SecTEE: A Software-based Approach to Secure Enclave Architecture Using TEE

Shijun Zhao
Institute of Software Chinese Academy of Sciences
Beijing, China

Qianying Zhang*
College of Information Engineering, Capital Normal University
Beijing, China

Yu Qin
Institute of Software Chinese Academy of Sciences
Beijing, China

Wei Feng
Institute of Software Chinese Academy of Sciences
Beijing, China

Dengguo Feng
Institute of Software Chinese Academy of Sciences
Beijing, China

ABSTRACT

Secure enclaves provide a practical solution to secure computation, and current approaches to secure enclaves are implemented by extending hardware security mechanisms to the CPU architecture. Therefore, it is hard for a platform to offer secure computation if its CPU architecture is not equipped with any secure enclave features. Unfortunately, ARM CPUs, dominating mobile devices and having increasing momentum in cloud markets, do not provide any security mechanisms achieving the security equivalent to modern secure enclave architectures. In this paper, we propose SecTEE, a software-based secure enclave architecture which is based on the CPU's isolation mechanism and does not require specialized security hardware of the CPU architecture such as memory encryption engines. SecTEE achieves a high level of security even compared with hardware-based secure enclave architectures: resistance to privileged host software attacks, lightweight physical attacks, and memory access based side-channel attacks. Besides, SecTEE provides rich trusted computing primitives for enclaves: integrity measurement, remote attestation, data sealing, secrets provisioning, and life cycle management. We implement a SecTEE prototype based on the ARM TrustZone technology, but our approach can be applied to other CPU architectures with isolation mechanisms. The evaluation results show that most overhead comes from the software encryption and the runtime overhead imposed by trusted computing primitives is acceptable.

KEYWORDS

Secure enclave, TEE, ARM TrustZone, Board-level physical attacks, Memory access based side-channel attacks

*Corresponding author.

ACM Reference Format:
Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. 2019. SecTEE: A Software-based Approach to Secure Enclave Architecture Using TEE. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3319535.3363205

1 INTRODUCTION

Board-level physical attacks are becoming practical threats to computer systems, such as cold boot attacks [42], bus monitoring attacks [44, 46, 59, 78] and DMA attacks [19, 105]. These attacks only require inexpensive attack tools, most of which are publicly available [25, 76, 78, 97], resulting in that hackers can easily reproduce these attacks. As a result, these attacks present a big challenge to the security of computer systems, which are usually equipped with security measures only against software attacks. To tackle this challenge, academic and industrial communities propose the secure enclave architecture, which supports secure computation under physical attacks and privileged software attacks.

Secure enclaves are implemented as secure isolated execution environments. Usually, their security is guaranteed by hardware security mechanisms of CPUs. A specialized memory encryption engine on the CPU encrypts DRAM regions belonging to secure enclaves, and thus no code/data is stored outside CPU in plaintext form. This mechanism prevents physical attacks against hardware components outside CPU die, such as DRAM and buses. Besides, the isolation mechanism of CPU isolates address spaces of secure enclaves to prevent software attacks from the host OS and applications. This paper mainly considers the security and functionality of secure enclave architectures, so we refer to "secure enclave architecture" meaning all the technologies that help to achieve the above security level, no matter whether the isolated execution environment is within the address space of its host application or not.

Since secure enclaves provide a high level of security, most CPU giants deploy this feature to their products. Apple extends a secure enclave coprocessor within its SoC. Intel proposes the Software Guard Extensions (SGX) technology and deploys it to all Core Processors (6th-generation and later). SGX is the most widely used secure enclave technology, and a variety of SGX-based security solutions [3, 4, 10, 18, 23, 54, 58, 90, 93, 95, 106, 111] have been proposed. AMD implements its secure enclave architecture by incorporating a coprocessor called AMD Secure Processor (AMD-SP) into the processor. The weakness of AMD's secure enclave architecture is that it does not provide integrity for enclaves' DRAM regions. IBM proposes the SecureBlue [87] and SecureBlue++ [6] technologies that can be built into its processors to protect devices from physical attacks.

ARM CPUs, which dominate mobile devices and gain increasing momentum in cloud platforms, however, do not support secure enclaves. Unfortunately, mobile devices could be lost easily and cloud platforms could be tampered with by cloud server providers, so ARM platforms have a strong requirement for a secure enclave architecture. Current approaches to secure enclaves require modifying the CPU and cannot be applied to commodity ARM platforms.

There are two issues to address when building a secure enclave architecture for ARM platforms. The first one is the security problem. ARM proposes TrustZone [2] as the security pillar of its CPU architecture, which separates a secure world called trusted execution environment (TEE) for security-critical code, but TrustZone is designed to only resist software attacks and is unable to resist physical attacks. Besides, software side-channel attacks are also becoming practical threats to secure enclaves, especially the page fault based side-channel attacks and the cache based side-channel attacks [9, 14, 24, 32, 41, 73, 110, 114], which we refer to in this paper as the "memory access based side-channel attacks". This paper focuses on the memory access based side-channel attacks because of two reasons. First, these attacks are the most threatening ones against ARM platforms [68, 98, 99, 112], which even have been developed against ARM TrustZone [68, 122]. Although there are other kinds of software side-channel attacks, such as side channels based on speculative execution (Foreshadow [107], SgxPectre [13], et al.), based on page directory [115], based on branch prediction (BranchScope [21], Branch Shadowing [61], et al.), based on TLB [33], these attacks usually target Intel CPUs, and it is better to protect against them in hardware. Second, Ge et al. [27] have found that side channels are caused by the inherent insecurity of the hardware, and that OS itself is powerless to close all side channels and it can prevent side-channel attacks only when the hardware provides it with sufficient protection mechanisms. For existing ARM platforms, it is difficult to prevent all kinds of side-channel attacks under current hardware protection mechanisms. Therefore, we only consider resisting memory access based side-channel attacks in this paper, and our resistance to these side-channel attacks can be seen as an instance of preventing side-channel attacks by leveraging CPU's partition mechanism for memory management and caches.

The second issue is that ARM platforms lack necessary trusted computing features required by secure enclaves. For example, an enclave should be able to attest itself to a remote user that it is issued by a legal entity, runs on a genuine platform, and its state is trustworthy enough to be provisioned with secrets.

In this paper, we design a software-based secure enclave architecture for ARM platforms, named SecTEE. We achieve our goal by leveraging software-based security primitives of resisting physical attacks and side-channel attacks, and do not require modifications to the CPU hardware and only require some basic security hardware resources which are common on commodity CPUs. In contrast to specious arguments that software-based approaches are unable to offer the same security guarantees as hardware-based approaches [70], SecTEE illustrates a software-based approach to a secure enclave architecture providing strong security execution environments: 1) based on the SoC-bound execution environment technology and TEE's isolation capability, SecTEE offers the same security properties as Intel SGX, namely resistance to privileged host software attacks and lightweight physical attacks; 2) based on the page coloring technique [55, 84] and CPU's hardware support on cache maintenance, SecTEE protects ARM TrustZone from memory access based side-channel attacks, including cross-core cache attacks.

Although there are some solutions that leverage the page coloring technique to resist cache based side-channel attacks [31, 55, 92], their approaches cannot be used directly in the ARM TrustZone context. In their contexts, there exists a privileged system (the hypervisor, for instance) controlling and managing the whole physical memory, so they can divide the physical memory into pieces that are guaranteed to not contend in the cache. However, in the context of ARM TrustZone, the host OS in the normal world and the TEE OS in the secure world manage the normal memory and the secure memory respectively, and the normal world and the secure world share caches, therefore, the host OS, which can be compromised in the ARM TrustZone's threat model, can always perform cache based side-channel attacks by manipulating a piece of normal memory sharing the same cache with the victim's memory in the secure world. Especially, when the attacker and the victim run on separate CPU cores, the attacker can monitor the cache lines of the victim while the victim is running, so cleaning the victim's cache during the victim's context switch is useless. Therefore, it is a technical difficulty to resist cache based side-channel attacks in the context of ARM TrustZone.

SecTEE is designed to be incorporated into the TrustZone software architecture. Compared to Intel SGX, a main difference of the SecTEE architecture is that there is a specialized OS, i.e., TEE OS, for enclave management, and all enclave management functionality, such as memory management, enclave loading, and initialization, is moved from host system software to the TEE OS. This approach allows system designers not to expose memory management and scheduling of enclaves to host system software, and further allows us to deploy a mechanism of resisting memory access based side-channel attacks in the memory management service. Another benefit of this approach is that it reduces host applications' complexity and allows host application developers to focus on the software logic and ignore the burden of enclave management.

SecTEE extends to the TEE OS critical trusted computing features required by secure enclaves, including *enclave identification, enclave measurement, remote attestation, data sealing, secrets provisioning*, and *life cycle management of enclaves*. Secure enclaves run as trusted applications (TA) on the security-enhanced TEE OS. The extended trusted computing features expose their interfaces to enclaves as system calls. When an enclave is going to be loaded into the system, the security-enhanced TEE OS verifies the enclave's identity, measures and checks its integrity. After loaded, the enclave could invoke the extended system calls to store its sensitive data and attest its identity and integrity to an external entity. After the attestation, the external entity can provision secrets to the enclave.

From an enclave developer's perspective, SecTEE is a security-enhanced TEE architecture. A developer builds his enclave like programming a TA and exposes the interfaces of the enclave to host applications as TA commands. Host applications use the services of the enclave by invoking the corresponding TA commands. Besides, the developer needs to compute the initial measurement of the enclave and signs the measurement using his signing key. The measurement and identity of the enclave will be verified when it is loaded into SecTEE.

We have implemented SecTEE on a TrustZone-enabled platform, NXP i.MX6Q development platform. We evaluate the performance overhead introduced by SecTEE using a mature TEE test suite tool Xtest [67] and some security enclaves. The evaluation results show that the overhead mainly comes from software encryption and that the trusted computing features only introduce acceptable overhead. In summary, the key contributions of this paper include

- A new secure enclave architecture for commodity ARM platforms, SecTEE, which can be incorporated into ARM TrustZone software architecture and achieves the highest level of security for secure enclaves, that is, resistance to board-level physical attacks, strong isolation, and resistance to memory access based side-channel attacks.
- The page coloring technique cannot prevent cache based side-channel attacks against TrustZone, especially the cross-core attacks. To address this technical difficulty, we design a locking mechanism which locks the enclave pages in the cache and combine it with the page coloring technique to resist memory access based side-channel attacks against TrustZone. This approach demonstrates that hiding the memory management of secure enclaves from host software is a practical way to eliminate memory side channels.
- An approach to adding rich trusted computing features to TEE systems, which enables TEE systems to identify, measure, attest security applications, seal sensitive data, and enables users to provision secrets to them.
- An implementation of the SecTEE architecture on a mature TEE system, showing that it is feasible to deploy SecTEE on mature TEE systems. The evaluation results show that memory encryption incurs most performance overhead, and that the extended trusted computing features impose acceptable overhead.

The rest of the paper is organized as follows. Section 2 gives the background information related to this paper. Section 3 describes the threat model. Section 4 lists the design goals of SecTEE and outlines an overview of SecTEE design. Section 5 illustrates the details of SecTEE architecture. Section 6 implements and evaluates our prototype system. Section 7 surveys related work. Section 8 concludes this paper.

2 BACKGROUND

2.1 Intel SGX

Intel SGX is a set of CPU instructions for creating, running, and managing secure enclaves. SGX separates a memory region from DRAM, called PRM (Processor Reserved Memory), for enclaves. Enclaves' code and data are stored in Enclave Page Cache (EPC) pages of the PRM. SGX enforces an access-control policy on PRM to prevent PRM from being accessed by non-enclave software. So enclaves do not need to rely on the security of system software. SGX also provides a specialized Memory Encryption Engine (MEE) [39] to encrypt and perform integrity checks on PRM, which prevents physical attacks from reading or manipulating enclaves' code and data. In a word, SGX offers a high level of software and physical security for enclaves.

An enclave is mapped to a reserved memory area of the virtual address of a host application, called ELRANGE (Enclave Linear Address Space). The virtual address outside ELRANGE is used to map non-enclave code and data. SGX prevents non-enclave software from accessing ELRANGE, while the enclave software in the ELRANGE is able to access the non-enclave address space.

SGX leverages host system software to manage enclaves, such as creating, loading, and scheduling enclaves. The system software needs to allocate EPC pages for a newly created enclave, load the initial code and data into the enclave, and establish the memory mapping between ELRANGE and EPC pages using its page tables. The system software is also able to interrupt and resume the enclave like a normal process. As the system software that manages enclaves is untrusted, the SGX hardware needs to measure the loaded code and data of an enclave and check the measurement results with the value specified by the enclave developer. Unfortunately, as SGX allows the untrusted system software fully controlling enclaves, practical side-channel attacks [9, 32, 73, 91, 94, 114] are proposed, in which attackers can infer information of an enclave by scheduling it and manipulating its page tables.

2.2 ARM Cache Architecture

ARM CPU is a modified Harvard architecture and typically has two levels of caches. The level one (L1) cache consists of two separate caches, an instruction cache (I-cache) and a data cache (D-cache). The level two (L2) cache is unified and holds both instructions and data. For a cache architecture, if all data from lower levels must be stored in a higher level cache, it is called inclusive; if data can only reside in one of the cache levels, it is called exclusive; if the cache is neither inclusive nor exclusive, it is called non-inclusive. Unlike most modern CPUs, which have either inclusive last-level caches (LLC) (Intel CPUs) or exclusive LLC (AMD CPUs), ARM CPUs do not fix their policy on cache inclusiveness: caches can be inclusive, exclusive, or non-exclusive.

A cache line is the unit of data transfer between the cache and main memory. ARM CPU caches are organized as N-way set associative caches. The cache is divided into N equally-sized pieces, called ways, and each way consists k cache lines with indexes 0 ~ k-1. The cache lines from all ways with the same index compose a cache set. The main memory is divided into blocks, and the size of a block is equal to the size of a cache way. The i^{th} entry of a memory block can be loaded into any one of the N cache lines in the $(i \ mod \ N)^{th}$ cache set. On CPUs with TrustZone extensions, each cache line is extended with an NS bit, indicating the cache line belongs to the secure world or the normal world.

The ARM CPU architecture provides programmers with cache maintenance operations: *invalidation*, *cleaning*, *zero*, and *preload*. Invalidation of a cache means to clear its data; cleaning a cache means to write its contents to the next level of cache or to main memory; zero a cache means to zero a block of memory with the cache; preload instructions allow programmers to preload memory content to cache. In TrustZone-enabled systems, operations performed from the normal world only affect the non-secure cache lines, while operations performed from the secure world can affect all cache lines.

2.3 SoC-bound Execution Environments and the OP-TEE Pager System

SoC-bound execution environments are software-based approaches to resisting physical attacks. The idea is to leverage the memory inside SoC, such as CPU registers [26, 74, 75, 96], CPU caches [37, 38, 120, 121], GPU registers and caches [109], or on-chip memory (OCM) [11, 15, 36, 43, 82, 119, 123], to build a secure execution environment for security-critical code. Sensitive data shall be encrypted when it is swapped out of the environment, and thus physical attacks against hardware components outside the SoC are prevented.

OP-TEE [65] is a popular open source TEE OS maintained by Linaro, which implements a Trusted Execution Environment using ARM TrustZone technology and is compatible with GlobalPlatform TEE specifications. To prevent physical attacks, an SoC-bound execution environment called *Pager* [66] is proposed.

Technically, Pager is a demand paging system separated from the OP-TEE kernel, which is responsible for maintaining execution of the rest components of OP-TEE kernel and TAs. It sets the OCM as the working memory for CPU to execute the OP-TEE system, and uses the DRAM as a backing store. Pager runs on the OCM, and the other components of the OP-TEE kernel and TAs are encrypted and stored in the DRAM. Pager manages swapping between the OCM and DRAM: when code or data stored in DRAM is demanded, Pager decrypts and performs integrity check on the corresponding page, and loads it into the OCM; when an OCM page needs to be swapped to DRAM, Pager encrypts it.

3 THREAT MODEL AND HARDWARE REQUIREMENTS

3.1 Threat Model

SecTEE aims to achieve the same security level of modern secure enclave architectures, which protects the confidentiality and integrity of enclaves from an adversary who has full control of the system software and hardware components outside of the SoC, such as DRAM and peripherals. We require that the platforms where SecTEE is deployed support the ARM TrustZone technology.

At the software level, the commodity OS in the normal world is untrusted and potentially compromised. The adversary can access interfaces of the TEE system. He can create and load malicious enclaves to the system. However, we assume that the TEE OS is trustworthy and provides isolation for TAs running on it because we do not aim to increase the security of TEE OS regarding software attacks.

At the hardware level, we assume that the SoC is trusted, and all components outside of the SoC are assumed to be vulnerable, including DRAM, address and data buses between CPU and DRAM, other I/O devices, and so on. So the adversary is able to probe the CPU-DRAM bus, observe and tamper with the DRAM contents, and launch malicious peripherals. In particular, the following practical physical attacks are within our threat model: cold boot attacks, bus probing attacks, and DMA attacks. We do not consider attacks against internal states of the SoC, which are sophisticated and require expensive equipment.

3.2 Hardware Requirements

We list the hardware primitives required by SecTEE, which are common on modern mobile devices.

Device Sealing Key (*DSK*). It is a symmetric key generated in the SoC during manufacturing. It is only known by the device, and even the manufacturer does not know it. DSK is used to protect secrets that are related to a device. As no other devices know this key, secrets protected by it are bound to the device, and other devices cannot get them. This key may be referred to as other terminologies, such as the *Seal Secret* in SGX [16] or *Device-Unique Hardware Key* (*DUHK*) in Samsung's KNOX [88].

Device Root Key (*DRK*). It is an asymmetric key pair generated at manufacture time and signed by the manufacturer's root key through a certificate. Its private part should be stored in the secure storage of the device, such as processor's eFuses. As DRK is device-unique and signed by the manufacturer, it can be used to identify and authenticate the device.

Manufacturer's Public Key. The public part of the manufacturer's root key should be hard-coded into the SoC. Software developed by the manufacturer should be signed by the private part of the manufacturer's root key, including TEE OS and some special TAs. The hard-coded public key can be used to verify whether a loaded software component is issued by the manufacturer, and thus the secure boot and TCB of the TEE system can be established.

4 DESIGN OVERVIEW

4.1 Design Goals

Security. The designed system should be able to resist all kinds of practical attacks against secure enclave architectures. First, it should provide complete isolation of enclaves from host software, including privileged system software. Second, it should prevent practical physical attacks, in particular, attacks against hardware components outside of the chip. Third, it should be able to resist memory access based side-channel attacks.

Compatibility. The design should be compatible with standard ARM TEE system architectures, such as GlobalPlatform TEE system architecture [30], so that it can be deployed to existing TEE systems.

No requirements for specialized hardware memory protection mechanisms. We want our design to be deployed on commodity ARM platforms, which are not provided with hardware protection mechanisms to prevent physical attacks and memory access based side-channel attacks. So the design should achieve the security goal under the assumption that only a common CPU architecture is provided and no specialized hardware protection mechanism is available.

Rich trusted computing features. Trusted computing features, such as integrity measurement, remote attestation, and data sealing, are becoming indispensable means for modern computing platforms to guarantee their security. It is necessary for a secure enclave architecture to provide rich trusted computing features.

4.2 Overview of SecTEE

SecTEE, illustrated in Figure 1, is a TEE system that satisfies enclaves' both security requirements and functionality requirements for trusted computing. SecTEE leverages the SoC-bound execution environment technology and isolation mechanism provided by ARM TrustZone to achieve the physical and software security respectively, and its kernel provides the mechanism of resistance to memory access based side-channel attacks.

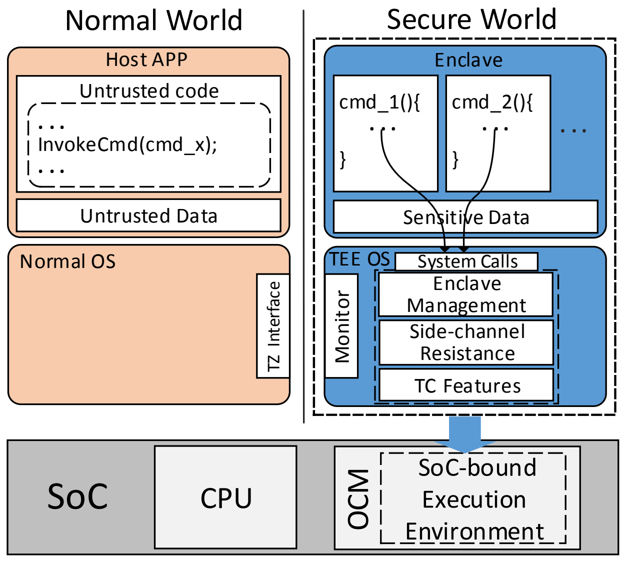

Figure 1: An Overview of SecTEE

Secure enclaves are implemented as TAs in the TEE system. To create an enclave for a host application, the enclave author first identifies the sensitive code and data of the host application, then creates a TA, and finally moves the sensitive code and data into the TA. The enclave exposes its interfaces to the host application as TA commands, and the host application can run the sensitive code by invoking the TA commands.

As enclaves are implemented as applications of the TEE OS, it is natural to integrate the enclave management into the TEE OS kernel. So our design does not expose enclave management interfaces to host system software (while Intel SGX does), but directly leverages the TA management functionality of the TEE OS. The enclave management functionality is responsible for memory management of enclaves and invoking, interrupting, resuming, and scheduling enclaves.

SecTEE extends trusted computing features to the TEE OS kernel, which can be used to identify, measure, and attest enclaves, and to protect sensitive data of enclaves. These features expose their interfaces to enclaves as system calls (Table 1). We identify the following trusted computing features that modern secure enclave architectures should provide.

- *Enclave identification.* A platform supporting secure enclaves should be able to identify the author of an enclave loaded on it. To achieve this goal, an enclave author should sign his enclaves using his own signing key, and the platform identifies the enclave's author by verifying the signature.
- *Enclave measurement.* The platform should measure the integrity of the enclave before running it. The measurement results present the good/bad state of enclaves, and they can be used by the platform to perform remote attestation.
- *Remote attestation.* This is a key feature of a trusted computing system, by which a platform can convince a verifier that an enclave is in a good state and runs on a trusted system. This feature can be achieved by a signature on the measurement of the attesting enclave using a certified attestation key. Since we assume that the TEE OS is trustworthy, TEE OS rollback attacks are not considered in SecTEE's design and implementation. To prevent such attacks, hardware monotonic counters or rollback prevention fuses [88] are needed.
- *Data sealing.* This feature is used to bind sensitive data to an enclave and a platform, and ensures that only the particular enclave running on the specific platform can access the bound data.
- *Secret provisioning.* This feature enables a remote data owner to provision his sensitive data to an enclave. The remote data owner is convinced that the enclave is running on a trusted platform and the confidentiality of the sensitive data that will be provisioned can be guaranteed. Typically, this feature is conducted through remote attestation and a secure channel.

Comparison with Sanctuary [8]. Sanctuary is also a secure enclave architecture for ARM CPUs. It provides isolated compartments for security-sensitive code based on TrustZone and enables SGX-like usage of these compartments. We compare SecTEE with it in the following aspects: security, trusted computing primitives, and the TCB size.

- **Security.** Sanctuary only provides isolation for enclaves, while SecTEE achieves a much higher security level: in addition to isolation, SecTEE provides both resistance to board-level physical attacks and resistance to memory access based side-channel attacks, which are requirements of modern secure enclave architectures, such as Intel SGX, Komodo, and Sanctum. Besides, Sanctuary adopts a design similar to Intel SGX: it leverages the system software in the normal world to manage enclaves' resource, such as memory allocation. From lessons we learned from the-state-of-art architectures (Section 4.3), it is hard for this kind of design to add protection mechanisms against memory access based side-channel attacks.
- **Trusted computing primitives.** SecTEE provides more comprehensive trusted computing primitives than Sanctuary, such as the key hierarchy of platforms and secrets provisioning, and illustrates all the details of these trusted computing primitives.
- **TCB size.** Sanctuary does not increase the TCB of TEE systems because it executes enclaves in isolated compartments in the normal world. Although SecTEE has an increase of TCB, the increase is acceptable (evaluated in Section 6.1), and we give suggestions on how to decrease TCB in Section 6.1.

In conclusion, SecTEE has a larger TCB than Sanctuary, but the increase of TCB is acceptable, and SecTEE achieves the highest

Table 1: SecTEE System Calls

System Calls	Description of SecTEE Operations
`syscall_request_AK(void *pubAK, void *sigDRK)`	Generate an attestation key AK and sign it with the device root key.
`syscall_seal_AK(bool flag, void *SealedAK)`	Receive a flag indicating whether the generated attestation key is approved by the manufacturer, and if it is true, seal the attestation key.
`syscall_import_AK(void *SealedAK, void *sigDRK)`	Import a sealed attestation key SealedAK.
`syscall_remote_attestation(char *report_data, void *attest_sig)`	Perform attestation on report_data, and store the attestation result in attest_sig.
`syscall_seal(char *data, char *ciphertext)`	Seal data and return the result to the invoking enclave.
`syscall_unseal(char *ciphertext, char *data)`	Unseal the sealed data ciphertext and return the result to the invoking enclave.
`syscall_provisioning(void *DH_A, void *DH_B, void *sigAK, void *DH_shared)`	Attest the trustworthiness of the invoking enclave to a remote party, and establish a secure channel for data provisioning with the party.

security level of modern secure enclave architectures and provides more comprehensive trusted computing primitives.

4.3 Lessons from Other Architectures

We analyze some popular secure enclave architectures in industrial and academic areas: Intel SGX, Sanctum [17], and Komodo [22], and describe lessons we have learned from these architectures, which help us to design SecTEE.

Actually, all of these secure enclave architectures provide similar trusted computing features and management functionality for enclaves. The main difference among them is the way of managing enclaves. Figure 2 shows the design overviews of SGX, Sanctum, and Komodo.

Intel SGX (Figure 2 (a)) implements the management functionality and trusted computing features in the processor and exposes their interfaces as ISA (Instruction Set Architecture) extensions. In SGX, enclaves are implemented as isolated execution environments embedded in the address spaces of host applications. The OS is responsible for managing enclaves, such as allocating memory for enclaves, managing virtual-physical address mappings for enclaves, loading initial data and code into enclaves, and scheduling enclaves. However, the abilities of controlling memory management and scheduling enclaves lead to memory access based side-channel attacks against SGX. One type of side-channel attacks learns about the memory page usage of an enclave by exploiting page faults of the enclave [94, 114]. The other type of these attacks is cache attacks [9, 32, 73, 91] which learn the memory access patterns of an enclave.

Sanctum (Figure 2 (b)) is an SGX-like secure enclave architecture for the RISC-V architecture. It achieves the same level of software security as SGX, but does not offer protection against any physical attacks due to the lacking of memory encryption engines. One important contribution of Sanctum is that it adds full protection against memory access based side-channel attacks such as page fault monitoring attacks and cache attacks. It achieves this goal by setting individual page table for each enclave and modifying the cache hardware to ensure that each enclave uses distinct cache sets. Compared to SGX, one advantage of Sanctum is that it leverages the hardware-software co-design to achieve minimal hardware modifications. Most of its trusted computing features are implemented in a trusted software secure monitor and invoked through monitor calls that mirror SGX ISA instructions.

Komodo (Figure 2 (c)) is a hardware-software co-design of secure enclave architecture. It aims to disentangle the enclave management (such as memory management) and trusted computing features (such as measurement and remote attestation) from basic hardware mechanisms (such as isolation and memory protection). Komodo delegates the enclave management and trusted computing features to a privileged software monitor, making it easy to update and patch security flaws. Although Komodo isolates enclaves' page tables from the untrusted OS, it still relies on the untrusted OS to manage and schedule enclaves. So it is vulnerable to cache attacks, and that's why memory access based side-channel attacks are excluded in its threat model.

Lessons. From the above analysis, we obtain the following two lessons. First, implementing the whole secure enclave architecture in hardware is inflexible. For example, it is hard to fix up memory side channels for CPUs that have been shipped. Software is much more malleable than hardware, so it is better to combine necessary hardware security mechanisms with software to implement the architecture in a flexible and updatable way. Both Sanctum and Komodo adopt this approach. Second, relying on the untrusted OS to manage enclaves (especially memory management and scheduling) enables attackers to launch software side-channel attacks, such as side-channel attacks against SGX. To prevent this kind of attacks, the above three secure enclave architectures have to modify their CPU hardware, such as Sanctum's cache partitioning scheme. The lessons we learn from these architectures help us to present a better design of SecTEE. First, SecTEE only requires basic hardware security components, and most of its functionality is implemented in software, which makes it flexible for manufacturers to fix up security flaws and add new features. Second, SecTEE puts the enclave management functionality inside the TEE OS, so the host software is unable to control memory management of enclaves or schedule enclaves. This design makes it possible for system designers to enforce mechanisms of resisting memory access based side-channel attacks. Another reason for us to put the enclave management inside TEE OS is that, in the TrustZone software architecture, the TEE OS is designed to manage TAs. So SecTEE can be easily incorporated into the ARM TrustZone architecture.

5 SECTEE ARCHITECTURE

This section describes the details of the SecTEE architecture.

5.1 Memory Protection

To protect enclaves from physical attacks, SecTEE leverages the SoC-bound execution environment technology, such as OP-TEE Pager, to provide memory protection on the whole TEE system. The

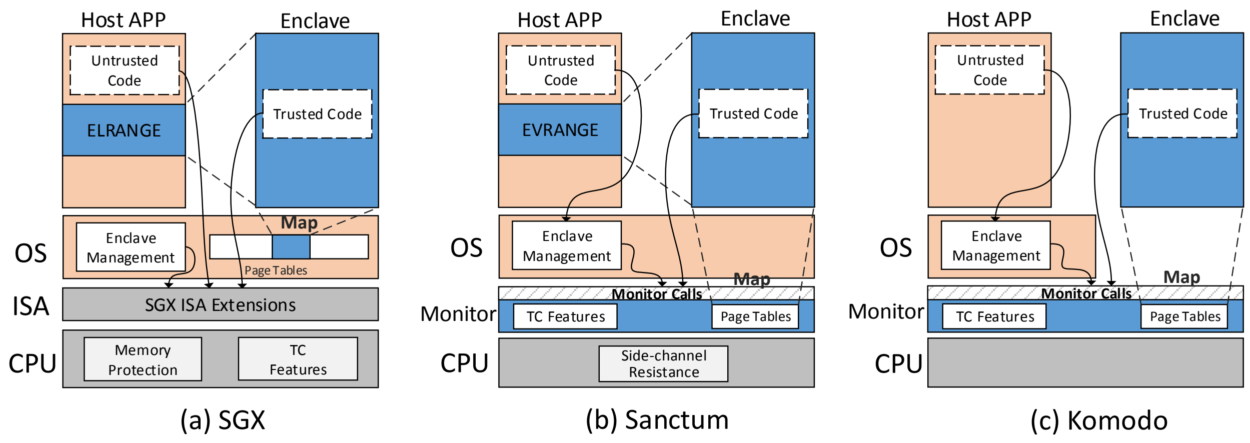

Figure 2: Design Overviews of SGX, Sanctum, and Komodo

SoC-bound execution environment is a demand paging system with memory protection mechanism. It runs the whole TEE system on the OCM, and uses the DRAM as a backing store for the TEE system. It also guarantees the confidentiality and integrity properties for the backing store: when a page in the OCM is going to be swapped out of the OCM, the memory protection component encrypts and hashes the page; when a page in the backing store is demanded and swapped into the OCM, the memory protection component decrypts the page and performs integrity check on it.

5.2 Side-channel Resistance

Since all page faults of enclaves are handled by the SecTEE kernel, host software is unable to learn memory page usage by manipulating page tables, thus page fault based side-channel attacks are prevented directly. So we focus on how to prevent cache attacks.

5.2.1 Resisting attacks from the secure world. The basic requirement of launching cache attacks from the secure world is to load an attack enclave to memory locations which share the same cache sets with the victim enclave. Then the attacker can use the attack methods – *Evict+Time*, *Prime+Probe* [80], *Flush+Reload* [116], *Evict+Reload* [35], and *Flush+Flush* [34] – to learn the victim enclave's memory access patterns, and can even launch cross-core attacks [40, 47, 49–51, 68, 69, 113, 116] by exploiting the LLC.

We prevent cache attacks from the secure world by a page coloring mechanism: modifying the memory management service of SecTEE kernel to make different enclaves never share cache sets. Then no matter how the attacker manipulates the attack enclave (including launching cross-core attacks), it will not affect the cache of the victim enclave. As the working memory of enclaves is the OCM, we only need to guarantee that all the OCM pages assigned to an enclave do not have collisions in cache sets with OCM pages of other enclaves.

We propose a separation scheme for OCM to achieve the above goal (Figure 3). Suppose the cache is an N-way set associative cache, the total size of the cache is S_T, the size of each cache line is S_{CL}, the size of each cache way is $S_W = S_T/N$, and the size of one page is S_P. We divide each cache way into p ($p = S_W/S_P$) page-sized blocks, and blocks from all ways with the same index compose a

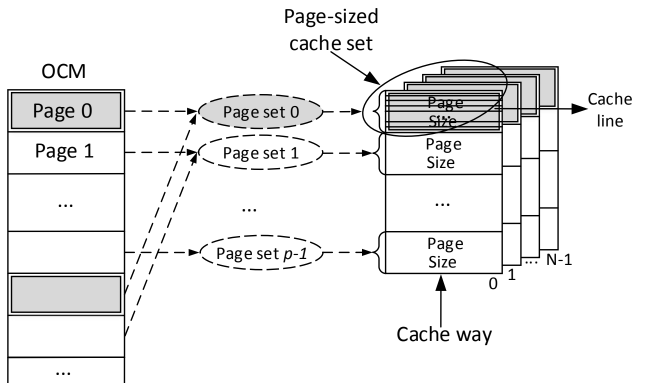

Figure 3: OCM Separation Scheme

page-sized cache set. Suppose the size of OCM is S_{OCM}, and we call all pages that map to the same page-sized cache set a *page set*. As pages from different page sets do not map to the same page-sized cache set, we load enclaves into different page sets to guarantee the isolation of their cache sets:

(1) When an enclave is invoked, SecTEE assigns a free page set for it, and pages of this enclave will be loaded into pages of this page set. If there are no free page sets, SecTEE chooses a page set that has been used least recently (LRU), swaps all the pages of the page set into the DRAM, and finally uses cache maintenance operations to clean and invalidate the page-sized cache set of the page set. Figures 4 and 5 show the assembly code of using the *DCCISW* operation to clean and invalidate a range of cache sets on ARMv7 and ARMv8 architectures respectively.
(2) When an enclave needs to load a page from DRAM to OCM and it happens to run out of all pages of its page set, SecTEE allocates a new page set for the enclave if there are free page sets, and if no free page sets are left, SecTEE frees an OCM page by swapping the enclave's least recently used page from the OCM to the backing store and loads the demanded page into the just freed OCM page.

```
/* R0: the maximum way number;
 * [R1, R2]: the set range of the enclave;
 * R3: 32 - Log2(ASSOCIATIVITY);
 * R4: Log2(LINELEN);
 * R10: Cache number;
 * ASSOCIATIVITY and LINELEN are parameters defined in CCSIDR
 */
   SUB R10, #1
   LSL R10, R10, #1
LOOP_WAY:
   MOV R7, R2
LOOP_SET:
   ORR R11, R10, R0, LSL R3 ;factor in the way number and cache
                            number into R11
   ORR R11, R11, R7, LSL R4 ;factor in the set index number
   MCR p15, 0, R11, c7, c14, 2 ;DCCISW operation
   SUB R7, R7, #1 ;decrement the set number
   SUBS R8, R7, R1
   BGE LOOP_SET
   SUBS R0, R0, #1 ;decrement the way number
   BGE LOOP_WAY
```

Figure 4: Cache Clean and Invalidate for ARMv7

```
/* W0: the maximum way number;
 * [W1, W2]: the set range of the enclave;
 * W3: 32 - Log2(ASSOCIATIVITY);
 * W4: Log2(LINELEN);
 * W10: Cache number
 */
   SUB W10, #1
   LSL W10, W10, #1
LOOP_WAY:
   MOV W7, W2
LOOP_SET:
   ORR W11, W10, W0, LSL W3 ;factor in the way number and
                            cache number into W11
   ORR W11, W11, W7, LSL W4 ;factor in the set index number
   DC CISW, X11 ;DC CISW operation
   SUB W7, W7, 1 ;decrement the set number
   SUBS W7, W7, W1
   B.GE LOOP_SET
   SUBS X0, X0, 1 ;decrement the way number
   B.GE LOOP_WAY
```

Figure 5: Cache Clean and Invalidate for ARMv8

Take the NXP i.MX6Q platform based on which we implement our prototype as an example, it has a 16-way 1MB L2 unified cache and 256KB OCM, and the page size is 4KB. We divide the OCM into $p = S_W / S_P = 1MB/16/4KB = 16$ page sets, and the size of each page set is $256KB/16 = 16KB$. So we can run 16 enclaves simultaneously at most, which is enough for the host OS.

5.2.2 Resisting attacks from the normal world. In ARM TrustZone, cache maintenance operations only affect non-secure caches, so attackers from the normal world cannot leverage cache maintenance operations to manipulate caches of the secure world. So the cache attacks relying on cache maintenance operations such as *Flush+Reload* and *Flush+Flush* will not succeed in the normal world. Thus, attackers from the normal world can only launch cache attacks based on memory operations, such as *Prime+Probe*.

Unfortunately, the page coloring technique cannot protect ARM TrustZone from cache attacks because in the context of ARM TrustZone, it does not partition all the memory but only the secure memory into separate page sets, and the normal world's memory, which shares all caches with the secure world, can be leveraged to launch cache attacks.

To prevent cache attacks from the normal world, SecTEE cleans and invalidates all the cache lines of the invoked enclave when the CPU switches from the secure world to the normal world. As ARM architecture does not use inclusive LLC, SecTEE needs to clean and invalidate all cache levels.

Cleaning the caches of the invoked enclave when CPU returns to the normal world only prevents a local attacker on the same core from learning the access patterns of the enclave, but cannot prevent cross-core cache attacks: the attacker can monitor the cache usage of a victim enclave by manipulating a spy program which shares the same cache sets with the victim enclave and runs on a different core. Since the spy program performs its attack during the execution of the victim enclave, the cache cleaning operations, which happen only when the core switches back to the normal world, cannot prevent this attack.

To prevent cross-core cache attacks against ARM TrustZone, we design a locking mechanism to lock the OCM pages of enclaves in the cache: when an enclave is invoked, SecTEE preloads its pages into the cache when they are loaded from DRAM to OCM and then uses the ARM cache locking mechanism [120] to lock the corresponding cache lines. Since the caches of the enclave are locked, manipulating the memory in the normal world cannot probe or manipulate the cache lines of the enclave. The details of the locking mechanism are as follows.

(1) When SecTEE assigns a free page set for some enclave, it cleans and invalidates the page-sized cache set of the page set.
(2) When a page is loaded from DRAM to OCM, SecTEE chooses a free page-sized cache block from the cache set that corresponds to the OCM page, uses the preload operation to load the OCM page into the page-sized cache block, and then locks the cache block.
(3) When an OCM page is swapped out to DRAM, SecTEE unlocks, cleans, and invalidates the corresponding page-sized cache block. However, as ARM only provides a coarse-grained cache locking mechanism that only locks certain cache ways, SecTEE needs to re-lock the way which the cache block belongs to if the cache way still contains any preloaded OCM page.

Take the NXP i.MX6Q platform as an example, we split its L2 cache into 16 page sets, so each page set contains $256KB/4KB/16 = 4$ pages. Since the L2 cache has 16 cache ways, each page set can be preloaded into the cache entirely.

5.2.3 Analysis of Different Timing Caused by Cache Maintenance. Cache maintenance exhibits different timing: attackers can distinguish whether SecTEE cleans the cache of an enclave by probing the memory locations sharing the same caches with the enclave (see the probing timing on OP-TEE and SecTEE in Section 6.5.2). However, cache operations of SecTEE clean all the caches of an enclave (the target of cache attackers), so attackers do not know which memory address is accessed by the enclave. Our experimental results of probing operations on SecTEE (Section 6.5.2) show that attackers cannot distinguish whether a memory location is accessed by the enclave. Similar results are presented by our cache attacks on AES (Section 6.5.3): attackers cannot distinguish which T-table entry is accessed.

5.3 Key Hierarchy

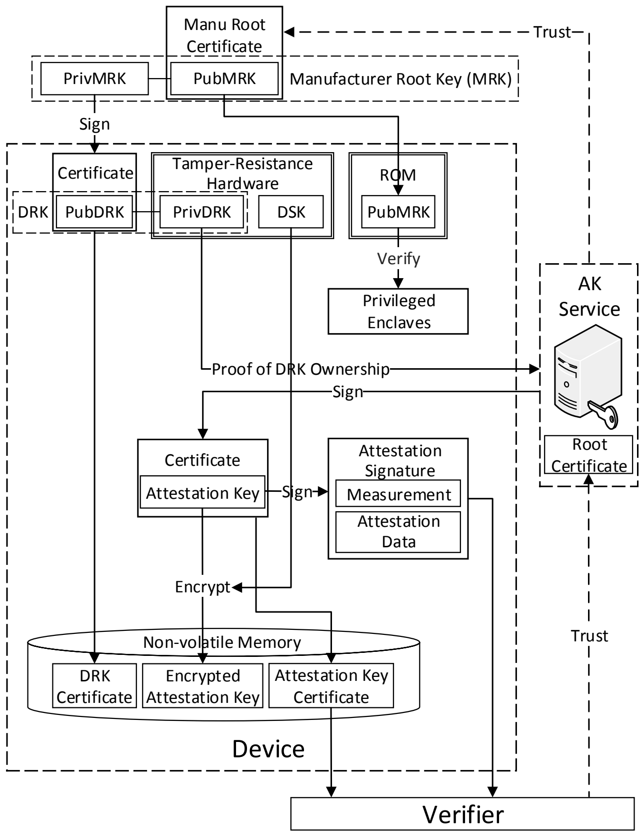

Figure 6: Key Hierarchy of SecTEE

The key hierarchy of SecTEE (Figure 6) is based on three hardware keys: device root key (*DRK*), device sealing key (*DSK*), and the public key of the manufacturer (*PubMRK*). All the three keys are generated during manufacture time. *DRK* and *DSK* should be stored in the secure hardware of the device, such as Battery-backed RAM (BBRAM) or eFuses. *PubMRK* is required to be tamper-resistant only, so it can be hard-coded into the ROM.

DRK is an asymmetric key (*PubDRK*, *PrivDRK*) and unique to devices. During the manufacturing process in the factory, the device generates *DRK* and applies for a certificate from the manufacturer; after receiving the request, the manufacturer signs the public part of *DRK* (*PubDRK*) with its signing key *PrivMRK*, produces a certificate $CERT_{DRK}$, and stores $CERT_{DRK}$ in the non-volatile memory of the device, such as flash. $CERT_{DRK}$ and *DRK* will be used to prove to remote parties that the device is a genuine trusted device. *DRK* is only accessible by specially privileged software within the secure world, i.e., the TEE OS.

DSK is a device-unique symmetric key used to protect device-related sensitive data, such as attestation keys. As the key is device-unique, data encrypted by it is bound to the device and not accessible to other devices.

We refer to the enclaves provided by the manufacturer as privileged enclaves, and there are some operations that only can be performed by these enclaves, such as importing an attestation key. *PubMRK* can be used to verify whether a software component is approved by the manufacturer, and SecTEE uses *PubMRK* to identify privileged enclaves.

5.4 Enclave Identification and Measurement

Before running an enclave, SecTEE needs to recognize the identity of the enclave and make sure that the enclave is intact and exactly the one published by the author, so a scheme for enclave identification and integrity measurement is needed.

SecTEE requires that each enclave author should have a signing key. After the author has built the software image of the enclave, he first measures the software image and generates the standard integrity value of the image. Then he assigns a software ID to the enclave, and generates a certificate for the standard integrity value and software ID by signing them with his signing key. Finally, the author appends his public key and the certificate to the end of the software image and publishes the enclave.

During runtime, SecTEE maintains an information table storing critical information of the enclaves running on it. The table stores the enclave ID, integrity, and a bool flag indicating whether the enclave is privileged. The enclave ID is composed of the public part of the author's signing key and the software ID assigned by the author, and the bool flag is set by comparing the public part of the author's signing key with the *MPK* in the SoC. SecTEE takes the following steps to load an enclave. First, it uses the public key of the author to verify the certificate of the enclave, and only when the verification succeeds, it decodes the certificate and extracts the software image, software ID, and standard integrity value of the enclave. Second, it measures the software image and verifies the measurement result using the standard integrity. Third, if the integrity verification passes, SecTEE loads the software image into the enclave address space. Finally, SecTEE allocates and fills an entry of the information table for the enclave.

5.5 Remote Attestation

Intel SGX implements the remote attestation mechanism as a privileged Quoting Enclave. As it is impossible for enclaves to share sensitive pages in SGX, a prover enclave is unable to communicate with the Quoting Enclave. SGX solves this problem with a local attestation mechanism, which is implemented as an enclave instruction *EREPORT*. The *EREPORT* generates an attestation report which can be transferred to the Quoting Enclave via the host application, and the Quoting Enclave leverages the report to generate the real remote attestation report. In SecTEE, as each enclave can perform remote attestation by invoking the corresponding system call, the local attestation mechanism is unnecessary.

SecTEE implements the remote attestation mechanism in kernel, and system calls which can be used to request and import attestation keys and perform remote attestation are exposed to enclaves (the `syscall_request_AK`, `syscall_seal_AK`, `syscall_import_-AK`, and `syscall_remote_attestation` system calls listed in Table 1). Attestation keys are critical for secure enclave architectures because they are used to attest the trustworthiness of the platform and all enclaves. So it is reasonable to perform the operations of

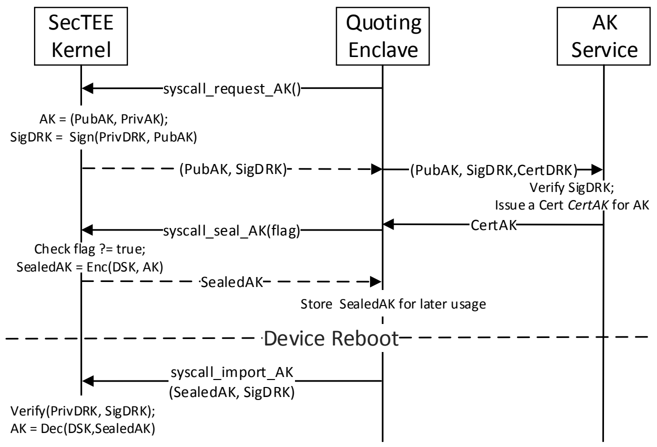

Figure 7: The Workflow of Quoting Enclave

requesting and importing an attestation key by trusted software, such as software components approved by the manufacturer. To this purpose, we implement a privileged Quoting Enclave to request and import attestation keys, and other enclaves are forbidden to perform these operations.

Quoting Enclave. The Quoting Enclave is a privileged enclave published by the manufacturer. It is responsible for applying for certificates for SecTEE's attestation keys. The workflow of the Quoting Enclave is as follows (Figure 7).

(1) If the device has no attestation keys, the Quoting Enclave invokes `syscall_request_AK` to request the kernel to generate an attestation key $AK = (PubAK, PrivAK)$. After generating AK, the kernel signs $PubAK$ with DRK, and returns the signature $SigDRK$ and $PubAK$ to the Quoting Enclave. $SigDRK$ is used to prove AK is generated in a trusted device.
(2) The Quoting Enclave sends the $PubAK$, $SigDRK$, and the certificate of DRK ($CertDRK$) to the AK service, which verifies $SigDRK$. If the verification succeeds, the AK service issues a certificate $CertAK$ for AK and sends $CertAK$ to the Quoting Enclave.
(3) The Quoting Enclave invokes `syscall_seal_AK`, which sends a bool value *flag* indicating whether the AK service issues a certificate for AK. If $flag = true$, the kernel sets AK as its attestation key, seals AK using DSK, and returns the sealed attestation key $SealedAK$ to the Quoting Enclave. The Quoting Enclave saves $SealedAK$ in the non-volatile memory.
(4) When the device is rebooted, the Quoting Enclave can import AK to the kernel by invoking `syscall_import_AK`. The kernel verifies the $SigDRK$ and unseals the $SealedAK$ with DSK to obtain AK.

One step is not described in the above workflow: when the kernel is invoked by `syscall_request_AK`, `syscall_seal_AK`, or `syscall_import_AK`, it verifies whether the public key of the invoking enclave is MRK, and only after the verification succeeds, the kernel runs corresponding services. This verification is used to ensure that only privileged enclaves can manage attestation keys.

Remote Attestation. Each enclave can perform remote attestation by invoking the system call `syscall_remote_attestation`. The kernel receives *report_data* that the invoking enclave wants to attest, and signs *report_data* together with *Measurement* (the integrity value of the invoking enclave stored in kernel), with the attestation key AK: $attest\_sig = Sign(PrivAK, report\_data||Measurement)$. After receiving *attest_sig* from the kernel, the invoking enclave sends *attest_sig*, $PubAK$ and AK's certificate $CertAK$ to the verifier. The verifier first validates the attestation key using $CertAK$, and then verifies the *attest_sig* using $PubAK$. After the verification, the verifier ensures that the integrity value of the enclave that communicates with him is *Measurement*.

5.6 Data Sealing/Unsealing

Each enclave can use the sealing/unsealing system calls to bind sensitive data to it and ensure that only enclaves with the same integrity state can use the data. When an enclave, denoted by $\hat{A}$, wants to seal a piece of data *data*, it invokes the system call `syscall_seal`. The kernel first derives a sealing key from the DSK and the integrity value of $\hat{A}$: $K_{\hat{A}} = HKDF(DSK, Measurement_{\hat{A}}, klen)$, where $HKDF$ is an $HMAC$-based key derivation function whose output length is $klen$, and $Measurement_{\hat{A}}$ is the measurement result of $\hat{A}$ stored in the kernel; then the kernel encrypts *data* using $K_{\hat{A}}$: $ciphertext = Enc(K_{\hat{A}}, data)$, and returns *ciphertext* to $\hat{A}$. When the kernel receives the system call `syscall_unseal`, it first derives the sealing key $K_{\hat{A}} = HKDF(DSK, Measurement_{\hat{A}}, klen)$, then uses $K_{\hat{A}}$ to unseal *ciphertext*: $data = Dec(K_{\hat{A}}, ciphertext)$, and finally returns *data* to $\hat{A}$.

The key used for sealing and unsealing operations is derived from the device-unique key DSK and the measurement of the enclave, so other devices and enclaves with different integrity values are unable to derive the sealing key, and thus they are unable to obtain the sealed data.

5.7 Secrets Provisioning

A remote data owner, denoted by $\hat{D}$, can leverage the secrets provisioning mechanism to provision sensitive data to an enclave (denoted by $\hat{E}$) whose integrity state he can validate. The secrets provisioning mechanism can be seen as a combination of the SIGMA authenticated key exchange protocol [57] and the remote attestation mechanism. ① the data owner $\hat{D}$ generates a DH key $A = g^a$ and sends the public key A to the enclave $\hat{E}$; ② the enclave invokes the system call `syscall_provision` which transfers A to the kernel; ③ the kernel generates a DH key $B = g^b$, performs remote attestation on A, B, and the measurement of the enclave *Measurement*: $SigAK = Sign(PrivAK, A||B||Measurement)$, computes a shared secret $sk = g^{ab}$, and returns B, $SigAK$, and sk to the enclave; ④ the enclave derives a session key $k_s = HKDF(sk, \text{"}session\ key\text{"}, klen)$ and a MAC key $k_m = HKDF(sk, \text{"}mac\ key\text{"}, klen)$, computes $MAC_{k_m}(\hat{E})$, and sends B, $\hat{E}$, $SigAK$, $MAC_{k_m}(\hat{E})$, $PubAK$, and $CertAK$ to the data owner; ⑤ the data owner validates the attestation signature $SigAK$, and after the validation, he computes a shared secret $sk = g^{ab}$, derives a session key $k_s = HKDF(sk, \text{"}session\ key\text{"}, klen)$ and a MAC key $k_m = HKDF(sk, \text{"}mac\ key\text{"}, klen)$, and uses k_m to verify $MAC_{k_m}(\hat{E})$; ⑥ the data owner signs A and B with its signature key $PrivDO$: $SigDO = Sign(PrivDO, A||B)$, and sends $\hat{D}$, $SigDO$, its certificate $CertDO$, and $MAC_{k_m}(\hat{D})$ to the enclave; ⑦ the enclave verifies *SigDO* and $MAC_{k_m}(\hat{D})$, and then the data owner and enclave can transfer sensitive data to each other in the secure channel established by k_s.

5.8 Enclave Management

Figure 8 illustrates the life cycle of an enclave and the enclave management functions that trigger transitions of the enclave states. Only three of the enclave management functions — functions used to open, invoke, and close enclaves — expose their interfaces to the normal world. To keep compliant with GP TEE Client API specification [29], the three interfaces are wrapped as `TEEC_OpenSession`, `TEEC_InvokeCommand`, and `TEEC_CloseSession` (interfaces in the rectangles of Figure 8). All the other management functions do not expose interfaces to the normal world.

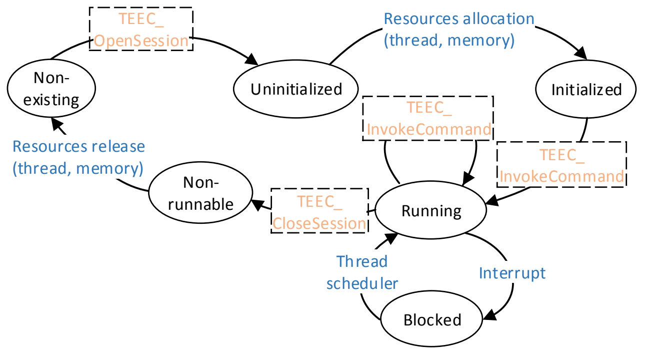

Figure 8: The Life Cycle of an Encalve

Unlike Intel SGX, which exposes all enclave management instructions to host software, SecTEE only exposes three basic management interfaces to host software, and most events that trigger enclave management functions are captured and addressed by the TEE OS directly. This kind of design helps SecTEE to resist memory access based side-channel attacks because it eliminates the abilities to manage enclaves' memory and schedule enclaves from host software, which are two critical ways for malicious hosts to launch software side-channel attacks against enclaves.

5.9 Other Security Considerations

Denial-of-Service attacks. SecTEE designs a preemptive allocation scheme (Section 5.2.1) for page sets. So even if a malicious host application occupies all page sets by loading multiple enclaves, when a legal application tries to load its enclave, SecTEE will free the least recently used page set and designate it to the enclave.

Interrupt based high-resolution channels. Like most TEE systems, SecTEE designates IRQ interrupts to the normal world and FIQ interrupts to the secure world. To prevent attackers from leveraging IRQ interrupts to implement high-resolution channels, such as attacks against SGX [73, 108], SecTEE disables IRQ interrupts when running in the secure world by masking them. An attack enclave may implement high-resolution channels based on the FIQ interrupts, but since all enclaves do not share caches, it can obtain nothing about cache usage of other enclaves.

6 IMPLEMENTATION AND EVALUATION

We implement a prototype based on OP-TEE v2.4.0 and leverage the OP-TEE Pager system to provide an SoC-bound execution environment for enclaves. All the trusted computing features, i.e., enclave identification, remote attestation, data sealing/unsealing, and secrets provisioning, are extended to the OP-TEE kernel, and the system calls listed in Table 1 are provided to enclaves. A Quoting Enclave is implemented to manage attestation keys. We implement two types of attestation keys: RSA-based keys and ECC-based keys, whose lengths are 2048 bits and 256 bits respectively, and implement the sealing/unsealing keys using 256-bit AES keys. Our prototype is built on the NXP i.MX6Q Sabre-SD platform, which has an i.MX 6Quad SoC with 4 ARM Cortex-A9 1.2 GHz CPUs, 16-way 1MB L2 unified cache, 256 KB OCM, and 1 GB DRAM. As the platform does not satisfy the hardware key requirements of SecTEE, we simulate the *DRK* and *MRK* by hard-coding two 2048-bit RSA keys in software and simulate the *DSK* by hard-coding a 256-bit AES key.

We evaluate the TCB size of SecTEE, the performance overhead imposed by SecTEE, and the side-channel defense effectiveness of SecTEE by performing well-known cache attacks on AES. The performance evaluation is performed on four systems: the OP-TEE OS without any trusted computing features, the OP-TEE Pager system without any trusted computing features (denoted by Pager), SecTEE without memory protection (denoted by SecTEE-plain), and SecTEE. Pager is used to evaluate the performance overhead of the trusted computing features to SecTEE, and SecTEE-plain is used to evaluate the performance overhead incurred by the memory protection mechanism to SecTEE.

6.1 TCB Size

Although our prototype is based on OP-TEE, SecTEE's design is not limited to it, and it can be applied to any other TEE OSes. So the large codebase of OP-TEE does not indicate that any SecTEE's implementation has a large TCB. Furthermore, the large TCB and its trustworthiness issues can be solved by the microkernel approach: TEE OS based on formally verified OS has been proposed, such as MicroTEE [52], a TEE OS based on seL4 [56].

To comprehensively evaluate the TCB introduced by SecTEE, we measure the lines of source code of all the components of SecTEE (Section 5) except the enclave management component: memory protection, side-channel resistance, and trusted computing primitives. We also measure the code size of cryptographic primitives required by memory protection and trusted computing primitives, including AES-GCM, SHA256, and RSA. Note that the enclave management, memory protection, and cryptographic primitives are of OP-TEE. We do not measure the enclave management component because it is actually a combination of memory management, process scheduling, and interrupt handling functionalities of an OS. Since these functionalities are common for any TEE OS, we believe that the enclave management component does not bloat the TCB. The memory protection component has ~2000 LOC, the side-channel resistance component has ~200 LOC, the trusted computing primitives have ~1700 LOC, and the cryptographic primitives (we only measure the primitives required by SecTEE) have ~3500 LOC. So SecTEE adds ~7400 LOC to the TCB. Note that the cryptographic primitives take up about half of the code, and their trustworthiness can be improved by leveraging formally verified cryptographic libraries, such as EverCrypt [83], HACL* [124], Vale [7], and Vigilant's CRT-RSA [86].

In conclusion, the whole TCB of SecTEE can be decreased by leveraging the microkernel approach; SecTEE's design increases the TCB in an acceptable magnitude, and the trustworthiness of the added TCB can be improved by adopting formally verified cryptographic libraries.

6.2 Overhead of Trusted Computing Features

When host software invokes `TEEC_OpenSession`, SecTEE authenticates the identity of the required enclave and measures its integrity, so the execution time of `TEEC_OpenSession` represents the performance overhead of enclave identification and measurement. The overhead of other trusted computing features can be evaluated by measuring the execution time of the corresponding system calls (Table 1). For the system calls performed by attestation keys (`syscall_request_AK`, `syscall_remote_attestation`, and `syscall_provisioning`), we measure their performance when the types of attestation keys are RSA and ECC respectively. The evaluation results (Table 2) show that most system calls take acceptable time except `syscall_request_AK` and `syscall_provisioning`, and that the performance of `syscall_request_AK` is greatly improved when the attestation key is the type of ECC.

Table 2: Performance of Trusted Computing Features (millisecond)

	RSA-based AK	ECC-based AK
TEEC_OpenSession	90.73	–
syscall_request_AK	23254	744.37
syscall_seal_AK	1.40	–
syscall_import_AK	10.47	–
syscall_remote_attestation	196.61	507.69
syscall_seal	0.90	–
syscall_unseal	0.90	–
syscall_provisioning	1186	1508.89
World Switch	0.08	–

6.3 Xtest Performance Evaluation

Xtest [67] is a test framework designed by Linaro for OP-TEE. It contains two kinds of performance benchmarks: the trusted storage benchmark and the crypto benchmark. It also contains comprehensive tests of features of OP-TEE, including OS related tests, socket related tests, crypto related tests, shared memory tests, storage tests, GP shared memory tests, key derivation & management tests, and secure storage tests. We perform the benchmarks and all feature tests for the four systems: OP-TEE, Pager, SecTEE-plain, and SecTEE. We run each test 100 times and compute the geometric mean of the results.

6.3.1 *Xtest Benchmarks.* Figures 9 and 10 illustrate the results of trusted storage and crypto benchmarks for the four systems. The results are calculated by inputting data of different sizes to secure services. Pager and SecTEE achieve similar performance, which shows that the extended trusted computing features introduce little overhead. For the trusted storage benchmark, SecTEE-plain is 1.2 times slower than OP-TEE, and SecTEE is 2.2 times slower than OP-TEE. For the crypto benchmark, SecTEE-plain is 11.8 times slower than OP-TEE, and SecTEE is 53.5 times slower than OP-TEE. The reason that SecTEE's performance impact on trusted storage operations is less than on crypto operations is that trusted storage operations need to invoke the file system service of the normal world to store data, which takes a large part of the whole execution time but is not affected by SecTEE.

6.3.2 *Xtest Tests of OP-TEE's Features.* Figures 11 illustrates the results of Xtest tests of OP-TEE's features for the four systems. SecTEE is 3.9 times slower than OP-TEE on average, and SecTEE-plain and OP-TEE achieve similar performance on average (SecTEE-plain is 1.06 times slower than OP-TEE). The results demonstrate that most performance overhead is caused by the memory protection mechanism.

6.4 Enclave Performance Evaluation

To evaluate SecTEE's performance impact on enclaves, we build the following three security enclaves and run them on the four systems.

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

We evaluate both the entire execution time and the service runtime of enclaves (Figure 12). The entire execution time includes all the time of loading an enclave into SecTEE, allocating resources for it, executing enclave services, and destroying the enclave. The service runtime only includes the time of running enclave services (TA commands).

For the entire execution time of secure enclaves, SecTEE-plain is 4.4 times slower than OP-TEE on average, and SecTEE is 43.7 times slower than OP-TEE on average. So the results demonstrate that most performance overhead comes from the memory protection mechanism. As SecTEE is 1.12 times slower than Pager, the trusted computing features (not including memory protection) of SecTEE only introduce 12% overhead on average.

6.5 Side-channel Defense Evaluation

To demonstrate the effectiveness of SecTEE's protection against memory access based side-channel attacks, we perform the public cache attacks [77] (based on the libflush library of ARMageddon [68]) against the OpenSSL AES implementation on OP-TEE and SecTEE respectively.

6.5.1 *Experiment Preparation.* We port the libflush library and attack tools to SecTEE. We extract the source code of the T-table based AES implementation of OpenSSL, and port it to SecTEE as an AES static TA. We leverage the *cycle count register* (PMCCNTR) of the performance monitoring unit (PMU) to measure the time of memory access. The cache hit and miss histograms (Figure 13) of normal DRAM (N-DRAM), secure DRAM (S-DRAM), and secure OCM (S-OCM) show that cache hits and cache misses are clearly distinguishable in both the normal world and the secure world (the

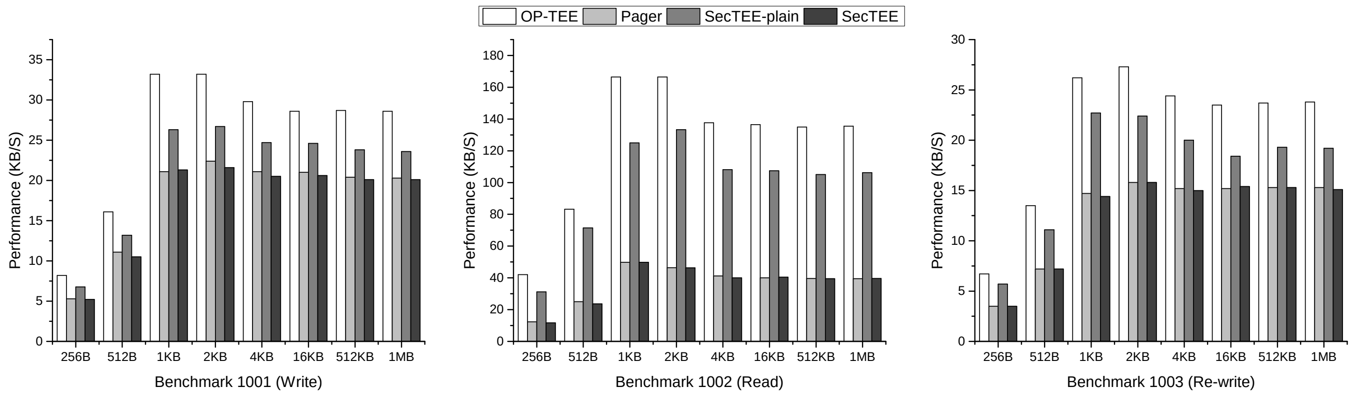

Figure 9: Xtest Trusted Storage Benchmarks

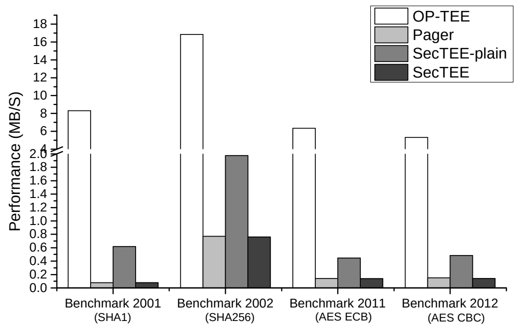

Figure 10: Xtest Crypto Benchmarks

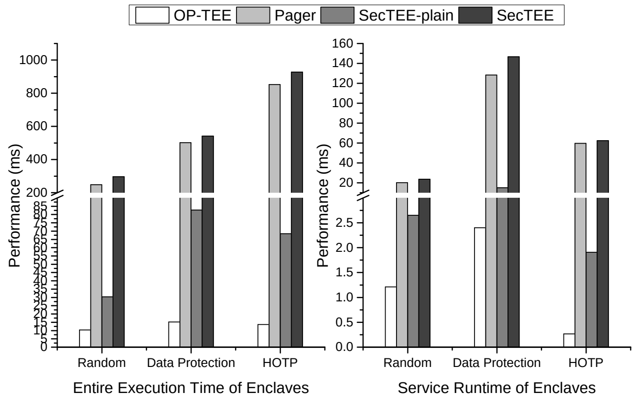

Figure 12: Performance of Security Enclaves

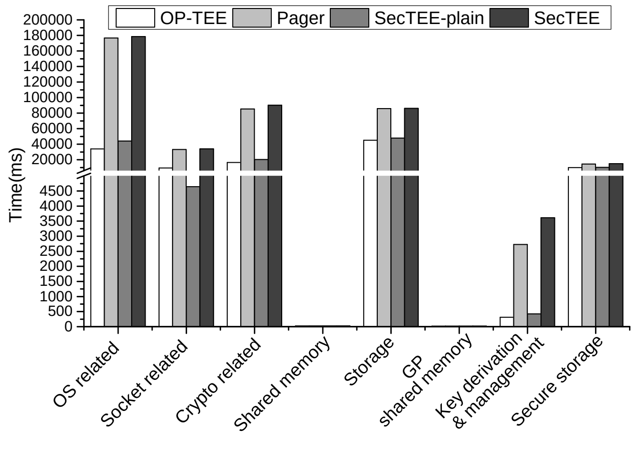

Figure 11: Xtest Test of OP-TEE's Features

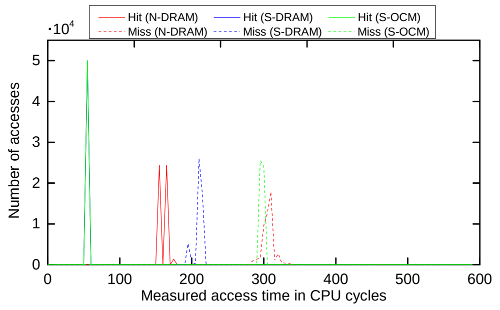

Figure 13: Histograms of cache hits and cache misses in the normal world and the secure world (the X-Axis represents the latency of a memory access in CPU cycles)

cache hit histogram of the secure DRAM overlaps that of the secure OCM). The results illustrate an interesting phenomenon: cache hits of the secure memory (about 55 CPU cycles) take much less time than cache hits of the normal memory (about 160 CPU cycles).

6.5.2 Evaluating Attacks from the Normal World. We first perform the cache attacks against AES, and find that the methods provided by the libflush library, such as *Prime+Probe* and *Eviction+Probe*, do not provide high enough resolution to recover the AES key in the secure world. So we perform *Prime+Probe* to check whether the normal world can detect memory accesses in the secure world, i.e., prime before switching to the secure world and probe after switching back to the normal world. Figure 14 shows the

Prime+Probe histograms for cache hits and cache misses. On OP-TEE, we observe a higher execution time if the secure world accesses a congruent memory address (addresses that map to the same cache set are considered congruent), while on SecTEE, the execution time is the same no matter whether the secure world accesses a congruent memory address or not. So SecTEE prevents attackers in the normal world from learning the access patterns of the secure world.

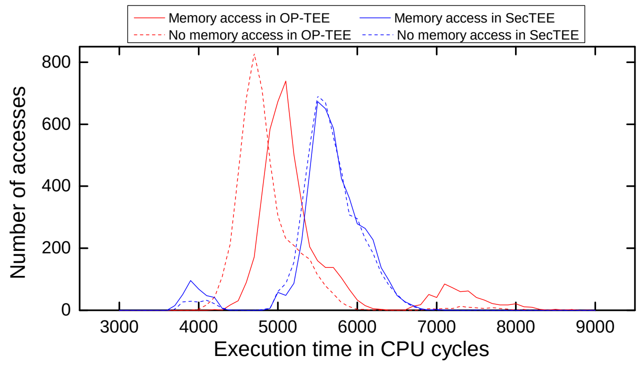

Figure 14: Histograms of *Prime+Probe* timings on OP-TEE and SecTEE (the X-Axis represents the execution time of the *Probe* operation of libflush)

6.5.3 Evaluating Attacks from the Secure World. We implement the T-table based implementation of AES as a victim static TA, and use the attack tools [77] to implement an attacker static TA which can invoke the AES TA. Since the cache attack requires the victim and the attacker to share AES T-tables, we put the T-tables on a shared memory page. Our experiment shows that the attacker TA can recover the 128-bit AES key after 256,000 encryptions on OP-TEE, while it cannot recover any byte of the key on SecTEE. Figure 15 illustrates the candidate scores of the first byte of the last round key on OP-TEE and SecTEE respectively: the correct value of the first byte of the last round key is 0x98=152. The attacker TA correctly guesses the last round key of AES on OP-TEE, while it cannot guess the key on SecTEE. Especially, Figure 15 shows that the attacker TA learns no information about the memory access patterns of the AES TA: all candidates get almost the same score (1 or 0).

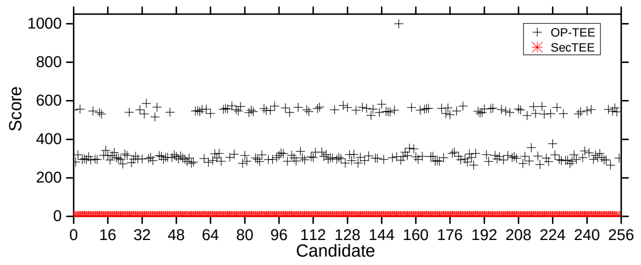

Figure 15: Candidate Scores of the 1st byte of the last round key on OP-TEE and SecTEE

7 RELATED WORK

7.1 Security Applications of TrustZone

This section introduces the security applications of ARM TrustZone, including TEE virtualization, mobile OS protection and monitoring, and security services for mobile devices.

TEE Virtualization. ARM TrustZone itself does not support virtualization, preventing its application on server markets. vTZ [45] addresses this problem by creating secure VMs as guest TEEs for guest VMs and leveraging TrustZone and the hypervisor to enforce strong isolation between the guest TEEs. TEEv [64] designs a TEE virtualization architecture for ARM TrustZone, which supports multiple TEE OSes running concurrently in the secure world.

Mobile OS Protection and Monitoring. TZ-RKP [5] provides real-time protection of the mobile OS by removing critical system control instructions from the mobile OS kernel and simulating these instructions in TEE. TruZ-Droid [118] incorporates the generic TrustZone support in Android so that allows Android applications leveraging TrustZone to protect users' secrets and interaction information without installing app-specific TAs. Sprobes [28] presents an introspection mechanism in the TEE to detect mobile OS kernel rootkits. TrustShadow [36] and CryptMe [11] protect mobile applications from physical attacks using a lightweight runtime system in the TEE. TrustDump [102] develops a memory acquisition mechanism in the TEE to perform memory dump and malware analysis of the mobile OS. TrustICE [103] enables execution of security-sensitive code in isolated environments in the normal world without increasing the TCB of TEE.

Security Services for Mobile Devices. TLR [89] provides a small runtime engine interpreting .NET managed code in the TEE and enables mobile applications to implement security use cases using high-level languages like C#. AdAttester [62] is a verifiable mobile advertisement framework which guarantees that the advertisement is displayed intact and timely. TrustOTP [101] proposes a secure one-time password solution achieving both the flexibility of software-based solutions and the security of hardware-based solutions. fTPM [85] presents the design and implementation of a TPM 2.0 chip in TEE, and it has been used in millions of mobile devices. VButton [63] verifies the authenticity of sensitive user operations to prevent malicious mobile apps and rootkits from forging legitimate user inputs.

7.2 Secure Enclave Architectures

Secure enclave architectures have been an active field of research over the past ten years, most of them are implemented by extending security mechanisms to microprocessors, and some of them require a small security kernel. AEGIS [100] provides secure execution environments and fundamental trusted computing features for security-sensitive code. OASIS [81] offers an isolated execution environment with basic trusted computing features (attestation and data binding) on minimally modified commodity CPUs. One disadvantage of OASIS is that its execution environments are limited by the cache size. Bastion [12] is a hardware-software security architecture for security-critical tasks. It extends a memory authentication mechanism to the microprocessor to provide basic protection against both software and physical attacks. Intel SGX is the most popular secure enclave architecture. Although it achieves high levels of software and physical security, it is vulnerable to a variety of software side-channel attacks [9, 14, 32, 41, 73, 110, 114]. Iso-X [20] is a hardware-software co-design that provides isolated compartments with the remote attestation mechanism.

Flicker [72] is a trusted computing architecture based on the Late Launch technology [1, 48] and does not need to modify the CPU. The TrustVisor [71] system overcomes Flicker's performance disadvantage by using a software-based "micro-TPM", executing at high speed on the platform's primary CPU, to provide basic trusted computing primitives. Sancus [79] presents a trusted computing architecture for low-end systems, which provides rich trusted computing primitives with minimal (hardware) TCB, such as remote attestation, strong integrity, and authenticity guarantees.

Sanctum [17], Keystone [60], Komodo [22], and Sanctuary [8] are modern secure enclave architectures proposed recently, which aim to provide the same or higher security features as Intel SGX. Sanctum [17] offers the same software security and trusted computing features as SGX for the RISC-V CPU architecture. It does not prevent physical attacks, but adds protection against software side-channel attacks. After Sanctum, the Keystone project [60] is proposed, whose goal is to make an open end-to-end framework for secure enclaves on the RISC-V architecture. Komodo [22] presents an approach to secure enclave architecture in formally verified software. Based on TrustZone, Sanctuary [8] provides SGX-like user-space enclaves without requiring any hardware modifications and does not increase the TCB of the TEE system, but it does not consider recently raised threats such as physical attacks and memory access based side-channel attacks.

7.3 The Page Coloring Technique

Jin et al. [53] present a two-dimension page coloring mechanism which can improve both on-chip miss rate and cache access latency. Tam et al. [104] implement a software page coloring mechanism in the OS which allows for partitioning of the shared L2 cache. Shi et al. [92] protect crypto keys in virtualized clouds by proposing a dynamic cache coloring mechanism. StealthMem [55] presents a system-level protection mechanism against cache side channel attacks in the cloud. Godfrey et al [31] design and implement two defenses against the sequential and parallel types of cache-based side-channel attacks respectively for cloud systems. COLORIS [117] implements an efficient page re-coloring framework in production systems, such as Linux.

8 CONCLUSION

In this paper, we present a software-based secure enclave architecture, SecTEE, for the ARM architecture. SecTEE leverages the TrustZone isolation mechanism and the SoC-bound execution environment technology to provide protection against software and physical attacks and offers necessary trusted computing features required by secure enclaves. SecTEE also provides protection against memory access based side-channel attacks by modifying the kernel's memory management service to avoid cache conflicts of enclaves and locking the working enclave pages into the cache, showing that the design of moving the enclave management functionality such as memory management and enclave scheduling to a dedicated secure OS is an efficient way to resist memory access based side-channel attacks. We implement a prototype system on a TrustZone-enabled platform, but SecTEE can be applied to other CPU architectures with isolation mechanisms, such as the RISC-V architecture with TEE extensions. Although our prototype is based on OP-TEE, SecTEE is not limited to it and can be directly applied to other TEE systems by re-implementing the SoC-bound execution environment and applying the mechanisms of resistance to side-channel attacks and trusted computing primitives to them. Our evaluation results show that the trusted computing features of SecTEE introduce acceptable performance overhead on the runtime of security-critical code and most performance overhead comes from the memory protection mechanism.

ACKNOWLEDGEMENT

We would like to thank our anonymous reviewers for their valuable comments. This research was supported by the National Natural Science Foundation of China (61802375, 61602325, 61872343, 61602455, U1636216), the National Key R & D Program of China (2018YFB0904900, 2018YFB0904903).

REFERENCES

[1] AMD64 Virtualization. Secure Virtual Machine Architecture Reference Manual. *AMD Publication*, 33047, 2005.

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

[3] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. Stillwell, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI'16*, pages 689–703. USENIX Association, 2016.

[4] P.-L. Aublin, F. Kelbert, D. O'Keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. Eyers, and P. Pietzuch. TaLoS: Secure and Transparent TLS Termination inside SGX Enclaves. *Imperial College London, Tech. Rep*, 5, 2017.

[5] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *21st ACM SIGSAC Conference on Computer and Communications Security, CCS'14*, pages 90–102. ACM, 2014.

[6] R. Boivie and P. Williams. SecureBlue++: CPU Support for Secure Execution. *Technical report*, 2012.

[7] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson. Vale: Verifying High-Performance Cryptographic Assembly Code. In *26th USENIX Security Symposium, USENIX Security 17*, pages 917–934. USENIX Association, 2017.

[8] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf. SANCTUARY: ARMing TrustZone with User-space Enclaves. In *26th Network and Distributed System Security Symposium, NDSS 2019*, 2019.

[9] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies*. USENIX Association, 2017.

[10] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *17th International Middleware Conference*, pages 14:1–14:13. ACM, 2016.

[11] C. Cao, L. Guan, N. Zhang, N. Gao, J. Lin, B. Luo, P. Liu, J. Xiang, and W. Lou. CryptMe: Data Leakage Prevention for Unmodified Programs on ARM Devices. In *International Symposium on Research in Attacks, Intrusions, and Defenses, RAID 2018*, pages 380–400. Springer, 2018.

[12] D. Champagne and R. B. Lee. Scalable Architectural Support for Trusted Software. In *16th IEEE International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[13] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. Lai. SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution. In *4th IEEE European Symposium on Security and Privacy*, pages 142–157. IEEE, 2019.

[14] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjá Vu. In *12th ACM on Asia Conference on Computer and Communications Security*, pages 7–18. ACM, 2017.

[15] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. De Lara, H. Raj, S. Saroiu, and A. Wolman. Protecting Data on Smartphones and Tablets from Memory Attacks.

In *20th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'15*, pages 177–189. ACM, 2015.
[16] V. Costan and S. Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive, 2016/086*, 2016.
[17] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium, USENIX Security 16*, pages 857–874. USENIX Association, 2016.
[18] Y. Ding, R. Duan, L. Li, Y. Cheng, Y. Zhang, T. Chen, T. Wei, and H. Wang. POSTER: Rust SGX SDK: Towards Memory Safety in Intel SGX Enclave. In *24th ACM SIGSAC Conference on Computer and Communications Security, CCS'17*, pages 2491–2493. ACM, 2017.
[19] L. Duflot, Y.-A. Perez, G. Valadon, and O. Levillain. Can You Still Trust Your Network Card. *CanSecWest/core10*, pages 24–26, 2010.
[20] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley. Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-47*, pages 190–202. IEEE Computer Society, 2014.
[21] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *23rd International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'18*, pages 693–707. ACM, 2018.
[22] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *26th Symposium on Operating Systems Principles, SOSP'17*, pages 287–305. ACM, 2017.
[23] B. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov. IRON: Functional Encryption using Intel SGX. In *24th ACM SIGSAC Conference on Computer and Communications Security, CCS'17*, pages 765–782. ACM, 2017.
[24] Y. Fu, E. Bauman, R. Quinonez, and Z. Lin. SGX-LAPD: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults. In *20th International Symposium on Research in Attacks, Intrusions, and Defenses, RAID 2017*, pages 357–380. Springer, 2017.
[25] FuturePlus System. DDR2 800 Bus Analysis Probe. http://www.futureplus.com/download/datasheet/fs2334_ds.pdf, 2006.
[26] B. Garmany and T. Müller. PRIME: Private RSA Infrastructure for Memory-less Encryption. In *29th Annual Computer Security Applications Conference, ACSAC'13*, pages 149–158. ACM, 2013.
[27] Q. Ge, Y. Yarom, and G. Heiser. No Security Without Time Protection: We Need a New Hardware-Software Contract. In *9th Asia-Pacific Workshop on Systems*, pages 1:1–1:9. ACM, 2018.
[28] X. Ge, H. Vijayakumar, and T. Jaeger. Sprobes: Enforcing Kernel Code Integrity on the TrustZone Architecture. *arXiv preprint arXiv:1410.7747*, 2014.
[29] Global Platform Device Technology. TEE client API specification version 1.0. http://globalplatform.org, 2010.
[30] GlobalPlatform. GlobalPlatform Device Technology: TEE System Architecture. Technical report, GPD_SPE_009, 2017.
[31] M. M. Godfrey and M. Zulkernine. Preventing Cache-Based Side-Channel Attacks in a Cloud Environment. *IEEE Transactions on Cloud Computing*, 2(4):395–408, 2014.
[32] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache Attacks on Intel SGX. In *10th European Workshop on Systems Security*, pages 2:1–2:6. ACM, 2017.
[33] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *27th USENIX Security Symposium, USENIX Security 18*, pages 955–972. USENIX Association, 2018.
[34] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+ Flush: A Fast and Stealthy Cache Attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.
[35] D. Gruss, R. Spreitzer, and S. Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *24th USENIX Security Symposium, USENIX Security 15*, pages 897–912. USENIX Association, 2015.
[36] L. Guan, C. Cao, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. 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*, 16(3):438–453, 2018.
[37] L. Guan, J. Lin, B. Luo, and J. Jing. Copker: Computing with Private Keys without RAM. In *21st Network and Distributed System Security Symposium, NDSS 2014*, pages 23–26, 2014.
[38] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang. Protecting Private Keys against Memory Disclosure Attacks using Hardware Transactional Memory. In *36th IEEE Symposium on Security and Privacy, S&P 2015*, pages 3–19. IEEE, 2015.
[39] S. Gueron. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptology ePrint Archive, 2016/204*, 2016.
[40] D. Gullasch, E. Bangerter, and S. Krenn. Cache games–Bringing access-based cache attacks on AES to practice. In *30th IEEE Symposium on Security and Privacy, S&P 2011*, pages 490–505. IEEE, 2011.
[41] M. Hähnel, W. Cui, and M. Peinado. High Resolution Side Channels for Untrusted Operating Systems. In *2017 USENIX Annual Technical Conference, USENIX ATC 17*, pages 299–312. USENIX Association, 2017.
[42] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. *Communications of the ACM*, 52(5):91–98, 2009.
[43] M. Henson and S. Taylor. Beyond Full Disk Encryption: Protection on Security-Enhanced Commodity Processors. In *11th International Conference on Applied Cryptography and Network Security*, pages 307–321. Springer, 2013.
[44] G. Hotz. PS3 Glitch Hack. http://www.eurasia.nu/wiki/index.php/PS3_Glitch_Hack, 2010.
[45] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vTZ: Virtualizing ARM TrustZone. In *26th USENIX Security Symposium, USENIX Security 17*, pages 541–556. USENIX Association, 2017.
[46] A. Huang. Keeping Secrets in Hardware: The Microsoft Xbox^{TM} Case Study. In *4th International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2002*, pages 213–227. Springer, 2002.
[47] M. S. Inci, B. Gülmezoglu, G. I. Apecechea, T. Eisenbarth, and B. Sunar. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. *IACR Cryptology ePrint Archive, 2015/898*, 2015.
[48] Intel Corporation. LaGrande Technology Preliminary Architecture Specification. *Document No. 315168 002*, 2006.
[49] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES. In *36th IEEE Symposium on Security and Privacy, S&P 2015*, pages 591–604. IEEE, 2015.
[50] G. Irazoqui, T. Eisenbarth, and B. Sunar. Cross Processor Cache Attacks. In *11th ACM on Asia conference on computer and communications security*, pages 353–364. ACM, 2016.
[51] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Wait a Minute! A fast, Cross-VM Attack on AES. In *17th International Workshop on Recent Advances in Intrusion Detection, RAID 2014*, pages 299–319. Springer, 2014.
[52] D. Ji, Q. Zhang, S. Zhao, Z. Shi, and Y. Guan. MicroTEE: Designing TEE OS Based on the Microkernel Architecture. In *18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2019*. IEEE, 2019.
[53] L. Jin and S. Cho. Better than the Two: Exceeding Private and Shared Caches via Two-Dimensional Page Coloring. In *1st Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2007.
[54] V. Karande, E. Bauman, Z. Lin, and L. Khan. SGX-Log: Securing System Logs With SGX. In *12th ACM on Asia Conference on Computer and Communications Security*, pages 19–30. ACM, 2017.
[55] T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *21st USENIX Security Symposium, USENIX Security 12*, pages 189–204. USENIX Association, 2012.
[56] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *22nd ACM Symposium on Operating Systems Principles, SOSP'09*, pages 207–220. ACM, 2009.
[57] H. Krawczyk. SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and its Use in the IKE Protocols. In *23rd Annual International Cryptology Conference, CRYPTO 2003*, pages 400–425. Springer, 2003.
[58] K. A. Küçük, A. Paverd, A. Martin, N. Asokan, A. Simpson, and R. Ankele. Exploring the Use of Intel SGX for Secure Many-Party Applications. In *the 1st Workshop on System Software for Trusted Execution*, pages 5:1–5:6. ACM, 2016.
[59] M. G. Kuhn. Cipher Instruction Search Attack on the Bus-encryption Security Microcontroller DS5002FP. *IEEE Transactions on Computers*, 47(10):1153–1157, 1998.
[60] D. Lee, D. Kohlbrenner, K. Cheang, C. Rasmussen, K. Laeufer, I. Fang, A. Khosla, C.-C. Tsai, S. Seshia, D. Song, and K. Asanovic. Keystone Enclave: An Open-Source Secure Enclave for RISC-V. https://keystone-enclave.org/, 2018.
[61] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *26th USENIX Security Symposium, USENIX Security 17*, pages 557–574. USENIX Association, 2017.
[62] W. Li, H. Li, H. Chen, and Y. Xia. AdAttester: Secure Online Mobile Advertisement Attestation Using TrustZone. In *13th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'15*, pages 75–88. ACM, 2015.
[63] W. Li, S. Luo, Z. Sun, Y. Xia, L. Lu, H. Chen, B. Zang, and H. Guan. VButton: Practical Attestation of User-driven Operations in Mobile Apps. In *16th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'18*, pages 28–40. ACM, 2018.
[64] W. Li, Y. Xia, L. Lu, H. Chen, and B. Zang. TEEv: Virtualizing Trusted Execution Environments on Mobile Platforms. In *15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE'19*, pages 2–16. ACM, 2019.
[65] Linaro. OP-TEE: Open Portable Trusted Execution Environment. https://www.op-tee.org, 2014.

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

[67] Linaro. OP-TEE Xtest Framework. https://github.com/OP-TEE/optee_test, 2016.

[68] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium, USENIX Security 16*, pages 549–564. USENIX Association, 2016.

[69] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *36th IEEE Symposium on Security and Privacy, S&P 2015*, pages 605–622. IEEE, 2015.

[70] P. Maene, J. Götzfried, R. De Clercq, T. Müller, F. Freiling, and I. Verbauwhede. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. *IEEE Transactions on Computers*, 67(3):361–374, 2018.

[71] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *31th IEEE Symposium on Security and Privacy, S&P 2010*, pages 143–158. IEEE, 2010.

[72] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 315–328. ACM, 2008.

[73] A. Moghimi, G. Irazoqui, and T. Eisenbarth. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *19th International Conference on Cryptographic Hardware and Embedded Systems, CHES 2017*, pages 69–90. Springer, 2017.

[74] T. Müller, A. Dewald, and F. C. Freiling. AESSE: A Cold-boot Resistant Implementation of AES. In *3rd European Workshop on System Security*, pages 42–47. ACM, 2010.

[75] T. Müller, F. C. Freiling, and A. Dewald. TRESOR Runs Encryption Securely Outside RAM. In *20th USENIX Security Symposium, USENIX Security 11*, volume 17. USENIX Association, 2011.

[76] T. Müller and M. Spreitzenbarth. Frost: Forensic Recovery of Scrambled Telephones. In *12th International Conference on Applied Cryptography and Network Security*, pages 373–388. Springer, 2013.

[77] E. Nascimento. Cache Side-channel Attack AES. https://github.com/enascimento/cache_side-channel_attack_aes, 2017.

[78] 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.

[79] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base. In *22th USENIX Security Symposium, USENIX Security 13*, pages 479–498. USENIX Association, 2013.

[80] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: the Case of AES. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[81] E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vasudevan. OASIS: On Achieving a Sanctuary for Integrity and Secrecy on Untrusted Platforms. In *20th ACM SIGSAC Conference on Computer and Communications Security, CCS'13*, pages 13–24. ACM, 2013.

[82] P. Papadopoulos, G. Vasiliadis, G. Christou, E. Markatos, and S. Ioannidis. No Sugar but All the Taste! Memory Encryption Without Architectural Support. In *22nd European Symposium on Research in Computer Security, ESORICS 2017*, pages 362–380. Springer, 2017.

[83] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, T. Ramananandro, A. Rastogi, N. Swamy, C. Wintersteiger, and S. Zanella-Beguelin. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. Technical report, IACR Cryptology ePrint Archive, 2019/757, 2019.

[84] H. Raj, R. Nathuji, A. Singh, and P. England. Resource Management for Isolation Enhanced Cloud Services. In *1st ACM workshop on Cloud computing security*, pages 77–84. ACM, 2009.

[85] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Löser, D. Mattoon, M. Nyström, D. Robinson, R. Spiger, S. Thom, and D. Wooten. fTPM: A Software-Only Implementation of a TPM Chip. In *25th USENIX Security Symposium, USENIX Security 16*, pages 841–856. USENIX Association, 2016.

[86] P. Rauzy and S. Guilley. A Formal Proof of Countermeasures against Fault Injection Attacks on CRT-RSA. *Journal of Cryptographic Engineering*, 4(3):173–185, 2014.

[87] 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.

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

[89] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM TrustZone to Build a Trusted Language Runtime for Mobile Applications. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'14*, pages 67–80. ACM, 2014.

[90] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *36th IEEE Symposium on Security and Privacy, S&P 2015*, pages 38–54. IEEE, 2015.

[91] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2017.

[92] J. Shi, X. Song, H. Chen, and B. Zang. Limiting Cache-based Side-channel in Multi-tenant Cloud using Dynamic Page Coloring. In *IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops, DSN-W 2011*, pages 194–199. IEEE, 2011.

[93] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska. S-NFV: Securing NFV States by Using SGX. In *2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pages 45–48. ACM, 2016.

[94] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing Page Faults from Telling Your Secrets. In *11th ACM on Asia Conference on Computer and Communications Security*, pages 317–328. ACM, 2016.

[95] S. Shinde, D. Le Tien, S. Tople, and P. Saxena. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *24th Network and Distributed System Security Symposium, NDSS 2017*, 2017.

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

[97] Solutions EPN. Analysis Tools for DDR1, DDR2, DDR3, Embedded DDR and Fully Buffered DIMM Modules, 2014.

[98] R. Spreitzer and T. Plos. Cache-Access Pattern Attack on Disaligned AES T-Table. In *4th International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 200–214. Springer, 2013.

[99] R. Spreitzer and T. Plos. On the Applicability of Time-Driven Cache Attacks on Mobile Devices. In *7th International Conference on Network and System Security*, pages 656–662. Springer, 2013.

[100] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 357–368. ACM, 2014.

[101] H. Sun, K. Sun, Y. Wang, and J. Jing. TrustOTP: Transforming Smartphones into Secure One-Time Password Tokens. In *22nd ACM SIGSAC Conference on Computer and Communications Security, CCS'15*, pages 976–988. ACM, 2015.

[102] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia. TrustDump: Reliable Memory Acquisition on Smartphones. In *19th European Symposium on Research in Computer Security, ESORICS 2014*, pages 202–218. Springer, 2014.

[103] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang. TrustICE: Hardware-Assisted Isolated Computing Environments on Mobile Devices. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2015*, pages 367–378. IEEE, 2015.

[104] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing Shared L2 Caches on Multicore Systems in Software. In *2nd Workshop on the Interaction between Operating Systems and Computer Architecture*, pages 26–33, 2007.

[105] A. Triulzi. The Jedi Packet Trick Takes over the Deathstar. *Central Area Networking and Security , CANSEC 2010*, 2010.

[106] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference, USENIX ATC 17*, pages 645–658. USENIX Association, 2017.

[107] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium, USENIX Security 18*, pages 991–1008. USENIX Association, 2018.

[108] J. Van Bulck, F. Piessens, and R. Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *2nd Workshop on System Software for Trusted Execution*, pages 4:1–4:6. ACM, 2017.

[109] G. Vasiliadis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Pixelvault: Using GPUs for Securing Cryptographic Operations. In *21st ACM SIGSAC Conference on Computer and Communications Security, CCS'14*, pages 1131–1142. ACM, 2014.

[110] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *24th ACM SIGSAC Conference on Computer and Communications Security, CCS'17*, pages 2421–2434. ACM, 2017.

[111] S. Weiser and M. Werner. SGXIO: Generic Trusted I/O Path for Intel SGX. In *7th ACM Conference on Data and Application Security and Privacy*, pages 261–268. ACM, 2017.

[112] M. Weiß, B. Heinz, and F. Stumpf. A Cache Timing Attack on AES in Virtualization Environments. In *16th International Conference on Financial Cryptography and Data Security*, pages 314–328. Springer, 2012.

[113] M. Weiß, B. Weggenmann, M. August, and G. Sigl. On Cache Timing Attacks Considering Multi-core Aspects in Virtualized Embedded Systems. In *6th International Conference on Trusted Systems*, pages 151–167. Springer, 2014.

[114] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *36th IEEE Symposium on Security*

and Privacy, S&P 2015, pages 640–656. IEEE, 2015.

[115] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas. Attack Directories, Not Caches: Side-Channel Attacks in a Non-Inclusive World. In *40th IEEE Symposium on Security and Privacy, S&P 2019*. IEEE, 2019.

[116] Y. Yarom and K. Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *23rd USENIX Security Symposium, USENIX Security 14*, pages 719–732. USENIX Association, 2014.

[117] Y. Ye, R. West, Z. Cheng, and Y. Li. COLORIS: A Dynamic Cache Partitioning System using Page Coloring. In *23rd International Conference on Parallel Architecture and Compilation Techniques, PACT'14*, pages 381–392. IEEE, 2014.

[118] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du. TruZ-Droid: Integrating TrustZone with Mobile Operating System. In *16th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'18*, pages 14–27. ACM, 2018.

[119] M. Zhang, Q. Zhang, S. Zhao, Z. Shi, and Y. Guan. SoftME: A Software-Based Memory Protection Approach for TEE System to Resist Physical Attacks. *Security and Communication Networks*, 2019, 2019.

[120] N. Zhang, H. Sun, K. Sun, W. Lou, and Y. T. Hou. CacheKit: Evading Memory Introspection Using Cache Incoherence. In *1st IEEE European Symposium on Security and Privacy, EuroS&P 2016*, pages 337–352. IEEE, 2016.

[121] N. Zhang, K. Sun, W. Lou, and Y. T. Hou. Case: Cache-Assisted Secure Execution on ARM Processors. In *37th IEEE Symposium on Security and Privacy, S&P 2016*, pages 72–90. IEEE, 2016.

[122] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou. TruSense: Information Leakage from TrustZone. In *IEEE Conference on Computer Communications, IEEE INFOCOM 2018*, pages 1097–1105. IEEE, 2018.

[123] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng. Minimal Kernel: An Operating System Architecture for TEE to Resist Board Level Physical Attacks. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019*, pages 105–120. USENIX Association, 2019.

[124] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A Verified Modern Cryptographic Library. In *24th ACM SIGSAC Conference on Computer and Communications Security, CCS'17*, pages 1789–1806. ACM, 2017.