Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Taxonomy of Trusted Execution Environment Technologies

Author
Yifan Wu

Supervisor:
Lluis Vilanova

Submitted in partial fulfillment of the requirements for the MSc degree in Advanced Computing of Imperial College London

May 2022

Contents

Chapter 1

Introduction

Given that the importance of the data security is growing, Intel, ARM and other vendors all have been published Trusted Execution Environment (TEE) extensions on their processors to protect the secrets of PC users or cloud service tenants. However, there are huge differences among these TEE implementations since they are based on different platforms and their targets are not equal. Therefore, it is hard to compare these implementations when we want to find a most suitable one to satisfy a specific application requirement.

There were some surveys[17, 13], but they only concentrated on a specific hardware platform or a small number of individual properties. Therefore, we cannot gain a comprehensive understanding of these technologies through them. By contrast, the target of this report is to treat the TEE as an complete entity and build a framework to reason about it systematically. After that, it will be easier to understand the similarities and differences between these TEE implementations.

The main contributions of this report are:

- Providing detailed background knowledge of TEE to readers who are not familiar with TEE in Chapter 2.
- Reasoning about the TEE in a systematic way and extracting several functionality and performance metrics used for evaluating TEE implementations. These metrics are also categorized properly so that they are easy to be extended in Chapter 3.
- Evaluating several TEE implementations from various platforms as examples to find the relationship between these metrics in Chapter 3.

Chapter 2

Background

2.1 Basic Definitions

In this section, we list and explain several basic terminology used throughout this report. To begin with, we should know what do we mean by **trust**. When we say that we trust a component or a system which can interact with us and handle our requests, it means that we believe that this component or system can never be compromised by any adversary in any way. It also means that we believe that the functional and security requirements the component or system guarantees will always be satisfied independent of the way we interact with it. Generally, in a complicated system, some hardware of software components are trusted while other components are untrusted. In order to guarantee security of the whole system, we should prioritize protecting these untrusted components since they are assumed to be more likely to be compromised.

Based on the definition of trust, now we can introduce **Trusted Execution Environment (TEE)** and **Trusted Computing Base (TCB)**. TEE, the main topic of this report, is an abstraction of an execution environment where security-sensitive programs can be securely executed, which means that TEEs are isolated from the untrusted components of the system so that the confidentiality and integrity of the code and data of these programs can be guaranteed. We will give precise definition of confidentiality and integrity later. TEE's functionality and security requirements are implemented by TCB, which is a set of software and hardware components which are trusted by the system designer in the system.

Additionally, the term **Enclave** is also frequently used in this report, which is firstly used by Intel[14] to refer to the TEE implementation on a SGX-enabled platform. However, a number of later research(e.g., PENGLAI Enclave [9] and Sanctum[8]) also used it to name their own TEE implementations. Therefore, we take advantage of it to refer to all TEE implementations across various platforms.

2.2 Security Requirements

In the TEE's context, there are four main security requirements:

- **Integrity**: Adversary cannot modify the content of TEE without being detected.
- **Confidentiality**: No secret can be revealed from the content of TEE by any adversary.
- **Authenticity**: It must be assured that the TEE is based on a trusted hardware platform and TEE's code and data is just as expected before interacting with the TEE.
- **Availability**: The availability of TEE-related services like creating a new TEE or calling APIs provided by a TEE when users request these services.

2.3 Protection Methods

2.3.1 Core Mechanisms

Having discussed what the TEE is and what security requirements the TEE should guarantee, in this section we will discuss what protection methods are widely utilized to construct a TEE. There are 2 core mechanisms which are essential for ensuring the security of TEE: **hardware-based isolation** and **cryptographic algorithms**.

Hardware and software co-design has become mature in building a modern secure computer system, with the hardware side being more reliable than the software side since it is more difficult to compromise the hardware. A well-known example is that the Memory Management Unit (MMU) in the CPU is used to create an isolated address space for each process so that the code and data of one process can only be accessed by itself if the process does not share its data with other processes via some inter-process communication methods such as pipes or shared memory. Under some cases, even the privileged software like kernel cannot access the address space of a process directly.

Similarly, we create isolated execution environments based on dedicated hardware modules, which is called hardware-based isolation. There are several ways to provide isolation required by TEE based on hardware. The first one is called hardware-based access control, which means that the accessible resources and allowed operations are limited and checked by the hardware depending on the current state of the CPU. For example, on RISC-V platform[22], the software executed under User mode is not allowed to execute privileged instructions such as *sfence.vma* or access privilege Control and State Registers (CSRs). If it tries to do that, an invalid instruction exception will be generated and handled by privileged software of Supervisor or Machine mode, leading to the result that the execution of the possibly malicious User mode software is halted. How can we implement a TEE based on this mechanism? In TEE's perspective, the whole computer system is divided into two parts, which are secure and non-secure, respectively. The secure and non-secure parts are also called Trusted World and Normal World respectively in ARM TrustZone Technology[23]. All the software and hardware components in the Trusted World are trusted while those in the Normal World are untrusted. In order to ensure the

security requirement of the Trusted World, the hardware can prevent the code executed in the Normal World from accessing the resources in the Trusted World. The resources may include the part of DRAM specially allocated to the Trusted World, on-chip storage which can only be accessed by the Trusted World and interrupts from trusted peripherals which should only be caught and handled by the Trusted World. The second one is called physical resource partition, which means that the hardware resource such as part of DRAM or cache accessed by different Worlds are physically isolated. Therefore, there is no way for the code in the Normal World to affect the execution of the Trusted World directly. Oppositely, the last way is called secure resource sharing. In this method, some hardware resources are shared by multiple Worlds. However, the secrets stored in these hardware resources must be moved to elsewhere and then cleared by hardware or software when CPU switches between Worlds. In this way, the execution of the Trusted World can also be protected against the Normal World.

Although hardware-based isolation is strict and versatile, only relying on it is not sufficient to meet all the security requirements of TEE. This is because the hardware-based isolation requires privileged software such as the operating system or hypervisor to cooperate with the hardware (e.g., managing hardware resources) so that we need to trust these privileged software. Nevertheless, it is not reliable to do so since the privileged software such as Linux kernel is very complicated today and it does contains a number of vulnerabilities. Consequently, it is likely to be partially or even fully controlled by adversaries. Therefore, the recent trend is that a part of privileged software is removed from TCB and put into the Normal World. In this case, cryptographic algorithms is an necessary complement to hardware-based isolation. The main idea of cryptography is creating a secure communication channel which guarantees confidentiality and integrity of the message passed through the channel from a non-secure medium where none of security requirements are guaranteed. There are several applications of cryptography in TEE implementation. For instance, symmetric/asymmetric encryption, digital signature, authorized certificate, attestation are all basic cryptographic primitives widely used in TEE technologies, which we will discuss in detail later.

2.3.2 Cryptography-Derived Methods

To make it clear, in this section we discuss some protection methods in TEE technologies derived from cryptography we mentioned above.

Secure Boot. Secure Boot is used to ensure the trustworthiness of the software part of the TCB. Given that these software are stored on persistent storage before the system boots and the software has not been loaded to the memory, there is a risk that the software may be maliciously modified or substituted by an adversary, making the TCB no longer reliable. Secure Boot means that all the stages of the booting process should be verified. We start from a trusted secret stored in on-chip storage, which is called the Root of Trust (RoT). Most of the time, the RoT is a kind of secret key. We trust RoT without any condition since it is stored on-chip so that there is no way for an adversary to read or modify it. Based on the RoT, before loading the code

and data of the each booting stage into memory and executing the code, we should ensure the integrity of the code and data by calculating the hash of the code and data and then compare it with a pre-calculated value. If they do not match, then the system cannot be booted or sensitive operations are disabled. In other words, Secure Boot expands the TCB from only the RoT to multiple layers of software. Even if Secure Boot is essential, in this report we do not put it into consideration.

Data Encryption. Before data with secrets flows from trusted software/hardware to untrusted software/hardware, the data should be encrypted. Otherwise, the confidentiality of the data cannot be guaranteed.

Integrity Check. When data flows back from the Normal World to the Trusted World and it is expected to stay the same as the data when it leaves the Trusted World, we should check whether the data has been modified by the adversary in the non-secure world to ensure the integrity of the data in the Trusted World.

Attestation. It is a procedure that a TEE prove to its user that the TEE is running on a platform where hardware-based protection is enabled and the code and data of the TEE is expected by the user. Mention that there are 2 types of attestation: One type is between 2 TEEs, the other one is between a TEE and a local/remote client.

Chapter 3

Taxonomy of TEE technologies

3.1 Metrics

In this chapter, we start to evaluate and compare the effectiveness of some representative TEE implementations of commercial platforms such as Intel SGX and ARM TrustZone or published academic works. At first, we need to determine the metrics of the evaluation, which can be further divided into functionality metrics and performance metrics.

3.1.1 Functionality Metrics

The functionality metrics describe to what extent the given TEE implementation can guarantee the security requirements mentioned in Chapter 2. In other words, it depends on whether the TEE implementation can protect the execution with secrets against a variety of attacks.

Unverified Lines of Code of the software in the TCB (ULoC). All the software in the TCB is assumed to be trusted for simplicity of the threat modeling. However, sometimes it is not true. For example, there may be vulnerabilities in these software, leading to a risk that the software is compromised and misused by the adversary, which means that the security requirements of the whole system are no longer guaranteed due to the TCB. In order to detect and eliminate the vulnerabilities in these software, we can leverage some mature methods like fuzzing, dynamic symbolic execution, sanitizers, static program analysis and even formal verification to verify the correctness of the software implementation at compile time or runtime. If the ULoC in a TEE implementation is high, this means that its TCB is not secure to some extent. Mention that ignore of the effects of the compiler or toolchain since they are out of the boundary of this report.

Confidentiality and integrity of the data on DRAM. It has been a consensus that we can trust the on-chip storage since it is embedded in the System-on-Chip (SoC) so that we believe that it can never be compromised by any adversary. However, the on-chip storage's capacity is extremely limited, which is of only several KiBs or MiBs, which means that we also need a DRAM for data transferring and temporary storage. Unfortunately, there have been a number of attacks against the DRAM,

making it untrusted. For example, the adversary can read or modify the content of the DRAM even without the CPU since he can substitute the DRAM with another one which works like a normal DRAM but it is controlled by the adversary. The adversary can also control the bus between the CPU and the DRAM to snoop or modify the signal on the bus. If the cloud service provider is an adversary, the confidentiality and integrity of cloud tenants' data will be challenged for the reason that the cloud service provider can physically control a range of hardware including the DRAM and the bus.

The attacks above are transparent to software. This is because the software cannot predict whether these hardware is substituted. As a result, we need dedicated on-chip hardware module to protect the data against these attacks in 2 ways. The first way is *data encryption*. before the data which contains secrets leaves CPU (e.g., from Last-Level Cache to the bus), the data should be encrypted by the hardware module so that no secrets can be revealed from the encrypted data even if it is stored on an untrusted DRAM. When the CPU fetch data from DRAM on a cache miss, the hardware module decrypts the data so that the original data is visible to CPU and instructions. We can see that the data encryption can ensure the confidentiality of the data. The second way is *integrity check*, which aims to guarantee the integrity of the data, which means that if the data stored on DRAM is modified in an unexpected way, it should be detected. Integrity check can be implemented by storing the hash of the data which needs to be protected on-chip before the data leaves CPU. In this way, we can compute the hash of the data on DRAM and check if the computed hash matches the hash stored on-chip previously.

There are a lot of technical details about how data encryption and integrity check are implemented, e.g. how to manage the secret keys used in encryption and hashing and choices of cryptographic algorithms, through which we can see the trade-off between security and performance.

Confidentiality and integrity of the data related to I/O. Similar to DRAM, CPU also need to transfer data between all kinds of peripherals. If the I/O devices are not trusted or maliciously controlled by adversaries, then the same issues as those in DRAM occurs. Apparently, We can take advantage of data encryption or integrity check on the near-CPU side to prevents similar attacks. However, sometimes it is more convenient and efficient to protect the data directly on the peripheral side. For example, we can attach a dedicated module called accelerator on the peripheral to protect the communication between CPU and the peripheral.

Secure interrupts. Interrupts raised by trusted peripherals can only be handled by trusted software, while these interrupts should be hidden for untrusted software. It is important in some scenarios and it requires hardware support such as the specialized interrupt controller.

Resistance of side-channel attacks. Side-channel attacks are a type of attacks based on the defects of the CPU microarchitecture design. These attacks can break the intended isolation semantic provided by the CPU by observing the target data indirectly through hardware resources shared across isolated executions like caches or even the branch predictor in the CPU. Relying on the software can only partially address this issue. For example, the Kernel Page Table Isolation (KTPI)[16] in the Linux

kernel can mitigate the Meltdown attacks[12] on mainstream platforms. Essentially, we need to further enhance the effectiveness of the isolation semantic based on hardware modifications such as physical resource partition or secure resource sharing. However, we have to take the extra cost of the resource management and latency of context switches.

Support for attestation. As we mentioned before, attestation is widely used and critical for ensuring the authenticity of the TEE, and it also requires hardware support.

Here we can briefly summarize these functionality metrics. One way of classifying them is with respect to the structure of TEE of three layers. The underlying layer is the TCB that the TEE relies on, which we evaluate using the ULoC. The intermediate layer is about the internal state (i.e, code and data) of the TEE, we need to protect the internal state against the untrusted hardware including DRAM and peripherals as well as software when the TEE is interacting with them. The upper layer is about the quality of services provided by the TEE. Attestation fails in this layer since it can significantly increase the trustworthiness of the TEE when the TEE is used for confidential computing. Another way is just dividing all the attacks into direct or indirect attacks. The representative of direct and indirect attacks are physical attacks on DRAM/bus and side-channel attacks, respectively. However, the boundary between direct and indirect attacks is vague.

This report focuses on the most important metrics. Therefore, the availability and control flow integrity of TEE are not covered in this report. The adversary can conduct Denial-of-Service attacks to break the availability of the TEE, but the confidentiality and integrity of the data are not affected. For the control flow integrity, it is subtle thus out of the boundary of this report. Moreover, we decide not to discuss attacks related to the API provided by the privileged software. For example, the Iago attack[6] is that the compromised operating system can affect the execution of applications by just changing the return values of the system calls. The Iago-like attack also makes sense even if the privilege software is assumed to be untrusted in the threat model. This is because the classical design of the system call interfaces implies that the privilege software is trusted to some extent. We can see that the interaction between unprivileged software and privileged software is very complicated so that we do not discuss it further.

3.1.2 Performance Metrics

In addition to the functionality metrics, we also need to evaluate the performance overhead of TEE as an extra level of abstraction, which can be divided into 3 categories: management overhead, scalability limitation and external access overhead.

Management Overhead

Similar to processes and threads, TEE can be seen as a kind of *container* where code executes inside it. Thus, TEE should be properly managed by the system, i.e., the system should allocate resources to TEEs, which incurs some management overheads.

The management overhead is dominated by TEE creation latency and context switch overhead.

Container creation latency. The first one is Container creation latency, which is critical especially when TEE technologies are used for cloud computing. For example, in serverless computing, we dynamically create a TEE to execute a code fragment or a function on a request from clients. However, as mentioned in [9], based on Intel SGX[14] platform, it takes several seconds to create a TEE while the total execution time of the code is less than one second most of the time, which means that the TEE creation becomes the bottleneck.

Context switch overhead. The second one is context switch overhead. According to the implementation of the TEE, some extra works should be done on a context switch between TEEs or between TEE and the external environment. For instance, in order to enhance the isolation semantic provided by the CPU, we can leverage secure resource sharing, which means that the data with secret stored in the shared resources should be cleared on a context switch. In addition, depending on the structure of the software stack, a change of the privilege level of the CPU may be required on a context switch, leading to the processor state pollution mentioned in [19], which significantly breaks the locality of caches in the CPU and degrades the overall performance.

Scalability Limitation

The scalability of the TEE is critical since we want to keep the same level of resource utilization after adding more machines to the system to maximize the performance. However, in some TEE implementations, the scalability may be limited due to some reasons. There are 4 metrics related to the scalability:

- *Concurrency*: The level of concurrency can be evaluated using the maximum number of concurrent enclaves at a time in the system. A higher level of concurrency means that the system can accept more requests over a fixed time period.
- *Parallelism*: The level of parallelism can be represented by the maximum number of threads running the same enclave at a time. A higher level of parallelism means that the execution of a single enclave can be more efficient based on multithreading.
- *Storage overhead*: In order to implement the TEE, some extra metadata should be stored as well as the data. These metadata may be stored on-chip, on DRAM, or on external storage, leading to the consumption of storage space, which also limits the scalability.
- *Performance degradation on huge workloads*: Some workloads may be large in terms of the size of the code and data. In some TEE implementations, the performance degrades significantly on huge workloads. For example, when a workload is larger than 128MiB on Intel SGX platform[14], the performance degrades a lot.

External Access Overhead

It is not sufficient for a TEE to only work on its own code and data. Sometimes the TEE also need to interact with the external environment to meet its functionality requirements. The TEE mainly accesses the external environment in 2 ways: calling external functions and accessing DRAM or peripherals.

Calling external functions. In order to guarantee the security requirements of the TEE, the functionality of the code inside the TEE is somehow limited. Thus, the TEE need to call the external functions provided by another TEE or the Normal World. Similar to the context switch overhead, this may requires a change of the privilege level of CPU. Additionally, some intermediate functions also need to be called for security consideration, leading to an extra overhead.

Accessing DRAM or peripherals. Transferring data with DRAM or peripherals is a core part of the functionality of TEE. Since we assume that the DRAM and peripherals are all untrusted, we cannot directly access them. Otherwise, the confidentiality and integrity of the data cannot be guaranteed, which conflicts with the definition of TEE. As we mentioned earlier, we can take advantage of data encryption and integrity check to ensure the confidentiality and integrity of the data when communicating with DRAM and peripherals, respectively. Obviously, these 2 methods incur extra computational costs.

3.2 Classical Containers

Before evaluating TEE technologies, we analyze 2 classical containers, i.e. process and virtual machine, using the functionality and performance metrics given in section 3.1 to set up a baseline so that we can compare it with other TEE technologies to show the effectiveness of TEE technologies. For this reason, we will measure the functionality metrics in the most conservative way to match the non-secure environment TEE works in, which means that we assume that the hardware except for the CPU and the software which is not in the TCB are both untrusted so that they can be a start point for the adversaries to attack the system and break the security requirements.

3.2.1 Process

Process is the most important abstraction since the birth of Unix. It implements the isolation semantic between processes, which means that each process believes that it exclusively own all the resources in the computer system so that the process can use these resources freely. In fact, this is just an illusion created by the underlying software and hardware layers. To be specific, the operating system creates an virtual address space for each process and ensures that every physical memory page is mapped to at most one address space with the support of MMU. In this way, one process cannot access the data of another process without being detected since they are not in the same address space. This can be done through inter-process communication such as message queues or pipes which requires cooperation of involved

processes. When it comes to I/O, a process cannot access peripherals directly and it should rely on system calls provided by the operating system. The OS is responsible to ensure that the peripheral is assigned to at most one process and only this process is allowed to access the peripheral. Therefore, we can say that the confidentiality and integrity of the data on DRAM and peripherals are guaranteed if the OS and other privileged software layers are all trusted.

Nevertheless, we cannot assume that the privilege software including OS is trusted since it can be fully or partially compromised by the adversary, which is no longer rare today. For example, if the OS is controlled by the adversary, the OS can directly access the DRAM and peripherals, which is transparent to the processes. This means that the confidentiality and integrity of data on DRAM and peripherals cannot be guaranteed. In addition, the process itself do not support secure interrupts, defending against side-channel attacks or attestation. Briefly speaking, the process model is so vulnerable when facing with all kinds of attacks we mentioned before and this is why we need TEE to satisfy these security requirements. Fortunately, there is no TCB here so that the ULoC is 0.

As for performance, we can see that all the performance metrics are great for process. The creation latency is dominated by loading executable from persistent storage and copying the code and data into a new virtual address space, which is required. The context switch may involve a change of privilege level, which is time-consuming and inevitable. However, it is more lightweight compared to other TEE implementations. When it comes to scalability, we know that the number of processes, the number of threads per process are limited by resources rather than a certain value. The metadata of processes is minimal and it does not incur extra overheads if the size of code and data of the process is large. Now we discuss the external access overhead of processes. We know that the process uses system calls to access functions provided by the OS or other processes or interact with peripherals. At the same time, when the process accesses its virtual memory space, the virtual address should be translated to physical address by MMU if the virtual address cannot be found in the Translation Look-aside Buffer (TLB). Admittedly, the system calls and address translation leads to some overhead but they are compulsory and still more efficient than other TEE implementations.

3.2.2 Virtual Machine

Virtual machine means that multiple OS instances can run concurrently in a system. These OS instances are managed by the hypervisor or the Virtual Machine Monitor (VMM), which is an extra abstraction layer between OS and the hardware platform. Mention that the hypervisor run on a unique privilege level which different from the privilege level that applications or the OS runs on. The functionality of hypervisor is virtualizing the CPU, DRAM, I/O and interrupts to create an illusion for the OSs that they all exclusive own the whole computer. Therefore, all the context switches, external function calls and accesses of resources should bypass the hypervisor thus an extra change of the privilege level is required, which means that these operations are less efficient than the process. At the same time, one more address translation is

required when the application in a virtual machine accesses its virtual address space. To be specific, the virtual address needs to translated into an intermediate address which is further translated to the physical address. In opposite, the creation latency and the scalability of the virtual machine is not affected.

The virtual machine abstraction prevents one OS instance from accessing the resources allocated to another OS instance without detection. In other words, the isolation between OS instances can be guaranteed. However, similar to process, it assumes that the hypervisor is trusted. If the hypervisor is compromised, then the confidentiality and integrity of the data on DRAM and I/O cannot be guaranteed. The secure interrupts, defense against side-channel attacks and attestation are also not supported by definition of virtual machine. However, some extension of secure virtualization has been enabled on recent processors, which we will discuss in section 3.7.

3.3 Intel SGX

Software Guard Extension (SGX)[14] is an important extension published by Intel, which aims to protect the confidentiality and integrity of the data in enclaves, the name of the TEE implementation on Intel SGX-enabled processors, as well as satisfy other security requirements under the condition that only the processor itself is trusted. By contrast, the DRAM, bus, peripherals and the whole software stack except for the SGX Software Development Kit (SDK) are all untrusted in SGX's threat model. Therefore, cloud service providers can leverage SGX to make their remote customers believe that the confidentiality and integrity of the customers' data are guaranteed. Additionally, SGX can also be used to protect users' secrets against malicious processes or compromised OS on a desktop platform. For example, online payment and chat applications are both based on SGX. However, SGX has been deprecated on recent 11th and 12th Generation of Intel Core Desktop Processors and Intel Trusted Domain Extension (TDX) can be a substitute to SGX, which is discussed in [4].

SGX can ensure the confidentiality and integrity of the data on DRAM based on an on-chip hardware module called Memory Encryption Engine (MEE)[10] as an extension to the Memory Controller (MC) between the LLC and the DRAM. Some memory regions on the DRAM are used to store the data in enclaves, thus they should be protected by MEE. For the confidentiality, the MEE encrypts the data on the cache before the data is written back to DRAM. The integrity is more difficult to guarantee compared to the confidentiality since we have to record some information about the data written to DRAM so that we can check if the data has been replaced when the data is fetched back to the cache later. To be specific, a Message Authentication Code (MAC), which is similar to a kind of hash value, is recorded for every memory block (e.g., of 512 bits). In this way, the attacks that simply modifies the memory content can be detected. However, there is another attack called replay attack which is more subtle. Basically, it means that the adversary can observe the DRAM and record historical pairs of content and MAC of a memory block. After that, the adversary can substitute the content and MAC at the same time with a historical pair, which can

pass the MAC check.

In order to solve this problem, the MME takes advantage of a counter-based Merkle Tree design[18]. Every memory block has its own local counter and there is also a global counter. On every write access to a memory block, the local counter of this memory block and the global counter both increments. the MAC of a memory block is calculated using both its content and local counter. The local counters are organized as a multi-layer tree whose root is the global counter stored on-chip which all software is not allowed to read or modify directly. Every layer on the tree has its MAC as well. When we need to check the integrity of a memory block on the DRAM, we firstly ensure the integrity of its local counter by verifying the tree layer by layer starting from the root. After that, we can check the integrity of the memory block by computing the MAC using its content on DRAM and the local counter and compare it with its MAC stored on DRAM. In this way, we can ensure the confidentiality and integrity of the data on DRAM. However, this can incur a huge overhead. Mention that we need multiple times of calculation of MAC each time the data leaves or enters the cache, making the memory accesses 5.5% slower on average[10]. Furthermore, around 25% of the size of protected region should be used to store the Merkel Tree on DRAM, and about $\frac{1}{2^{15}}$ of that should be used to store the root of the Merkle Tree on-chip[10], which is an acceptable storage overhead.

The integrity tree and the protected region are in a DRAM area called Processor Reserved Memory (PRM) which only the enclaves are allowed to access. The size of the PRM is only 128MiB by default. When the total size of all concurrent enclaves is greater than 128MiB, the data will be moved between PRM and other regions on DRAM frequently similar to the issue of cache capacity misses, leading to the significant degradation of the performance.

SGX does not work well on other security requirements. SGX does not support secure I/O and interrupts by design although it is mentioned that SGX can reject DMA accesses in the protected region in [7]. Additionally, SGX cannot defend against side-channel attacks and some successful side-channel attacks on SGX can be found in [15]. This is because SGX does not clear secrets stored on shared resources[7], making the context switches efficient while making the system vulnerable to side-channel attacks. By contrast, SGX supports local and remote attestation, but it takes the cost of the high latency of the enclave creation. This is because when creating an enclave, all the data inside the enclave need to be accessed and a corresponding hash value should be calculated and stored for further attestation, which leads to high computing and memory overhead.

While SGX does not need to trust privilege software such as the OS or the hypervisor, SGX need to trust the SGX SDK[1] published by Intel which TEE applications are built based on, which is actually the TCB of SGX and is not verified. Cloud service providers may publish their own modified version of the SGX SDK to fit into their own software stack(e.g., [20]).

3.4 ARM TrustZone

TrustZone[23] is the TEE extension on ARM architecture since Armv6k and it is supported by Armv7-A and Armv8-A. Before TrustZone, there were 4 Exception Levels (ELs) in the privileged architecture of Arm-A: EL0 for applications; EL1 for OS kernels; EL2 for hypervisors; EL3 for the secure monitor. TrustZone further divides the privileged architecture into 2 parts: Trusted World and Normal World, which CPU runs in secure and non-secure mode respectively. The EL3 is always in the Trusted World and is the most essential in the TCB. By contrast, the EL0-2 are divided into 2 states depending on which world the EL resides in. For example, EL0 is divided into S.EL0 and NS.EL0 in Trusted World and Normal World, respectively. The software executed in these 2 states are different. The overhead of a context switch between Trusted World and Normal World is expensive since we need to bypass EL3 where an extra change of EL is required. All the software in the Trusted World are assumed to be trusted and they are in the TCB, but they are not verified in most systems. Therefore, the ULoC is high.

According to TrustZone's hardware architecture[21], the RAM, peripherals, interrupts are also divided into secure and non-secure parts. The secure part of these resources can only be accessed by the Trusted World while the Trusted World can access both secure and non-secure parts of these resources, which is guaranteed using the hardware-based access control. The TrustZone architecture contains a trusted RAM which is an on-chip SRAM of several hundreds of kilobytes. The trusted RAM enlarges the on-chip storage and may increase the area and power consumption of the chip. Since it is on-chip, we trust it so that the confidentiality and integrity of the data on it can be guaranteed even if it is directly and efficiently accessed without data encryption and integrity check. Moreover, the interconnection bus (e.g. AMBA AXI) and the Generic Interrupt Controller (GIC) are responsible for ensuring the secure I/O and secure interrupts, respectively. However, TrustZone itself does not support attestation and TrustZone are vulnerable to side-channel attacks[5].

The TrustZone's software stack is complicated since we have 7 different processor states. To be specific, in the Normal World, we have applications (NS.EL0), rich OS (NS.EL1) and virtual machine monitor (NS.EL2). In the Trusted World, we have trusted applications (S.EL0), trusted kernels (S.EL1), secure partition monitor (S.EL2) and the secure monitor (EL3). Since we are familiar with the Normal World, we can focus on the Trusted World. The trusted applications aims to provide secure services to normal applications and they are managed by the trusted kernels. However, for a normal application, successfully calling a function provided by a trusted application requires bypassing rich OS, virtual machine monitor and trusted kernels, and vice-versa. This means that the cost of calling an external function is high.

3.5 RISC-V

RISC-V[22] is an open-source architecture, which means that its specification is discussed and maintained by the community. There are also some open-source RISC-V processor implementations such as Rocket Core[2] or XiangShan[3], which allows

researchers to do some hardware-level modification to come up with a co-design consisting of both hardware and software. It is difficult for researchers to do so on Intel SGX or TrustZone since the hardware cannot be modified and some details about the hardware design are not published. Next, we discuss the following systems based on RISC-V architecture.

3.5.1 Sanctum

Sanctum[8] aims to simulate the enclave semantic from Intel SGX on RISC-V so that Sanctum is similar to Intel SGX. However, Sanctum is different from Intel SGX in these aspects:

- Sanctum's TCB only contains a secure monitor which run on machine mode. The secure monitor is only responsible for validating the resource allocation by the OS, thus it is more lightweight than the SGX SDK although the secure monitor is not verified either.
- Sanctum only considers software attacks and assumes the correctness of the implementation of the off-chip hardware including the DRAM. Therefore, it does not use data encryption or integrity check. As a result, the confidentiality and integrity of the data on DRAM can only be partially guaranteed while the overhead of accessing the DRAM is lower.
- Sanctum can protect the data against cache timing side-channel attacks by flushing per-core cache on a context switch between enclave and non-enclave mode, which also increases the overhead of context switches.
- Sanctum is not sensitive to huge workloads.

3.5.2 Keystone

Keystone[11] is an open TEE framework, which can customize the enclaves on demand. Keystone has a large TCB including the secure monitor, the runtime per enclave and the enclave application. Like TrustZone, Keystone takes advantages of a huge on-chip storage of 2MiB on FU540 which is exclusively used by one enclave. Keystone supports the secure interrupts in theory but it has not completely support it. Keystone supports attestation like SGX. Keystone can protect the data against cache side-channel attacks by cache partitioning, which is an example of physical resource partitioning.

3.5.3 PENGLAI Enclave

PENGLAI enclave[9] can guarantee the confidentiality and integrity of the data on DRAM while taking a significantly less cost than SGX. Based on the Host Page Table (HPT), a huge number of access checks can be avoided, which makes the memory accesses more efficient. Based on the Mountable Merkle Tree (MMT), PENGLAI

Table 3.1: Taxonomy Table of TEE Technologies

TEE	Functionality						Performance							
	ULoC	DRAM	I/O	Intr	SC	A	Creation	CS	C	P	S	H	EF	Access
Process	🟢	🔴	🔴	🔴	🔴	🔴	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢
VM	🟢	🔴	🔴	🔴	🔴	🔴	🟢	🟡	🟢	🟢	🟢	🟢	🟡	🟡
Intel SGX	🟡	🟢	🔴	🔴	🔴	🟢	🔴	🟢	🟢	🟢	🟡	🔴	🟡	🔴
TrustZone	🔴	🟢	🟢	🟢	🔴	🔴	🟢	🟡	🟢	🟢	🔴	🟢	🔴	🟢
Sanctum	🟢	🟡	🔴	🔴	🟡	🟢	🔴	🔴	🟢	🟢	🟡	🟢	🟡	🟢
Keystone	🟡	🟢	🔴	🟡	🟡	🟢	🔴	🔴	🟢	🟢	🔴	🟢	🟡	🟢
PENGLAI	🟢	🟢	🔴	🔴	🟡	🟢	🟢	🔴	🟢	🟢	🟡	🟢	🟡	🔴

supports huge workloads of a size up to 512GiB with minimal storage overhead. PENGLAI also supports attestation, but it can initialize an enclave very fast with shadow enclaves in a fork-style. PENGLAI can defend against cache side-channel attacks using cache line locking, which increase the overhead of context switches since special-purpose instructions should be executed.

3.6 Analysis

Table 3.1 concludes all the TEE technologies mentioned in this chapter. For functionality metrics, **DRAM** means confidentiality and integrity of data on DRAM while **I/O** means that related to peripherals. **Intr**, **SC** and **A** represents secure interrupts, resistance to side-channel attacks and attestation, respectively. For performance metrics, **CS** means overheads of context switches. **C, P, S, and H** mean Concurrency, Parallelism, Storage overhead and performance degradation on Huge workloads, respectively. **EF** represents overheads of accessing External Functions while **Access** means overheads of accessing DRAM or peripherals. The green, yellow and red bullets mean that the TEE under evaluation performs greatly, not badly, and terribly in terms of the metric of the current column, respectively.

It can be seen from the table that all TEEs fully or partially guarantee the data security on DRAM, while only TrustZone can ensure the I/O and interrupt security due to its special hardware architecture. In order to guarantee the data security on DRAM, either a trusted on-chip RAM (e.g., TrustZone and Keystone) or data encryption/integrity check is needed, with the latter increasing the overhead per memory access. Support for attestation leads to a high creation latency except for PENGLAI since it utilizes a fork-style fast enclave initialization. The ULoC and EF depend on the complexity of the software stack. Finally, in order to prevent side-channel attacks, we should take the extra cost of context switches during which the shared resources should be flushed.

Chapter 4

Conclusion

The target of this report is to provide a comprehensive understanding of a variety of TEE technologies. At the beginning, we explain some basic definitions and security requirements related to TEE. Then, we consider how these security requirements can be satisfied. We list and explain a number of protection methods which can be categorized into hardware-based isolation and cryptographic algorithms in detail. After that, we try to build a framework of evaluating TEE implementations, of which the functionality and performance metrics are the most important. On top of that, we analyze several TEE technologies from different platforms. Finally, we observe the taxonomy table to try to find the relationship between these metrics.

Due to the lack of time, only a tiny part of TEE technologies were analyzed and secure virtualization technologies (e.g., Intex TDX and AMD SEV) were not covered at all. In addition, more metrics can be added in the future.

References

[1] https://github.com/intel/linux-sgx. Accessed: 2022-04-24.

[2] https://github.com/chipsalliance/rocket-chip. Accessed: 2022-04-24.

[3] https://github.com/OpenXiangShan/XiangShan. Accessed: 2022-04-24.

[4] *Are TDX going to replace SGX?* https://github.com/Maxul/Awesome-SGX-Open-Source/blob/master/SGX-vs-TDX.md. Accessed: 2022-04-24.

[5] Sebanjila Kevin Bukasa et al. "How TrustZone could be bypassed: Side-channel attacks on a modern system-on-chip". In: *IFIP International Conference on Information Security Theory and Practice*. Springer. 2017, pp. 93–109.

[6] Stephen Checkoway and Hovav Shacham. "Iago attacks: Why the system call API is a bad untrusted RPC interface". In: *ACM SIGARCH Computer Architecture News* 41.1 (2013), pp. 253–264.

[7] Victor Costan and Srinivas Devadas. "Intel SGX explained". In: *Cryptology ePrint Archive* (2016).

[8] Victor Costan, Ilia Lebedev, and Srinivas Devadas. "Sanctum: Minimal hardware extensions for strong software isolation". In: *25th USENIX Security Symposium (USENIX Security 16)*. 2016, pp. 857–874.

[9] Erhu Feng et al. "Scalable Memory Protection in the {PENGLAI} Enclave". In: *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 2021, pp. 275–294.

[10] Shay Gueron. "A memory encryption engine suitable for general purpose processors". In: *Cryptology ePrint Archive* (2016).

[11] Dayeol Lee et al. "Keystone: An open framework for architecting trusted execution environments". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–16.

[12] Moritz Lipp et al. "Meltdown". In: *arXiv preprint arXiv:1801.01207* (2018).

[13] Pieter Maene et al. "Hardware-based trusted computing architectures for isolation and attestation". In: *IEEE Transactions on Computers* 67.3 (2017), pp. 361–374.

[14] Frank McKeen et al. "Innovative instructions and software model for isolated execution." In: *Hasp@ isca* 10.1 (2013).

[15] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. "A survey of published attacks on Intel SGX". In: *arXiv preprint arXiv:2006.13598* (2020).

[16] *Page Table Isolation*. https://www.kernel.org/doc/html/latest/x86/pti.html. Accessed: 2022-05-01.

[17] Sandro Pinto and Nuno Santos. "Demystifying arm trustzone: A comprehensive survey". In: *ACM Computing Surveys (CSUR)* 51.6 (2019), pp. 1–36.

[18] Brian Rogers et al. "Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly". In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE. 2007, pp. 183–196.

[19] Livio Soares and Michael Stumm. "{FlexSC}: Flexible System Call Scheduling with {Exception-Less} System Calls". In: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. 2010.

[20] *TEE-based confidential computing*. https://www.alibabacloud.com/help/en/container-service-for-kubernetes/latest/tee-based-confidential-computing-tee-based-confidential-computing. Accessed: 2022-04-24.

[21] *TrustZone System Architecture*. https://developer.arm.com/documentation/102418/0101/System-architecture?lang=en. Accessed: 2022-04-24.

[22] Andrew Waterman, Krste Asanovic, and John Hauser. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Document Version 20211203*. https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf. Accessed: 2022-04-24.

[23] *What is TrustZone?* https://developer.arm.com/documentation/102418/0101/What-is-TrustZone-?lang=en. Accessed: 2022-04-24.