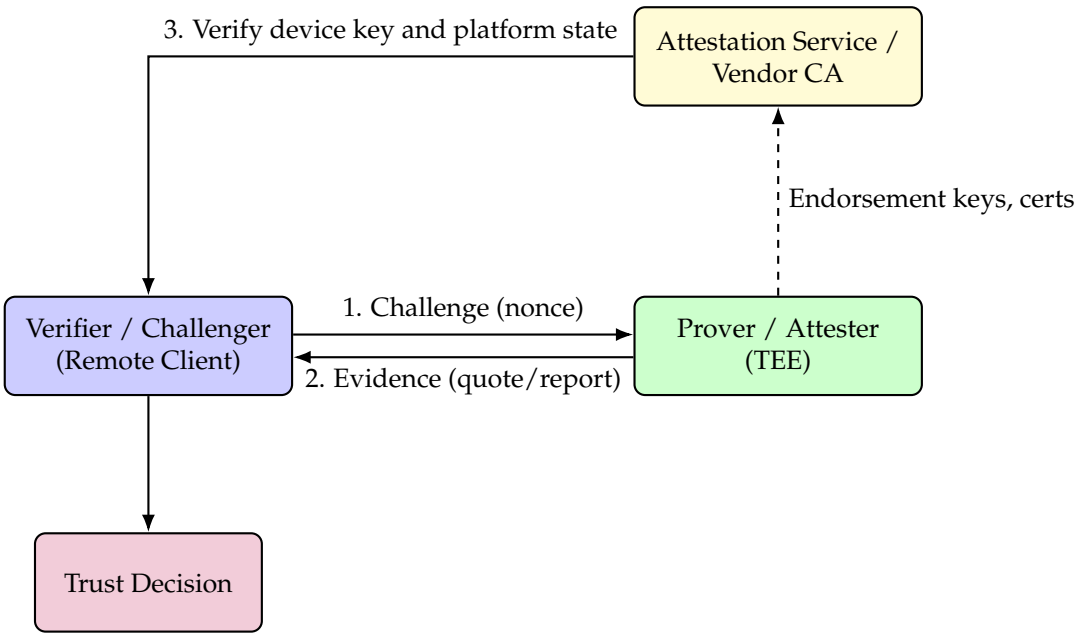

FIGURE 2.3: Attestation flow between Prover (TEE), Verifier (remote client), and Attestation Service.

2.2 Examples of TEEs

Going to describe some of the popular TEE-Enabling Hardware Technologies. Given that the main scope of this research was Intel TDX. We will delve deeper into the nuances of this technology in a later chapter. For now, we describe its predecessor, Intel SGX, its alternatives, such as ARM TrustZone and AMD SEV.

2.2.1 Intel SGX

Intel SGX is an implementation of a Trusted Execution Environment (TEE), provided as a set of instructions for specific Intel CPUs that support hardware-enforced isolation. It provides an isolated execution environment called an enclave, which is a protected memory region separated from the rest of the system. Enclaves are designed to safeguard sensitive data and computations from both external and internal threats, including the operating system, hypervisor, or firmware. SGX's primary goal is to provide confidentiality and integrity for code and data, even if the privileged software on the platform is malicious or compromised.

Since this is an Intel solution, it requires the hardware isolation provided by specific Intel CPUs (add list here)

An enclave is a protected area of execution in memory that was designed to run sensitive code and data. It ensures that no code outside the enclave, regardless of privilege level, can access its protected memory regions. These areas are isolated from the rest of the system so that even if the main system is compromised, what is inside the enclave is secure. Applications built for SGX typically consist of trusted and untrusted components.

- The trusted component resides inside the enclave and contains the sensitive code and data, which are encrypted in memory and decrypted only when executed by the CPU.
- The untrusted component operates outside the enclave and includes the remainder of the application, which interacts with the system, OS, and virtual machine monitor (VMM).

Attestation

In SGX, attestation is what allows another party to verify the integrity and authenticity of an SGX enclave alongside the platform it is running on.

It gives the user who requested the attestation cryptographic proof that the code running inside the enclave is the unmodified code that the developer of the application placed inside, and it has not been tampered with/altered. It also gives proof that this enclave is running on a platform that is SGX-enabled and has the required microcode/software updates to ensure security, and any configuration of the machine is correctly set.

There are two main mechanisms for attestation:

- Local attestation or intra-platform attestation:

  This allows two enclaves inside the same platform to verify each other.

- Remote attestation or inter-platform attestation:

  This occurs when a remote party needs to verify the authenticity of an enclave running on a different platform. This is more common and used for confidential computing in cloud environments.

This process is what allows two enclaves to trust one another, it closes the gap between hardware-isolation within an enclave and the need for a remote party to guarantee that the isolation of the environment was correctly done and that the enclave is running the trusted code.

Sealing

Sealing in SGX is used to store data in persistent storage and encrypt it so that the enclave that wrote the data to the disk can, at some point later in time, decrypt it and access it. This serves the purpose of protecting sensitive/private data across the life cycle of the enclave, either an application restart or a system reboot.

It works by using a sealing key that is derived internally by the CPU secret and the enclave's identity of the sealing identity (MRENCLAVE or MRSIGNER). This ensures that only enclaves with the same identity or signed by the same authority can unseal the data and access it.

This also provides confidentiality and integrity for the data that the enclave reused after it had been outside the trusted zone.

Deprecated

Since there was a low adoption by consumers, Intel SGX was deprecated on consumer-grade core processors. In 2022, Intel removed SGX support from 12th Gen Alder Lake consumer CPUs, although it is still supported on some server and Xeon processors.

Developer complexity was another main cause for the decision taken by Intel to deprecate this technology, building SGX-enabled applications required a different programming model, given that any application needs to have the trusted and untrusted components. Developers had to adapt since enclaves run in a restricted subset of instructions in the CPU. A complex Software Development Kit was necessary to build enclave applications, this was used to build proxy functions in C code that handled the context switching from both worlds.

Interactions with the outside world/ normal world must be made through ECALLS and OCALLS, special API instructions made to communicate with enclaves, which are enclave calls and outside calls, respectively. This adds a layer of boilerplating to the code and another layer of complexity.

Memory restrictions also made it increasingly difficult to develop these applications, since developers had to deal with the enclave memory being limited to 128MB in the enclave page cache. This need for more memory introduced a new problem, page swapping, which has a massive performance penalty.

Complex attestation process, remote attestation requires the use of Intel attestation service DCAP/IAS. Development issues, debugging is hard since there is a limited view of the state of the enclave from the normal world.

Security vulnerabilities and the attack surface of enclaves are also a part of why Intel SGX was deprecated. Side-Channel Attacks, although having a strong isolation, SGX enclaves have been the target of multiple security researchers. Over its lifespan, there were numerous side channel attacks have been discovered, Foreshadow [10], SGAxe [11], MicroScope [12] and Spectre [13]. Demonstrating that it was possible to leak data from inside the enclaves and even extracting attestation keys. Even though Intel kept patching these flaws in the SGX, the complexity of mitigating these attacks required microcode updates and software patches, and changes in system design, which were hard to consistently deploy for all consumers.

Usages and Future

Intel SGX is being used by cloud providers such as Microsoft Azure and IBM Cloud. These providers offer SGX-based virtual machines for confidential computing. These environments ensure the confidentiality and privacy of workloads and applications, keeping sensitive data secure at all times. They also enable confidential AI training, allowing data processing, business analytics, and machine learning models to run within the secure enclaves of SGX. SGX provides a trusted platform for digital assets, supporting secure custody, storage, and transfer of high-value assets in reliable and scalable environments.

Intel is shifting its confidential computing strategy from application isolation (SGX) to virtual machine isolation (TDX). We will delve deeper into this new solution, Intel TDX in the next chapter. It shifts the focus of securing an application and its memory to protecting a whole virtual machine.

2.2.2 ARM TrustZone

ARM processors have their own hardware-based security extension, ARM TrustZone [14], turned more towards mobile devices, IoT devices and embedded systems. Similarly to Intel's solution, their purpose is to create a TEE by separating system hardware and software's access to resources into two distinct worlds: a Secure world and a Normal World.

- Secure world: This is used for sensitive operations and is a much smaller, isolated, and controlled environment. It is designed to host security code and data.
- Normal World: this is where user applications will run, it is an environment with a large attack surface, one example is OS Android.

Similar to other TEE implementations, TrustZone uses hardware isolation. It divides the entire system into two halves: one secure world and one normal world, unlike SGX, which creates many small and parallel isolated compartments.

The ARM processor is designed to be TrustZone aware. The processor can switch between worlds controlled by a special instruction called SMC (secure monitor call) that handles the switching and enforces the isolation between normal and secure worlds. Every SMC instruction is made to the secure monitor, which is a small and trusted piece of code on the firmware that runs at the highest privilege level. It is capable of saving the state of the world it was leaving, validating the instruction, and restoring the state of the world it is entering.

TrustZone power comes from a system-wide integration, ranging from almost all components of the System on a Chip. Every operation has an additional bit called the Security state bit or Non-secure bit (NS bit), which indicates if the operation is non-secure or secure, or if it originated from the Normal World or the Secure World. This NS bit propagates throughout the entire memory system and down to peripherals.

When the CPU is in a secure state, it has access to all resources, and when it's not, it is restricted to non-secure resources. The Secure World runs its own dedicated software stack, completely separate from the Normal World. At the foundation of this stack is the Trusted Firmware-A (TF-A), the open-source firmware that handles the secure boot process and initializes the Secure Monitor. Above this firmware is a Trusted OS such as OP-TEE (Open Portable TEE), which creates a protected environment for secure operations. Finally, the Trusted Applications execute inside this Trusted OS.

The system first boots into the secure world, TrustZone, then can help to verify the integrity and authenticity of the normal world boot-loader and all other software components before they execute, creating a chain of trust from firmware to OS.

By design, a malicious or fully compromised Normal World operating system has no visibility into or access to the Secure World's isolated memory or its dedicated peripherals.

However, TrustZone's security model has inherent limitations. The entire security of the system relies on the integrity of the Secure World's software stack. A vulnerability in the Secure Monitor or the Trusted OS is catastrophic, as it would compromise all secrets and applications protected within it. This design philosophy necessitates keeping the Secure World's codebase minimal to reduce its attack surface. Furthermore, like other hardware-based isolation technologies, TrustZone does not inherently protect

against hardware-based side-channel attacks, which can infer secret information by observing physical effects of computation.

ARM TrustZone is widely deployed in mobile and embedded devices to secure a range of critical functions that require isolation from the main operating system. A use case is in Digital Rights Management (DRM), where the Secure World stores and uses the cryptographic keys needed to decrypt video content. It is also fundamental to mobile payments, securing sensitive biometric data for fingerprint or facial recognition and processing transactions for platforms like Apple Pay and Google Wallet in a protected environment. Additionally, TrustZone establishes the device's root of trust by holding the keys necessary for the Secure Boot process, ensuring the device only loads authentic software. Finally, it enhances user security in applications like password managers, allowing them to store and auto-fill credentials without ever exposing them to the potentially compromised Normal World OS.

2.2.3 AMD SEV

AMD Secure Encrypted Virtualization [15] is designed to enhance the confidentiality and integrity of virtual machines in multi-tenant cloud environments, leveraging hardware-based security features. SEV provides hardware-level memory encryption for VMs. Data that belongs to a VM is encrypted while it is in the host system's RAM. Traditional virtualization gives the hypervisor full access to all virtual machines' memory. In cloud environments, different customers' machines all run on the same memory in the physical machine.

SEV addresses this by encrypting each VM's memory with a unique key managed by the onboard AMD Secure Processor (ASP). Memory pages are encrypted upon being written to DRAM and decrypted upon being read by the CPU, ensuring the hypervisor can manage the VM's life cycle and I/O without being able to access its data in plaintext. This ensures strong isolation; even if one VM's key is compromised, others remain secure.

The hypervisor still manages the life cycle of the VMs and controls I/O, but cannot read plaintext from the VM memory.

Initial versions of SEV focused on protecting memory confidentiality. To provide a more robust security, AMD introduced new versions of SEV. AMD SEV-ES (Encrypted State) extended the protection of memory to include the CPU register state, preventing memory leaks during VM interrupts. AMD SEV-SNP (Secure nested paging) gave

memory integrity and rollback protection, protecting against a malicious hypervisor from remapping or tampering with a running VM's memory.

Attestation is also present in this TEE, giving the ability for a VM owner to verify these protections. The ASP can generate a signed report that proves that a VM was launched with the correct configuration and with the specific initial software. Allowing the owner to trust in the execution environment before transmitting sensitive data for computation.

Similar to Intel SGX, AMD SEV has been utilized in cloud services to enhance data security. Google Cloud Confidential Machines utilize AMD SEV technology to ensure data confidentiality during processing. Amazon and Microsoft Azure also utilize AMD SEV in their EC2 instances, allowing customers to validate the instances state and identity.

Chapter 3

Attestation and Trusted Execution

Most modern applications rely on processing sensitive data, which makes protecting it a critical challenge. The greatest risk lies in maintaining confidentiality when the data is in use, that is, while it is being processed and cannot remain encrypted unless executed within a controlled environment. This vulnerability is especially pronounced in cloud settings, where a compromised host, malicious hypervisor, or even a cloud administrator could access or tamper with workloads. Confidential computing directly addresses this problem by ensuring both the confidentiality and integrity of sensitive data during execution.

A pure cryptographic technique for a Secure Multiparty Computation and outsourced computation incurs a performance penalty, making it slower than unprotected data processing. This makes it very difficult for large-scale data and demanding AI workloads. There are alternatives talked about, but they come with their own challenges, such as data distribution and model poisoning [16].

This is where hardware-based TEEs emerge as a solution. Technologies such as Intel TDX and AMD SEV come as a perfect alternative to the cryptographic algorithms. These technologies, as we already talked about, use hardware-based memory isolation/encryption to create a TEE with a negligible loss in performance. This research will focus on Intel TDX [17] as the solution similar to the AMD SEV-SNP technology analyzed by this paper [16].

They give a keen insight into the architectural and procedural complexities of enabling secure computations. A critical focus of the research done in that paper is the attestation

process, the involved parties in SMC, and the fact that the attestation processes are designed to be ”performed by a single guest owner”. Meaning that the group that is interested in participating in the computation would have to elect one member to act as a leader. This leader would be the one performing the attestation and sharing the results with other members, shifting the trust the group had only in the outsourced infrastructure to this leader.

We will work on explaining how Intel TDX works and how it can be used to attest a multi-client workload environment. Also, we will see how we can use the attestation provided by TDX in the systems that are available and what kinds of guarantees that attestation gives to the end user. We will be examining the mechanics of Intel TDX, especially the attestation. Understanding this is going to allow us to demonstrate what the process of attesting a machine gives in a perfect world. Laying the groundwork for a more trust-distributed model for secure computation within public cloud environments.

3.1 Intel TDX

In cloud environments, traditional virtual machines are only isolated from each other, but the hypervisor and the OS which they are hosted on still have full access to the memory and CPU state of each VM.

This means that cloud providers or any party that can either compromise the hypervisor and have physical access to the machines could access private data.

This is the problem Intel TDX is aiming to solve, using confidential computing and introducing a hardware-level isolation Trusted Execution Environment for these VMs, called Trust Domains TDs, protecting sensitive data and applications from unauthorized access from the host OS and hypervisor, and any malicious threat actor with physical access to hardware.

We are going to give a brief overview of how TDX works, and we will follow with explaining every component that is important for the workings of TDX. The main innovation with TDX is that the virtual machine is itself a secure, trusted execution environment. Comparing it to a normal Virtual machine that communicates only with the virtual machine manager/monitor, a TD interacts with a new module made by Intel that is made to add a layer of security to all virtual machines inside a machine. Looking at Figure 3.1 we can see that the calls made from the traditional VM are made directly to the VMM, whereas the calls from the TD are made to the module and only then to the VMM.

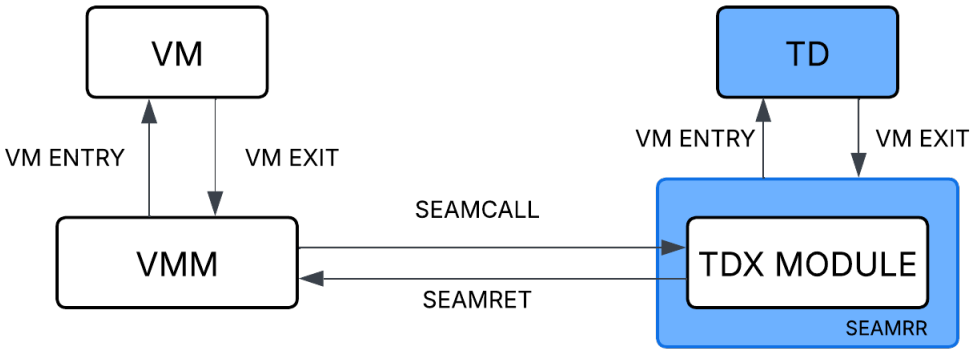

FIGURE 3.1: VM and TD interaction with VMM

3.1.1 Trust Domain (TD)

These are virtual machines with enhanced security via hardware-level isolation, which is the fundamental building block in TDX. Where the guest OS, the operating system that is running inside the VM, and applications can execute their code and operations over sensitive data. Unlike a traditional VM that communicates with VMM and hypervisor, TDs are protected by the CPU and the TDX Module. We can see this difference between a normal VM and a TD in Figure 3.1, where the calls to the VM and TD are made from different components. Each TD acts as a self-contained space, the code and data inside are protected from external access. This is where the guest operating system will run securely within Intel TDX. From the perspective of the applications and user workloads, the guest OS is just as a traditional OS, but it operates in a secure environment. This guest OS only has to have a few modifications to be able to work, since Intel provides us with the Intel-TDX-kernel `https://github.com/intel/tdx-linux`, which helps users build their guest OS.

3.1.2 SEAM mode

To enforce and manage these TD, Intel introduced a new security module and a new CPU mode, since CPUs require a special and higher privileged execution environment that is isolated from the rest of the platform. This is where the Secure-Arbitration Mode (SEAM) comes into play.

SEAM is a new CPU mode, it differs from the traditional CPU protection ring 0-3, levels such as user and kernel, VMX root/non-root, and SMM. It serves as a hardware-enforced privileged level. It runs code in a reserved CPU mode that any of the other privileges can access. This isolation is crucial for protecting the integrity of the TDX module and the data it manages.

The code and data for the SEAM reside in a reserved, hardware-isolated memory region called SEAM Range Register SEAMRR, as illustrated in Figure 3.1.

Any attempt to access any memory in the SEAMRR is aborted by the CPU, be it by external software, hypervisor, BIOS, SMM or DMA from hardware devices. Since this memory space is made to secure the whole environment, we have a finer control of the TCB, compared against just securing an application. SEAM is designed to host an Intel provided and digitally signed new security module, TDX module.

3.1.3 TDX module

The Intel TDX module runs inside the SEAM mode in the CPU, as shown in Figure 3.1 with the blue area. This module is a digitally signed, Intel-provided binary that is loaded into SEAM during the boot process by a special secure loader, SEAMLDR, which validates its cryptographic signatures to ensure authenticity before execution. This is the trusted component that mediates all interactions between the untrusted hypervisor and the TD.

If a hypervisor wants to manipulate or change a TD state or memory registries, they must do it through special API instructions, such as SEAMCALL in Figure 3.1, to request services from the TDX module. Instructions that would previously be made to the VMM, such as creating a new VM, assigning pages for memory scheduling tasks, must all go through the new module. The TDX module verifies each request to ensure it complies with strict security policies, preventing the hypervisor from performing any malicious operation against a TD.

To sum up these two main components, the SEAM provides the execution environment while the TDX module acts as the secure control and a new layer for the TD, which runs inside the new CPU mode. Together, they enforce the policies for security in TDX. Although the hypervisor still has the responsibility of allocating resources for every machine, it cannot do it directly.

3.1.4 TME-MK memory encryption

As a key important part of TEE is securing the memory, Intel TDX leverages a technology called Intel Total Memory Encryption, Multi-Key (TME-MK), that helps enforce memory protection for TDs. This engine ensures that the data of a TD cannot be read or accessed by the hypervisor or any other TD or anyone outside the TD itself.

This is crucial, as the goal is precisely to ensure that the cloud host is unable to access sensitive resources, as it can become the target of attacks. Indeed, our model considers it to be adversarial. Which is consistent with our requirements, as the cloud host is, himself, the adversary.

This helps with memory confidentiality and integrity, since each TD when it is created is assigned a unique memory encryption key that is generated and maintained by the hardware. The Memory Encryption Engine (MEE) transparently encrypts all data before writing it to DRAM and decrypts it on-the-fly when loaded back into the CPU. This helps prevent physical attacks, such as cold boot, bus snooping, DIMM probing and software attacks such as an hypervisor trying to map TD pages. Since each TD is assigned a different key it ensures isolation across tenants.

In the white paper for Intel TDX [17], we can see there are two integrity models:

- Cryptographic Integrity:

  This uses AES-XTS combined with a SHA-3-based MAC on each cache line in addition to AES-XTS 128-bit memory encryption. This gives the guarantee that the data is not only encrypted but is resilient to tampering because if an attacker tries to modify the ciphertext in DRAM, the MAC check fails when decrypted.

- Logical Integrity:

  This relies on who owns the data and the access control bits that the CPU keeps track of. "To help prevent ciphertext disclosure, a 1-bit TD-ownership tag is maintained with each cache line to identify if the line is associated with a memory page assigned to a TD" [17]. This is less computationally heavy and provides lightweight protection.

3.1.5 Memory isolation

Regarding memory isolation to protect the memory of a TD from the hypervisor and any entity outside the TCB, TDX memory is split into private and shared memory 3.2.

This distinction is made by the shared bit in the guest physical address. If the bit is '0', the memory is private and is encrypted with the TD's unique keyID, and it is only "visible" to the TD who owns it. If the bit is '1', the memory is shared, which allows the TD to communicate with untrusted entities such as the hypervisor.

To prevent the hypervisor from remapping the TD's private memory, Intel TDX uses Address translation integrity, which is enforced by hardware. This uses two separate Extended page tables one secure Extended Page Tables and shared Extended Page Tables. The first one translates private memory addresses and is managed by the TDX module, while the hypervisor manages the Shared EPT. This ensures the hypervisor cannot remap TD memory. The TDX module uses a structure called Physical-Address-Metadata Table (PAMT) that keeps track of ownership of each physical page.

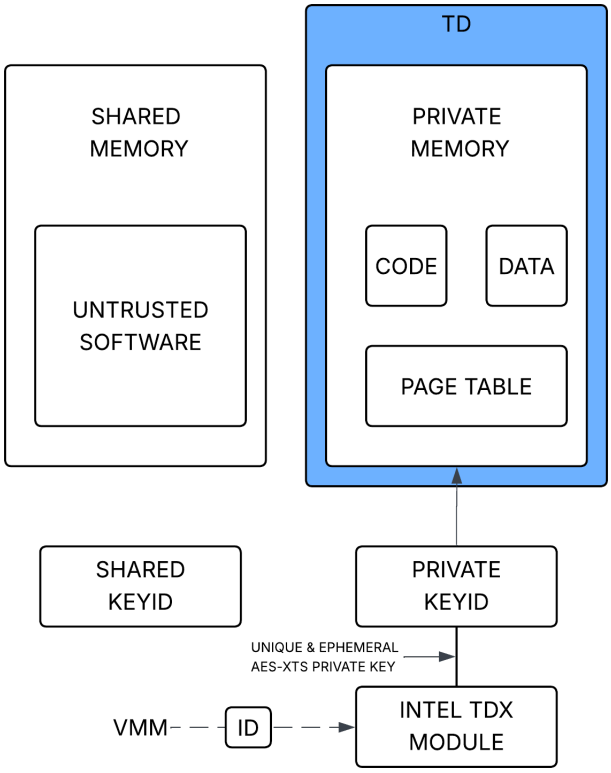

FIGURE 3.2: Private memory accessed with KeyID for each TD

With this, memory isolation reduces both the granularity of the hypervisor and the size of the TCB. Having this design limits the hypervisor to specific operations while preventing it from tampering with the data itself. This leads to the hypervisor not being part of the TCB for increased confidentiality and integrity.

3.1.6 TD creation

As mentioned before, the hypervisor is still responsible for the virtual machine resource allocation; it just has to do it through requests made to the TDX module via SEAMCALL. The TD is assigned a unique KeyID, which has its corresponding private memory encryption key, that is used by the hardware to encrypt and decrypt that TD's memory. The TDX module also records the initial measurements for the TD These will be a part of the evidence used in attestation that will be talked about in the following sections.

3.1.7 CPU state and interrupts

If a TD is not executing, either because its context was switched or it was interrupted, the CPU state must be protected from the hypervisor and saved securely. Whenever there is a TD exit the TDX module saves the contents of its registers into a dedicated, encrypted state-save area. Even privileged structures like the VMCS Virtual Machine Control Structure have their TDX corresponds (TDVMCS) that is encrypted and managed by the TDX module. Before returning to a normal operating mode by the hypervisor the TDX module scrubs the physical CPU registers to prevent leakage of sensitive information. "The module then scrubs these registers before returning execution control to the VMM, to help prevent leakage of TD state." [17]

3.1.8 New instructions

These are the instructions that we can see in Figure 3.1 between the VMM and Intel TDX module.

- SEAMCALL: Used by the hypervisor to request services from the TDX module (create a TD, pages, manage execution). OR The instruction used by the hypervisor to enter SEAM and invoke a function within the TDX module.
- SEAMRET: Return from SEAM back to normal execution mode after completing a secure operation. OR The instruction used by the TDX module to exit SEAM and return control to the hypervisor
- SEAMREPORT: Generates a cryptographically signed measurement/report of the TD's current state. This is used in local attestation between TDs or in remote attestation with external verifiers.

3.2 Threat Model and TCB

Let's remember the main goal of TDX, that is to create hardware-isolated virtual machines called Trust Domains (TDs), so tenant code and data are protected from the platform owner, hypervisor/VMM, and other host software.

In traditional virtualization environments, the TCB is large since it includes every part of the software and hardware of the host machine, even device drivers. This broad scope makes it difficult to achieve a strong security assurance.

Intel TDX changes this, it has a minimized TCB from the perspective of the tenant running inside a TD. With TDX, the hypervisor, host OS, and other platform software are considered untrusted. As represented in the Table 3.1, the TCB or trust boundaries for TDX are reduced to the Intel TDX module, Intel Authenticated code modules, TD Attestation Software and Intel CPU Hardware.

TRUSTED BY TD	**NOT TRUSTED BY TD**
INTEL® TDX MODULE	PLATFORM ADMIN
INTEL AUTHENTICATED CODE MODULES (ACM)	DISCRETE AND INTEGRATED DEVICES
TD QUOTING ENCLAVE	ALL OTHER SOFTWARE
INTEL CPU HARDWARE	OTHER PLATFORM FIRMWARE
	HOST-OS/VMM
	BIOS/SMM

TABLE 3.1: Trust Boundaries for TDX

With this reduced TCB, Intel TDX provides a tighter security boundary, so that the tenant needs to trust only on a small and precisely defined set of components and Intel itself, rather than the entire cloud software stack.

Now that we have the TCB clearly scoped, Intel TDX defines a threat model that assumes the following:

- **Untrusted components**: The hypervisor/VMM, host operating system, host firmware (outside SEAM), and all other privileged software are not trusted and may be fully adversarial. Other tenants and their workloads are likewise untrusted.

- **Adversary capabilities**: An attacker may gain full control of the host software stack, inspect or modify shared memory, attempt to inject interrupts or privileged instructions, and manage I/O paths and devices. The adversary may also have the ability to snapshot, pause, or migrate VMs at will.
- **In-scope protections**: TDX aims to ensure that the adversary cannot read or modify a TD's private memory or CPU state, cannot forge TD attestation, and cannot trick the tenant into running on an outdated or revoked platform configuration.
- **Out-of-scope attacks**: Certain classes of hardware-based attacks, such as invasive physical probing of CPU internals, side-channel leakage via micro architectural channels, and denial-of-service by the hypervisor, are not fully mitigated by TDX and must be addressed at higher layers or accepted as residual risks.

By articulating this threat model, Intel TDX provides both tenants and cloud providers with a clear understanding of what is protected, what is not, and how attestation can be used to establish trust before sensitive workloads or secrets are provisioned. For tenants to get strong guarantees of confidentiality and integrity, cryptographic values are intrinsically connected to the software and hardware of the infrastructure where these operations are being executed. These mechanisms are called attestation, that gives a tenant a better understanding of the platform it's using.

3.3 Attestation in TDX

The purpose of attestation is to allow an external party to cryptographically confirm that their data either, code or workload, is running inside a genuine and trustworthy Intel TDX-enabled TD, and that the platform where it is hosted (infrastructure) is correctly configured.

We are going to give a detailed analysis on how this process works, then we are going to be examining how a Quote and report are built and what assurances do they give us.

3.3.1 Attestation flow

We are first going to describe what is the flow of attestation in TDX as illustrated in Figure 3.3.

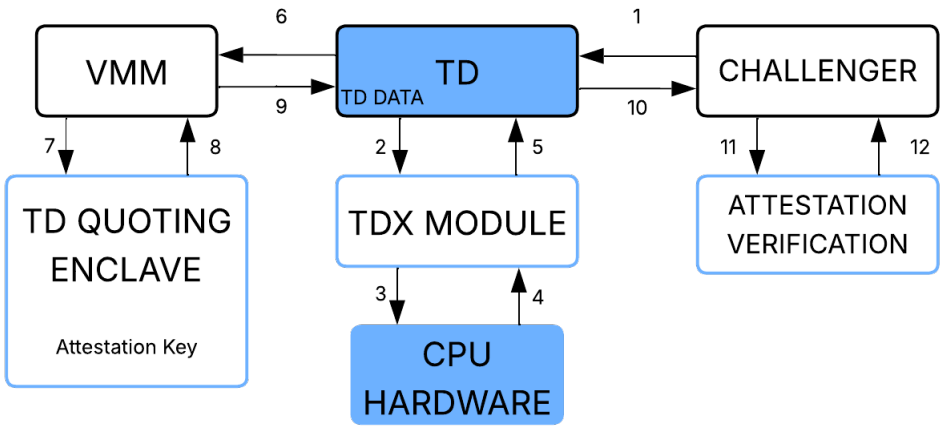

FIGURE 3.3: Intel TDX Remote Attestation Flow

The first step for attestation is the request that can be made by the application inside the TD or a remote party outside the machine (1). Then the TD issues a request to the TDX module (2), which will generate a TD report using the SEAMREPORT instruction (3). The report is a representation of the current state of the TD and it is bound cryptographically to the platform where it was generated. This links a TD report to the machine where it was generated at the hardware level. This report is then handed back to the TDX module (4) and back to the TD (5). Now the TD has to request the VMM (6) to convert this report into a remote attestation quote. This process is made by the TD Quoting Enclave (7,8 and 9). We will explain later how this works in detail. The next step is where the quote returns to the challenger (10) and this party has to be designed to use an attestation verification service (11 and 12).

Now, let's expand on what happens when the report leaves the TD, to become a quote (6,7,8 and 9).

After this report is generated, it has to leave the TD to be delivered to the TDQE quoting enclave. Leaving the TD can be seen as an unsafe step, as it leaves the TCB.

When the TD generates a report with SEAMREPORT, that report is hardware-bound. The CPU uses a secret key, known only to the TDX module, to compute a MAC over the report. Because this key never leaves the processor package, neither the VMM nor any other software can forge or modify the report without detection.

This is why it is considered safe for the TD to pass the report out of its protected memory and through the untrusted VMM on the way to the TD Quoting Enclave (TDQE).

The VMM can observe and forward the report, but it cannot alter it.

The TDQE is responsible for converting the hardware-bound TD report to a verifiable quote. There are two main steps for the quote generation, validation of the report and quote generation, as shown in Figure 3.4.

- **Validation:** This is where the enclave checks the integrity and authenticity of the TD report using the CPU-rooted key. This confirms that the report genuinely originates from the TDX module on that physical platform.
- **Generation:** After validation, the TDQE restructures the report into a standardized "quote" format. The quote is then signed with the platform's attestation key (1), an ECDSA key derived from Intel's endorsement infrastructure. The TDQE generates its own Attestation Keys and provides the Provisioning Certification Enclave (PCE) with the attestation public key. The PCE is designed to authenticate the request and issue a structure identifying the TDQE and the Attestation Key (3). Each platform has a Provisioning Certification Key (PCK) certificate, which chains up to Intel's Provisioning Certification Service (PCS) and ultimately to the Intel Root CA. This structure would then be signed using the PCK (4).

Once the quote is generated, it returns to the TD, where it is transmitted to the challenger, where it will perform freshness, authenticity and integrity checks.

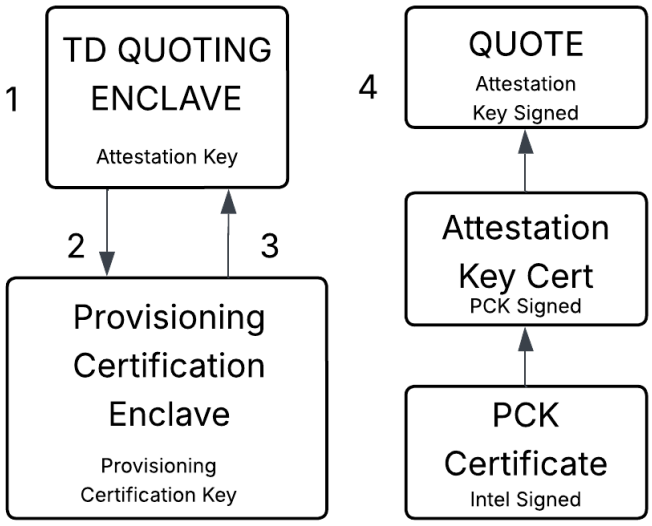

FIGURE 3.4: Quote generation with TDQE

For freshness, the quote must contain the correct nonce that the verifier originally sent to the TD. This helps prevent replay attacks where an attacker could send a valid but old quote. For authenticity, the ECDSA signature is checked on the quote, using Intel's certificate chain which proves that the Quote was signed by a genuine Intel TDX platform and has not been forged by software. Lastly, integrity, the verifier compares all fields of the quote, the TD measurements, attributes, RTMR values, and TCB SVN values against reference values,

3.3.2 Quote and report structure

The quoting mechanism binds the contents of its report to a concrete platform and hardware. As such, it is crucial to assess what this report actually reflects from the code being executed. When the TD requests a report from the TDX Module, it uses the SEAMREPORT instruction as we already explained.

The TD report structure taken from the Intel TDX Module Base Architecture Specification [18] can be seen in Figure 3.5 and the most important measurements are:

- **TD measurements**:

  As we already mentioned at TD Creation, the Intel TDX module is designed to initialize the measurement registers for that TD. These are hashes that represent the state of the TD at initialization. There are various hashes such as, the initial pages loaded into the memory of the TD, configuration parameters (TD policies) and any static data that would be defined during the TD Building stage. This is inside the TDINFO in Figure 3.5.

  These components are all bundled into a static measurement register called the TD-measurement register (TDMR). This would help to ensure that the data/code inside the TD matches the one that the tenant defined and it has not been tampered with.

  The TDX module also seeks to provide inside the report a set of registers dedicated to being extendable with additional code and data at runtime (when the TD loads new modules or runtime libraries). These registers are called Runtime-Extendable-measurement registers (RTMR).

  They function similarly to TPM Platform Configuration Registers (PCRs) [19], that are special registers used to store integrity measurements of a system. Instead of being overwritten whenever there is a new update to the system the PCR is extended,

using a cryptographic hash function. Each update extends the RTMR with a cryptographic hash, creating a cumulative measurement that reflects the full runtime history.

This ensures that the challenger can not only confirm the initial state of the TD but also its runtime evolution. This protects against code injections and reconfiguration after startup. Additionally, these properties are especially important in scenarios such as collaborative computation, where code and data from multiple parties are being executed inside the same TD. Having the initial state and the runtime measurements of the TD compressed into a cryptographic measurement and a means to verify it allows for a way to all parties ensure that all other parties cannot inject malicious code that could subvert the computation. Thus, providing these fields in the report allows for all participants to preserve the trust and ensure that the code running inside is the agreed-upon.

- **TD attributes**

  This is where the security configuration and operating mode of the TD are defined. Debug mode for whether the TD was created as a production or a debug TD, this is important because when a TD is in debug mode it has additional interfaces that can be used to extract sensitive information. Memory configuration, whether the TD is using the cryptographic-integrity or the logical-integrity protection mode using the TME-MK memory encryption. If the TD can handle privileged instructions and I/O. TD identity and ownership, metadata of the TD, this attribute (MROWNER), would not be measured but is included to verify if the TD corresponds to the correct tenant identity. All of this information is also captured inside the TDINFO in Figure 3.5.

- **TCB SVN Values**

  Trusted Computing Base Security Version Numbers (TCB SVN) Values These values are the versions and patch levels of the critical components in the TDX, they include CPU microcode Firmware for BIOS UEFI and the TDX components module SEAMLDR. This allows the verifier to confirm that the platform is up to date with all the security patches needed to prevent any known vulnerability from compromising the system. If a component is missing an update the SVN will not match and attestation will fail.

- **TD report data**:

This is a small space of 64 bytes, where optional user provided values such as a public key or nonce are stored. The nonce is used as a way to verify for freshness when the correct nonce appears in the final quote and the verifier compares values. The pair of private and public keys that can be generated inside the TD, the public key can be used to verify the identity of the TD instance.

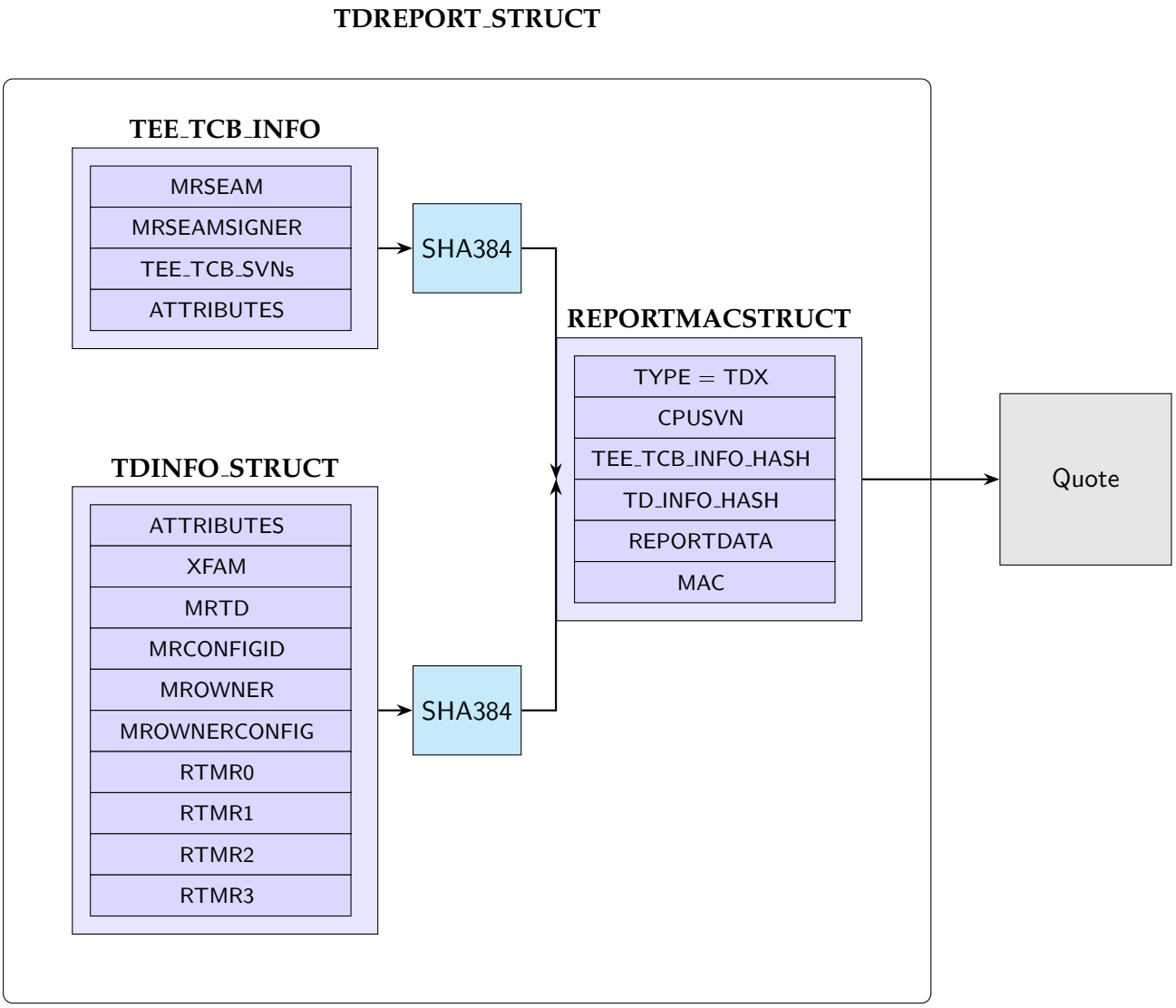

FIGURE 3.5: TDreport struct

In the table below (Table 3.2), we provide a concise description of each attribute contained in the Intel TDX report structure. These attributes collectively capture both the immutable measurements taken at TD build time (e.g., `MRTD`, `MRCONFIGID`), runtime measurements extendable by the TD itself (e.g., `RTMR0--3`), and platform information related to the underlying TEE (e.g., `MRSEAM`, `CPUSVN`, `TEE_TCB_SVNs`). Together, they enable a verifier to

establish the integrity of the TD's initial state, track security-relevant runtime events, and bind attestation evidence to a specific platform configuration.

Attribute	**Description**
MRSEAM	Measurement of the SEAM module loaded on the platform.
MRSEAMSIGNER	Measurement of the public key of the entity that signed the SEAM module. Identifies the signer.
TEE_TCB_SVNs	Security Version Numbers (SVNs) of the TEE's trusted components, used to assess TCB level.
ATTRIBUTES	Flags describing the TD's execution environment. Immutable once TD is finalized.
XFAM	Extended Features Allowed Mask
MRTD	Measurement of the initial contents of the TD.
MRCONFIGID	Software-defined ID describing the configuration of the TD.
MROWNER	Software-defined ID representing the TD owner.
MROWNERCONFIG	Software-defined ID for the TD owner's configuration or policy values.
RTMR0–3	Runtime Measurement Registers, extendable registers that TD software can use to measure runtime state.
TYPE = TDX	Identifies the report type.
CPUSVN	CPU Security Version Number, representing the microcode level of the processor.
TEE_TCB_INFO_HASH	Hash of the TEE TCB information structure, binds the report to the TCB description.
TD_INFO_HASH	Hash of the TDINFO structure.
REPORTDATA	64-byte field provided by the TD software, used to bind the attestation report to an application-specific context.
MAC	MAC protecting integrity of the report, keyed with a CPU-internal key so only hardware can generate it.

TABLE 3.2: Fields in the Intel TDX Report Structure

For an end user running workloads on a cloud provider, some of the most important attributes in a TDX quote are going to be those that directly establish trust in the platform. Specifically, MRSEAM and MRSEAMSIGNER prove that the SEAM module running is genuine and signed by Intel, anchoring trust in the TDX environment. TEE TCB SVNs, CPUSVN, and TEE TCB INFO HASH provide assurance of the microcode and TCB security levels, so that users can know they are protected against known vulnerabilities in their system. MRTD, MRCONFIGID, MROWNER, and MROWNERCONFIG bind the measurement to the specific workload and configuration the user expects, this ensures that the VM they are using matches the intended configuration. Finally, ATTRIBUTES, XFAM, and RTMRs confirm that the runtime environment has not been tampered with.

Taken together, these fields allow a tenant to verify both the integrity and the confidentiality of the TD and workload inside, despite running in an untrusted cloud infrastructure.

3.4 TDX Attestation in Practice

In this section, we are going to describe the work that was done to better understand the attestation process and what could be extracted from a quote in a TDX-enabled environment. First, we are going to address what our type of deployment environment was for testing and researching attestation, and what kinds of environments are there for developers to have applications that are enhanced by these mechanisms.

During this research, I used the Google Cloud Platform (GCP) [20] that allows users through their confidential computing VMs to create a TEE, they offer AMD SEV, AMD SEV-SNP, Intel TDX and NVIDIA Confidential Computing [21].

In Listing 3.1, we can see how to set up a confidential VM that is TDX enabled.

```
gcloud compute instances create instance-20250106-170005 --confidential-compute-type=TDX
--machine-type=c3-standard-4
--zone=europe-west9-a
--confidential-compute-type=TDX
--maintenance-policy=TERMINATE
--image-family=ubuntu-2404-lts-amd64
--image-project=ubuntu-os-cloud
--project=pure-beach-447016-q2
```

LISTING 3.1: Google cloud machine setup

In the GCP, when trying to set up a TDX-enabled machine, the only type of image for the guest OS available is ubuntu-2404.

With the help of go-tpm-tools [22] we managed to get quote generation and verification. The process for attesting and verifying can be seen in Listing 3.2

```
nonce=$(head -c 16 /dev/urandom | xxd -p)
sudo ./gotpm attest --nonce $nonce --format textproto --output quote.dat
sudo ./gotpm verify debug --nonce $nonce --format textproto --input quote.dat --output vtpm_report
```

LISTING 3.2: Generating Quote and Verifying it

We generate a nonce, for freshness purposes, and then we can use the attest function, using the standard formatting protocol from google textproto. This generates a quote.dat file that contains what we already saw in the Figure 3.5. Verifying that file yielded a

output file that has the same format as the quote, if the quote was valid, if the quote was not valid the command would output an error.

We extracted attestation quotes from the VM and verified those quotes and confirmed that we could indeed validate the identity of a machine. The attributes that we saw in the previous section that reflected the initial state and identity of the machine. The quotes all contained these static values:

- MRTD (build-time measurements): initial code pages, static config, TD build state.
- TD attributes: configuration of the TD (debug mode, integrity mode, etc.).
- TCB SVN values: microcode, BIOS, SEAM loader, TDX module patch levels.

When I took a closer look at the quotes started to look at the measurements and their meaning. I understood that the quotes I was getting only reflected the boot-time and creation time state of the TD, the values above. The fields for the RTMR were always the same, even if the system had updated or there were programs running in the background.

To test this further I went through 3 steps: take a measurement to establish the baseline, try to modify the system, get a new measurement and compare it.

1. I extracted a quote of the system and stored the values for the RTMR.
2. Created a script that would output to a file random 16 byte, key-like values. Made this script to execute in the background.
3. Extracted our second quote, and again stored the values for the RTMR.

Comparing those two files, the hashes for RMTR0–3 were the same in both, meaning that we could not attest a running program inside our TD. This is a major problem for tenants that want to use these environments to run protocols or join in multi-party computations, since there is no way to attest anything else apart from the start boot point.

This is where I encountered most of my limitations. Without a way to extend attestation to higher-level software running inside the TD developers are very limited in these cloud environments. Attempts to include or extend the RTMR were unsuccessful.

Without the ability to attest what software we are running on the machine after the boot process, this means that the verifier can only confirm that the TD started in a known valid state. But can not verify anything after the boot, including applications that have

started after the boot. Meaning that a user after requesting a quote from the TD can be interacting with a malicious TD.

I asked in forums about how I could extend the software I was using, even an initial sample code, to be included in the attestation, without any success. https://discuss.google.dev/t/runtime-attestation-with-intel-tdx-in-a-google-cloud-cvm/180226/1#M10380

Intel TDX provides RTMR which we took as the intended way for a TD software to extend with hashes of dynamically loaded code. However, the tools used and available did not provide a way for those measurements to be extended. Attempts were made with the Go-tpm-tools and the sample code from Intel TDX.

The go-tpm tool command offers a command to extract attestation evidence and send it to Google Cloud Attestation. Upon testing the command in Listing 3.3 realized in an Intel TDX machine the tool did not work and did not produce a token. So further researching the google confidential computing VMs did not allow for an easy attestation by using these google specific mechanisms [23].

```
sudo ./gotpm token > attestation_token
```

LISTING 3.3: Token generation

In the end identified the main key gap: Attestation process is not something trivial, it requires setup and runtime attestation of application workloads, which is not automatic, it requires explicit measurement and extension of RTMRs, which is not well-documented or supported in existing tooling. This gap in what Intel TDX promises and what we are able to do in these cloud environments raises a problem in what type of guarantees these environments are able to give an end user about the application and the system it's using.

3.5 Intel TDX in MicroKernels

Intel TDX provides hardware-enforced isolation and attestation for VMs, it gives a way for a client to verify the integrity of the machine he is using, through remote attestation.

Despite these guarantees, there are still limitations in controlling the execution environment. Since the guarantees that are given through attestation, about the system and what's executing, are few and not clarifying enough, to ensure a level of trust that is comprehensive, one would need stronger assurances that cover the broader software stack, the persistence of state, and the actual runtime behavior of the system.

A natural solution to further enhance the strength on security and reduce the TCB is the use of micro-kernels within a TDX-enabled environment. These solutions minimize the code executing in privileged mode, only exposing the essential interfaces for communication, execution and attestation.

Since attestation allows a user to verify the entire system, not just the application, we need to ensure that the evidence retrieved from the machines gives the tenant a better trust and understanding of the environment. This can be achieved by a micro-kernel where the environment we engage with is stripped to the bare minimum it needs to function and has all the capabilities it needs to operate, if the environment that the user interacts with is minimal. Since it would lead to the creation of an environment with a reduced TCB and it would be much easier to attest the system. This environment would still need to be capable of hosting the software that enables clients to send their workloads (communication), process the data (execution) and produce quotes (attestation).

To have a complete system capable of running for example a joint computation of multiple parties without any disclosure of information, this environment would also need a minimal file system, a Linux kernel that was TDX aware and the code for the computation of the workloads, and a few other components depending on the type of communication required for the protocols used.

Following this, there are two possible approaches, either start from a full guest OS and minimize the TCB by stripping the system to the bare minimum or build the micro-kernel from scratch, with only the components that are important for our system, with a reduced TCB in mind.

Given that access to a bare metal machine that was Intel TDX enabled was restricted, and cloud environments had a restricted and limited guest image options, to be able to replace the guest OS for example the Ubuntu TDX given by Canonical [24] with a micro-kernel, emulation of the TDX environment was needed. The cloud environments, such as Azure [25] and Alibaba [26], face the same problem as in the GCP, they all have a limited selection of TDX-enabled machines with their pre-built guest OS. Altering the image of the OS in cloud environments is impossible.

3.5.1 Infrastructure Components

Turning this into a working development requires the following classes of components to be able to function properly. First the host machine and its hypervisor that can create

and manage TDs, also the virtual firmware and guest OS that need to be a TDX-enabled kernel to boot and run inside the TD, and finally a practical way to test/debug without the constraint of being tied to a TDX-capable hardware.

Host and Hypervisor:

For an Intel TDX environment to function, the host system requires a specific software stack capable of interacting with the TDX hardware module. The central component is the hypervisor or Virtual Machine Monitor (VMM), which is responsible for the life-cycle of VMs. Although the hypervisor itself is outside the trust boundary of a running TD, it must be architecturally aware of TDX to create, manage, and terminate TDs.

Hypervisors can be split into two main categories:

- **Type 1 (Bare-metal):** This is the hypervisor runs directly on the host hardware. Some examples are Hyper-V and Xen.
- **Type 2 (Hosted):** Runs on top of a host OS in user-space.

In TDX systems, the type 2 are most common, with the usage of Kernel-based virtual machines (KVM) and Quick Emulator (QEMU). These are the most common combination for virtualization in Linux environments.

KVM is a Linux Kernel module, that transforms the host this turns the host OS into a hypervisor, allowing the kernel to manage VMs like a regular processes. Providing low-level virtualization support for the CPU. For this to work in the TDX environment, KVM needs to be patched with a support update [27], without this patch the kernel would not be able to manage TD.

QEMU a user-space VMM, that provides device models emulation, I/O emulation, and VM life-cycle management. This is usually used together with KVM, for a fast performance and hardware-acceleration virtualization. QEMU is responsible for handling TD creation, by interacting with KVM and exposes the TDX VM type only when the host supports it. Also provides the tools for configuring the guest firmware and launching TDs VMs. QEMU or other VMM must be built with TDX support, since QEMU only allows for the creation of TDX VMs unless the host shows KVM TDX support.

Together, KVM and QEMU form the necessary stack for the development of TDX environments, KVM is a Linux kernel module that allows the host kernel to schedule VMs directly, QEMU is the emulator and user-space VMM that, when combined with KVM, delegates CPU and memory execution usage.

Running real TDs the physical machine must have a CPU that allows for TDX to run. This is needed for QEMU and KVM to function, since they only offer the TDX VM type when the host kernel and CPU expose the TDX module. Although the hypervisor is outside the TD's trust boundary, it must support the TDX architecture, to be able to interact with Intel's hardware module. Simulation without TDX hardware is not supported [28].

Using the KVM paired with QEMU has several advantages, they are both mature, have robust support, as they are the primary development platform for TDX, both are open-source software, with this contributing to the security and maintenance of the projects, both benefiting from a strong community and vendor support.

However they also present challenges, developers must patch and compile a custom Linux kernel and rebuild QEMU with TDX support, adding significant setup complexity.

There are alternatives beyond QEMU, that are still in development, Cloud Hypervisor [29], which focuses on running modern, cloud workloads, with minimal hardware emulation. Despite it being less mature than QEMU it has growing support to develop TDX enabled VMS.

Guest OS and TD:

A TD Virtual Firmware (TDVF) [30] must also be included in the TD, this is what allows the TD to set up page tables and boot the guest kernel. The guest OS must be TDX aware, in this case, its a micro-kernel, so building the image for the VM needs to be done using a special kernel. Intel gives the Intel TDX kernel [27], this is a good starting point to later add parts of the micro-kernel that are needed, this way a controlled TCB can be maintained. A full guest OS can be generated by following the Intel official guide [31].

Hardware dependency:

There is no practical way to simulate the need for the host OS to be TDX hardware capable, limiting the testing of these environments. QEMU will refuse the creation of virtual machine types of TDX when the host does not present TDX-capable features.Creating a micro-kernel that could run inside a TDX TD is a promising approach to solve this problem and to reduce the TCB, and make the limited runtime attestation environments more comprehensible and trustworthy.

Since TDX provides a hardware boundary protecting the entire guest VM from the host, the image inside the TD can often remain a large and complex environment, acting as a vulnerable component. By replacing the standard OS with a micro-kernel we can construct a far more secure and easily verifiable environment.

3.5.2 seL4 Microkernel as a solution

Popular solutions for micro-kernels, such as seL4 [32], that is a unique OS that is the first with a formal proof of correctness. This enforces strong security boundaries for applications running on the OS, while maintaining a high performance that some deployed systems need.

Other solutions for micro-kernels can also be used, for example MINIX 3 [33] and QNX [34], that both offer good security guarantees and could have a good attestation synergy, but since seL4's primary use case is high assurance security we will develop on that first.

Even though the seL4 kernel offers a minimalist core, it provides the most fundamental mechanisms needed to build an OS, capabilities such as thread management, virtual memory and inter-process communication. While device and network drivers and the file system run as isolated user-space processes.

Access to any resource and any kernel service is controlled by a capability. In seL4 a `syscall` are capabilities that are made referencing or invoking on an object. There are also objects that Capability Nodes that contain a list of capabilities which are pointers to other objects in the system [35]. This ensures a principle of least privilege, since a process starts with no capabilities it can only do what it was explicitly given permission to do. Compared to a normal system, a process running as root can automatically access any file, whereas in seL4 you must have the specific capability for the specific file or resource you want to access.

Having the formal proof of isolation that seL4 provides, helps enforce confidentiality, integrity and availability, by guaranteeing that processes are completely isolated unless given permission to communicate. The formal verification rules out the majority of common vulnerabilities in the kernel, buffer overflows, race conditions and null pointer dereferences. It has a minimal TCB making the task of verification of the kernel incredibly feasible, with around 10,000 lines of code for the kernel.

SeL4 is used across a wide range of sectors in commercial products [36] such as in the automotive industry where companies like NIO use a seL4-based SkyOS an operating system for high safety and security. It is also being used in research environments [37] to showcase the virtualization use-cases.

Combining this micro-kernel with TDX will create a multiple-layer security in the system, where each component addresses a different threat.

- TDX protects the entire guest environment, from any part that is untrusted by the TD. The memory encryption capabilities and integrity protection ensure that nothing outside the TD can tamper with the micro-kernel and the applications inside.
- seL4, job is to protect the sensitive applications from each other and any other potential untrusted component introduced in the TD.

The enforcement of policies given by seL4 protects the TD against a potential malicious user who would try to subvert the TD behavior by injecting malicious code.

When a TD, running the seL4 OS, runs, the static attestation of the TD given by the initial state measured in MRTD is easily verified by the client. Furthermore, the client can not only verify that they are not just running a TD with the correct configuration, but also that the OS inside the TD is a mathematically proven secure kernel. Whenever there are multiple users inside the same TD, the attestation given by TDX and the formal proof of seL4 would allow the user a better trust in the security of their sensitive data, because any participant would be under a set of policies that restricts the operations they can do inside the machine and the binaries and data they can send over to the TD.

When it comes to the runtime attestation of the machine, our main problem, this combination between seL4 and TDX, addresses the limitations identified. Designing an application inside the user space that allows the creation of a verifiable and unforgeable measure of every piece of code that is inside the TD, is extremely important. For this the Application Loader would need to receive a binary that the client wishes to execute inside the TD, This would be the entry point for any new code or data, strictly enforced by seL4 model. Then the Loader would measure the binary, retrieve its hash, which can be used to either compare to a set established by the client of permitted binaries or to extend a RTMR. The Loader would make a call to the seL4 kernel and using the extend instruction it would update one of the hardware RTMRs, `TDG.MR.RTMR.EXTEND`.

This has its advantages that were already mentioned, creating an immutable ordered chain, that combines the previous values with the new measurement. Having the seL4 kernel being the only entity that is trusted to request this hardware-level instruction, ensures that any untrusted app cannot alter or misuse the operation.

Only after this measurement has been updated and the application confirms it, does the kernel give the Loader permission to execute the binary.

Whenever a client requests an attestation, this application would probe the measurements MRTD and RTMR, and send them in the quote, the client would then be able to

verify those values and that the kernel is the correct seL4 one, alongside the correct workload and binary files.

This design would follow the principle of least privilege on both hardware and software levels. Bringing these two technologies together strengthens the trustworthiness of the execution environment. In the end we would have a minimalist single-purpose environment, that can create a TD and run applications securely. On top of being easily verifiable, it gives software and hardware guarantees. In a multiparty computation scenario, code from multiple parties could be loaded as separate processes within the same TD, while the seL4's formal verification would ensure that one party's code cannot interfere with another's while TDX ensures the cloud provider or the infrastructure owner cannot tamper with the machine.

Chapter 4

Scalable and replicated TE

The increasing reliance on TEEs in cloud and multi-tenant infrastructures has highlighted the need for replication protocols that are both secure and efficient. This need exists because although TEEs are resistant to attacks they can still crash or be restarted. Hardware or software failures, or even an adversarial cloud operator, may cause a TEE instance to stop.

Traditional crash-fault tolerant (CFT) protocols fall short in protecting against rollback attacks, while Byzantine fault tolerant (BFT) protocols, although secure, impose significant performance penalties.

The Restart-Rollback fault model [38], works by capturing the precise fault behavior of TEEs with external state: namely, crash failures combined with the possibility of state rollback after a restart. This model enables the design of protocols that provide the confidentiality and integrity guarantees of TEEs while also ensuring the freshness of state.

We have designed and implemented a prototype that demonstrates how parts of the RR protocol can be integrated into a practical system. The implementation explores how leader-based replication can be achieved under the RR model, ensuring that clients communicate only with the leader while replicas maintain a consistent state with rollback protection.

As mentioned in the Introduction one of the critical motivations for this work is the need for a tenant to attest a machine in cloud environments, even if that machine has other dependencies that also need to be attested.

In the remainder of this chapter I will explain the architecture model of our design and also discuss the attestation flow required in multi-tenant settings with a number of replicas.

4.1 Architecture

The system is a client-server application built around a leader-replica architecture. With a focus on multiple clients being able to communicate with one entity that can delegate and distribute workloads to other servers. The main focus was that the clients only needed to interact with one entry point, and that the distributed system behind remained hidden but still verifiable. Its primary goal is to provide a continuous service to clients even in the event of a leader server failure. A single Leader node actively serves all client requests, while multiple passive Replica nodes maintain a synchronized copy of the system's state and have the ability to manage a workload.

If the leader fails, the replicas can initiate an election protocol to promote one of them to become the new leader, ensuring service continuity. The system incorporates a remote attestation mechanism for the leader to verify the integrity of replicas and a fault injection capability for testing purposes.

There are three main components as seen in Figure 4.1:

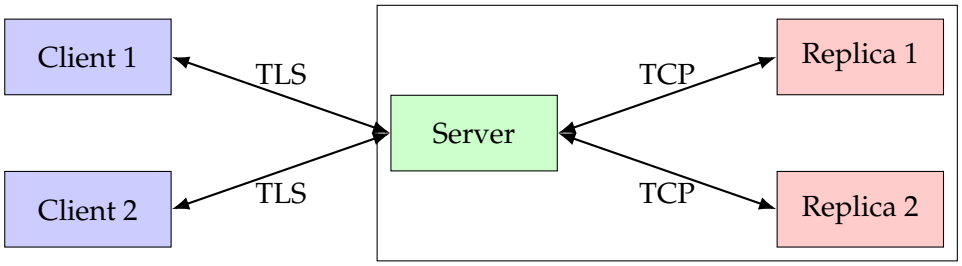

FIGURE 4.1: Client–Server–Replica Communication

Let's start explaining what the client does.

The end user who interacts with this service has a limited interface. It connects to the a leader server using a TLS Session. It must identify itself to the leader with a unique ID that is persistent to enable session and state recovery across shutdowns and disconnects. It can send commands to the server to submit workloads, check server status, or trigger administrative actions like remote attestation and fault injection. It can also be configured with a list of potential server IPs to use a for a potential leader. If the connection to the current leader is lost, it will connect to the next IP on the list. Since most of the testing was done on a local machine, the leader IP was static.

Client message protocol

- Client_ID: used to identify the client and its workloads.

- Request_quote: for requesting a quote from all the machines.
- Status: used to see the status of the server, what replicas are active and what are the workloads.
- Workload: used to send a workload to the server
- Fault_inject: used to send a fault injection on a specific replica using its ID.

As for the Server it is the other two components, the process can act as two roles, a Leader or a Replica.

LEADER:

This is the central hub for communication, since it's the only component that communicates directly with the clients, even though it can change identity The leader has the responsibility of creating the TLS dedicated connection for each client. Also listening and accepting incoming connections from replica servers through TCP.

It has to process every client command for every user, as well as maintain the correct state for all client workloads. It has to communicate every state change to all connected replicas in real time. Some of the updates are made only when they exist others go along a Heartbeat type message to the replicas. It must share the updates to workloads of every client to all replicas, and if a new replica joins it must notify all other replicas of that new one. It is also responsible for initialing and validating the remote attestation quotes from all replicas, ensuring their integrity, and managing a list of all active and suspicious replicas.

Leader responsibilities

- Client Handling
- Replica Handling
- Workload Management
- Replica Coordination
- Quote Validation
- Send Quotes

Replica:

This is a passive node that mirrors the leader state and can be turned into a part of a decentralized server for workload processing. When a replica is initialized it attempts to connect to a leader from the potential IP addresses list. If the connection is successful it must generate an attestation quote of itself and deliver it to the leader so that it is accepted as a trusted replica, proving integrity. It will then receive passive state updates for all clients and workloads, maintaining a consistent local cache. Responds to the leader's heartbeat messages and command queries and continuously monitors the leader's health via a heartbeat timeout mechanism. If a leader's failure is detected, it participates in an election to become the new leader.

Replica responsibilities

- Connect to Leader
- Receive Heartbeats
- Respond to Commands
- Send Quotes

Process Model:

The system consists of three types of processes: Clients, a single Leader, and multiple Replicas. The Leader and Replica roles are dynamic. A server process starts as either a Leader or a Replica, but a Replica can be promoted to a Leader through an election process.

Communication Model:

The communication model is asynchronous, meaning there are no guaranteed upper bounds on message delivery time. Communication occurs over reliable, ordered channels (TCP).

- Client-to-Leader: Communication is point-to-point over a TLS-encrypted channel, providing authentication, confidentiality, and integrity.
- Leader-to-Replica: The leader broadcasts state updates to all connected replicas over unencrypted TCP channels. Communication is one-to-many, implemented as a series of point-to-point connections.

Failure Model:

The system is designed to tolerate the crash of the leader node.

- Failure Type: The model assumes crash-stop failures, where a failing process halts permanently and does not perform any incorrect actions. It does not handle Byzantine failures, except for the specific case of a replica providing a bad attestation report.
- Failure Detector: The system uses a timeout-based heartbeat mechanism to detect leader failures. A replica suspects the leader has crashed if it does not receive a HEARTBEAT_FROM_LEADER message within a specified REPLICA_HEARTBEAT_TIMEOUT.
- Network Failures: Link failures between nodes are detected by the underlying TCP protocol, resulting in connection closure.

State Machine Replication (SMR) Model:

The system implements the Primary-Backup model of state machine replication to ensure fault tolerance.

- State: The replicated state is the complete set of all client workloads, stored as a map for each client.
- Operations: Client commands, primarily WORKLOAD submissions, are the operations applied to the state machine.
- Consistency: The system provides Sequential Consistency. The single leader serializes all incoming client operations and broadcasts them to replicas in a fixed order. Since replicas apply these updates in the same order they are received, their state remains consistent with the leader's.
- Restart: The model supports restarts through leader election. When a primary (leader) fails, the backups (replicas) elect a new primary, which then restores the replicated state from its local cache and resumes service.

Security and Trust Model:

The model defines different levels of trust for its communication channels.

- Client-Leader Channel: This external channel is considered untrusted. Trust is established through TLS mutual authentication and encryption.
- Leader-Replica Channel: This internal channel is unencrypted, since all server-to-server communication is assumed to occur within a secure, trusted network environment.

- Replica Integrity: While replicas are generally trusted, the leader uses a remote attestation mechanism (GET_QUOTE) to periodically verify their integrity, either on restart or when the client requests it. A replica that fails this check is marked as suspicious and is effectively excluded from the trusted group until it can be revalidated.

Now we are going to explain some of the more important parts of our implementation. I did not have access to an infrastructure where I could set up different IPs, all the work was done locally. First what happens when a replica joins the network. Since the process for a replica and a Leader is the same, when the process for the server starts it will check for an IP and port, if that IP and port accept a connection that means that there is already a Leader so the process starts up as a replica, if not it will act as the Leader. When a server acts as a replica it must send a quote for the leader to validate it status. In Figure 4.2 we can see that the replica sends an attestation, the leader verifies it, if its valid the Leader will mark the replica as Active, if not it will mark the replica as Suspicious.

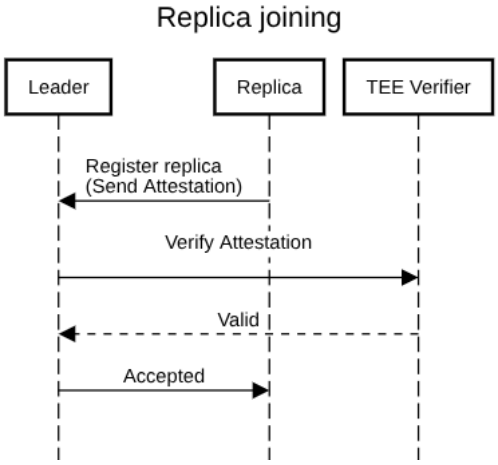

FIGURE 4.2: Replica joining

In Figure 4.3, we can see the attestation flow when a client requests a quote from the Leader. First the Leader will query all replicas, active or not, collect all the quotes and verify them against the TEE verifier and if any of the replicas have a bad quote it will mark them as Suspicious, otherwise the replica remains active. After verifying all quotes from the replicas, the Leader will generate a quote of itself and send to the Client. The Client then must verify this quote.

For all the quotes during testing we used a placeholder for a quote, since most of the testing was not done in a TDX-enabled machine, which not generate reports or quotes, so we used a simple string with the ID of the replica.

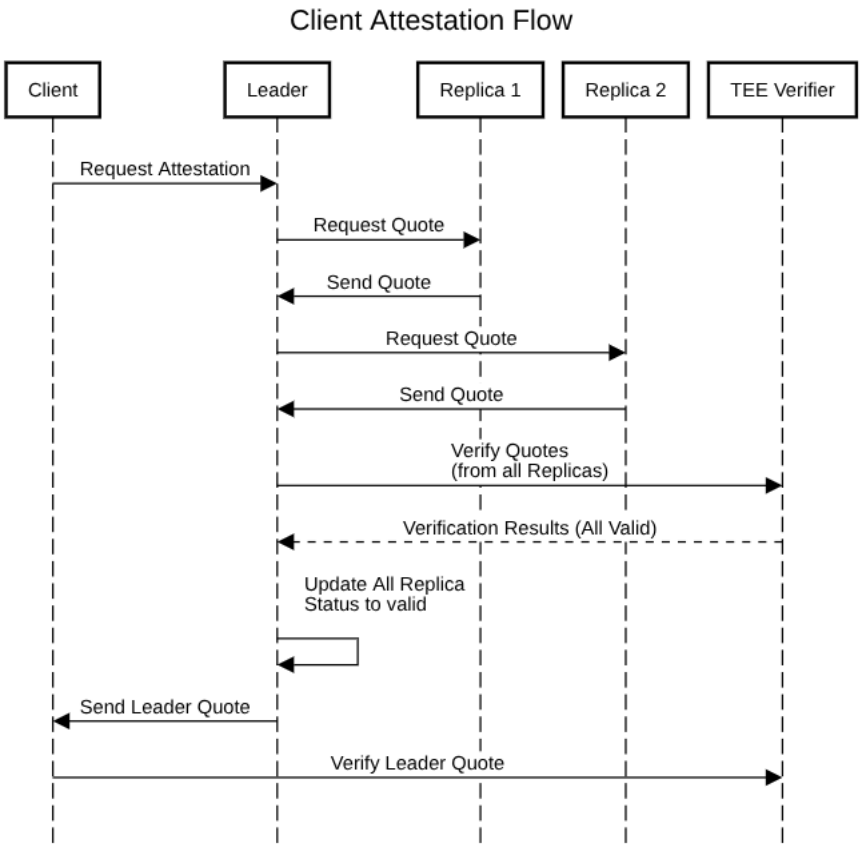

FIGURE 4.3: Client requesting attestation flow

In Figure 4.4, we can see our fault injection model, for testing, a client can choose from a list of replicas using their ID, a replica to target and prime its next quote to be a bad quote. This is useful to see the behavior of the replicas and the attestation process. A client sends the Leader the fault injection command, the leader then communicates to the replica that the Client chose to prepare the next quote to be invalid, the replica acknowledges when it has been primed. The next time the Client requests attestation from the Leader, when the Leader is querying all replicas, the one that has been primed will send back a faulty quote when all other replicas send a good quote. The Leader verifies all quotes, checks for one replica with a bad quote, updates the list of replicas with now one suspicious replica, and the process ends the same as the normal attestation. When the client checks the status of the server, it will see that one replica is marked as suspicious.

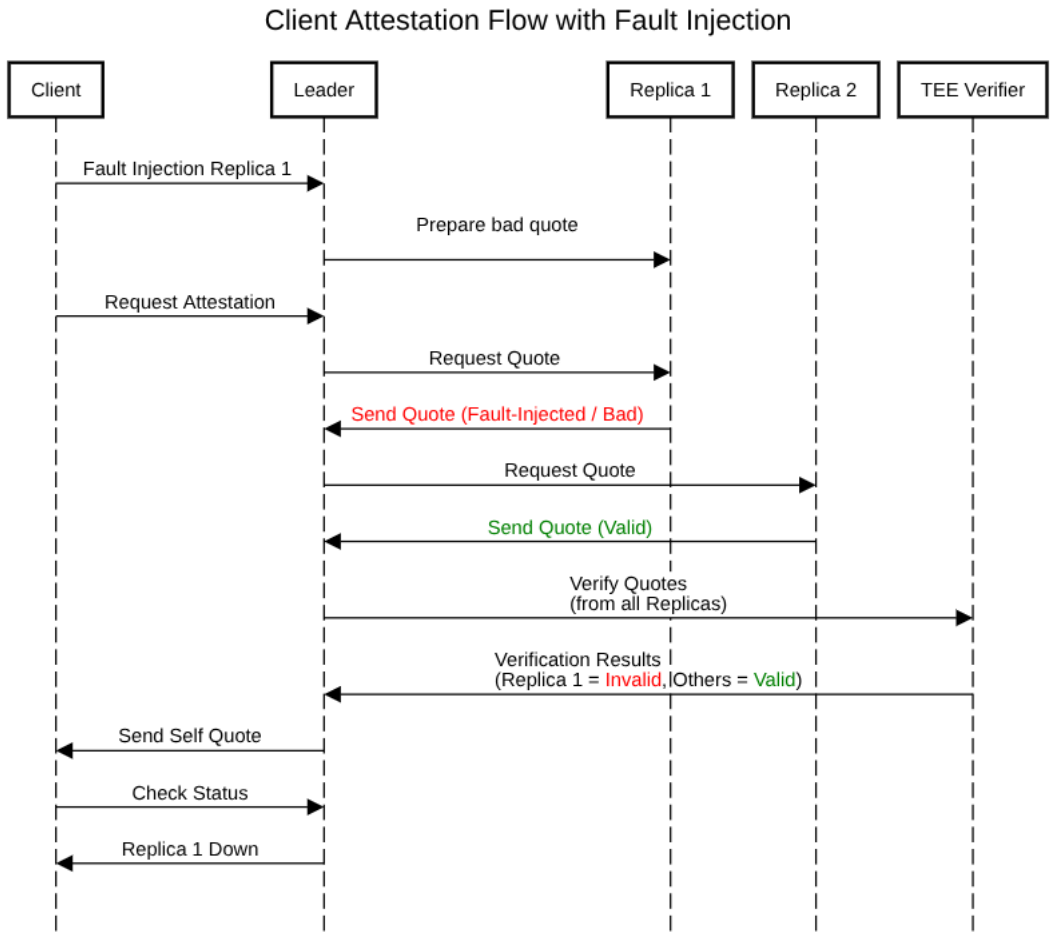

FIGURE 4.4: Client Injecting a Fault into a replica

In Figure 4.5, we can see the leader-based replication of the workloads sent by the client. When a client sends a workload, it is identified by the ID for the workload itself and the ID of the client who sent it. The first step, the client submits the workload to the leader, sending first the ID=1, with the data, in our test the workload were and array of integers, the leader then forwards it to all replicas connected to the network. In the second step the client can update the same workload by again using the same identifier ID=1, with new data, the leader only sends to the replicas the new update, not the whole workload. This ensures that the replicas always reflect the state of the client's latest workload.

Expanding and generalizing This client-server architecture can be implemented to create a more robust environment capable of executing arbitrary code and processes. The abstract workloads that were used as a simple array of data, could be defined to execute jobs each identified by a unique number. A client would submit a job to the leader, containing either data or code to be processed, in the form of a script/function or a precompiled binary. Then the leaders responsibilities would have to expand from the simple

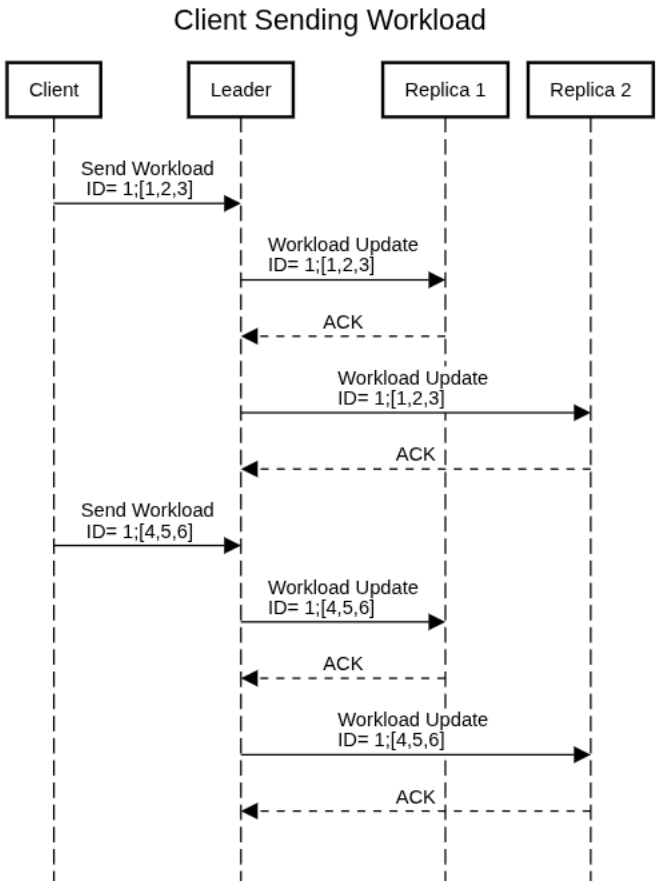

FIGURE 4.5: Client Sending a Workload

state synchronization and attestation mechanism, to include:

- **job reception and validation**: This would entail, parsing more complex data structures and validating such data types, from the client.
- **Increased state management**: Maintaining a more detailed overall state, which would include active, pending and completed jobs.
- **Delegation and Execution**: The leader would have to allocate resources (replicas) for the execution of these jobs. Changing the passive nodes that just mirrored the state of the leader, to become active workers to process jobs.
- **Result delivery**: When the leader gathers all results from the workloads it must update the state of the job and make the ending output available for the correct client

This would turn this design into a fully distributed computation platform where a client can offload a task to a fault-tolerant and scalable network that is based in trusted hardware.

To successfully manage the state of the jobs being executed, two primary strategies can be improved, a better state replication, and a state partitioning for performance that fits the jobs/workload being processed.

State replication

Every replica in the network maintains an identical, copy of the entire system. This is the primary backup model. The main goal is to tolerate a crash-stop failure of the leader. If a leader node fails, the system can elect a new leader from the active pool of leaders. Since every replica maintains a copy of the leader data, it can assume its place seamlessly and continue serving the clients.

State partitioning

This is a strategy that es more on scalability and efficiency, allowing our design to handle workloads that would exceed one single machine's capacity. Instead of every replica maintaining the same state that the leader has, the leader would divide the total state into smaller, independent partitions. It would then assign each replica a responsibility over a partition of the entire workload. That replica would be in charge of completing the job and maintaining its state, communicating it to the leader. This would improve performance and scalability, allowing for more complex workloads and heavier data being processed in the server. Increasing the throughput of the system, as more replicas are added.

4.2 Experimental results

In this experiment there were two testing environments, one local, and the cloud environment we talked about in the previous chapter. The setup for the local machine can be seen on Table 4.1

Similarly, the setup for the machine on the cloud is demonstrated on the Table 4.2

Since I was limited to a cloud environment the results I got were minimal, but we managed to conclude that the time for getting and verifying a quote was still the same using our Client-Server Model.

For our baseline, the recorded measurements were done in the environment, at the time of getting a quote:

Component	Specification
Hardware	
CPU	Intel(R) Core(TM) Ultra 7 155H
Cores/Threads	16 / 22
Memory (RAM)	16 GB
Software	
Operating System	Ubuntu 24.04.3 LTS
Kernel Version	Linux 6.14.0-32-generic
Compiler	g++ (Ubuntu 13.3.0-6ubuntu2∼24.04) 13.3.0
OpenSSL Version	OpenSSL 3.0.13 30 Jan 2024 (Library: OpenSSL 3.0.13 30 Jan 2024)

TABLE 4.1: System Hardware and Software Specifications

Component	Specification
Hardware (GCP c3-standard-4)	
vCPUs	4
Cores / Threads	2 / 4
Memory (RAM)	16 GB
CPU Model	Intel Xeon (Sapphire Rapids)
Software (customizable)	
Operating System	ubuntu-2404-lts-amd64
Kernel Version	Linux 6.14.0-32-generic
Compiler	g++ (Ubuntu 13.3.0-6ubuntu2∼24.04) 13.3.0
OpenSSL Version	OpenSSL 3.0.13 30 Jan 2024 (Library: OpenSSL 3.0.13 30 Jan 2024)

TABLE 4.2: Specifications of GCP `c3-standard-4` VM

- **Quote generation**: 10 ms
- **Quote verification**: 20 ms

In our experiment, we use the go-tpm-tools command line interface directly, on the code, to produce and validate, the time is the same as if any user were requesting it themselves.

There can be extra time that can be attributed to latency issues, when deploying on different server machines, with the quote verification process if the mechanism requires communication with a machine outside of the host network.

Whenever a server goes down, either a leader or a replica, and there is a restart the first thing a the server does upon reconnect is send an attestation quote to verify its integrity, this as we already saw is constant, now the leader after accepting the replica must send the entire of the workload data for every client and a list of all replicas, this time grows with the size of the workloads.

On the Table 4.3 we can see the time it takes for the client to establish a TLS connection, to the leader and send an attestation request for the system, we can also see the constant time for generating and verifying the quote. Over 50 runs with the client connecting and sending a request command and waiting for a reply, we can see that the constant time of the quote generation and verification process is growing with the number of replicas there are to attest. Since the leader must verify every replica before answering the client with the full attestation of the system.

Replicas	Total Median Time (ms)	TLS + Request	Quote
1 Replicas	122 ms	92 ms	30 ms
2 Replicas	132 ms	92 ms	40 ms
3 Replicas	142 ms	92 ms	50 ms
4 Replicas	153 ms	93 ms	60 ms

TABLE 4.3: Median connection and attestation time over 50 runs for different numbers of replicas.

The following table 4.4 shows the time cost of sending a workload from the client to the leader, which after receiving it spreads to all replicas. There is not much latency on the workload commands since it's just parsing an array first. The low number of replicas did not have much impact on the time it took for the workload to reach all replicas.

Workload (digits)	Latency of 2 and 4 replica
10	0.40 – 0.53 ms
100	0.40 – 0.53 ms
1000	0.40 – 0.53 ms
10000	0.45 – 0.58 ms
100000	0.50 – 0.64 ms

TABLE 4.4: End-to-end latency from client to replicas for different workload sizes.

As we can see in Figure 4.6 the graph shows the minimum, average and maximum time for 50 operations, for the process of a client sending a workload to the leader and the leader to send it to all replicas. This showcases the time it takes for the whole network to have the same workload as the client sent to the leader, we used 4 replicas for this test.

As we can see the graph shows how the average time to complete an operation grows with the size of the workload. For small inputs (100 or 1,000 elements), the operation completes in less than a millisecond, but as the input size reaches 10,000 elements the time jumps into the tens of milliseconds, and at 1,000,000 elements it grows into the hundreds of milliseconds.

We can take from this that the performance cost is negligible for small workloads but increases significantly as the data size grows. Since the workload, in a real use case would be sent either with or before asking for an attestation of the system, we show that this attestation cost will become insignificant as we move to 'real' requests, with many instructions and larger states.

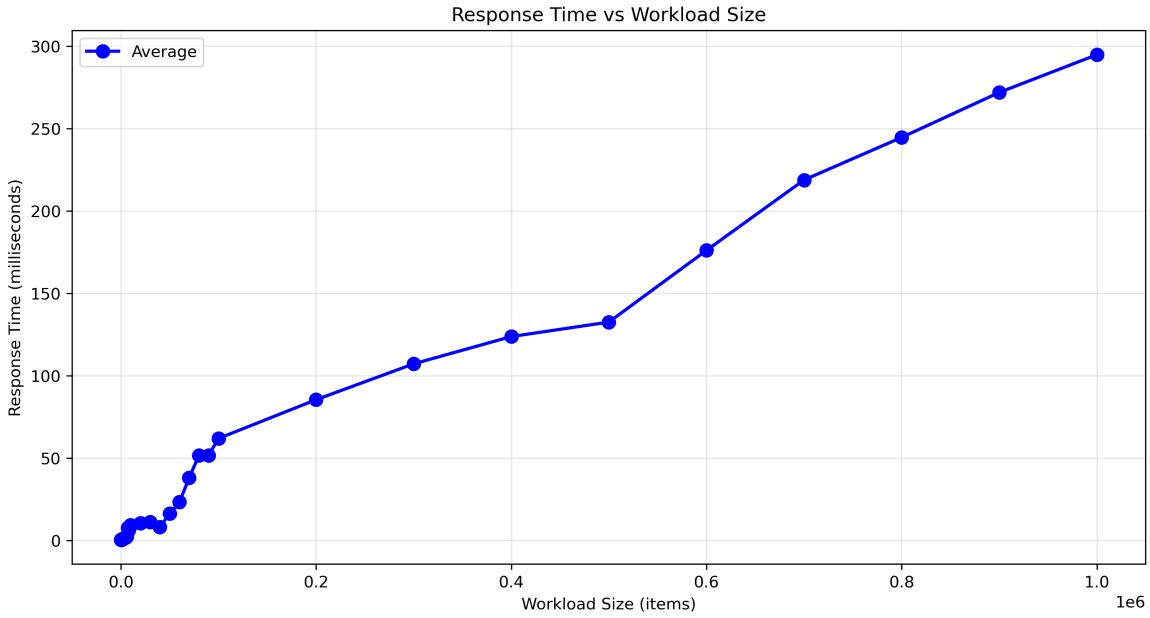

FIGURE 4.6: Increasing Workload Operation Time

The communication between the leader and replicas, was made in TCP given that it was simpler to test since the number of replicas is dynamic and not fixed, a TLS solution would entail generating certificates for every replica.

If the processing of more quotes brings too much latency to the system, verifying every quote could be made faster by the use of threads that would speed the process. Since testing was done using at most the max of 4 replicas that problem did not occur at most we got more 60ms.

This constant overhead of the remote attestation process, fits the use of a practical implementation for secure operations and protocol strengthening. Combining these measurements with a micro-kernel based TD environment, as we previously talked, ensures that code offloaded to the TD cannot access sensitive memory directly, ensuring confidentiality across the environment for all tenants. Consequently, the integrity guarantees provided by the quote remain meaningful and enforceable, even as the system scales to larger datasets or more complex operations.

Chapter 5

Conclusion

5.1 Conclusion

In this final chapter, we will look at the objectives proposed and what we hoped to achieve with this research, looking back at the Introduction (Section 1.1), we established these goals:

1. Analyse and compare competing Trusted Hardware devices and their applications in the market and academia
2. Study and experiment with the SDK of a chosen device, particularly its remote attestation protocols
3. Design and implement the provisioning manager and client protocols, enabling client-server and inter-enclave attestation
4. Evaluate the application performance in comparison with the state-of-the-art, in particular single-server solutions

For the first goal, we explored the competing Trusted hardware devices SGX, ARM TrustZone and AMD SEV, with special detail on SGX. This choice led to a better understanding of the successor of SGX, Intel TDX, explored the inner workings of this new technology and its remote attestation protocols and solutions it provided.

Designed a provisioning manager and client protocol, although the implementation is very rough and bare boned, it allowed us to understand what is possible to do with an environment that allows for attestation of multiple clients and machines.

Where there is work to be done is in the evaluation of the application that was designed, since the testing done was minimal, due to the fact that there was not a platform easily available to test and modify.

All of this investigation into Intel TDX revealed that while its hardware-level protections offer strong guarantees for isolating entire virtual machines, a significant gap exists in its practical application, when it comes to the solutions provided by cloud providers, specifically the GCP.

The attestation mechanisms provided in the GCP environment, primarily verified the initial, boot-time state of a TD. We demonstrated the importance of runtime measurements, and how crucial it is to verify the integrity of applications after boot. Most of the available tools are not readily supported to do that. This has a huge limit on the level of trust a tenant can place in a running workload.

To address this challenge, we proposed a solution that joined two technologies, pairing the hardware isolation of TDX with the software security of a micro-kernel, seL4. This would create the minimal TCB and a formally verifiable environment inside a TD, enabling the use of applications to execute workloads and mechanisms for runtime attestation, where every piece of code and data loaded into the TD can be measured and verified.

5.2 Future work

The leader-replica prototype developed in Scalable and replicated TE (Section 4) serves as a practical demonstration of the architectural requirements for scalable, fault-tolerant TEE systems.

In the model designed, a central leader is responsible for managing and attesting a dynamic set of replicas, abstracting the complexity of attesting the entire distributed system from the client. Evidently the trust and security of the architecture implemented is entirely dependent on the quality of the attestation evidence it is capable of collecting.

The implementation attestation flow, has room for improvement, for the leader to truly guarantee the integrity of a replica, it must be able to verify not just that the replica's TD booted correctly, but that the specific replica software running inside it is uncompromised at that moment.

This is precisely where the micro-kernel proposal becomes essential. Implementing our leader-replica system within a micro-kernel-based TD would bridge the gap between

architectural need and practical capability. The micro-kernel would provide the strictly controlled environment necessary to measure the replica application at runtime, extend the hardware's RTMRs, and generate a comprehensive attestation quote. A possible cloud solution using this would entail developing the micro-kernel, to be able to receive any binary or workload and have the attestation and processing capabilities fully functional.

In conclusion, while technologies like Intel TDX provide the necessary hardware foundation, achieving truly secure and verifiable confidential computing at scale requires a co-design of the entire stack. Future work should focus on implementing this micro-kernel-based TD environment and integrating it into our distributed framework to create a system that is not only scalable and resilient but also holistically trustworthy from the silicon to the application.

Bibliography

[1] M. U. Sardar and C. Fetzer, "Confidential computing and related technologies: a critical review," *Cybersecurity*, vol. 6, no. 1, p. 10, 2023. [Cited on page 1.]

[2] A. Mohan, M. Ye, H. Franke, M. Srivatsa, Z. Liu, and N. M. Gonzalez, "Securing ai inference in the cloud: Is cpu-gpu confidential computing ready?" in *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, 2024, pp. 164–175. [Cited on page 1.]

[3] A. C. Yao, "Protocols for secure computations," in *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, ser. SFCS '82. USA: IEEE Computer Society, 1982, p. 160–164. [Cited on page 1.]

[4] C. Kocaoğullar, T. Marjanov, I. Petrov, B. Laurie, A. Cutter, C. Kern, A. Hutchings, and A. R. Beresford, "A confidential computing transparency framework for a comprehensive trust chain," 2024. [Online]. Available: https://arxiv.org/abs/2409.03720 [Cited on page 2.]

[5] R. Zhang, A. Cheu, A. Gascon, D. Moghimi, P. Schoppmann, M. Schwarz, and O. Suciu, "Farfetch'd: A side-channel analysis framework for privacy applications on confidential virtual machines," 2025. [Online]. Available: https://arxiv.org/abs/2506.15924 [Cited on page 2.]

[6] E. Machado de Sousa and A. Shahzad, "Data loss prevention from a malicious insider," *Journal of Computer Information Systems*, vol. 62, no. 6, pp. 1101–1111, 2022. [Cited on page 2.]

[7] N. Alhebaishi, L. Wang, S. Jajodia, and A. Singhal, "Modeling and mitigating the insider threat of remote administrators in clouds," in *Data and Applications Security and Privacy XXXII*, F. Kerschbaum and S. Paraboschi, Eds. Cham: Springer International Publishing, 2018, pp. 3–20. [Cited on page 2.]

[8] J.-C. Graf, S. Rüegge, A. Hajiabadi, and K. Razavi, "Vmscape: Exposing and exploiting incomplete branch predictor isolation in cloud environments." [Cited on page 3.]

[9] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, 2015, pp. 57–64. [Cited on page 7.]

[10] J. V. Bulck, F. Piessens, R. Strackx, M. Minkin, M. Silberstein, O. Weisse, D. Genkin, B. Kasikci, T. F. Wenisch, and Y. Yarom, "Foreshadow: Extracting the keys to the intel sgx kingdom," in *Proceedings of the 27th USENIX Security Symposium*, 2018. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck [Cited on page 18.]

[11] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom, "Sgaxe: How sgx fails in practice," 2020. [Cited on page 18.]

[12] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "Microscope: Enabling microarchitectural replay attacks," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 318–331. [Cited on page 18.]

[13] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19. [Cited on page 18.]

[14] ARM Limited, "Arm security technology: Building a secure system using trustzone technology," ARM Limited, White Paper, April 2009, available at: https://developer.arm.com/documentation/PRD29-GENC-009492C/. [Cited on page 18.]

[15] Advanced Micro Devices, Inc., "Amd memory encryption: Secure encrypted virtualization," Advanced Micro Devices, Inc., Tech. Rep., 2016. [Online]. Available: https://example.com/amd-memory-encryption.pdf [Cited on page 20.]

[16] D. Miladinović, A. Milaković, M. Vukasović, Z. Stanisavljevic, and P. Vuletic, "Secure multiparty computation using secure virtual machines," *Electronics*, vol. 13, p. 991, 03 2024. [Cited on page 22.]

[17] Intel Corporation, "Intel trust domain extensions (intel tdx) white paper," Intel Corporation, White Paper, February 2022, available at: https://cdrdv2-public.intel.com/690419/TDX-Whitepaper-February2022.pdf. [Cited on pages 22, 26, and 28.]

[18] Intel, "Architecture specification: Intel trust domain extensions (intel tdx) module," Intel Corporation, Tech. Rep., 2025, [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/733575. [Cited on page 33.]

[19] W. Arthur, D. Challener, and K. Goldman, *A practical guide to TPM 2.0: Using the new trusted platform module in the new age of security*. Springer Nature, 2015. [Cited on page 33.]

[20] G. Cloud. (2025) Google cloud platform (gcp). [Online; accessed 25-September-2025]. [Online]. Available: https://cloud.google.com [Cited on page 37.]

[21] G. C. Confidential. (2025) Confidential vm overview. [Accessed 25-September-2025]. [Online]. Available: https://cloud.google.com/confidential-computing/confidential-vm/docs/confidential-vm-overview [Cited on page 37.]

[22] Google. (2025) go-tpm-tools: Tools and libraries for using tpm 2.0 in go. https://github.com/google/go-tpm-tools. [Accessed 25-September-2025]. [Cited on page 37.]

[23] G. Cloud. (2025) Attestation token claims. [Online; accessed 25-September-2025]. [Online]. Available: https://cloud.google.com/confidential-computing/confidential-space/docs/reference/token-claims [Cited on page 39.]

[24] Canonical, "tdx: Intel tdx support for ubuntu," https://github.com/canonical/tdx, 2025, [Online; accessed 25-September-2025]. [Cited on page 40.]

[25] M. Azure. (2024) Confidential computing vm options in azure. [Accessed 25-September-2025]. [Online]. Available: https://learn.microsoft.com/en-us/azure/confidential-computing/virtual-machine-options [Cited on page 40.]

[26] A. Cloud. (2024) Build a tdx confidential computing environment. [Accessed 25-September-2025]. [Online]. Available: https://www.alibabacloud.com/help/en/ecs/user-guide/build-a-tdx-confidential-computing-environment [Cited on page 40.]

[27] Intel, "tdx-linux: Intel tdx support for the linux kernel," https://github.com/intel/tdx, 2025, [Online; accessed 25-September-2025]. [Cited on pages 41 and 42.]

[28] QEMU Project, "Intel trusted domain extension (tdx)," 2025, accessed: 2025-09-26. [Online]. Available: https://www.qemu.org/docs/master/system/i386/tdx.html [Cited on page 42.]

[29] "Cloud hypervisor: Run cloud virtual machines securely and efficiently," https://www.cloudhypervisor.org/, 2025, accessed: 2025-10-03. [Cited on page 42.]

[30] I. Corporation, "Intel® tdx virtual firmware design guide," Intel Corporation, Tech. Rep. 344991-004US, Dec. 2023, accessed: 2025-09-26. [Online]. Available: https://cdrdv2-public.intel.com/733585/tdx-virtual-firmware-design-guide-rev-004-20231206.pdf [Cited on page 42.]

[31] Intel, "Guest os setup," https://cc-enabling.trustedservices.intel.com/intel-tdx-enabling-guide/06/guest_os_setup, Jun. 2025, accessed: 2025-09-26. [Online]. Available: https://cc-enabling.trustedservices.intel.com/intel-tdx-enabling-guide/06/guest_os_setup [Cited on page 42.]

[32] G. D. C. Systems and Data61, "sel4 microkernel," https://sel4.systems/, 2025, accessed: 2025-09-26. [Cited on page 43.]

[33] A. S. Tanenbaum and M. Project, "Minix 3: A highly reliable, self-repairing operating system," https://www.minix3.org/, 2025, accessed: 2025-09-26. [Cited on page 43.]

[34] B. Limited, "Qnx — high-performance embedded solutions," https://blackberry.qnx.com/en, 2025, accessed: 2025-09-26. [Cited on page 43.]

[35] seL4 Foundation, "seL4 reference manual, version 13.0.0," seL4 Foundation, Tech. Rep., jul 2024, accessed: 2025-09-28. [Online]. Available: https://sel4.systems/Info/Docs/seL4-manual-latest.pdf [Cited on page 43.]

[36] seL4, "sel4 in use," https://sel4.systems/use.html, 2025, accessed: 2025-09-28. [Cited on page 43.]

[37] E. de Matos and M. Ahvenjärvi, "sel4 microkernel for virtualization use-cases: Potential directions towards a standard vmm," *Electronics*, vol. 11, p. 4201, 12 2022. [Cited on page 43.]

[38] B. Dinis, P. Druschel, and R. Rodrigues, "Rr: A fault model for efficient tee replication," in *The Network and Distributed System Security Symposium*. Internet Society, 2023. [Cited on page 46.]