Minimal Kernel: An Operating System Architecture for TEE to Resist Board Level Physical Attacks

Shijun Zhao[1], Qianying Zhang[2]*, Yu Qin[1], Wei Feng[1], Zhining Lv[3], *and* Dengguo Feng[1]
[1]*Institute of Software, Chinese Academy of Sciences, Beijing, China*
[2]*College of Information Engineering, Capital Normal University, Beijing, China*
[3]*Shenzhen Power Supply Bureau Co., Ltd., Shenzhen, China*

Abstract

ARM specifications recommend that software residing in TEE's (Trusted Execution Environment) secure world should be located in the on-chip memory to prevent board level physical attacks. However, the on-chip memory is very limited, placing significant limits on TEE's functionality. The *minimal kernel* operating system architecture addresses this problem by building a small kernel which executes the whole TEE system only on the on-chip memory on demand and cryptographically protects all the data/code stored outside of SoC. In the architecture, a small kernel is built inside the TEE OS kernel space and achieves the minimal size by only including the very essential components used to execute and protect the TEE system. The minimal kernel consists of a minimal demand-paging system, which sets the on-chip memory as the only working memory for the TEE system and the off-chip memory as a backing store, and a memory protection component, which provides confidentiality and integrity protection on the backing store. A Merkle tree based memory protection scheme, reducing the requirement for on-chip memory, allows the minimal kernel to protect large trusted applications (TAs). This OS organization makes it possible to achieve the goal of physical security without losing any TEE's functionality. We have incorporated a prototype of minimal kernel into OP-TEE, a popular open source TEE OS. Our implementation only requires a runtime footprint of 100 KB on-chip memory but can protect the entire OP-TEE kernel and TAs, which are dozens of megabytes.

1 Introduction

As the rapid development of mobile commerce and mobile applications for enterprises, such as BYOD (Bring Your Own Device), mobile devices store and process more and more security-sensitive data. To improve the security of mobile devices, ARM proposes the TrustZone security extension to its CPU architecture. By now, TrustZone technology has been widely studied [4, 30, 31, 33, 47, 50–52], and deployed in various of commercial products: almost all mainstream smartphone OEMs have integrated TrustZone in their products and leverage it to provide security services, such as Samsung pay, Huawei pay, and KNOX [46].

As TrustZone is becoming a popular hardware security architecture for mobile devices, it is important to ensure the security of TrustZone itself. TrustZone is designed to only resist software attacks and cannot resist physical attacks. Unfortunately, the feature that mobile devices can easily be stolen or lost puts sensitive data stored in TrustZone at the risk of physical attacks. What's worse, there exists a class of physical attacks which are low-cost and easy to set up: board level physical attacks.

Board level physical attacks target sensitive data in DRAM, such as cryptographic keys and passwords. This class of attacks usually is launched by tampering DRAM or snooping memory buses between CPU and DRAM. Several types of board level physical attacks have been developed: cold boot attacks [24], bus snooping attacks, and DMA attacks. There are some well-known successful attacks on commercial products: cold boot attacks on Galaxy Nexus smartphones [40], bus snooping attacks on Xbox [29] and PlayStation 3 [28], and there even exist attacks on security chips such as D-S5002FP [32] and TPM [41]. These attacks are inexpensive because they only requires some cheap tools, such as memory bus probes. Even worse, mature attack tools [14, 16, 40, 42] have been released publicly, with which hackers can reproduce attacks easily.

The way to prevent board level physical attacks is memory encryption. Most CPU manufactures have added this feature in their products, such as Apple's secure enclave technology [2], Intel's SGX [1, 37], IBM's SecureBlue [45] and SecureBlue++ [5]. Take Intel SGX for example, it addresses board level physical attacks by constructing an isolated memory enclave, which is transparently encrypted by a separate memory encryption engine (MEE). However, ARM CPUs, which dominate mobile devices, are not equipped with memory encryption technologies, so TrustZone cannot pre-

*Corresponding author

vent physical attacks. Fortunately, software-based approaches called *SoC-bound execution environments* are proposed, which run applications in CPU registers [17, 38, 39, 48], CPU cache [20, 21, 58, 59], GPU registers and cache [54], or on-chip memory (OCM) [8, 19, 25, 43].

Although the state-of-the-art software-based approaches can prevent some specific board level physical attacks, they cannot protect mature TEE OSes from board level physical attacks for the following two reasons. First, *limited space*: most SoC-bound execution environments can only protect a small piece of code because they require the protected application to be loaded into the execution environment entirely, while the size of CPU registers, CPU cache or on-chip memory is quite limited, usually about a few hundreds kilobytes. However, the size of a mature TEE OS is much larger. Take OP-TEE [35] for example, its kernel address space is about 1 megabyte, and its user address space can reach up to a few megabytes. Therefore, it is impossible to load the whole TEE software into such space-limited execution environments. Second, *insecurity under full power board level physical attacks*: board level physical attackers have abilities of intercepting information and injecting code via the buses between CPU and DRAM [12, 13], so they can inject malicious code into the address space where the SoC-bound execution environment is contained. Unfortunately, this security problem is not considered by most approaches. In most approaches, the address space containing the SoC-bound execution environment is the kernel space. Only the SoC-bound execution environment is located in the memory of SoC, and the other code and data of the kernel address space is located in DRAM in plaintext. An attacker can launch his attack as follows (Figure 1): when CPU runs code outside the SoC-bound execution environment, it will fetch the code from DRAM, and the attacker replaces the code transmitted on the bus with his malicious code. The injected malicious code is able to access sensitive data in the SoC-bound execution environment because they are in the same address space (kernel space).

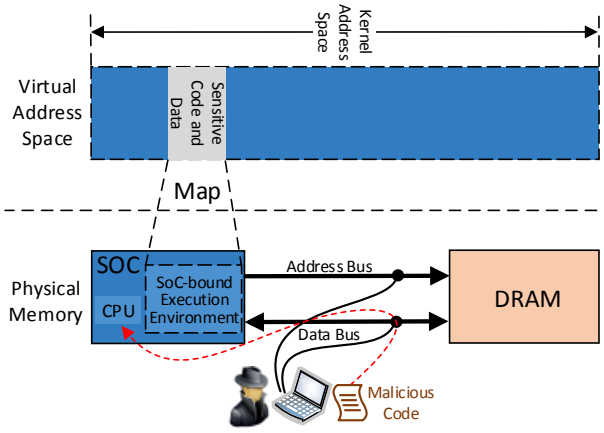

Figure 1: An Injection Attack against SoC-bound Execution Environment

The bus injection attack (Figure 1) shows that, under full power board level physical attacks, if some part of the address space where the SoC-bound execution environment resides is mapped to DRAM without memory protection, attackers can inject malicious code into the data bus when CPU fetches code from the unprotected address space. Thus, it is necessary to provide memory protection on the address space that is mapped to DRAM. Many approaches [17, 20, 21, 38, 39, 48] implement SoC-bound execution environments as kernel modules. Therefore, the rest kernel address space, which is located in the DRAM, should be protected. However, most approaches do not provide memory protection on the rest kernel address space. Some approaches [25, 59] can prevent board level physical attacks by creating an address space only including the SoC-bound execution environment, which would put a significant limit on the environment's functionality.

To overcome the above restriction, we have designed a new TEE OS architecture, the *minimal kernel*, which leverages the size-limited OCM to provide memory protection on the whole TEE system. The key design feature of the architecture is to extract a *minimal kernel*, consisting of the very essential components required to maintain the execution of the whole TEE system and cryptographically protect the whole TEE system, from the TEE OS kernel; the rest of the TEE OS kernel is called the *main kernel*. In the architecture, the OCM is set as the only working memory and DRAM as a backing store for the TEE system. The minimal kernel (permanently residing in the OCM) leverages the demand-paging mechanism to maintain the execution of the main kernel and TAs (initially stored in the DRAM), and protects the confidentiality and integrity of the main kernel and TAs.

One crucial requirement for the minimal kernel is that it should be as small as possible to reduce its demand for OCM. The minimal kernel also should be self-contained because its execution should not rely any components of the main kernel. To meet the above requirements, we propose a principle for the design of the minimal kernel, whose main idea is that a kernel component is tolerated inside the minimal kernel only if moving it outside the minimal kernel would prevent the CPU from running software normally.

In addition to the demand-paging functionality, the minimal kernel protects the confidentiality and integrity of the main kernel and TAs stored in DRAM. Two integrity protection schemes are designed: a trivial scheme requiring that all the integrity values should be stored in the OCM, and a Merkle tree based scheme that only requires secure storage of the root node but requires more computations. The two schemes offer system designers a trade-off between the size and the performance of the minimal kernel.

Leveraging the above demand-paging and memory protection mechanisms, the minimal kernel architecture protects the whole TEE system against physical attacks but only requires the minimal kernel to reside in the OCM, making it possible to protect mature TEE systems whose sizes are large.

We have implemented a prototype by retrofitting the minimal kernel architecture into a mature TEE OS: OP-TEE [35]. Although OP-TEE's kernel space reaches up to a few megabytes, the minimal kernel only requires a runtime footprint of 100 KB OCM, proving that our principle is very effective. We deploy the following security policy to protect the main kernel: for code sections, we guarantee their integrity; for data sections, we guarantee both of their confidentiality and integrity. Our prototype also provides memory protection for TAs. One benefit of enabling memory protection in the kernel level is enforcing TA encryption transparently, which reduces programmers' burden. Our evaluation shows that compared with OP-TEE's original memory protection solution, *Pager* [36], the minimal kernel architecture not only reduces 44% of OCM requirement but also improves performance greatly. The main contributions of this paper are:

- The minimal kernel architecture that provides a software-based approach to full system encryption for TEE systems. It enables a TEE system with rich functionality to obtain a high level of physical security without requiring specialized memory encryption hardware.
- A principle of building the minimal kernel, by which we can identify the very essential components that are required to maintain the execution of a TEE system and implement the minimal kernel with the minimum amount of software.
- A minimal kernel prototype based on a mature TEE OS: OP-TEE. We implement a minimal kernel for OP-TEE, called mTEE. mTEE resides in the OCM, restricts the execution of OP-TEE within the OCM by swapping pages between OCM and DRAM on demand, and protects the confidentiality and integrity of the whole OP-TEE system.
- Performance evaluations of our prototype under the following memory security policy: protecting the integrity of the code sections and the confidentiality and integrity of the data sections.

The rest of the paper is organized as follows. Section 2 gives the background information related with this paper. Section 3 describes the threat model. Section 4 illustrates the design of the minimal kernel architecture. Section 5 depicts the details of the implementation. Section 6 evaluates the performance overhead. Section 7 discusses how the minimal kernel architecture can be improved by hardware enhancement. Section 8 surveys related work. Section 9 concludes this paper.

2 Background

2.1 The Security of TrustZone

Attacks on computer systems can be classified into software attacks, physical attacks, and side channel attacks, and we do not consider side channel attacks in this paper. Physical attacks refer to board level attacks and chip level attacks. Board level attacks are launched at PCB (printed circuit board) level and leverage chip wired interfaces, such as the buses between DRAM and CPU. Chip level attacks target at on-chip secrets, and require depackaging of the chip to direct access to its internal components. We do not consider chip level attacks because they require high qualified specialists and specialized equipment such as microprobing and FIB workstations.

The TrustZone technology is able to resist software attacks by isolation. It partitions all resources of the platform (CPU, memory, and peripherals) into two worlds: the secure world and the normal world. TrustZone guarantees that no secure world resources can be accessed by normal world components, so software attacks even compromising the OS in the normal world cannot access resources in the secure world.

As for the physical security, the TrustZone technology does not have any countermeasures against chip level attacks, but it can resist board level attacks if system designers follow the recommendations of ARM TrustZone white paper [3]: storing all the code and data in the secure world in OCM, which is not subject to board level attacks. However, this approach will place significant limits on the functionality of TEE systems because the OCM is quite limited.

2.2 On-chip Memory

The OCM is an important building block in SoC. It has the following two advantages. First, it exhibits better access performance because it connects to the CPU via fast internal connection buses (AXI). Second, OCM is more secure against physical intrusions: it does not expose any physical pins or wires, which could be potentially tapped by board level physical attackers. Although OCM has the above attractive properties, it is prohibitively expensive to support a large OCM in SoC, so the OCM in commodity SoCs is limited.

We perform a survey on the OCM size of some popular platforms that support TrustZone (Table 1). Our survey shows that OCM is a general building block in an SoC, and the OCM of most platforms is about a few hundred kilobytes.

3 Assumptions and Threat model

In this section we first describe the hardware assumptions and threat model, then proves the OCM is secure under board level physical attacks, and at last show that most real-world examples of board level physical attacks are covered by our threat model.

3.1 Hardware Assumptions

We assume the existence of a device-unique symmetric key in each device. The key is generated at manufacture time, and should be stored in the secure storage of SoC, such as processor's eFuses. This key is common on mobile SoCs, such as the Device-Unique Hardware Key in Samsung's KNOX.

Platform	CPU (ARM)	OCM	Platform	CPU (ARM)	OCM	Platform	CPU (ARM)	OCM
NXP QorIQ-LS1021A	Cortex A5	128 KB	NXP WaRP7	Cortex A7	256 KB	Zynq 7000 ZC702	Cortex A9	256 KB
NXP i.MX6Q-SDB	Cortex A9	272 KB	Hikey	Cortex A53	72 KB	Zynq UltraScale+	Cortex A53	256 KB
NXP i.MX6UL-EVK	Cortex A7	128 KB	TI DRA7xx	Cortex A15	512 KB	Atmel SAMA5D2	Cortex A5	128 KB
NXP i.MX7-SABRE	Cortex A7	256 KB	TI AM57xx	Cortex A15	512+[1]KB	ARM Juno	Cortex A72	128 KB

[1] The '+' symbol means that some SoCs' OCMs are bigger than the size listed in the table.

Table 1: A Survey on the Size of OCM

3.2 Threat Model

We assume that mobile devices are exposed to a hostile environment where board level physical attacks are feasible. The main assumption of the threat model is that only the SoC is resistant to board level physical attacks and thus is trusted, and all components outside of the SoC are vulnerable, including DRAM, address/data buses between CPU and DRAM, I/O devices, and so on. We also assume that the software running in the secure world is trusted because we do not aim to increase the security of TEE regarding software attacks.

Board level physical attacks involve DRAM tampering and CPU-DRAM bus probing, which allow observation of the DRAM contents and injection of arbitrary data/code into buses or directly into the DRAM. In particular, attackers can perform *passive attacks* and *active attacks*. In passive attacks, attackers can intercept traffic between CPU and DRAM by bus probing or directly obtain data in DRAM, breaking memory confidentiality. In active attacks, attackers can actively modify or inject data/code into the bus, breaking memory integrity. Based on how an attacker chooses the injected data, we define three classes of active attacks:

1. *Spoofing attacks*: the attacker exchanges a memory block transmitted on the bus with an arbitrary fake one.
2. *Splicing or relocation attacks*: the attacker swaps a memory block transmitted on the bus with another memory block in DRAM. Such an attack can be viewed as a spatial permutation of memory blocks.
3. *Replay attacks*: a memory block located at a given address is recorded and injected at the same address later. This class of attacks replaces a memory block's value with an older one.

Our model defines a high level of physical security, the same as Intel SGX's. However, most current approaches [8,17,20,21,38,39,43,48,54] cannot achieve this security level. The reason is that they only focus on resisting one kind of board level physical attacks, instead of all-sided and comprehensive protection. For example, attackers in most of their models do not have the ability of injecting malicious code into buses. Under the attacker defined in our threat model, their security can be broken by the following spoofing attack: the attacker exchanges a memory block of the kernel – the address space where the SoC-bound execution environment is included – with a memory block containing malicious code designed to steal security data in the secure environment.

3.3 The Physical Security of OCM

We leverage OCM as the working memory for TEE systems, and it is the only memory where TEE software resides in plaintext. So we need to show that OCM is secure against board level attacks. As OCM has no address or data lines at physical pins, board level attackers, which launch attacks through interfaces outside of the SoC, such as off-chip buses and DRAM interfaces, cannot directly attack OCM through the passive and active attacks defined in our threat model. However, there are two types of attacks that can indirectly access OCM by inserting malicious components in the device: cold boot attacks and DMA attacks. Cold boot attacks can access OCM by rebooting the device and launching a malicious memory-dumping kernel that can retrieve memory contents. However, Sentry [8] and paper [60] show that OCM is cleared by BootROM upon boot up. BootROM is burned into the hardware of the device at manufacturing and is immutable. Since BootROM is the first piece of code running on the device, software running after it can only obtain the cleared content. Besides, real-world cold boot attacks usually load the malicious kernel by a non-secure bootloader (e.g., u-boot) [40], which can only access non-secure resources, so the malicious kernel cannot access OCM, which has already been assigned to the secure world by the secure bootloader. DMA attacks can access OCM by sending a DMA request to the DMA controller, but ARM TrustZone is able to prevent DMA attacks by denying all DMA requests from malicious devices of the normal world. Therefore, OCM is secure under DMA attacks.

3.4 Real-world Board Level Physical Attacks

Here we show that our threat model covers the most popular board level physical attacks in the real-world: cold boot attacks, bus probing attacks and DMA attacks.

3.4.1 Cold Boot Attacks

Cold boot attacks exploit the data remanence effect [23] of DRAM to recover sensitive data stored in it. The remanence effect says that memory contents fade away gradually over time. The fading speed slows significantly at low temperatures, so the attacker can retrieve the remaining data by cold-booting the device and re-flashing malicious code which steals sensitive data.

Cold boot attacks are covered in our threat model: the attacker obtains all the data stored in DRAM by performing passive attacks, which allow the attacker to directly read data from DRAM.

3.4.2 Bus Probing Attacks

By attaching a bus probing tool [14, 16, 42] to the bus, attackers can intercept and modify data transmitted on the bus, and even can inject malicious code into the bus. To the best of our knowledge, all current software-based approaches, which only protect a piece of code of a monolithic kernel, are susceptible to bus probing attacks. Although Sentry [8] claims to resist this kind of attacks, actually it only protects applications and the kernel is not protected. Thus, the attacker can inject malicious code into the kernel space, gain kernel privileges, and further compromise the whole OS and applications.

Bus probing attacks are covered in our threat model for the following reason: intercepting and modifying data on the bus, and bus-injecting are exactly the abilities that the passive and active attackers are given in our model.

3.4.3 DMA Attacks

By connecting DMA-capable peripherals to devices, attackers can gain access to physical memory without exploiting vulnerabilities present in software. Many peripherals have been exploited to launch DMA attacks, such as network interface cards [10] and video cards [53].

TrustZone can deny all DMA requests from the normal world, so DMA attacks are unable to access any memory in the secure world.

4 Minimal Kernel Design

In this section, we enumerate the main design goals that the minimal kernel architecture should achieve. Then, we discuss in detail the techniques that we use to achieve our goals.

4.1 Design Goals

Minimal size. The size of the minimal kernel should be minimized for the following reasons. First, OCM is quite limited (Table 1), and most platforms' OCMs are less than 256 KB, so the minimal kernel should be as small as possible to fit into such a small memory. Second, the less memory occupied by the minimal kernel, the more OCM remained as the working memory for TEE software stored in DRAM, which improves the performance of TEE OS and TAs.

Memory protection on the whole kernel address space. Figure 1 has shown that it is not enough to only protect the SoC-bound execution environment: if the whole address space where the environment resides is not protected, attackers are able to launch board level physical attacks. The minimal kernel resides in the kernel space, so the whole TEE OS kernel space should be protected.

Memory protection on TAs. Usually the TEE OS kernel only provides fundamental functionality, such as memory management and TA session management, while TAs provide rich functionality for users, such as secure storage and fingerprint authentication. So TAs contain much sensitive information such as cryptographic keys and fingerprints, and these valuable data should be protected.

Memory protection in the kernel level. Performing memory protection in the kernel level makes protection transparent to TAs, so TA developers can get rid of entangling with designing security policies on how to protect their data and code, reducing programming burden on developers. Besides, previous TAs can be reused after the TEE OS is retrofitted by the minimal kernel architecture.

4.2 System Overview

4.2.1 Overall Architecture

The minimal kernel architecture splits the kernel space into two parts (Figure 2.a): a minimal kernel mapped to the OCM, and a main kernel initially stored in the DRAM. The minimal kernel, which maintains the execution of the whole TEE system on the OCM and cryptographically protects the code and data stored in DRAM, consists of a *minimal demand-paging system* and a *memory protection component* (Figure 2.b).

The minimal demand-paging system sets the OCM as the only working memory for the TEE system, and sets the DRAM as a backing store for the TEE system. It is composed of a *core of minimal kernel* and an *OCM-DRAM Channel*. Unlike ordinary demand-paging systems, which are components of the kernel and included in the kernel space, the minimal demand-paging system is separated from the main kernel and cannot use any functions of the main kernel, so it should be self-contained and contain all components required by the CPU to maintain execution of the TEE system. Another crucial design requirement for the minimal demand-paging system is that it should contain minimum amounts of components to fit into the size-limited OCM. All the required components comprise the *core of minimal kernel*.

When the CPU accesses code/data of the main kernel or TAs, the *OCM-DRAM channel* component of the minimal demand-paging system automatically intercepts the access and loads the demanded page into the OCM. The reason for explicitly building a software-based channel between OCM and DRAM is as follows: memory protection (encryption/decryption and integrity computation) should be triggered when data is transferred in or out of the trust boundary (i.e., the SoC), but unlike hardware-based memory protection approaches [22, 49], which provide memory protection by interposing a hardware protection engine on the boundary

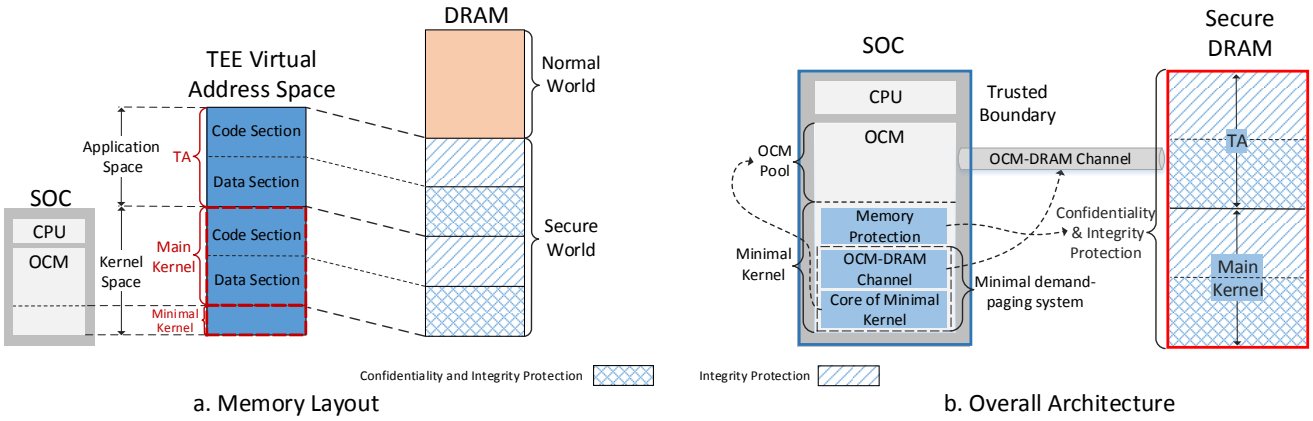

Figure 2: The Memory Layout and Overall Architecture of Minimal Kernel

between cache and DRAM, i.e., the cache-DRAM transmission channel, software-based memory protection approaches cannot leverage the cache-DRAM channel because they are unable to modify hardware, so they have to build a channel between OCM and DRAM in software, based on which memory protection can be performed.

The memory protection component is responsible for protecting the part of the TEE system stored in the backing store. Specifically, it guarantees the integrity of the code to prevent attackers from tampering the software, and it guarantees both the confidentiality and integrity of the data to prevent attackers from obtaining and modifying sensitive information.

4.2.2 Workflow

The workflow of the minimal kernel is as follows (Figure 3).

1. When the CPU accesses data/code of the main kernel or TAs, if the data/code is not in the OCM, the *minimal demand-paging system* automatically intercepts this access. The details of how to intercept the access automatically are described in Section 4.3.2.
2. The *minimal demand-paging system*, which keeps a pool of free OCM frames, allocates an OCM frame from the pool, loads the DRAM page storing required data/code into the OCM frame. During loading, if the virtual address of the required data/code belongs to a code section, the *memory protection component* performs integrity check of the DRAM page; if the virtual address belongs to a data section, the *memory protection component* decrypts the loaded page and performs integrity check.
3. After loading the DRAM page into OCM, the *minimal demand-paging system* changes the page table entry (PTE) which maps the virtual address of the required data/code page, and re-maps the page to the allocated OCM frame. At this time, the page is loaded into OCM thoroughly.
4. The CPU re-accesses the data/code from OCM.
5. When the OCM pool is empty, the *minimal demand-paging system* picks one page from OCM, frees it, and adds it to the pool. If the selected page is a code page, it is freed directly; if it is a data page, before freeing it, the *memory protection component* encrypts the page, stores the encrypted page to DRAM, computes and stores the hash value of the page in a reserved integrity area of the OCM.

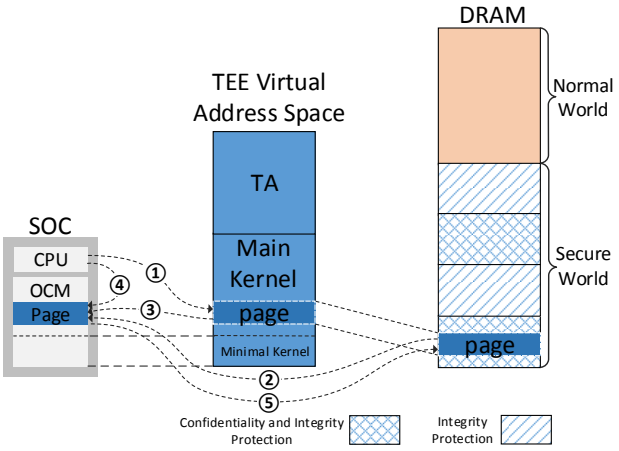

Figure 3: The Workflow of Minimal Kernel

4.3 The Minimal Demand-paging System

The minimal demand-paging system consists of the core of minimal kernel and the OCM-DRAM channel.

4.3.1 The Core of Minimal Kernel

As described in Section 4.2.1, the core of minimal kernel should only contain the very essential components required by the CPU to maintain execution of the TEE system. The difficulty of designing such a kernel lies in how to distinguish all the required components from unnecessary ones. Inspired by the minimality principle for the microkernel architecture

[34], we propose a principle for designing the core of minimal kernel: *one kernel component is tolerated inside the core of minimal kernel only if moving it out would prevent the CPU from running software normally.*

For a component not in the working memory and therefore not accessible to the CPU, if the CPU can access the component later by leveraging the demand-paging mechanism and the delayed-access does not cause software crash, according to the above principle, this component should not be included in the core of minimal kernel. Thus, the core of minimal kernel should only contain the components that must be directly accessed by the CPU to run software. From the above observation, we give the definition of essential components:

Definition 1 (Essential Components) Essential components are components that the CPU has to access directly when running the kernel, and if the CPU fails to directly access these components, the kernel would crash immediately.

From Definition 1, we identify all the essential components as follows, which comprise the core of minimal kernel.

- Page tables. Programs use virtual addresses, and the CPU obtains physical addresses by a virtual-physical address translation. The mappings between virtual addresses and physical addresses are stored in page tables, and the MMU (Memory Management Unit) uses page tables to perform address translations. If page tables are not directly accessible, the CPU will be unable to access any physical memory. So the page tables of the TEE OS are essential components.
- The exception vector table. When an exception occurs, the CPU immediately jumps to the exception handler entry defined in the exception vector table. If the CPU cannot access the exception vector table directly, the kernel will not address any exceptions. So the exception vector table for the TEE OS is an essential component.
- Low level exception handlers. Upon entering into the exception handler, the CPU has to save the context of the non-banked registers immediately. Otherwise, the CPU cannot return back to the point where the interrupt occurs. So the low level exception handlers which save contexts are essential components.
- The kernel stack. One usage of the stack is to store calling conventions. The ARM procedure call standard [11] defines that subroutines should use the stack to receive parameters and return results when there are insufficient argument registers available. As subroutine/function calls are trivial in the kernel, the stack is an essential component.

4.3.2 The OCM-DRAM Channel

The OCM-DRAM channel needs to intercept the data transmission between the CPU and DRAM using software, so it requires a software-controllable mechanism to interrupt the CPU's access to the DRAM data. Based on the mechanism, the OCM-DRAM channel can be implemented by the following steps: first, interrupt the access to DRAM; second, load the required data into the OCM in the interrupt handler. Fortunately, we find two software-controllable interrupt mechanisms based on the virtual memory technology. When the CPU accesses the DRAM, the MMU first translates the virtual address of the data into a physical address by a hardware page table walk. The translation procedure can be interrupted by configuring any one of the following two fields in the PTE:

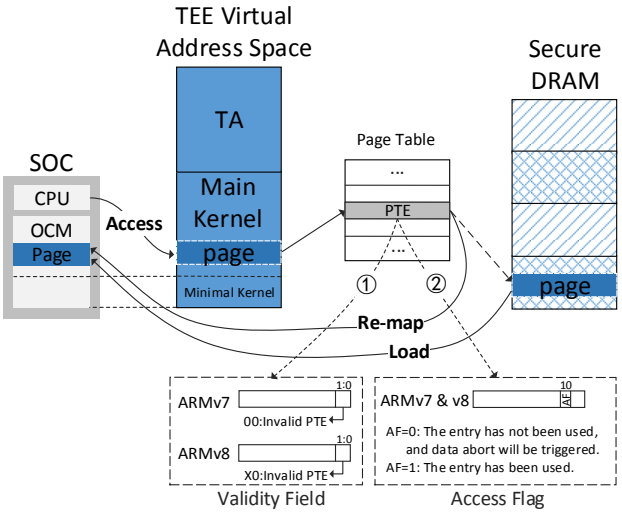

Figure 4: The OCM-DRAM Channel

① Validity field. This field indicates whether the PTE is valid. If the field is set to invalid, when CPU accesses the virtual address mapped by this PTE, the access will cause the CPU to generates a translation fault which will be handled by the data abort exception handler, so the access is interrupted. We describe how to set the field invalid for ARMv7 and ARMv8 architectures in Figure 4.

② Access flag (AF). This flag indicates whether the PTE is used for the first time. If this flag is cleared, when the corresponding page is accessed, the CPU will generate an access flag fault and transfer control to the data abort exception handler. In the exception handler, the AF is set, and then the CPU accesses the page normally. Therefore, this flag can be used to interrupt the access to DRAM. One disadvantage of the access flag mechanism is that ARM specifications allow the access flag being managed by hardware [26, 27], in which case the translation procedure cannot be interrupted by software. The details of AF are depicted in Figure 4.

Based on the above interrupt mechanisms, we design the OCM-DRAM channel as follows (Figure 4):

1. When the CPU accesses some data stored in DRAM, the channel interrupts this access by invalidating the validity field or clearing the AF of the PTE mapping the virtual address of the accessed data, and then the CPU jumps to the data abort exception handler.
2. In the data abort exception handler, the channel copies the accessed page into OCM, re-maps the accessed virtual

address to the page in OCM, and makes the page accessible to the CPU by validating the validity field or setting the AF of the PTE.

3. When the data abort exception handler completes, the CPU re-accesses the data from the OCM.

4.4 Memory Protection

The memory protection component is triggered when code/-data in the main kernel or TAs is transferred into OCM or stored back into DRAM through the OCM-DRAM channel, and it provides memory protection on the main kernel and TAs (Figure 5): for code sections, it protects their integrity to prevent memory tampering; for data sections, it protects their confidentiality and integrity to prevent information leakage and data tampering.

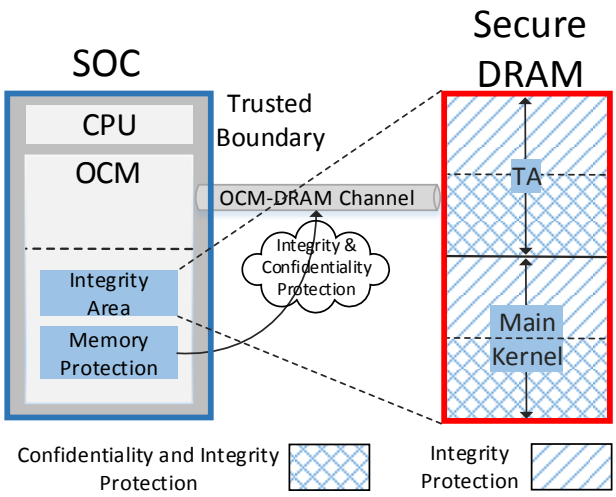

Figure 5: The Memory Protection Component

The memory protection component needs to store the legal integrity values of the main kernel and TAs to perform integrity verification. A direct method is to store all the integrity values of the DRAM pages in the OCM. This method gains high performance because it only needs to compute the integrity value of the loaded page for each integrity verification. Its disadvantage is the large storage for the integrity values. Take a 1M size TA for example, if SHA-256 is used to compute the integrity value, each page requires 32B OCM, and the TA requires 8KB OCM to store its integrity values. If multiple TAs are supported, more OCM will be occupied. The method of Merkle trees [18] can address this disadvantage because it only needs to store the root node of the Merkle tree, but it introduces high performance overhead: it needs to re-compute all the nodes in the path from the leaf to the root of the tree for each integrity verification.

Based on the above two integrity verification methods, we design two memory protection schemes: a trivial memory protection scheme which stores all integrity values in the OCM, and a Merkle tree based memory protection scheme which leverages the Merkle tree technique to reduce the requirement for OCM and makes it possible to protect large TAs. The two schemes can be used to trade-off between the performance of the minimal kernel and its requirement for OCM.

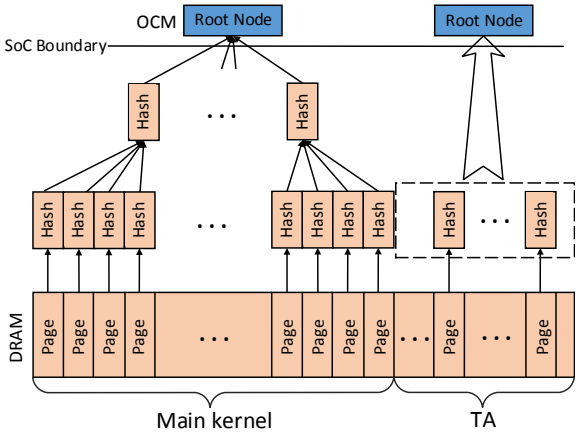

Figure 6: The Merkle Tree Based Integrity Protection

4.4.1 Trivial Memory Protection

The memory protection component allocates an area in the OCM to store integrity values, and uses an encryption key k to protect the confidentiality of the pages in DRAM. The encryption key is derived from the device-unique device key (denoted by k_d): $k = HKDF(k_d, \text{“}memory\ encryption\text{”}, klen)$ where $HKDF$ is an $HMAC$-based key derivation function whose output length is $klen$.

The memory protection component works as follows. When an OCM page is going to be transferred to DRAM by the OCM-DRAM channel, if the page belongs to data sections, the memory protection component encrypts the page using k, computes its integrity tag and stores the integrity tag in the integrity area, and if the page belongs to code sections, the memory protection component does nothing. When a page is going to be loaded into the OCM, if the page belongs to data sections, the memory protection component decrypts the page using k, computes its integrity tag, and compares it with the legal integrity value, and if the page belongs to code sections, the memory protection component only computes and checks its integrity tag.

4.4.2 Merkle Tree Based Memory Protection

The Merkle tree based memory protection scheme works similar to the trivial memory protection scheme except for the integrity protection procedure (Figure 6). We list the differences between them as follows.

1. The Merkle tree based scheme generates a Merkle tree for the main kernel and one tree for each TA. The integrity tag

of a page is a leaf node. Only root nodes are stored in the OCM, and other nodes are stored in the DRAM.

2. When a data page is going to be transferred to DRAM, the Merkel tree based scheme needs to update the corresponding tree: all the nodes in the path from the leaf to the root node (including the leaf and root nodes) are updated.
3. When a page is going to be loaded into the OCM, the Merkel tree based scheme verifies its integrity as follows. First, compute the integrity tag of the page. Then compute all the nodes in the path from the leaf to the root. Finally, verify the integrity of the page by comparing the computed root node with the root node stored in the OCM.

5 mTEE: A Minimal Kernel Prototype

We present a concrete implementation of the minimal kernel architecture, named mTEE, on a popular open source TEE OS: OP-TEE. In this section, we first introduce the architecture of OP-TEE OS, and then give our implementation of mTEE.

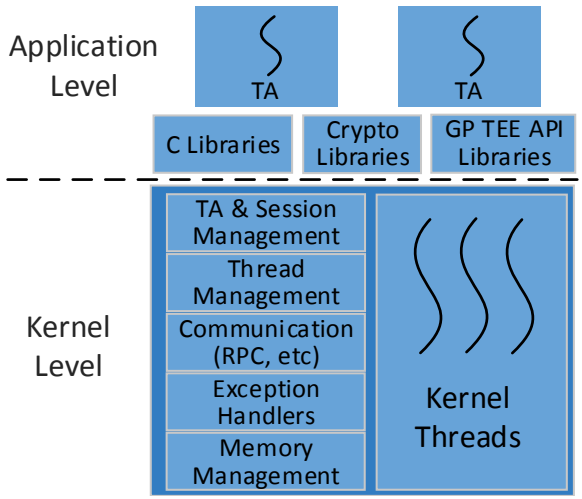

Figure 7: The Overall Architecture of OP-TEE

5.1 Architecture of the OP-TEE OS

The OP-TEE project publishes some documents introducing the OS components briefly but lacks the description of the whole kernel architecture, implementation details of kernel components, and interactions between the kernel components, which are needed to implement the minimal kernel architecture. So we go deep into the details of the OP-TEE source code to obtain the above information. However, the OP-TEE kernel is composed of more than 46,000 lines of C code and 7,000 lines of assembly code, which makes our task difficult.

The overall architecture of OP-TEE is illustrated in Figure 7. When the normal world sends a TA request, the thread management component captures the request and allocates a thread from the thread pool. The new thread first runs TA & session management routines to open the demanded TA, and then initializes a session for this request, and finally executes the TA. OP-TEE supports two kinds of TAs: static TAs, which are statically linked in the OP-TEE kernel image and run in kernel threads, and user TAs, which are loaded from normal world and run in user threads. During execution, the TA can communicate with the normal world through the communication component which supports RPC (Remote Procedure Call) and shared memory mechanisms. The exception handlers deal with software and hardware exceptions. The memory management component manages memory pools, and allows dynamical memory allocation for threads. OP-TEE also provides a collection of libraries for user TAs, such as C libraries and cryptography libraries.

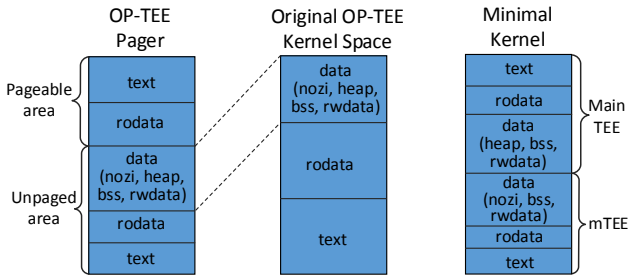

Figure 8: The Memory Layout of OP-TEE, Pager, and Minimal Kernel

Figure 8 (middle) depicts the memory layout of the original OP-TEE kernel, which consists of text, rodata and data sections. The data section is composed of the rwdata, bss, heap, and nozi sections, and the nozi section consists of page tables and stacks. Figure 8 (left) illustrates the memory layout of the OP-TEE's memory protection solution (Pager): it builds an unpaged area residing in the OCM, which plays a role similar to the minimal kernel, and the remaining code and data form the pageable area, which resides in DRAM. As Pager lacks the design principle described in Section 4.3.1, it contains many unnecessary components. For example, Pager puts all the rwdata, heap and bss sections into OCM, which makes it very large and unsuitable for devices with small OCM.

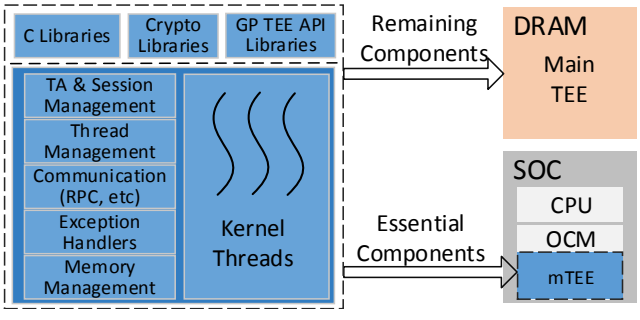

Figure 9: The Implementation of mTEE

Our implementation (Figure 9) divides OP-TEE into two parts: *mTEE* and *main TEE*. The mTEE is built by following

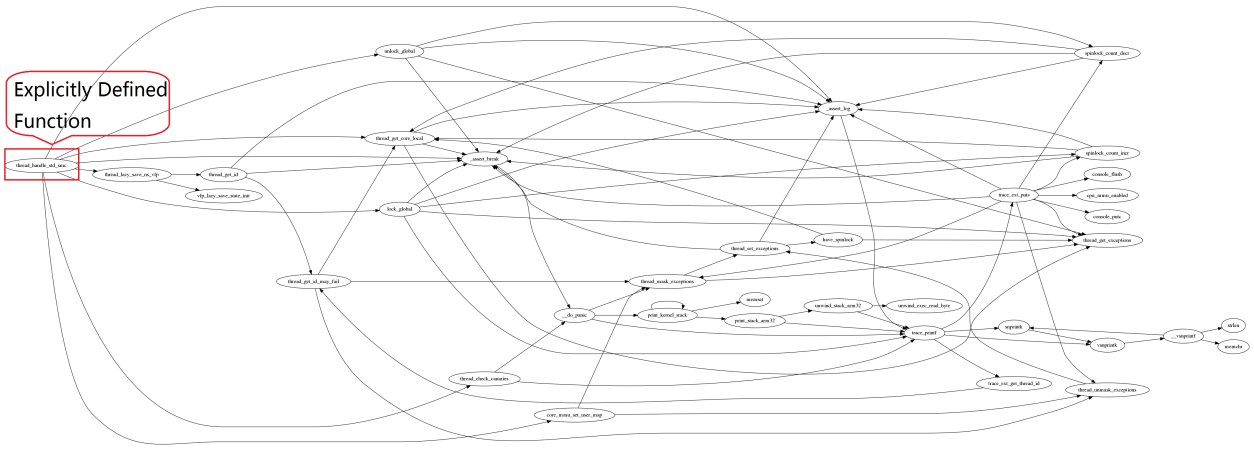

Figure 10: The Call Dependency Graph of *thread_handle_std_smc* in OP-TEE

our principle in Section 4.3.1. It consists of the very essential components listed in Section 4.3.1, the OCM-DRAM channel and the memory protection component. As mTEE is self-contained, the functions and data on which these components are dependent should be included in mTEE. Figure 11 shows the call dependency graph of a function explicitly defined in these components, and all the functions and data that are called by the function should be linked in the mTEE image. We use some tricks of GCC and GNU linker (ld) to deal with this dependency problem. First, in the compilation phase, we use GCC *-ffunction-sections* and *-fdata-sections* options to generate a separate section for each function and data item in the source file. Then, in the linking phase, we first use GNU ld *--undefined* option to force the function and data sections that are explicitly defined in the essential components, OCM-DRAM channel and memory protection component, into the mTEE image; and then we use the GNU ld *--gc-sections* option to link all the code and data sections referenced by the sections identified by the *--undefined* option and eliminate unused code and data sections from the mTEE image. After the above phases, the output image is self-contained and only includes the necessary code and data. To prove the effectiveness of our minimal kernel design, we compare the numbers of sections that are included in the mTEE image with the Pager image. The results show that the mTEE image includes 329 sections, while the Pager image includes 511 sections, i.e., our implementation eliminates 36% sections of Pager. The call dependency graph of the *thread_handle_std_smc* function in OP-TEE (Figure 10) shows that the function depends on other 35 functions or data items, while the call dependency graph of the same function in mTEE (Figure 11) shows that the function depends on 23 functions or data items. The comparison shows that for the single function *thread_handle_std_smc*, our design eliminates about 34% unnecessary dependency functions or data items.

All sections of mTEE are linked in the following sections: *text, rodata, bss, nozi* and *rwdata*. In the linker script, we create separate *text, rodata*, and *data* sections for mTEE, and the remaining code and data of OP-TEE are linked in sections of the main TEE (Figure 8).

In the memory protection component, we use the authenticated encryption algorithm AES-GCM to protect the confidentiality and integrity of data sections, and use the SHA-256 algorithm to protect the integrity of code sections. Each Merkle tree is implemented as a 4-ary tree, and each leaf node is the hash value of a 4K DRAM page.

6 Evaluation

We first evaluate the size of mTEE, which concerns whether mTEE can run on the size-limited OCM, and then we evaluate mTEE's performance overheads on cryptographic computations and TAs. We evaluate five systems for TA performance tests: Original OP-TEE, OP-TEE Pager system, mTEE using trivial memory protection (denoted by mTEE), mTEE using Merkle tree based memory protection (denoted by mTEE-MT), and mTEE without memory protection (denoted by mTEE-plain), and use the experimental results to infer the factors that lead to performance overhead. The mTEE-plain system is used to evaluate the performance overhead introduced by the memory protection. Experiments are conducted on the NXP i.MX6Q Sabre-SD board, which has an i.MX 6Quad SoC with 4 ARM Cortex-A9 1.2 GHz CPUs and 256 KB OCM, and 1 GB DRAM. The TEE OS is OP-TEE v2.4.0, and the normal world runs Linux.

6.1 Size of mTEE

Table 2 lists the size of each section of original OP-TEE, Pager, and mTEE. The evaluation results show that mTEE only requires a runtime footprint of 100 KB OCM to maintain the execution of the whole TEE OS, whose image is more

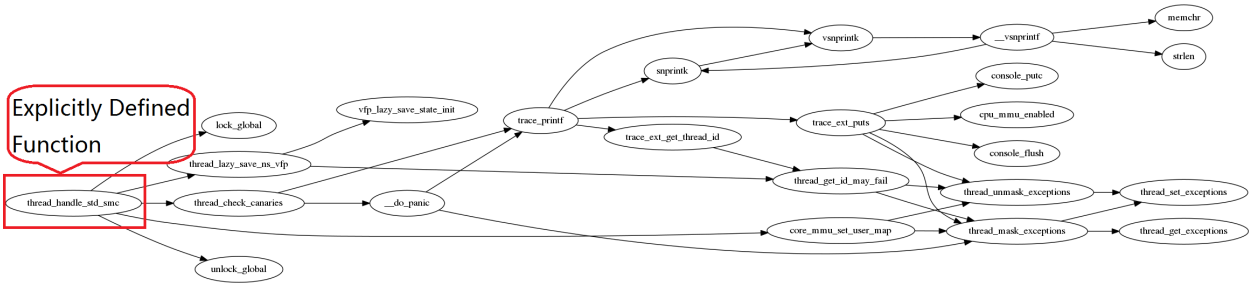

Figure 11: Call Dependency Graph of an Explicitly Defined Function

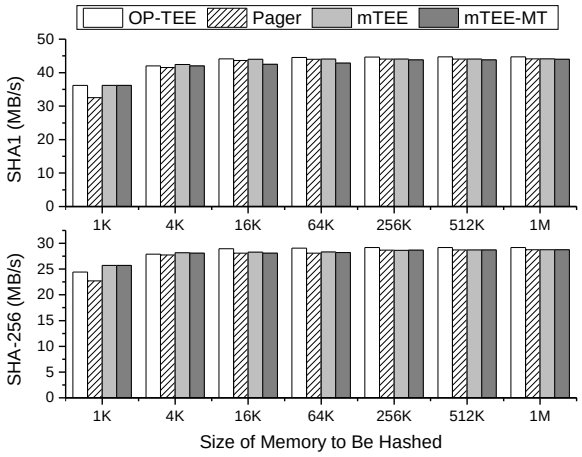

Figure 12: Throughput Comparison of SHA

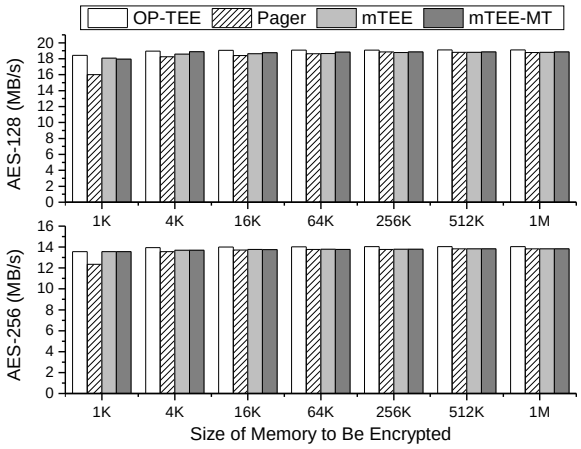

Figure 13: Throughput Comparison of AES

than 400 KB. The evaluation results also show that mTEE only needs about half (56%) of OCM required by Pager.

	text	rodata	rwdata	bss	heap	nozi	others	total
OP-TEE	147	81	8	60	76	56	12	440
Pager	35	19	8	7	65	36	10	180
mTEE	26	16	0	12	0	36	10	100

Table 2: Sizes of OP-TEE, Pager and mTEE (KB)

6.2 Crypto Evaluation

To evaluate the performance overhead of mTEE on cryptographic computations, we measure the performance of SHA1, SHA-256, AES-128, AES-256, and RSA on four systems: original OP-TEE, OP-TEE Pager, mTEE, and mTEE-MT. Figures 12, 13, and 14 depict the evaluation results.

The four systems achieve similar performance on AES and SHA, except that when the memory to be hashed or encrypted is small (less than 1KB), Pager is slightly slower than the other three systems. This is because Pager has smaller working memory, which makes the overhead of demand-paging be non-negligible compared to the cryptographic computations when the hashed/encrypted memory is small.

For RSA operations, mTEE and mTEE-MT perform much better than Pager. Especially in RSA verification, mTEE and mTEE-MT is 7 times faster than Pager. This is due to that RSA verification requires less cryptographic computations than signing, so the overhead of demand-paging takes up a large proportion of the whole overhead.

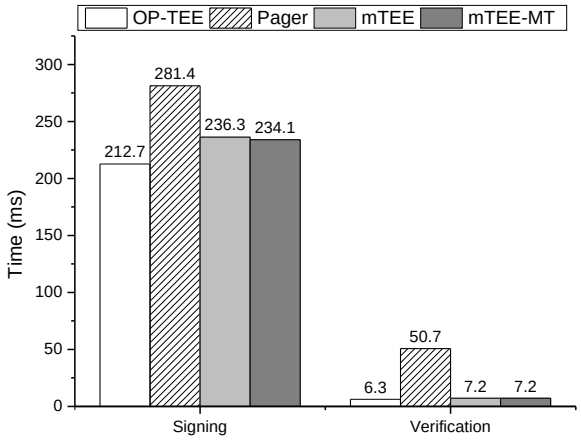

Figure 14: Performance Comparison of RSA

6.3 TA Evaluation

To evaluate the impact that mTEE has on TAs, we run the following three TAs that perform security-related computations on the five systems.

- *Random TA:* generate random numbers for applications in the normal world.
- *Data Protection TA:* use AES to encrypt the provided data, and return the ciphertext to the normal world.
- *One Time Password (OTP) TA:* receive a shared key from the normal world, and compute HMAC-based OTPs.

We measure both the entire execution time of TAs and the execution time of TA services (Figure 15). The entire execution time includes the time of loading a TA into the memory, mapping it to the TA address space, creating a session for the TA, running the TA service in the session, and destroying the session. The execution time of TA services only includes the time of running the TA service. Compared to OP-TEE, the performance of running the whole TA on Pager is 7.9 $\sim$ 21.4 times slower, mTEE is 4.0 $\sim$ 4.4 times slower, and mTEE-MT is 4.4 $\sim$ 5.0 times slower; the performance of TA service on Pager is 20 $\sim$ 200 times slower, mTEE is 3.3 $\sim$ 7.5 times slower, and mTEE-MT is 3.5 $\sim$ 7.7 times slower.

The evaluation results show that both mTEE and mTEE-MT achieve better performance than Pager. In the aspect of running whole TAs, mTEE is 1.8 $\sim$ 5.3 times faster than Pager, and mTEE-MT is 1.6 $\sim$ 4.7 times faster. In the aspect of TA service runtime, mTEE is 6.0 $\sim$ 60 times faster than Pager, and mTEE-MT is 5.7 $\sim$ 54 times faster. This is because Pager only offers 76KB (256KB - 180KB = 76KB) OCM to run TAs, while mTEE and mTEE-MT offer more OCM, about 156KB (256KB - 100KB = 156KB).

We also observe that, for TA services, the performance advantage of mTEE and mTEE-MT over Pager is very significant: mTEE can reach up to 60 times faster than Pager. This is because that Pager's free OCM (76KB) is smaller than the size of a TA (about 100KB) and therefore the execution of a TA service requires a lot of page-swappings, whose overhead is much more than the execution time of the pure TA service.

Another interesting observation beyond our expectation is that mTEE-MT gains almost the same performance with mTEE. This is because the major performance overhead of the memory protection mechanism is introduced by cryptographic computations (decryption, encryption or hash) on the 4KB pages swapped between OCM and DRAM, and mTEE-MT only adds a few hash computations on tree nodes (only dozens of bytes) for each page swapping, which are negligible compared to the cryptographic computation on the 4KB page.

From the above observations, we come to the following conclusions: for a software-based memory protection scheme, providing a large working memory for the protected software is critical to improve its performance, and the Merkle tree is a good choice for integrity protection because it reduces the required physical secure memory while only introducing a slight increase on performance overhead.

6.4 Evaluation of the Impact of TA Size on Performance

Since the available OCM is quite limited, the performance of mTEE is sensitive to the size of TAs, especially when the working set size is larger than the available OCM because in this case paging is required. To evaluate the impact of TA size on the performance of mTEE, we leverage a random code generator tool Csmith [55] to generate a series of test TAs whose sizes range from 100KB to 1MB. We measure the entire execution time and service runtime of these test TAs on the five systems. Figures 16 and 17 show the results for the Pager, mTEE, mTEE-MT, and mTEE-plain systems relative to the OP-TEE system.

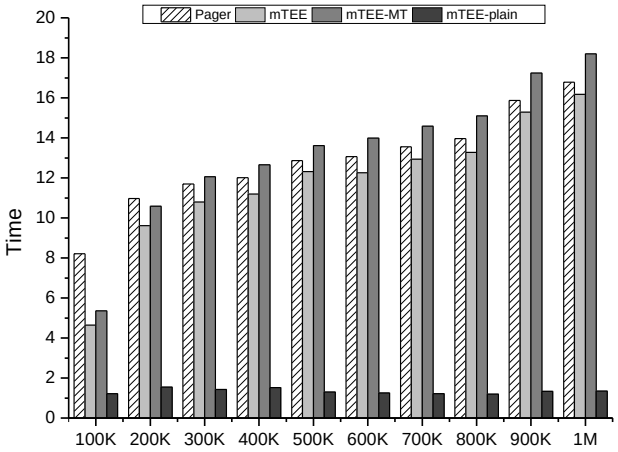

Figure 16: Performance of Test TAs

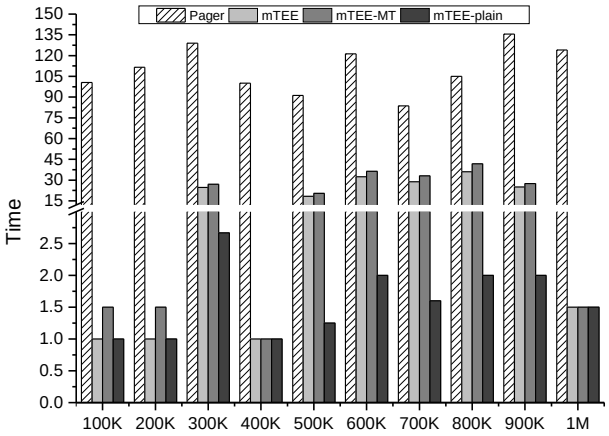

Figure 17: Performance of Test TA Services

Figure 16 shows that, in the aspect of executing whole TAs, Pager, mTEE, and mTEE-MT are about 10X times slower relative to the baseline OP-TEE, and as the size of TA increases, the performance overhead of these three systems

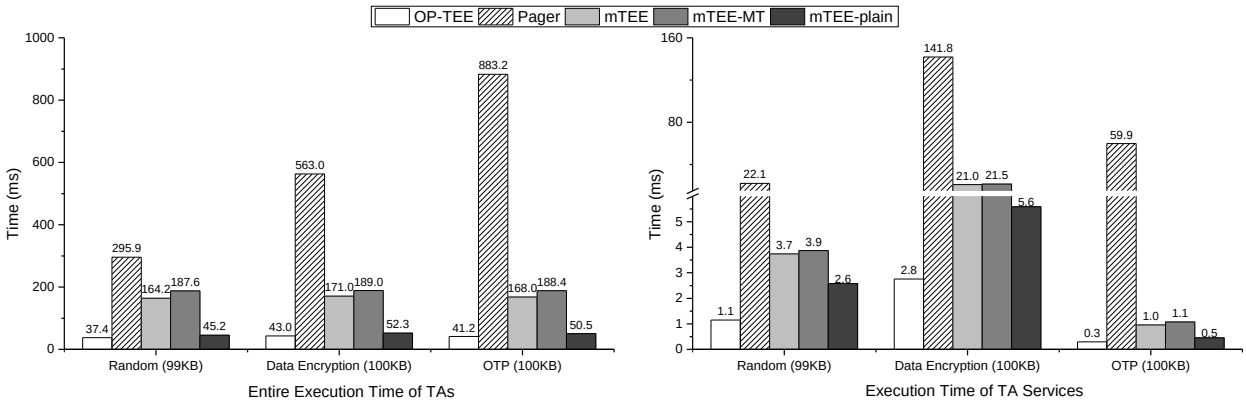

Figure 15: Performance Comparison of TA

grows. We also observe that, in most cases, mTEE performs better than Pager and mTEE-MT; the reason is that mTEE has more available OCM than Pager and needs less cryptographic computations than mTEE-MT.

Figure 17 shows that, in the aspect of TA services, Pager performs worst, and its performance is about 100X times slower than OP-TEE; we infer that this is because Pager itself occupies too much OCM, and the left OCM is far less than the working set size of all test TAs. For mTEE and mTEE-MT, they achieve almost the same performance in all test cases; in the 100KB, 200KB, 400KB and 1MB test cases, they achieve similar performance to OP-TEE; this is because Csmith programs often contain some dead code, so the active code of these four test cases can fit into the free OCM left by mTEE and mTEE-MT; in the rest six test cases, mTEE and mTEE-MT are, at most, 42 times slower than OP-TEE, and are 4 times faster than Pager on average.

From the above observations, we get the same conclusions as Section 6.3: first, mTEE and mTEE-MT achieve much higher performance than Pager, especially for TA services, because they reserve much more working memory; second, for software-based memory protection schemes, the Merkle tree is a good choice for integrity protection because it saves much secure memory while introducing slight overhead.

6.5 Evaluation of the Overhead of Paging and Cryptographic Computations

The performance overhead comes from two aspects: the memory demand-paging between OCM and DRAM, and the cryptographic computations (encryption, decryption, and integrity check) during the memory swapping. Here we measure the overhead of each aspect.

Figures 15, 16 and 17 compare the performance of mTEE-plain (which only enables demand-paging) with the other four systems. The evaluation results show that the overhead introduced by the demand-paging mechanism takes up a small portion of the whole overhead. Take the Random, Data Protection, and OTP TAs for example, mTEE-plain introduces 61% overhead on average, while mTEE introduces about 344% overhead on average. So we conclude that most performance overhead comes from the cryptographic computations during memory protection.

7 Discussion

Just like other software-based memory protection solutions, the most challenge for deploying the minimal kernel architecture is its big performance overhead. Leveraging customized cryptographic hardware accelerators is a good way to reduce the performance overhead. For example, Intel SGX integrates a specialized memory encryption engine [22] in the CPU, which only imposes 5.5% performance degradation on average. Besides the performance overhead, the size of the minimal kernel is another key feature that concerns the application of the minimal kernel architecture: the minimal kernel should be small enough to fit into the OCM. From the experience we have learned from the design of the minimal kernel architecture and implementation of the mTEE prototype, we discuss and give suggestions on how CPU designers can revise or extend their designs to provide more efficient memory protection solutions.

Performance Improvement. The evaluation results in Section 6.5 show that the overhead mainly comes from cryptographic computations. One direct solution to reduce the overhead is leveraging hardware cryptographic accelerators, which have been integrated into most mobile SoCs. For example, the evaluation board used in this paper, NXP i.MX6Q, is integrated with a cryptographic accelerator, namely Cryptographic Acceleration and Assurance Module (CAAM), which provides basic cryptographic primitives, such as hash algorithms and symmetric block ciphers. CAAM is implemented as a separate co-processor, and data is transferred to it through DMA. Paper [6] evaluates the performance of CAAM, but the results show that CAAM is only efficient when data sent to the accelerator is larger than 100 KB each time, and if the

input length is smaller than 100 KB, CAAM is slower than software implementations of cryptographic algorithms. The reason is that the data communication overhead with DMA disadvantages the acceleration of CAAM. So we recommend that CPU designers build a cryptographic accelerator into the primary CPU and provide interfaces to software in the form of ISA extensions (such as AES-NI). One advantage of this approach is that, unlike SGX's MEE, which is dedicated to memory protection, it can serve to all software.

Reducing the Minimal Kernel. The ARM CPU architecture uses two sets of translation tables: TTBR0 and TTBR1. The N field of the register TTBCR determines the virtual address range translated by each set of translation tables. Usually, TTBR0 translation tables are used to map user space, and TTBR1 translation tables are used to map kernel space. No matter what TTBCR.N is set, the first level translation table of TTBR1 must contain a fixed number of PTEs to map the whole address space. Take the ARMv7 CPU architecture for example, the first level translation table of TTBR1 contains 4096 PTEs and takes up 16 KB. The minimal kernel architecture requires that the page tables mapping the kernel space should be contained in the OCM. However, as the kernel space of TEE OS is small, most of the 16 KB memory is wasted. Take OP-TEE for example, the kernel space is a few megabytes. Including the aliased mapping of the secure DRAM, which is about dozens of megabytes, only dozens of the 4096 PTEs of the first level page table are used, which means that no more than 1 KB of the 16 KB is utilized. We recommend that the CPU support small page tables like Sanctum [9], or SGX's Enclave Page Cache Map (EPCM) mechanism which stores the virtual-physical address mapping in a specialized data structure, and then the address mapping for the protected address space can be stored using smaller memory. As a result, the minimal kernel's size can be reduced.

8 Related Work

To protect computer systems from physical attacks without the support of specialized hardware memory encryption engines, some research works using software-based memory encryption mechansim to protect the whole address space have been proposed.

Memory protection for applications. Cryptkeeper [44] presents a software-encrypted virtual memory manager, which divides DRAM into a plaintext working set and a encrypted segment, and swaps pages between the two segments on demand. Cryptkeeper only mitigates but cannot fully prevent physical attacks because the working set is always in plaintext. Another weakness of Cryptkeeper is that the whole kernel is exposed to physical attacks because the kernel resides in DRAM permanently. Paper [43] implements main memory encryption for applications by static/dynamic instrumentation of load and store instructions with encryption and decryption instructions. However, under physical attacks, just protecting applications is not enough: attackers can attack and control the kernel, which is more severe. TrustShadow [19] and CryptMe [7] protect applications in the normal world from physical attacks using a lightweight runtime system, whose main task is to maintain execution environments and provide memory protection for applications. Ginseng [56] protects small sensitive data of applications by static protection extended to the compiler and runtime protection in the secure world, which are used to keep sensitive data in CPU's registers when they are used and encrypting them when switching context.

Memory protection for whole computer systems. Bear OS [25] is a microkernel which requires only about 35 KB secure memory and can be entirely loaded into the OCM. It encrypts all code and data outside of the chip boundary at section granularity. SoftME [57] protects tasks on embedded platforms by locating the embedded OS in the OCM and extending a task scheduler and a memory protection engine in the embedded OS. Both Bear OS and SoftME only work for small embedded OSes, and they are not applicable to mature OSes whose sizes are bigger than OCM. OP-TEE Pager is the most closely related system to our work. It implements a demand-paging system, which runs in the OCM and protects all code and data stored in DRAM. However, Pager is implemented without a design principle that determines which components are necessary and which components are unnecessary, causing it to include some unnecessary components, so the size of Pager exceeds OCM of many platforms. Komodo [15] presents an approach to implementing a formally verified enclave architecture in software, which achieves the security equivalent to Intel SGX. However, Komodo assumes a memory encryption engine to prevent physical attacks, which does not exist in ARM CPUs. So our work can be seen as the solution to the assumption.

9 Conclusion

In this paper, we present the minimal kernel architecture, which protects the whole TEE OS kernel and TAs against board level physical attacks. As the whole kernel address is protected, the minimal kernel achieves the same security level as SGX. Our principle of designing the minimal kernel makes it feasible to build a small kernel running inside the chip in the minimum amount of software, and our prototype fits into most commodity SoCs' OCM. By reserving OCM for TA's working memory as much as possible, mTEE achieves better performance than OP-TEE's original memory protection solution Pager: mTEE is several times faster, and for TA services, mTEE can reach up to 60 times faster. We give suggestions on how to modify or extend CPU hardware to improve the performance of software-based memory protection schemes. We expect that the minimal kernel architecture can help system designers to revise their CPUs in an economical way and propose efficient SGX-like solutions by hardware-software co-designs.

Acknowledgement

This research was supported by the National Key R & D Program of China (2018YFB0904900, 2018YFB0904903) and the National Natural Science Foundation of China (61802375, 61872343, 61602455, 61602325).

References

[1] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, volume 13. ACM New York, NY, USA, 2013.

[2] Apple. iOS Security. https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf, 2018.

[3] ARM. Security Technology - Building a Secure System using Trustzone Technology. *ARM Technical White Paper*, 2009.

[4] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 90–102. ACM, 2014.

[5] Rick Boivie and Peter Williams. SecureBlue++: CPU support for Secure Execution. *Technical report*, 2012.

[6] Aymen Boudguiga, Witold Klaudel, and Jimmy Durand Wesolowski. On the Preformance of Freescale i.MX6 Cryptographic Acceleration and Assurance Module. In *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, page 8. ACM, 2015.

[7] Chen Cao, Le Guan, Ning Zhang, Neng Gao, Jingqiang Lin, Bo Luo, Peng Liu, Ji Xiang, and Wenjing Lou. CryptMe: Data Leakage Prevention for Unmodified Programs on ARM Devices. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 380–400. Springer, 2018.

[8] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal De Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Protecting Data on Smartphones and Tablets from Memory Attacks. *ACM SIGARCH Computer Architecture News*, 43(1):177–189, 2015.

[9] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security Symposium*, pages 857–874, 2016.

[10] Loïc Duflot, Yves-Alexis Perez, Guillaume Valadon, and Olivier Levillain. Can You Still Trust Your Network Card. *CanSecWest/core10*, pages 24–26, 2010.

[11] Richard Earnshaw. Procedure Call Standard for the ARM Architecture. *ARM Limited, October*, 2003.

[12] Reouven Elbaz, David Champagne, Catherine Gebotys, Ruby B Lee, Nachiketh Potlapally, and Lionel Torres. Hardware mechanisms for memory authentication: A survey of existing techniques and engines. In *Transactions on Computational Science IV*, pages 1–22. Springer, 2009.

[13] Reouven Elbaz, Lionel Torres, Gilles Sassatelli, Pierre Guillemin, Michel Bardouillet, and Albert Martinez. A Parallelized Way to Provide Data Encryption and Integrity Checking on a Processor-Memory Bus. In *Proceedings of the 43rd annual Design Automation Conference*, pages 506–509. ACM, 2006.

[14] EPN Solutions. Analysis Tools for DDR1, DDR2, DDR3, Embedded DDR and Fully Buffered DIMM Modules. http://www.epnsolutions.net/ddr.html, 2014.

[15] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 287–305. ACM, 2017.

[16] FuturePlus System. DDR2 800 Bus Analysis Probe. http://www.futureplus.com/download/datasheet/fs2334_ds.pdf, 2006.

[17] Behrad Garmany and Tilo Müller. PRIME: Private RSA Infrastructure for Memory-less Encryption. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 149–158. ACM, 2013.

[18] Blaise Gassend, G Edward Suh, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Caches and Hash Trees for Efficient Memory Integrity Verification. In *In Proceedings of the Ninth International Symposium on High-Performance Computer Architecture, (HPCA)*, pages 295–306. IEEE, 2003.

[19] Le Guan, Chen Cao, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. Building a Trustworthy Execution Environment to Defeat Exploits from both Cyber Space and Physical Space for ARM. *IEEE Transactions on Dependable and Secure Computing*, 2018.

[20] Le Guan, Jingqiang Lin, Bo Luo, and Jiwu Jing. Copker: Computing with Private Keys without RAM. In *In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, pages 23–26, 2014.

[21] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. Protecting Private Keys against Memory Disclosure Attacks using Hardware Transactional Memory. In *In Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP)*, pages 3–19. IEEE, 2015.

[22] Shay Gueron. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptology ePrint Archive*, 2016:204, 2016.

[23] Peter Gutmann. Data Remanence in Semiconductor Devices. In *Proceedings of the 10th USENIX Security Symposium*, page 4. USENIX Association, 2001.

[24] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. *Communications of the ACM*, 52(5):91–98, 2009.

[25] Michael Henson and Stephen Taylor. Beyond Full Disk Encryption: Protection on Security-Enhanced Commodity Processors. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, pages 307–321. Springer, 2013.

[26] ARM Holdings. ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/, 2014.

[27] ARM Holdings. ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0487a.k_10775/index.html, 2016.

[28] George Hotz. PS3 Glitch Hack. http://www.eurasia.nu/wiki/index.php/PS3_Glitch_Hack, 2010.

[29] Andrew Huang. Keeping Secrets in Hardware: The Microsoft Xbox™ Case Study. In *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 213–227. Springer, 2002.

[30] Dongxu Ji, Qianying Zhang, Shijun Zhao, Zhiping Shi, and Yong Guan. MicroTEE: Designing TEE OS Based on the Microkernel Architecture. In *The 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2019.

[31] Kari Kostiainen, Jan-Erik Ekberg, N Asokan, and Aarne Rantala. On-board credentials with open provisioning. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 104–115. ACM, 2009.

[32] Markus G Kuhn. Cipher Instruction Search Attack on the Bus-encryption Security Microcontroller DS5002FP. *IEEE Transactions on Computers*, 47(10):1153–1157, 1998.

[33] Wenhao Li, Yubin Xia, Long Lu, Haibo Chen, and Binyu Zang. TEEv: Virtualizing Trusted Execution Environments on Mobile Platforms. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 2–16. ACM, 2019.

[34] Jochen Liedtke. On u-Kernel Construction. In *Proceedings of 15th ACM Symposium on Operating System Principles*, pages 237–250, 1995.

[35] Linaro. OP-TEE: Open Portable Trusted Execution Environment. https://www.op-tee.org/, 2014.

[36] Linaro. OP-TEE Pager. https://github.com/OP-TEE/optee_os/blob/master/documentation/optee_design.md, 2015.

[37] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. *In Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 10, 2013.

[38] Tilo Müller, Andreas Dewald, and Felix C Freiling. AESSE: A Cold-boot Resistant Implementation of AES. In *Proceedings of the Third European Workshop on System Security*, pages 42–47. ACM, 2010.

[39] Tilo Müller, Felix C Freiling, and Andreas Dewald. TRESOR Runs Encryption Securely Outside RAM. In *USENIX Security Symposium*, volume 17, 2011.

[40] Tilo Müller and Michael Spreitzenbarth. Frost: Forensic Recovery of Scrambled Telephones. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, pages 373–388. Springer, 2013.

[41] NCC Group. TPM Genie: Interposer Attacks Against the Trusted Platform Module Serial Bus. https://www.nccgroup.trust/us/our-research/tpm-genie-interposer-attacks-against-the-trusted-platform-module-serial-bus, 2018.

[42] NCC Group. TPM Genie Tool. https://github.com/nccgroup/TPMGenie, 2018.

[43] Panagiotis Papadopoulos, Giorgos Vasiliadis, Giorgos Christou, Evangelos Markatos, and Sotiris Ioannidis. No Sugar but all the Taste! Memory Encryption without Architectural Support. In *European Symposium on Research in Computer Security*, pages 362–380. Springer, 2017.

[44] Peter AH Peterson. Cryptkeeper: Improving Security with Encrypted RAM. In *Proceedings of the 22nd IEEE International Conference on Technologies for Homeland Security (HST)*, pages 120–126. IEEE, 2010.

[45] Rick Boivie, Eric Hall, Charanjit Jutla, Mimi Zohar. Secure Blue - Secure CPU Technology. https://researcher.watson.ibm.com/researcher/view_page.php?id=6904, 2006.

[46] Samsung. Whitepaper: Samsung KNOX Security Solution. 2017.

[47] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 67–80. ACM, 2014.

[48] Patrick Simmons. Security Through Amnesia: A Software-Based Solution to the Cold Boot Attack on Disk Encryption. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 73–82. ACM, 2011.

[49] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 357–368. ACM, 2014.

[50] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. TrustOTP: Transforming smartphones into secure one-time password tokens. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 976–988. ACM, 2015.

[51] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Sushil Jajodia. Trustdump: Reliable Memory Acquisition on Smartphones. In *European Symposium on Research in Computer Security*, pages 202–218. Springer, 2014.

[52] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 367–378. IEEE, 2015.

[53] Arrigo Triulzi. The Jedi Packet Trick Takes over the Deathstar. *Central Area Networking and Security (CANSEC 2010)*, 2010.

[54] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Pixelvault: Using GPUs for Securing Cryptographic Operations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1131–1142. ACM, 2014.

[55] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.

[56] Min Hong Yun and Lin Zhong. Ginseng: Keeping Secrets in Registers When You Distrust the Operating System. *NDSS*, 2019.

[57] Meiyu Zhang, Qianying Zhang, Shijun Zhao, Zhiping Shi, and Yong Guan. SoftME: A Software-Based Memory Protection Approach for TEE System to Resist Physical Attacks. *Security and Communication Networks*, 2019, 2019.

[58] Ning Zhang, He Sun, Kun Sun, Wenjing Lou, and Y Thomas Hou. CacheKit: Evading Memory Introspection Using Cache Incoherence. In *In Proceedings of the 2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 337–352. IEEE, 2016.

[59] Ning Zhang, Kun Sun, Wenjing Lou, and Y Thomas Hou. Case: Cache-assisted secure execution on arm processors. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP)*, pages 72–90. IEEE, 2016.

[60] Shijun Zhao, Qianying Zhang, Guangyao Hu, Yu Qin, and Dengguo Feng. Providing Root of Trust for ARM Trustzone Using On-chip SRAM. In *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*, pages 25–36. ACM, 2014.