Building a Trustworthy Execution Environment to Defeat Exploits from both Cyber Space and Physical Space for ARM

Le Guan, Chen Cao, Peng Liu, *Member, IEEE,* Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, *Member, IEEE,* and Trent Jaeger, *Member, IEEE*

Abstract—The rapid evolution of Internet-of-Things (IoT) technologies has led to an emerging need to make them smarter. However, the smartness comes at the cost of multi-vector security exploits. From cyber space, a compromised operating system could access all the data in a cloud-aware IoT device. From physical space, cold-boot attacks and DMA attacks impose a great threat to the unattended devices.
In this paper, we propose `TrustShadow` that provides a comprehensively protected execution environment for unmodified application running on ARM-based IoT devices. To defeat cyber attacks, `TrustShadow` takes advantage of ARM TrustZone technology and partitions resources into the secure and normal worlds. In the secure world, `TrustShadow` constructs a trusted execution environment for security-critical applications. This trusted environment is maintained by a lightweight runtime system. The runtime system does not provide system services itself. Rather, it forwards them to the untrusted normal-world OS, and verifies the returns. The runtime system further employs a page based encryption mechanism to ensure that all the data segments of a security-critical application appear in ciphertext in DRAM chip. When an encrypted data page is accessed, it is transparently decrypted to a page in the internal RAM, which is immune to physical exploits.

Index Terms—Malicious Operating Systems, ARM TrustZone, TEE, IoT, Physical Attack, Cold-boot Attack

1 Introduction

THE emerging Internet of Things (IoT) technologies have enabled more and more isolated "things" to collect, process, analyze, and exchange data. One trend in IoT evolution is that these connected devices are becoming smarter and smarter. To quickly prototype their feature-rich products, manufacturers seek ways to run product-ready applications without re-engineering efforts. As a result, they tend to build their products atop ARM-based multi-programming platforms, in which multiple programs run simultaneously on commodity Operating Systems (OSes) such as Linux. For example, smart gateways, embedded servers, and even automotive in-vehicle infotainment etc. are now empowered with more computation capability to run Linux. Except for performing their designed intrinsic functions, a lot of unnecessary functions that come with Linux are added to the code base of these systems.

- *L. Guan, C. Cao, P. Liu, X. Xing, and T. Jaeger are with The Pennsylvania State University, State College, PA 16801, USA.*
  E-mail: {lug14, cuc96, pliu, xxing}@ist.psu.edu, tjaeger@cse.psu.edu
- *X. Ge is with Microsoft Research, Redmond, WA 98052, USA.*
  E-mail: xing@microsoft.com
- *S. Zhang is with Florida Institute of Technology, Melbourne, FL 32901, USA. E-mail: zhangs@cs.fit.edu*
- *M. Yu is with The University of Texas at San Antonio, San Antonio, TX 78249, USA. E-mail: meng.yu@utsa.edu*

This manuscript is an extension of the conference version appearing in the Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '17) [1]. This manuscript presents a more detailed description of the system design, and augments the system to enable more comprehensive protection to trusted applications even in the presence of physical intrusions to the devices.
Manuscript received July 28, 2017; revised June 20, 2018.

Another distinct observation found in IoT system is that some devices are developed for remote sensing, monitoring or control, and thus left unattended. These devices include, but not limited to many sensors, road-side units, and credit card readers that appear in forest, street, and other non-trusted environments.

The aforementioned observations, i.e., full-blown software stack design and physically vulnerable placement, unfortunately, lead to exploits of the devices. First, it is commonly recognized that the security provided by commodity OSes is often inadequate [2]. Once an OS is compromised, attackers gain complete access to all the data on a system even if the applications are free of bugs. Second, physically exposing devices to attackers facilitates device capture attacks, in which attackers unseal the device body and manipulate hardware components to steal cryptographic keys or Intellectual Property (IP) [3], [4], [5], [6]. Since these devices may often deal with sensitive data, possibly subject to laws and regulations, this is particularly disconcerting.

To address these problems, a straightforward solution is to build a protected execution environment that is isolated from the OS (to defeat cyber-space exploits) and to encrypt sensitive data (to defeat physical-space exploits). To isolate from the OS, prior efforts on this explore executing applications that handle sensitive data in separate virtual machines (e.g., [2], [7], [8]), taking advantage of hardware features (e.g., [9], [10], [11]) or retrofitting commodity OSes (e.g. [12]). Unfortunately, none of them are applicable to the aforementioned IoT multi-programming platforms. First, these devices do not have the hardware features typically available on PCs. To be energy efficient, these devices generally incorporate ARM Cortex-A processors, making the techniques that rely on unique hardware completely futile (e.g., Haven [9] based on Intel SGX). Second, these devices do not have abundant computational resources in comparison with PCs or a data center in the cloud. Thus, it is not realistic to adopt to these devices those techniques that rely on more computing power (e.g., VirtualGoast [12]. Last but not least, some techniques previously proposed require radical changes to applications and OSes, which poses a substantial barrier to their adoption. This is especially true in the scenario where device manufacturers would like to retain compatibility with existing applications.

To encrypt memory, existing solutions either depend on hardware features [13], [14] or require modification to the OS kernel [15], [16], [17]. In the latter case, it is not realistic to assume a trusted kernel, especially considering the large attack surface of the bloated OSes. Therefore, we believe that a really working memory encryption solution should not depend on the trustworthiness of the already insecure OS.

In this paper, we propose `TrustShadow`, a system specifically designed for ARM-based IoT devices that shields legacy applications from both a compromised OS and physical attacks. First, by taking advantage of ARM TrustZone technology [18], our system constructs a trusted execution environment for security-critical applications. Different from some existing techniques, `TrustShadow` does not radically change existing OSes. Rather, it utilizes a lightweight runtime system to coordinate communications between applications and untrusted OSes. As such, `TrustShadow` requires no changes to existing applications either. Second, we incorporate the memory encryption into the trusted runtime system. Memory encryption and decryption are conducted solely by the runtime system, and the normal OS cannot access the ciphertext memory contents. As such, an attacker cannot bypass memory encryption even if he compromises the OS. This design is in line with commercial hardware-based memory encryption solutions such as Intel Software Guard eXtension (SGX) [13] and AMD SME [14]. That is, memory encryption does not rely on the OS.

More specifically, we develop `TrustShadow` with a runtime system running in the TrustZone of an ARM processor. The runtime manages the page tables for applications locally in an isolated secure environment, and ensures their virtual memory cannot be accessed by an untrusted OS outside the environment. To accommodate the execution of applications in a lightweight manner, the runtime does not incorporate complicated system services. Rather, it forwards application requests for system services to the untrusted OS, similar to Proxos [8]. To guarantee security, the runtime verifies return values of system services to defeat Iago attacks [19], and interposes context switches between the applications and the untrusted OS.

`TrustShadow` further takes advantage of the page table management mechanism in the runtime system to provide memory encryption service for memory regions containing secret data. In this way, a local intruder cannot directly read out the clear-text contents in the DRAM chip using physical attacks such as cold boot attacks [3], [4]. Deeply integrated with the runtime system, our memory encryption technique distinguishes itself from others in that it does not rely on a trustworthy OS. When an encrypted page is accessed, `TrustShadow` transparently decrypts it into a SoC component – `internal RAM (iRAM)`, which is immune to physical attacks.

With the design above, `TrustShadow` protects legacy applications from the untrusted OSes running them. As a result, developers no longer need to re-engineer applications in order to run them on IoT devices. Since `TrustShadow` does not implement system services itself, the complexity of Trusted Computing Base (TCB) is reduced, making `TrustShadow` less vulnerable to exploits. The memory encryption function can be enabled or not, depending on the device working environment. For example, in server rooms with physical safeguard, it is not necessary to enable this function. On the contrary, in public areas without monitoring, such as street, memory encryption can be invaluable for defending local intrusions. To the best of our knowledge, `TrustShadow` is the first solution on ARM-based IoT devices that allows an unmodified application to run protected from attacks from untrusted OSes, and works on encrypted memory to defeat local intrusions.

In summary, this paper makes the following contributions.

- We propose a system – `TrustShadow`– for ARM-based multi-programming platforms. It can protect security-critical applications from untrusted OSes without the requirement of re-engineering the applications.
- We introduce a runtime system within `TrustShadow`. It accommodates the execution of Linux applications with a lightweight forwarding-and-verifying mechanism.
- We design a memory encryption mechanism to protect application data from physical attacks. The memory encryption mechanism does not assume a trustworthy OS kernel.
- We implemented `TrustShadow` on a real chip (SoC) board with the ARM TrustZone support with only about 5.6K lines of code (LOC) in the secure world, and 320 LOC in the normal world. Using microbenchmarks and real-world softwares, we show that `TrustShadow` imposes only negligible performance overhead when the memory encryption feature is disabled. When running with full protection, we show that `TrustShadow` incurs acceptable overhead for real-world applications.

2 Related Work

As is described in Section 1, prior research on shielding trusted applications from an untrusted OS primarily focuses on taking advantage of virtual machines, hardware features and radical code re-engineering. In this section, we summarize these works and explain why they are not suitable for IoT devices with more details. Then we introduce relevant memory encryption techniques to defeat physical exploits.

2.1 Shielded Execution

Hypervisors and Virtual Machines. To protect an application from a compromised OS, one research effort focuses on utilizing hypervisor to construct trusted execution environment for applications. Systems following this design principle include `Overshadow` [2], `CHAOS` [20], `SP`^{3} [21], `Inktag` [7], etc. They encrypt address space for an application under protection through a hypervisor, so that a compromised OS can only view the address space of the application in ciphertext. Using the hypervisor, they also verify the integrity of memory contents, and thus ensure a compromised OS cannot jeopardize the execution of the application. Similar to these techniques, another research effort focuses on escalating protection with virtual machines. For example, `Terra` [22] and `Proxos` [8] allocate a dedicated VM for an application, making it resistant to a malicious OS.

While these systems have been shown to be effective in shielding applications, they are an overkill for resource-constrained IoT devices, and sometimes cannot be adopted by IoT devices. First, deploying a hypervisor-based system is relatively computation intensive, which cannot provide IoT devices with the best (native) performance [23]. Second, hardware virtualization extension is a new hardware feature in the ARM platform, but lacks support from many existing ARM devices[1]. In fact, even though ARM has recently released its new line of processors – specifically designed for smart embedded devices – the new architecture does incorporate virtualization extension [25]. This confirms our speculation that virtualization is not suitable for resource-constrained embedded devices.

From the security perspective, hypervisor or virtual machine based solutions relies on hypervisor, which is already struggling with its own security problems due to increasing TCB size [26], [27]. In this work, `TrustShadow` harnesses TrustZone technology to mediate communication between OS and applications, which eliminates complex, error-prone resource allocation in a hypervisor. More importantly, the new ARM architecture for IoT [25] has TrustZone support, which makes our solution broadly applicable for future hardware.

Hardware Features. Research in the past also explores using various hardware features to protect applications from untrusted OSes. For example, `Haven` [9] takes advantage of Intel SGX [13] to safeguard applications. More specifically, it harnesses SGX to instantiate a secure region called enclave, and then protects the execution of applications within the enclave from malicious privilege code. In addition to Intel SGX, Trusted Platform Module (TPM) is also used for shielding self-contained applications from a potentially malicious OS. For example, both `Flicker` [10] and `TrustVisor` [11] utilize TPM to isolate the execution of sensitive code. As is described in Section 1, IoT devices generally incorporate ARM processors which do not have the aforementioned hardware features. As a result, previous techniques based on those cannot be applicable.

Trusted Language Runtime (`TLR`) [28], CaSE [29], and `TrustOTP` [30] utilize ARM TrustZone technology for shielding applications. `TLR` implements a small runtime capable of interpreting .NET managed code inside the secure world. By splitting mobile application into secure part and non-secure part, the secure part of the app is never exposed to the untrusted OSes. CaSE [29] creates a trusted environment with TrustZone for self-contained applications. `TrustOTP` harnesses TrustZone to protect the confidentiality of the One-Time-Password against a malicious mobile OS. While these works take advantage of TrustZone as an isolation mechanism, they require radical modifications to applications. This poses a substantial barrier to their adoptions.

1. ARM released virtualization extension in the year 2010 [24].

Code Instrumentation. `Virtual Ghost` [12] is another research endeavor on protecting applications from a hostile OS. Different from those techniques discussed above, it uses compiler techniques and run-time checking to implement a mechanism similar to `OverShadow`. Since the compiler instrumentation and run-time checking introduce more privilege code to kernel, not only does it increase TCB of a computer system but also imposes performance overhead, making it not suitable to energy-efficient, computation-lightweight IoT devices.

2.2 Memory Encryption

By encrypting the memory in a computing system, an attacker who obtains an image of the system memory cannot extract any valuable information. Memory encryption can be performed either by software or hardware. In the latter case, the OS is oblivious of the underlying cryptographic calculations.

Software-based Approaches. Cryptkeeper [15] extends the traditional memory model by dividing the DRAM into two parts, a larger part holding encrypted data, and a smaller part holding a clear-text working set, which acts as a cache system for the encrypted part. RamCrypt [16] works on individual process. It attempts to encrypt the whole data segments of a protected process. In HyperCrypt [31], hypervisor is used to encrypt both the kernel and user space of a guest VM. In all these solutions, to execute the program, a small working set must be kept in clear-text, rendering data in this region insecure.

The following works address the problem of clear-text working set. Bear [32] hides the working set in the on-chip memory of ARM-based device. However, it focuses on a "from scratch" micro-kernel other than a commodity OS. Sentry [17] encrypts an Android application when the device is locked. To execute applications in background, it also employs on-chip caches.

All the existing solutions are based on the assumption that the OS kernel is trusted. However, this seldom holds because of the big TCB of commodity OSes. In `TrustShadow`, we enforce memory encryption using a lightweight runtime system that is isolated from the OS. With a substantially reduced TCB, `TrustShadow` affords a more secure and comprehensive memory encryption solution.

Hardware-based Approaches. By modifying the computer system architecture, it is possible to encrypt the DRAM chip solely by hardware. Execute-Only Memory (XOM) [33] is one of the first hardware-based efforts that uses full memory encryption to defeat against software piracy. The traditional fetch-decode-execute flow in the processor design is inserted with a decryption step. Researchers also proposed placing a specialized cryptography hardware between the processor and DRAM to encrypt and decrypt memory transactions [34], [35]. Although these solutions are efficient, they rely on hardware modifications. With increased cost, they are not feasible for incremental Commercial Off-The-Shelf (COTS) defense deployment.

As mentioned earlier, Intel processors with SGX extension [13] could isolate a security-critical application from an untrusted OS by placing it inside an enclave. The design of SGX also excludes DRAM out of its TCB. To achieve this, the DRAM page frames corresponding to an enclave are encrypted by hardware, and is only decrypted within the processor. In this work, we show that many SGX features, including memory encryption, can be supported with `TrustShadow` on ARM devices in a programmable and flexible way. We present a detailed comparison between `TrustShadow` and SGX in Section 9.5.

3 Preliminary

In this section, we present the background of ARM TrustZone technology, physical-space threats to IoT devices, and on-chip RAM available in ARM devices.

3.1 ARM TrustZone

We briefly describe the architecture of ARM TrustZone, and two key components, address space controller and memory management unit (MMU).

Architecture. ARM TrustZone partitions all of the System-on-Chip (SoC) hardware and software resources in one of two worlds - the secure world for the security subsystem, and the normal world for everything else. With this partition, a single physical processor core can safely and efficiently execute code from both the normal world and the secure world in a time-sliced fashion. When the processor executes code in the normal world, it enters a non-secure state in which the processor can only access resources in the normal world. Otherwise, it is in a secure state in which the processor can access resources resided in both worlds.

To determine whether program execution is in the secure or normal world, ARM TrustZone extends a Non-Secure bit (NS-bit) on the AMBA Advanced eXtensble Interface (AXI) bus. With this NS-bit, the processor can check permissions on the access. To manage switches to and from the secure world, TrustZone provides *monitor mode software* which ensures the state of the world that the processor is leaving is safely saved, and the state of the world the processor is switching to is correctly restored. The secure world entry to the monitor mode can be achieved by an explicit call via an `smc` instruction.

Address Space Controller. TrustZone Address Space Controller (TZASC) is an Advanced Microcontroller Bus Architecture (AMBA) compliant SoC peripheral. It allows a TrustZone system to configure security access permissions for each address region. In TrustZone, the access permissions are managed by a group of registers, the access to which must be from the secure world. In addition, TZASC controls data transfer between an ARM processor and Dynamic Memory Controller (DMC). To permit data transfer, it examines whether NS-bit matches the security settings of the memory region. Given a memory region set to *secure access only*, for example, an attempt to read returns all zeros and that to write has no change to the contents in that region.

Memory Management Unit. An ARM processor also provides MMU to perform the translation of virtual memory addresses to physical addresses. Since TrustZone partitions memory space into secure and normal worlds, a processor with TrustZone enabled provides two separated virtual MMUs which allow each world to map virtual addresses to physical addresses independently.

In the normal world, a process can only access physical memory in the non-secure state. In the secure world, it however can specify how to access physical memory by tuning NS-bit. For example, it could adjust the NS field in the first-level page table, and access the memory in either the secure or non-secure state. This flexibility augments a TrustZone system with an ability to efficiently share memory across the worlds.

3.2 Physical Memory Attacks

One of the easiest yet destructive physical attack targeting DRAM is cold boot attack [3], [4]. In a cold boot attack, an attacker quickly reboots the victim device and launches a malicious memory-dumping kernel that copies the contents of the system memory into the device permanent storage. The success of this attack relies on the remanence effect of the DRAM chips, which states that the DRAM memory content fades away slowly than expected (especially at low temperatures), leaving a large portion of residual contents even after reboot [3].

There are other physical attacks. In a bus monitoring attack, an attacker may attach a bus monitoring tool [36] to the memory bus to intercept the memory transactions. Bus monitoring also facilitates side channel attacks. For example, an attacker can deduce the cryptographic keys by merely observing memory access patterns [37]. In DMA (Direct Memory Access) attacks, an attacker can also utilize the debugging port of an UART controller [17] to issue DMA read requests to the device, bypassing security checks performed by the processor. This is because the DMA engine is independent of the processor, and can directly communicate with the DRAM chip.

3.3 On-chip RAM

Different from DRAM, SoC components are integrated into the chip. Therefore, they can exhibit better resilience to physical exploits. Utilizing cache [17], [29] or internal RAM [17], [32], many defense systems have been proposed.

CPU cache is a small amount of static RAM that sits in-between the processor and the DRAM. It has a much lower access latency than DRAM. Therefore it is widely deployed in computing systems to bridge the performance gap between the processor and the DRAM. When a processor loses power, all the cache contents become invalid. As a result, cache is inherently immune to cold boot attacks. Since cache data never appear on the memory bus, and DMA transfers data directly from DRAM without passing through cache, cache is also immune to bus monitoring and DMA attacks. a destructive physical attack targeting DRAM chip [3], [4].

On-chip RAM (OCRAM), also known as internal RAM (iRAM), is a small amount of static RAM tightly coupled with the SoC. Typically, it is used by the SoC's proprietary firmware to initialize the chip before DRAM can be used. IRAM has two features. First, it is encapsulated into the SoC package. Therefore, it is extremely difficult to disassemble the chip and expose the iRAM by physical attacks, such as bus-monitoring [36]. Second, whenever the device is rebooted, the device's firmware erases the iRAM [17]. This makes iRAM an ideal haven for storing temporary secret data to thwart cold boot attacks. With the secret data sheltered in iRAM, the malicious memory-dumping kernel could only obtain erased IRAM contents.

To exploit cache for data storage, existing solutions lock a portion of the cache to prevent the data from being evicted into the DRAM [17], [29]. However, cache lockdown is an obsoleted feature, as indicated in the official technical reference manual of ARM's latest processors (Chapter 6.1 and 7.1 in [38]). Moreover, monopolizing portions of cache impacts the overall system performance. We consider it sub-optimal to exploit cache for data storage, and instead choose iRAM to defeat physical attacks in TrustShadow.

4 Threat Model

TrustShadow shields a trustworthy application from both a hostile OS and a local intruder. We consider a completely compromised OS, which means the attacker can execute arbitrary hostile code with system privilege to interfere with the memory and registers of a process. For example, it may read/write any memory in victim process's address space, through either load/store instructions or Direct Memory Access [5], [6], causing memory disclosure and code injection attacks. As another example, OS could modify interrupted process state (e.g., the PC register) during exception handling and resume the execution from an arbitrary instruction to change the program execution's control flow. In addition, a hostile OS could change victim process's behavior by hijacking system services (e.g., forging system call responses), leading to Iago attacks [19].

Regarding a local intruder, we consider an attacker capable of dumping a memory image of the entire DRAM chip. In practice, this can be achieved by cold boot attacks [3], [4], bus monitoring [36], and DMA attacks [5], [6].

Availability is out of scope in this paper. In fact, a compromised OS could simply refuse to boot, or decline the time slices assigned to a trusted process to launch Denial-of-Service (DoS) attacks. We design our system with side channel attacks in mind. For example, we adopt hardware-based encryption mechanism to minimize the impact of side channel analysis. We also patch our system with kernel page table isolation technique to defeat Meltdown attack [39]. However, if the application developer adopts insecure software implementation subject to side channel attacks (e.g., sbox-based AES implementation), TrustShadow cannot prevent potential data leakage. Sophisticated side channel analysis through timing [40], energy consumption [41] and electromagnetic signal [42] are out of scope in this paper. We assume the runtime system running in the TrustZone is trusted. Throughout our design of TrustShadow, we keep its functionality simple and its code base minimal. This makes it easier to ensure its correctness through formal verification [43], [44] or manual review.

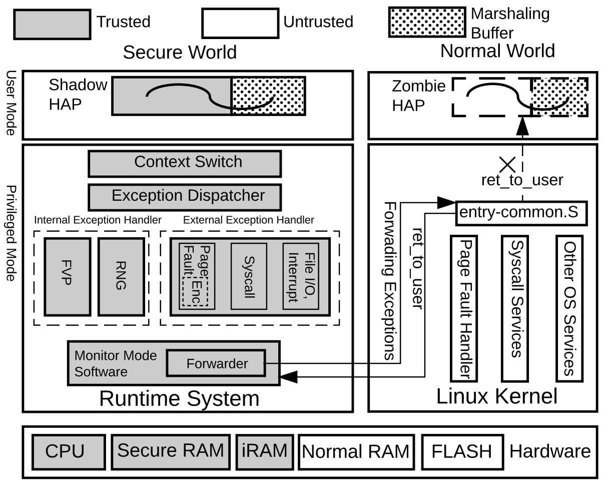


Fig. 1. The architecture of TrustShadow.

5 Overview

Figure 1 illustrates the architecture of TrustShadow, where the runtime system and Linux kernel run in the secure and normal world respectively. Within the secure world, the runtime system shields the execution of a High-Assurance Process (HAP), and all the trusted modules shown in the figure cannot be accessed by the ordinary Linux running in the normal world.

To be resistant to a hostile OS, a HAP needs to be initialized through a customized system call, which creates a "zombie" HAP and its "shadow" counterpart. In our design, the zombie HAP represents the application running in the normal world. However, it *never* gets scheduled to run. Rather, TrustShadow runs its "shadow" counterpart residing in the secure world. To support the execution of the shadow HAP, TrustShadow introduces a lightweight runtime system to the secure world.

The runtime system does not provide system services for shadow HAPs. Instead, it intercepts exceptions and forwards them to the Linux OS running in the normal world. In this way, the runtime system can maintain a trusted execution environment for HAPs without introducing a large amount of code to the secure world. To enable cross-world communications, TrustShadow introduces data structure task_shared to share data between the runtime system and the OS. In addition, TrustShadow sets aside data structure task_private to store sensitive metadata for shadow HAPs. It can only be accessed by the runtime system.

To accommodate the execution of HAPs and coordinate communications across two worlds, the runtime system is designed with various modules (see Figure 1). Serving as the gateway for all exceptions and their returns, the context switch module maintains the CPU hardware context for each HAP, and restores/clears general-purpose registers accordingly. It allows the runtime system to coordinate the execution of a HAP and avoid leaking sensitive data to the OS running in the normal world.

The runtime system also implements an internal exception handler module – indicated by FVP and RNG in Figure 1. They are designed to handle floating point computation and random number requests locally, for the reasons that cryptographic operation must rely upon trustworthy random number generation, and floating point computation necessarily exposes floating registers if TrustShadow relies upon the Linux OS for handling it. In Section 6.5, we describe this internal exception handler in detail.

In our design, the runtime system handles exceptions using three modules, including exception dispatcher, external exception handler, and forwarder. The exception dispatcher is responsible for dispatching exceptions to corresponding handlers. Except for floating point exception and random number requests, this module dispatches all the exceptions to the external exception handler which further redirects the exceptions to the forwarder module. To accommodate exception forwarding in a transparent manner, the forwarder module *emulates* an exception context for the normal world, pretending that exception is trigger by the zombie HAP. After receiving exceptions, the Linux OS handles them and returns results through task_shared. The external exception handler verifies the return results before reflecting them to the execution environment of the corresponding HAP. In Section 6.2, 6.3 and 6.4, we describe how the three modules coordinate and perform external exception handling.

The most important data structure for a process's execution environment is page tables. TrustShadow differentiates a public page (e.g., a code page) and a private page (e.g., an anonymous data page). In Section 6.3.1, we give a clear definition of public and private pages for use in this paper. With the page fault handler module, the runtime system loads a public page into a secure DRAM frame for isolation with the OS. For private pages, they appear as ciphertext in the normal DRAM. When such a page is accessed, TrustShadow transparently decrypts it into a secure iRAM page. In this way, we ensure that the entire address space of a HAP is isolated from the OS, and the private pages *never* appear in cleartext in the DRAM.

Since the normal OS does not have the privilege to access a shadow HAP, TrustShadow also introduces a world-shared buffer, indicated as the marshaling buffer in Figure 1. Through this buffer, not only does TrustShadow share the parameters of system calls with the ordinary OS but also retrieves the returns of the system calls. To retrieve the return value of a system call, TrustShadow copies the data in the buffer to the memory region corresponding to the system call, provided that the verifier module marks it valid.

6 Runtime System

In this section, we detail the runtime system illustrated in Figure 1. We begin with memory management for security. Then, we discuss how the aforementioned modules coordinate HAP execution.

6.1 Memory Management

Here, we describe how we partition physical memory regions, and specify the design of virtual memory system.

Physical Memory Partition. Using TZASC, TrustShadow creates four distinct physical memory regions. They are non-secure region ZONE_TZ_NORMAL as well as secure regions ZONE_TZ_RT, ZONE_TZ_APP, and ZONE_TZ_IRAM. The non-secure region can be accessed by both the normal and secure worlds, whereas the secure regions have to be accessed through the secure world. In our design, we designate secure regions ZONE_TZ_APP and ZONE_TZ_IRAM for HAPs, ZONE_TZ_RT for the runtime system, and non-secure region ZONE_NORMAL for the Linux OS, the encrypted data pages of HAPs and other ordinary processes. In our prototype, the whole ZONE_NORMAL can be mapped to lowmem of the Linux kernel virtual address space. Note that ZONE_TZ_IRAM is a special secure region that is backed by the iRAM. TrustShadow uses it to store the working set of a HAP that contains private data. We illustrate these four regions in Figure 2.

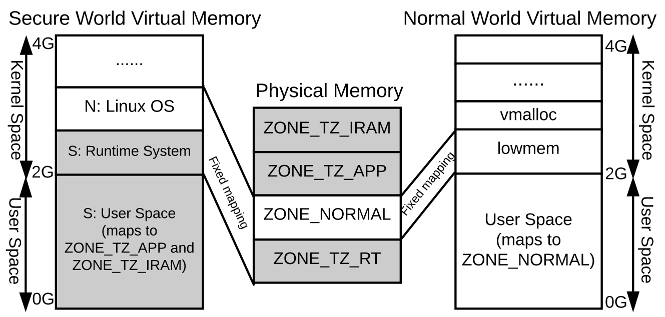


Fig. 2. Physical memory partition vs. virtual memory layout.

With the partition above, the runtime system, HAPs and Linux OS are all physically isolated, which provides the essential support for safeguarding the HAPs.

Virtual Memory Layout. TrustShadow supports executing legacy Linux code in the secure world. As a result, we design the virtual address of the secure world to follow the same user/kernel memory split as that in the Linux OS. With this design, legacy code can be offloaded to execute in the secure world without any code relocation. In our current design, both Linux OS and the runtime system maintain a 2G/2G virtual address split, as shown in Figure 2.

In the secure world, in addition to mapping itself to ZONE_TZ_RT, the runtime system maps the physical memory holding the Linux OS (ZONE_NORMAL) in the virtual address space. With this mapping, the runtime system can efficiently locate shared data from the OS (such as task_shared) by adding a corresponding offset.

6.2 Forwarding Exceptions

In general, a program is not self-contained. During execution, it might be trapped into the OS (e.g., calling a system service, encountering a page fault or interrupt). In the ARM architecture, system calls are requested by issuing the svc instruction which traps the processor into the privileged SVC mode to accomplish the system services. Likewise, other exceptions during execution would trap the processor into the corresponding privileged modes.

As is described in Section 5, except for float point computation and random number generation, the runtime system intercepts exceptions and redirects them to the

Linux running in the normal world. Here, we describe how `TrustShadow` performs exception forwarding.

ARM processors utilize current program status register (`cpsr`) to hold the current working mode of a processor (e.g., `USR` or `SVC`). When an exception is taken, a processor enters the target mode by performing the following operations. First, register `pc` points to the corresponding offset in the exception vector table. Then, the processor stores the value of previous `cpsr` to saved program status register (`spsr`) before setting `cpsr` to indicate the current working mode (i.e., the target mode). In the ARM architecture, `spsr` is a banked register and thus each processor mode has its own copy. Based on the value of `spsr`, an exception handler could get information about the pre-exception processor mode.

Since the monitor mode software can access resources in both worlds, the runtime system *re-produces* an exception as follows. Here, we take forwarding an `SVC` exception as an example. (i) The runtime system sets `spsr` in monitor mode to represent the target mode (`SVC`). (ii) It sets the target mode's `spsr` to represent user mode (`USR`). (iii) It issues `movs` instruction to jump to the target exception handler (`0xFFFF0008`). Here, `movs` is an exception return instruction. In addition to jumping to the target address, it copies `spsr` in the current mode (`SVC`, which is set in Step i) to `cpsr` in the target mode. As a result, the OS kernel catches the exception at the correct address (`0xFFFF0008`) in the right mode (`SVC`), with `spsr` indicating that the exception comes from user mode (set in Step ii). We provide a code snippet to demonstrate this implementation in Appendix A. Forwarding other types of exception can be implemented in a similar way.

6.3 Handling Page Fault

A page fault is one type of exception resulting from the failure of fetching an instruction or accessing data. In the ARM architecture, a page fault is also called an abort exception, raised by MMU, indicating that the memory accessed does not have a page table entry set properly. After such an exception is taken, an OS invokes its page fault handler which assigns an appropriate physical page and updates the page table entry accordingly. Typically, a page table entry includes the virtual-to-physical address mapping and the access permissions of the virtual memory.

In general, an OS maintains page tables for applications. However, considering that an OS might be hostile and can tamper with the page tables for applications, we isolate these page tables from the OS by placing them in the secure world. The runtime system updates their entries by taking advantage of the page fault handler provided by Linux OS.

Specifically, we modify the existing on-demand page fault handing mechanism in Linux. In particular, we hook the page fault handler so that it can store the context of page fault handling in the aforementioned shared memory `task_shared`[2]. After retrieving the updating information and before installing a page table entry, the runtime system

2. In our design, `task_shared` carries the updated page table entry value (which contains the address of the translated physical memory page), the influenced virtual address, and additional contextual information.

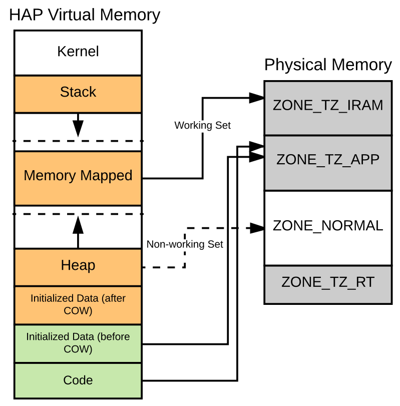


Fig. 3. Virtual address space of a HAP and its mapping to physical memory. Dotted line stands for a "logic" mapping to an encrypted page. Access to it triggers the page fault handler to decrypt it to a free page in `ZONE_TZ_IRAM`.

validates the returned information. In the following, we present the overall design goals of the page fault handling, followed by more details on several case studies.

6.3.1 Overall Design Goals

In Figure 3, we show the address space of a HAP and its mapping to the physical memory. An executable file, which is commonly assumed to be publicly available, contains the code region and initialized data region (such as global and static variables). When a process updates a variable, a Copy-On-Write (COW) happens, and the corresponding data page is duplicated and becomes private. Therefore, we consider code regions and initialized data regions before COW as public regions (shown in green in Figure 3), and consider others, including the anonymous memory segments (such as bss, heap, stack, and anonymously mapped memory segments) and initialized data regions after COW as private regions (shown in orange in Figure 3).

Guided by the information retrieved from the OS, the runtime system maintains the page table for a HAP such that the entire address space of a HAP is isolated from a hostile OS, and the private regions never appear in cleartext in the DRAM chip. In particular, The public regions are mapped to `ZONE_TZ_APP`, and `TrustShadow` ensures the integrity of each page by checking the pre-calculated hash value when it is loaded. In this way, a public page is loaded correctly, and cannot be touched by the OS during run-time. We do not protect the confidentiality of a public page from physical attacks.

The private regions, however, contains user-generated data, thus must be protected from physical exploits. We protect them cryptographically. When a private page is not active, it is mapped to an encrypted page frame in `ZONE_NORMAL`. When it is accessed, a page fault occurs, and the encrypted page is decrypted transparently to a page residing in `ZONE_TZ_IRAM`. To keep the integrity and freshness of a page, the runtime system maintains a Message Authentication Code (MAC), calculated with the encrypted page contents and an incremental generation number, for each inactive private page. We follow an encrypt-then-MAC design to avoid wasted CPU cycles in case of page manipulations. To avoid frequent swapping between a cipher-text

		Load-time Integrity	Run-time Isolation from OS			Physical Attack Immunity
			Confidentiality	Integrity	Freshness	
Private Regions	Initialized Data after COW	Hash Checking	Encryption	MAC	Incremental Generation Number	Encryption in DRAM
	Anonymous Memory (stack/heap, etc.)	N/A (all zero)				
Public Regions (code, initialized data before COW)		Hash Checking	Access Control using TrustZone			N/A (publicly available)

TABLE 1
SECURITY MECHANISMS ENFORCED ON DIFFERENT MEMORY REGIONS

DRAM page and a clear-text `iRAM` page, we also maintain a sliding window for the recently accessed private pages in `ZONE_TZ_IRAM`. When a memory access occurs in the sliding window, it is fulfilled directly. Otherwise, the oldest page in the sliding window is committed (i.e., encrypted back to the corresponding DRAM page), and the actual data is decrypted to the reclaimed `iRAM` page. In Table 1, we summarize the achieved security features for both the public regions and private regions of a HAP.

6.3.2 Page Table Update for Public Regions

A page fault may occur on different regions of the address space, and `TrustShadow` handles them differently to fulfill the design goals mentioned in Section 6.3.1. This section describes how `TrustShadow` handles page faults within public regions, which have an executable file as the backing media. In this case, `TrustShadow` relies on the OS to load the file contents to memory. As the OS may be hostile, the runtime system verifies the integrity of loaded contents. After being loaded in `ZONE_TZ_APP`, a public page can no longer be touched by the OS.

We take loading a code page as an example in Figure 4a. When a prefetch abort happens, the Linux page fault handler will eventually call `do_read_fault`, which locates the normal physical page $\mathbb{N}$ caching the corresponding code page (Step 1). In this context, a new secure world page $\mathbb{S}$ from `ZONE_TZ_APP` is allocated, and the physical addresses of both $\mathbb{N}$ and $\mathbb{S}$ pages are saved in `task_shared`. With this shared information, the runtime system first ensures that the $\mathbb{S}$ page is actually a fresh page from `ZONE_TZ_APP`. Then, the runtime system installs a new page table entry in the trusted page table (Step 2), copies the $\mathbb{N}$ page to the $\mathbb{S}$ page (Step 3) and verifies the integrity of the copied page (Step 4). Note that verification is performed on the $\mathbb{S}$ page, therefore, `TrustShadow` is resilient to `TOCTTOU` (Time Of Check To Time Of Use) attacks.

The described page table update with integrity check is the low level primitive for ensuring *load time program integrity*. `TrustShadow` enforces such checking on all the memory segments of type `PT_LOAD` in the ELF program images, including executables and dynamic libraries. In the following, we provide details on verifying the integrity of program images.

Verifying Executable Integrity. The Runtime system maintains a list of hash values in the format of `(vaddr, hash)`, which is initialized according to the bundled manifest (see Section 6.6). Once a page fault occurs in the covered range, the runtime system installs a secure page table entry as mentioned above. If validation is failed, the runtime system immediately terminates the process by sending an `_exit` system call to the OS. We note that such validation is a one-time effort, so it does not influence execution performance at run time after the program is warmed up.

Verifying Shared Library Integrity. Different from executables, shared libraries are position independent. To verify pages loaded for shared libraries, the runtime system maintains a system wide `(offset, hash)` list for all shared libraries frequently used. When a shared library image is mapped in the address space, the runtime system obtains the loaded base address `baseAddr` by monitoring the return values of the `mmap` system calls. Then, the integrity of the loaded page is verified at the address `(baseAddr + offset)`.

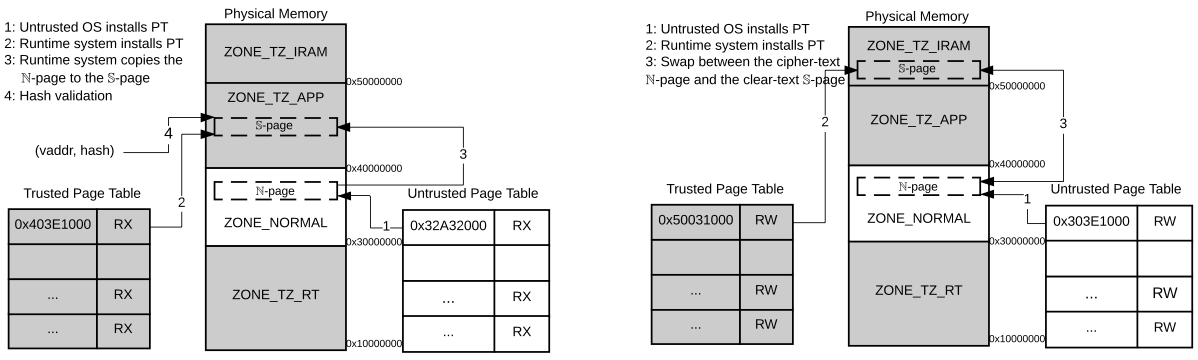
6.3.3 Page Table Update for Private Regions

As mentioned earlier, private regions contains user-generated data, and we protect them with cryptographic mechanisms. As shown in Figure 4b, the Linux page fault handler allocates a normal physical page $\mathbb{N}$ for the private page (Step 1). Then the runtime system finds a free page $\mathbb{S}$ in `ZONE_TZ_IRAM` from the sliding window, and installs a new page table entry in the trusted page table (Step 2). If the page fault is caused by COW, the runtime system copies the $\mathbb{N}$ page to the $\mathbb{S}$ page. Otherwise, the $\mathbb{S}$ page is initialized to all zeros. During execution, the clear-text $\mathbb{S}$ page and the cipher-text $\mathbb{N}$ page may swap the contents (Step 3). The runtime system maintains a MAC for each page to prevent OS from manipulating the encrypted page. Besides, the MAC algorithm also takes a generation number as input. The generation number increases with each swapping. Therefore, the runtime system is able to detect reply attacks from the OS.

File I/O Protection. Based on the page fault handling mechanism for private regions, `TrustShadow` supports encrypted file I/O operations such that a hostile OS only has access to the encrypted file contents. Specifically, `TrustShadow` allows developers to differentiate data files based on their the sensitivity levels. Only sensitive files that are specified in a manifest bundled with the application (see Section 6.6 for details) are protected.

Before elaborating the details, we first describe how `TrustShadow` manages protected files at a high level. All the operations accessing these files are transparently transformed into memory mapped I/O. To correctly map file descriptor offsets to virtual addresses, preceding pages of a file are reserved for storing meta-data. This includes the real file length, time stamp of the last access, along with the MAC. These preceding meta pages are protected by a per-application AES key that is provided by the manifest.

Encrypting a protected file is straightforward. When a non-present page of a protected file is accessed, the runtime system first verifies the MAC of the encrypted page $\mathbb{N}$ loaded by the OS. If the verification is passed, the runtime system decrypts the $\mathbb{N}$ page and writes it into a secure page $\mathbb{S}$ in `ZONE_TZ_IRAM`. When unmapping this page,



(a) Page Table Update for Public Regions.

(b) Page Table Update for Private Regions.

Fig. 4. Page Table Update (𝕊 page stands for a secure page while ℕ page stands for a normal page).

the runtime system encrypts the updated 𝕊 page into the original ℕ page, and recalculates and stores the MAC of the ℕ page. Finally, the ℕ page is written to the permanent storage by the OS.

6.4 Intervening System Calls

Two problems are raised when a system call is forwarded to the OS. First, due to encryption, the OS kernel cannot access the clear-text data of a shadow HAP, while some system call services rely on input data from user space. Second, the results returned by the OS are not trusted, which may lead to potential attacks. The runtime system coordinates the intervention between a HAP and the OS, provides the OS with essential service request data, and verifies the responses from an untrusted OS. For critical system services that cannot be served by the OS (e.g., random number generator), the runtime system implements them inside the secure world, which is discussed in Section 6.5.

6.4.1 Adapting System Calls

Memory isolation changes the way that the OS manages and accesses the memory of a HAP. Without the runtime system acting as an intermediator, it is impossible for the OS to access application data containing system call requests. We follow existing marshaling techniques available on x86 platform, in which system call parameters are adapted in a world-shared buffer [2]. This allows the OS to have temporary access to system call parameters. The design of parameter marshaling is common and straightforward. This section instead describes some remaining challenges that cannot be dealt with by parameter marshaling.

Signal. In signal handling, a signal delivery allows an untrusted OS to resume user space code at arbitrary location, thus compromising control flow integrity of a shadow HAP. In addition, the kernel function `setup_frame` needs to manipulate the process's stack to craft signal information and return code, while the OS has no privilege to do so.

`TrustShadow` addresses these problems by both OS instrumentation and runtime system support. Specifically, when a signal is registered, the runtime system inserts the handler address into the `task_private` structure of the shadow HAP. When a signal is caught by the OS, a reserved page in the marshaling buffer is used by `setup_frame` to set up a separate user mode stack specifically for signal handling. At the same time, the intended return address for signal handler is placed in `task_shared`. When the runtime system resumes, it first verifies that the address has been registered and that the `pretcode` on the signal stack is correct[3]. If so, the signal stack is copied to an unused virtual address backed by a secure page[4]. Then the hardware context of the normal control flow is saved in a temporary structure in `task_private`, and is replaced with the signal's hardware context. When the signal handler returns by issuing the `rt_sigreturn` system call, the hardware context of the normal control flow is restored.

Futex. Fast userspace mutex (futex) is another interesting kernel service that conflicts with process isolation. In Linux, a futex is identified by a four-bytes memory shared among processes or threads. It acts as a building block for many higher-level locking abstractions such as semaphores, POSIX mutexes, and barriers. If a thread fails to acquire a lock, it passes the lock's address along with its current value to a `futex wait` operation. This `futex` operation will block the thread if and only if the value in lock's address still matches the value it received. The blocked thread resumes when another thread releases the lock by issuing a `futex wake` operation, which unblocks all the threads waiting on a specific lock. Obviously, the `futex` system call needs to read the value of the lock which is in the user space of a HAP.

We observe that a thread never waits for more than one futex at a time[5]. Therefore, we hack the `futex` system call to always read from a fixed memory location in the marshaling buffer. Each time a `futex` wait operation is issued, the runtime system synchronizes the current futex value to that fixed address. In `TrustShadow`, we further handle a futex shared across processes by maintaining a system wide map that keeps physical addresses of involved memories. The runtime system queries this map to synchronize futex updates to different processes.

3. `pretcode` points to a piece of code calling the `rt_sigreturn` system call on sigpage. This code is common to all the processes.

4. `TrustShadow` reserves configurable number of secure pages specifically for this purpose.

5. A blocked thread can never issue another `futex` wait operation.

6.4.2 *Defeating Iago Attack*

As disclosed in [19], a compromised OS could subvert a HAP by manipulating the return values of system calls, thus leading to Iago attacks. For example, when a HAP requests a new memory region through the `mmap` system call, it expects that the returned region is disjoint with any existing mapping in the process's address space. However, a compromised OS could return an address that overlaps with the process's stack. Without proper checking on the return values, the following write on the new region would smash the stack and the process can be coerced into executing a return-oriented program [45].

With the runtime system sitting in-between the shadow HAP and the untrusted OS, it is straightforward to address known Iago attacks by interposing the system call interface and checking their results, providing we have a specification for that particular system call. Here, we take the aforementioned `mmap` system call as an example, which is also the classic Iago attack proposed in the original paper [19]. The runtime system maintains a trustworthy data structure of current virtual memory mapping for each HAP. Every return address of the `mmap` or `brk` system system call is compared with the current memory mapping. If an overlap is found, the HAP is immediately killed. The runtime system collects current memory mapping in three ways. First, the range of stack is obtained from current `sp`, because the stack spans from `sp` to the top of user space virtual memory. Second, heap limit can be monitored by examining return values of the `brk` system calls. Finally, the return value of each successful `mmap/munmap` system call is recorded.

We note that the current `TrustShadow` implementation does not address all the possible Iago attacks. There are so many system calls in the Linux kernel (386 in the kernel version 3.18.24 for ARM) that it is cumbersome and unrealistic to design and implement a specification for each system call like `mmap`. A clever solution was proposed by Baumann et al. [9], which adds a library OS to each application to narrow down the interface to be checked. In their prototype on Windows with Drawbridge, the interface is reduced to 22 calls, which makes comprehensive system call checking possible. In Linux, there is a similar work called Graphene [46] which exposes 43 calls. `TrustShadow` is completely compatible with this design – Graphene can be easily incorporated into our runtime system and as a result, we can check whether all the 43 system calls conform to the specification. However, we note that this also increases the TCB of the whole system. Drawbridge has a code base of over a million of LOC, while Graphene has a code base of 37,328 LOC.

6.5 Internal Exception Handling

In this section, we list security-critical exceptions that are handled directly inside the runtime system. Forwarding them to the OS would lead to security breaches.

Floating Point Computation. ARM architecture supports hardware floating point calculation by Vector Floating-Point (VFP) architecture extension. VFP introduces a set of registers and instructions specific for floating point calculations. The access to them is controlled by a privileged register `FPEXC`. In Linux, when a program accesses VFP for the first time, an `UNDEFINED` exception is raised and the kernel is responsible for enabling VFP support for this program. To support multiple processes accessing VFP concurrently, the kernel maintains a VFP context for each process in its kernel stack. This design obviously leaks user data contained in VFP registers to kernel. In `TrustShadow`, the runtime system duplicates the code handling VFP from the Linux OS. More specifically, the runtime system maintains a VFP context in the secure memory for each HAP that requires VFP calculation, and clears VFP registers whenever switching to the ordinary OS.

Random Number Generator. The Linux pseudo-Random Number Generator (LRNG) is the main source of randomness for many cryptographic applications, such as OpenSSL. Linux provides LRNG service by exposing `/dev/(u)random` devices to applications. Since using weak random values is a catastrophe for cryptographic systems, and an untrusted OS should not know the key materials used in the application, `TrustShadow` provides a trusted RNG service in the secure world. Specifically, the runtime system maintains a list of file descriptors that correspond to opened `/dev/(u)random` devices. Read operations on these descriptors are intercepted such that trusted random values are directly provided. The runtime system readily utilizes the on-board hardware random number generator RNG4 to generate strong random numbers.

6.6 Manifest Design

As mentioned in prior sections, each HAP is bundled with a manifest that provides metadata for the security features. We design a manifest to contain the following – a per-application secret key, the integrity metadata of the application (i.e., the `(vaddr, hash)` list), and a list of file names that should be cryptographically protected.

Since the manifest is stored on a local persistent storage which can be accessed by the OS, we design two mechanisms to ensure its security. First, we encrypt the per-application secret key using a per-device public key. Therefore, only the runtime system which has access to the per-device private key is able to decrypt it. Second, to ensure the integrity of the manifest, we append a digital signature calculated on the content of the manifest using a per-device private key. In a real deployment, we note that per-device public/private key pairs used for encryption and signature should be separated. In the presentation of this paper, we do not differentiate them for simplicity.

7 Implementation

We have implemented `TrustShadow` on a `Freescale i.MX6q` ARM development board that integrates an ARM Cortex-A9 MPCore processor, 1GB DDR3 DRAM and 256KB `iRAM`. As is discussed in the section above, `TrustShadow` involves operations on both the normal and secure worlds. In this section, we therefore describe our implementation details in turn.

7.1 Normal World

In the normal world, we made the following changes to the Linux kernel with version 3.18.24. (i) We added kernel parameter `tz_mem=size@start` which indicates the memory region used for HAPs, i.e., ZONE_TZ_APP. (ii) We tweaked zone-based allocator to allocate free pages from ZONE_TZ_APP when necessary. (iii) We added a tz flag to task_struct in order to make the OS capable of distinguishing HAPs. (iv) We implemented a new system call tz_execve in order to start an HAP in Linux. (v) We changed the control flow of ret_to_user and ret_fast_syscall, so that the Linux OS can pass the execution back to a corresponding shadow HAP instead of a zombie HAP. (vi) We hooked the page fault handler so that it can prepare page table update information for the runtime system. (vii) We augmented the data structure of page table so that it could recognize page faults caused by accessing encrypted pages. This is achieved by adding a new bit in the software page table entry. (viii) We modified the code handling signals in order to set up a signal stack in the marshaling buffer and make it ready for a HAP. In total, these changes introduce about 320 LOC to the Linux kernel.

7.2 Secure World

In the secure world, we implemented the aforementioned runtime system with about 4.8K LOC of ANSI C and 0.8K LOC of assembly. We used the Cryptographic Acceleration and Assurance Module (CAAM) shipped with our experiment board to perform AES encryption, hash, and HAMC calculations. For RSA, we ported mbed TLS [47] in our board. We implemented all the cryptographic operations within the SoC. Specifically, all the sensitive data, including the original keys, key schedules, and intermediate results are redirected into a single reserved iRAM page. Memory encryption works with AES-256 in CBC mode. The Initialization Vector (IV) is chosen as the virtual address of the encrypted page.

Sliding Window. The runtime system assigns a dynamic number of iRAM pages to each HAP. Starting from the first available iRAM page, the runtime system keeps a circular index to the next available iRAM page. Page faults occurred due to private region accesses continue to consume iRAM pages until the assigned pages are used up. In this case, the circular index points to the first iRAM page in the window, and swaps it for the new page fault request. After system boot, the first HAP monopolizes all the iRAM resource[6]. As more HAPs are created, they begin to share the iRAM resource. TrustShadow simply adjusts the amount assigned to each HAP to ensure an even distribution.

Secure Boot. We implemented a secure boot mechanism to guarantee the integrity of TrustShadow. Using High Assurance Boot (HAB), a non-bypassable proprietary boot ROM first loads the image of the runtime system. Then, it examines the integrity of the image. After passing the integrity check, the runtime system starts, using TZASC to configure the access policy of memory regions ZONE_TZ_RT, ZONE_NORMAL, and ZONE_TZ_APP. Since iRAM is not a part of DRAM, it cannot be configured using TZASC. Instead, we enabled OCRAM_TZ_EN bit in register IOMUXC_GPR10, and set access control policy in the low 8 bits of the CSU_CSL26 register in Central Security Unit (CSU)[7] so that ZONE_TZ_IRAM can only be accessed in the secure world from the processor. To guarantee that the policy cannot be maliciously altered, the runtime system locks the configurations. As a result, further modifications to the configured policies require a system reboot.

After the success of initialization, the runtime system loads the uboot bootloader into the normal memory region which further boots the Linux system. The Linux system runs in the normal world where it retrieves the manifest as well as the public/private key pair stored on the persistent storage. Note that, our implementation encrypts the public/private key pair in advance using the 256-bit Zeroizable Master Key (ZMK) stored on Freescale i.MX6q board. This ensures the key pair is not disclosed to the Linux in cleartext. We believe this implementation is a common practice for many device manufacturers [48].

To facilitate the secure boot, the Linux system passes the manifest and public/private key pair to the runtime system which further decrypts the key pair and installs the manifest. With this process completion, the runtime passes the execution back to the Linux system.

Kernel Page-table Isolation. The destructive Meltdown attack [39] has the potential to steal data from privilege execution domain. Therefore, it is possible that a HAP could be exploited to steal data from the runtime system running in kernel space. We applied the kernel page-table isolation technique from KASLR [49] to the implementation of runtime system. In particular, there are two page table base registers in ARM (TTBR0 and TTBR1). We configured TTBR0 to determine the address translation for the user space, while TTBR1 for the kernel space. Before a HAP is scheduled to run in user space, the runtime system configures the TTBR1 register to map a minimal set of kernel space, including the entries to the exception handler table. On taking an exception, the handler firstly restores the original TTBR1 register which maps the whole kernel space. Along with Meltdown, Spectre [50] is also relevant to TrustShadow. We defer the discussion of the implications of both attacks to Section 9.3.

8 Evaluation

In this section, we evaluate TrustShadow by conducting extensive experiments. Using microbenchmarks, we first explore the impact of TrustShadow upon primitive OS operations. Second, we quantify the overhead of I/O operations imposed by TrustShadow. Last, we show the results we measured with a real-world program – the Nginx web server. We conducted the aforementioned experiments on a Freescale i.MX6q board running both native Linux and TrustShadow. The performance of TrustShadow is measured in two modes, i.e., with and without memory encryption. When memory encryption is enabled, we assigned 48 pages in ZONE_TZ_IRAM as the sliding window. We treated the performance observed from native Linux as our baseline and compared it with those observed from TrustShadow.

6. There are 63 pages because one page is reserved for cryptographic operations.

7. CSU_CSL is a set of registers only accessible in the secure world. It can set access policies for individual slaves.

	Latency (μs)			Overhead	
Test case	**Linux**	**TrustShadow without ME**	**TrustShadow with ME**	**TrustShadow without ME**	**TrustShadow with ME**
null syscall	0.7989	1.6048	1.9736	2.01x	2.47x
open/close	29.2168	40.7886	42.3101	1.40x	1.45x
mmap (64m)	559.0000	784.0000	934.0000	1.40x	1.67x
pagefault	4.7989	7.9764	8.5795	1.66x	1.79x
signal handler install	1.6257	3.8294	4.9310	2.36x	3.03x
signal handler delivery	51.6111	57.0349	59.0811	1.11x	1.14x
fork+exit	987.0000	2328.6000	5002.3333	2.36x	5.06x
fork+exec	1060.3333	2509.0000	5163.7333	2.37x	4.87x
select (200fd)	15.0707	18.8649	20.1855	1.25x	1.34x
ctxsw 2p/0k	30.3700	32.7100	41.1300	1.08x	1.35x

TABLE 2
LMBENCH MICRO-BENCHMARK RESULTS

8.1 Microbenchmarks

Using LMBench [51], we studied the overhead imposed to basic OS operations. More specifically, we ran various system services against both native Linux and `TrustShadow`. To minimize the noise involved during our experiment, we ran each benchmark with 1,000 iterations and took the average as our measures.

Table 2 shows the results. Here, ME stands for memory encryption. First, we observe that `TrustShadow` in both modes introduce considerable overheads to individual operations. Most notably are `fork+exit`, `fork+exec` and `signal handler install`, all of which increase overhead by more than 2.36x. The high overhead introduced by the first two services is mainly due to the fact that `TrustShadow` optimizes the OS to populate all the marshaling buffer in one go when creating a new thread. And, the high overhead imposed by `signal handler install` results from copying a signal stack from the page in the normal world to one in the secure world.

Second, compared to `TrustShadow` without ME, we observe that `TrustShadow` with ME has slightly increased overheads in most of the measured system services except for `fork+exit` and `fork+exec`. For fork related system services, `TrustShadow` with ME needs to additionally prepare the clear-text working set in `ZONE_TZ_IRAM` for the new thread, which consumes considerable time.

While the overhead shown in the table appears large, it should be noted that, this does not imply that `TrustShadow` jeopardizes the performance of applications under protection. In fact, applications are significantly less sensitive to system services. As we will show later in the section, `TrustShadow` imposes only negligible to acceptable performance overhead on real-world application execution.

8.2 File Operations

To quantify the overhead imposed to I/O throughput, we conducted an experiment using Sysbench [52] in different modes. As is discussed earlier, `TrustShadow` allows developers to designate whether or not to protect a particular file. Thus, we conducted this experiment with and without file protection enabled.

We prepared 128 files, each of which has 8Mb, and tested both sequential write and random write. We measured the performance with and without cache. Results with cache enabled estimate the actual I/O performance of TrustShadow in real-world scenarios, while results with cache disabled reflect the raw I/O performance. We achieved this by tuning

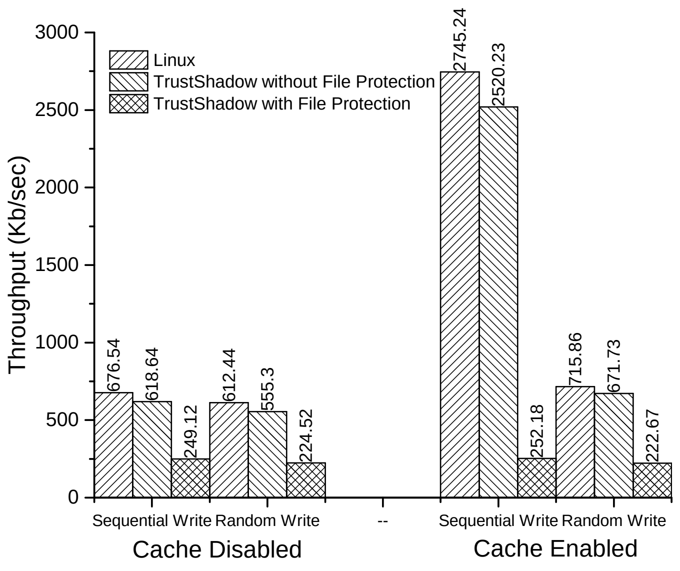


Fig. 5. File I/O performance as measured by sequential and random write.

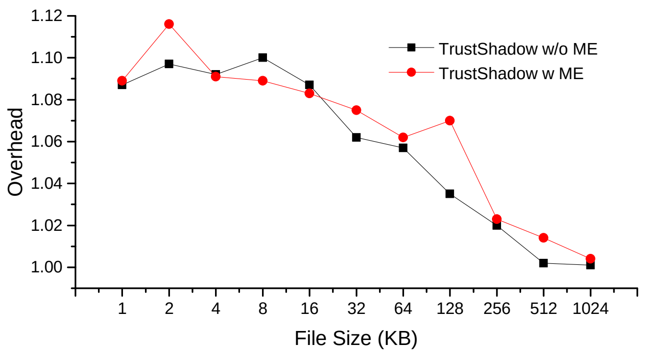


Fig. 6. Throughput overhead imposed by `TrustShadow` across HTML responses in different sizes.

parameters of Sysbench to work in write-through mode and force a call to `fsync()` after each write operation.

Figure 5 shows the results. When cache is disabled, sequential write did not exhibit significant advantage over random write. However, sequential write benefits a lot when cache is in effect (3.84x increase compared with random write). In both cases, the slowdown introduced by `TrustShadow` without file protection is moderate (around 1.09x), while the slowdown by `TrustShadow` with file protection is substantial. This is due to the fact that `TrustShadow` with file protection enabled involves heavy encryption and hashing computations when it synchronizes pages to persistent storage. Note that the difference between `TrustShadow` with and without file protection solely results from cryptographic operations. This indicates employing a more efficient cryptographic engine is a straightforward way of improving file I/O performance.

8.3 Application Benchmark

To study the impact of `TrustShadow` upon real-world applications, we mimic an embedded web server running on an IoT device. Many IoT devices expose to their users a

Bootup & world switch	1,157
External exception forwarding	2,273
Internal exception forwarding	387
Page table update	567
Iago attack checking (mmap manipulation)	890
Encryption	364
Total	**5,638**

TABLE 3
Source Code Breakdown of the Runtime System.

web interface to access the service or make configurations. For example, in an IP camera, users are able to monitor the videos on-line, or to control the movement of the cloud deck.

We ran the Nginx web server with version 1.9.15 on our testbed against both native Linux and `TrustShadow`. We configured Nginx to respond with HTML file in different sizes. To quantify the server throughput, we utilized the Apache benchmark [53] on another machine to connect to the server. We configured the benchmark program to create 10 connections simultaneously sending 10,000 HTTP requests. This experiment setting allows us to overwhelm the server and thus compare the throughput variation from the viewpoint of an end user.

Figure 6 shows the throughput overhead of Nginx. We observe that `TrustShadow` downgrades the server throughput by about 6% $\sim$ 12% when a client requests a file in a relatively small sizes. However, the throughput downgrade is alleviated when the requested file increases. As is shown in Figure 6, the server throughput drops only by 2% when a client requests a file with a size more than 256 KB. Overall, both `TrustShadow` with and without ME follow the same trend. However, `TrustShadow` without ME performs slightly better than that with ME as expected. Regarding latency, we measured the 95% percentile in each experiment, and found almost no latency overhead. The raw HTTP performance measurements can be found in Appendix B.

To show that this overhead is acceptable for real-world IoT applications, we configured the Nginx as a streaming server over HTTP. We successfully streamed a 1080p video through LAN without noticeable latency or glitch. Typically, 1080p streaming requires a bandwidth of 4000-8000 kbps [54], while the peak bandwidth in our prototype can achieve 400,000 kbps. Therefore, we would like to conclude that `TrustShadow` exploits the redundant computing powers of IoT devices to significantly improve system security, without disturbing the regular operations of such devices.

8.4 Trusted Code Base

To demonstrate the security of `TrustShadow` quantitatively, we analyze the TCB of our system, identify the share of each component, and compare its size with x86 alternatives.

Ultimately, all the code of a HAP itself must be trusted. Therefore, like all the other works in this line, user-space code is included in the TCB. Its size is highly dependent on the application's functionality and complexity. We rely on code review to ensure trust for this part of TCB.

The runtime system maintains the execution environment for a HAP, and thus must be included in the TCB. As mentioned earlier, our runtime system has only about 5.6K LOC, which we believe is small enough for manual review or formal verification. In Table 3, we show a breakdown of each component in our prototype. A substantial portion of implementation is for external exception forwarding. This is because forwarding system calls has to deal with different interface definitions of every possible system call. In comparison, previous x86 works have their own privileged code that must be trusted. Hypervisor-based solutions [2], [7], [20], [21] include the whole hypervisor in its TCB, bloating their TCBs by several hundreds of thousands of lines of code. Although thinner hypervisors exist [11], we are not aware of any similar system built on top of them. Haven [9] includes LibOS, a large subset of Windows in its TCB, resulting a TCB of millions LOC. VirtualGhost [12] includes about 5.3K LOC for their run-time system and LLVM passes. This is the only solution that has comparable TCB with `TrustShadow`.

9 Discussion

In this section, we first summarize how `TrustShadow` defeats OS-level and physical attacks to a HAP. Then, we analyze the security of the runtime system, and discuss the implications of the destructive Meltdown [39] and Spectre [50] attacks to `TrustShadow` and other attack surface. Finally, we compare our solution with Intel SGX, a processor extension for x86 platforms which provides similar functions.

9.1 HAP Security

`TrustShadow` protects an application from four aspects. (1) With a mechanism to verify the integrity of a program image, a malicious OS cannot manipulate application code/-data at *load time*. (2) With the isolation of code segments, the cryptographic protection of data segments, and the introspection mechanism, a malicious OS cannot interfere with HAP execution at *run time*. (3) With the cryptographic protection of private data segments, a local intruder cannot access a HAP's secrets through physical attacks. (4) With a cryptographic mechanism encrypting files and signing meta data, attackers can no longer read a file under protection or make any modification to it.

9.2 Runtime System Security

The protection above is established on the basis of the correctness and robustness of the runtime system. Our design enhances the security of the runtime system from three aspects. First, our design ensures the integrity of the runtime system at load time because the hash of the verification public key is burned in the chip's fuses and HAB uses this key to verify the signature of the runtime system image before loading it. Second, during execution, the runtime system is loaded into a secure physical region that is isolated from the Linux OS. Third, we design the interface to the runtime system to be as simple as possible. This greatly raises the bar for the attackers to manipulate the critical data structure of the runtime system. This is due to three reasons. (1) An application must undergo critical security reviews before being authorized to run as a HAP (i.e., providing it with a manifest with manufacture signature). (2) Even if a HAP has vulnerabilities that may be exploited to execute arbitrary code, it only runs with user privilege. (3) The interface exposed by the runtime system is narrow. Most of the time, the runtime system simply forwards the exceptions to the Linux OS. For example, crafted system call arguments that cause out-of-bounds memory access can only influence the Linux kernel, not the runtime system. The reduced functionality in the runtime system leads to the small code base of our prototype implementation, which makes formal verification of our code possible [43], [44].

9.3 Implications of Meldown and Spectre

Meltdown [39] and Spectre [50] have imposed serious security challenges to virtually all the computing systems. We explain their implications to ARM TrustZone, practically to TrustShadow. There are three variants disclosed in the original whitepapers. We analysis them one by one.

Meltdown (rogue data cache load, CVE-2017-5754). Meltdown takes advantage of out-of-order execution to load the cache lines corresponding to kernel memory, to which the attacker otherwise does not have privilege to access. Since the invalid access to kernel memory causes irreversible effect to cache lines in the processor, the attacker could infer the kernel memory by observing timing difference when accessing his own memory. There are two potential exploits to TrustShadow. First, as with the original Meltdown attack, a HAP running in the user space of secure world, could break privilege isolation and access arbitrary memory of our runtime system. However, as mentioned in Section 7.2, we have implemented a similar mechanism as KAISER (or KPTI in Linux) [49] that unmaps kernel memory when entering user space. This is easily achieved by updating TTBR1 which holds offset to page table corresponding to runtime system. Second, there are potential concerns that the untrusted Linux kernel could build a page tale entry that maps to secure physical memory and then launch Meltdown to infer secure memory. We argue that this is impossible because TZASC always returns zero for invalid accesses. As a consequence, the influence on observable cache is always fixed and so cannot be leveraged.

Spectra V1 (bounds check bypass, CVE-2017-5753). Spectra V1 takes advantage of speculative execution to bypass bounds checks to access memory that the code could not normally access. Again, cache side channel is used to infer the actual unauthorized data. Although the original Spectra V1 applies to ARM processors in general, it cannot break the boundary between secure world and normal world. This is because the invalid memory access is issued in the normal world, which returns zero. As with the case in Meldown, it cannot be leveraged in cache side channel attack.

Spectra V2 (branch target injection (CVE-2017-5715). Spectre V2 leverages shared branch prediction buffer (BPB) to influence the execution path of other tasks, potentially across exception levels. Unlike Meltdown and Spectre V1, invalid memory accesses are issued by the "legitimate" code in the correct exception level (although the results are discarded), which can leave observable trace in cache, causing side channel attacks. In practice however, spectre V2 is extremely hard to exploit. As a general mitigation technique, invalidating the BPB whenever switching from normal world to secure world is sufficient to defeat this attack.

9.4 Remaining Attack Surface.

To minimizes TCB, the runtime system does not implement system services itself, but relies on the OS. With full control of process scheduling, the OS can easily launch DoS attacks to a HAP. Similarly, to start a HAP, the OS may choose to invoke the normal `execve` system call instead of `tz_execve`. However, the process is executed in the normal world, so it cannot access cryptographically protected files.

We have designed a signing mechanism to verify the authenticity of a manifest. However, when a vulnerable program is updated, the corresponding manifest should be updated as well. A roll-back attack happens when an attacker executes the vulnerable version of the program with an older manifest. To prevent this from happening, one of our future work is to add a version number field in the manifest, and periodically communicate a list showing the updated version numbers of trusted programs between the runtime system and a remote server.

Last but not least, side channel attacks have been developed to extract information locally [40], [41], [42] or remotely [55], [56], [57]. `TrustShadow`'s current design may be subject to this line of side channel attacks. However, we can adopt existing techniques to mitigate such attacks [58].

9.5 Comparison With Intel SGX.

`TrustShadow` resembles Intel Software Guard Extensions (SGX) extension available in the newest Intel processors [13]. By creating an enclave, SGX-enabled platform is able to isolate the code and data of an application from the rest of system, including the OS kernel and even BIOS. At a low level, SGX is an architectural extension to x86 processors, which adds a set of new complex instructions. For example, dedicated instructions are used to create an opaque data structure called *Enclave Page Cache Map (EPCM)*, which defines the intended physical memory mappings of an enclave. Then the MMU circuit is instrumented to intercept page faults and to verify compliance with EPCM. Apparently, ARM processors follow a Reduced Instruction Set Computer (RISC) instruction set architecture, which is not likely to support these dedicated SGX instructions. `TrustShadow` fills the gap of providing similar security features for ARM-base IoT devices in a lightweight manner.

In the following, we compare `TrustShadow` with Intel SGX, in an attempt to figure out what falls short in our design, and possibly suggest architectural modifications on ARM to build a more robust defense solution. As the most fundamental feature, both solutions provide process isolation from the OS kernel. However, SGX goes further; an enclave is constrained in the CPU package, therefore, it is even isolated from BIOS and hypervisors. Both systems provide measurement, which ensures code and data are loaded in the memory correctly. They also use a similar approach (to be exact, cryptographic hash function) to achieve this goal. Based on measurement, SGX is able to generate a "quote", a proof to a remote entity that an enclave was instantiated

	Isolation	Measurement	Attestation	Sealing	Memory Encryption	Flexibility	Unmodified OS	Unmodified Program
TrustShadow	✓	✓	✗	✓	✓	✓	✗	✓
Intel SGX	✓	✓	✓	✓	✓	✗	✗	✗

TABLE 4
COMPARISON WITH INTEL SGX.

correctly, i.e., remote attestation [59]. This feature is missing in the current design of `TrustShadow`. However, our design can achieve the same security guarantee following a different approach. In particular, as mentioned earlier, there is an encryption key associated with each HAP. This key is contained in the manifest signed by the device manufacturer. With proper key distribution mechanism, this key (or a derived key) can be shared with the application provider, possibly the manufacturer itself. Then, if the HAP later communicates with its remote verifier on an encrypted channel protected by this key, the remote verifier is convinced that he is communicating with the authentic program. This is because only a correctly loaded HAP can execute in the secure world and have access to the encryption key. Seal is a security feature which enables the application secrets to be saved in non-volatile memory for future use. `TrustShadow` employs file I/O interception to transparently encrypt and decrypt data when accessing files, while SGX has dedicated instructions to support this. Regarding memory encryption, SGX implements a "tweaked" AES in Counter Mode [60], while `TrustShadow` utilizes the cryptographic acceleration hardware to implement AES-256 in CBC mode. Memory encryption is performed transparently to the OS kernel in both solutions. As the memory encryption algorithm can be programmed in our solution, `TrustShadow` is more flexible in terms of algorithm choice. In both solutions, the OS kernel needs to be patched to be benefited from the designed security features. Finally, `TrustShadow` is designed to support unmodified programs, while SGX needs substantial re-engineering efforts to wrap the program to harvest the security benefits [9]. We summarize the comparison in Table 4.

10 CONCLUSION

In this paper, we have presented `TrustShadow` that utilizes a carefully designed runtime system to shield applications running on multi-programming IoT devices. With `TrustShadow`, security-critical applications on these devices can be comprehensively protected even in the face of OS compromise and physical intrusions, which are major security concerns of IoT devices. Unlike techniques previously proposed, the design of `TrustShadow` does not require modification to applications. As a result, security can be guaranteed without the re-engineering efforts from the developers. Since we leverage TrustZone, a security feature available for future ARM IoT devices, we expect that our solution can be broadly applicable. Experiment results with both micro-benchmark and real IoT application show that `TrustShadow` imposes only negligible – and occasionally acceptable – overhead to IoT devices. With an increasing number of smart IoT devices developed, and the purse of hardware aided security solutions, we expect the design of `TrustShadow`, which provides major features of SGX, could inspire more research in the area of IoT computing.

REFERENCES

[1] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "Trustshadow: Secure execution of unmodified applications with arm trustzone," in *MobiSys '17*, 2017.

[2] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," in *ASPLOS '08*, 2008.

[3] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten, "Lest We Remember: Cold Boot Attacks on Encryption Keys," in *USENIX Security '08*, 2008.

[4] T. Müller, M. Spreitzenbarth, and F. Freiling, "FROST: Forensic recovery of scrambled telephones," in *11th International Conference on Applied Cryptography and Network Security*, 2013.

[5] M. Becher, M. Dornseif, and C. Klein, "Firewire: All your memory are belong to us," in *6th Annual CanSecWest Conference*, 2005.

[6] D. Hulton, "Cardbus bus-mastering: 0wning the laptop," in *Annual ShmooCon Convention*, 2006.

[7] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," in *ASPLOS'13*, 2013.

[8] R. Ta-Min, L. Litty, and D. Lie, "Splitting interfaces: Making trust between applications and operating systems configurable," in *OSDI'06*, 2006.

[9] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *OSDI'14*, 2014.

[10] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for tcb minimization," *SIGOPS Oper. Syst. Rev.*, 2008.

[11] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *2010 IEEE Symposium on Security and Privacy*, 2010.

[12] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual ghost: Protecting applications from hostile operating systems," in *ASPLOS'14*, 2014.

[13] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *HASP '13*, 2013.

[14] D. Kaplan, "AMD x86 memory encryption technologies." Austin, TX: USENIX Association, 2016.

[15] P. A. Peterson, "Cryptkeeper: Improving security with encrypted ram," in *Technologies for Homeland Security (HST), 2010 IEEE International Conference on*, 2010.

[16] J. Götzfried, T. Müller, G. Drescher, S. Nürnberger, and M. Backes, "Ramcrypt: Kernel-based address space encryption for user-mode processes," in *ASIACCS '16*, 2016.

[17] 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 *ASPLOS'15*, 2015.

[18] ARM Ltd., "Security technology building a secure system using trustzone technology (white paper)," 2009.

[19] S. Checkoway and H. Shacham, "Iago attacks: Why the system call api is a bad untrusted rpc interface," in *ASPLOS'13*, 2013.

[20] H. Chen, F. Zhang, C. Chen, Z. Yang, R. Chen, B. Zang, P.-c. Yew, and W. Mao, "Tamper-resistant execution in an untrusted operating system using a virtual machine monitor," Parallel Processing Institute, Fudan University, Tech. Rep. FDUPPITR-2007-0801, 2007.

[21] J. Yang and K. G. Shin, "Using hypervisor to provide data secrecy for user applications on a per-page basis," 2008.

[22] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *SOSP'03*, 2003.

[23] Felix Baum, "Why you don't necessarily need a hypervisor," 2014, http://embedded-computing.com/guest-blogs/why-you-dont-necessarily-need-a-hypervisor/.

[24] ARM Architecture Group, "Virtualization Extensions Architecture Specification," 2010, https://www.arm.com/products/processors/technologies/virtualization-extensions.php.

[25] ——, "ARMv8-M Architecture Simplifies Security for Smart Embedded Devices," 2015, https://www.arm.com/about/newsroom/armv8-m-architecture-simplifies-security-for-smart-embedded-devices.php.

[26] CVEdetails.com, "Xen: Vulnerability statistics," http://www.cvedetails.com/vendor/6276/XEN.html.

[27] ——, "Vmware: Vulnerability statistics," http://www.cvedetails.com/vendor/252/Vmware.html.

[28] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using arm trustzone to build a trusted language runtime for mobile applications," in *ASPLOS'14*, 2014.

[29] N. Zhang, K. Sun, W. Lou, and T. Hou, "Case: Cache-assisted secure execution on arm processors," in *The 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.

[30] H. Sun, K. Sun, Y. Wang, and J. Jing, "Trustotp: Transforming smartphones into secure one-time password tokens," in *ACM CCS'15*, 2015.

[31] J. Götzfried, N. Dörr, R. Palutke, and T. Müller, "Hypercrypt: Hypervisor-based encryption of kernel and user space," in *ARES '16*, 2016.

[32] M. Henson and S. Taylor, "Beyond full disk encryption: protection on security-enhanced commodity processors," in *International Conference on Applied Cryptography and Network Security*, 2013.

[33] D. Lie, C. A. Thekkath, and M. Horowitz, "Implementing an untrusted operating system on trusted hardware," *SIGOPS Oper. Syst. Rev.*, 2003.

[34] L. Su, S. Courcambeck, P. Guillemin, C. Schwarz, and R. Pacalet, "Secbus: Operating system controlled hierarchical page-based memory bus protection," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2009.

[35] W. Enck, K. Butler, T. Richardson, P. McDaniel, and A. Smith, "Defending against attacks on main memory persistence," in *ACSAC '08*, 2008.

[36] G. Gogniat, T. Wolf, W. Burleson, J.-P. Diguet, L. Bossuet, and R. Vaslin, "Reconfigurable hardware for high-security/high-performance embedded systems: the safes perspective," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2008.

[37] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *Journal of Cryptology*, vol. 23, no. 1, pp. 37–71, Jan 2010.

[38] ARM Ltd., "ARM Cortex-A57 MPCore Processor Technical Reference Manual," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0488d/index.html.

[39] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *ArXiv e-prints*, Jan. 2018.

[40] P. C. Kocher, *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*, 1996.

[41] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *CRYPTO'99*, 1999.

[42] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *CHES '01*, 2001.

[43] V. D. Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2008.

[44] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17, 2017, pp. 287–305.

[45] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *ACM CCS'10*, 2010.

[46] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, "Cooperation and security isolation of library oses for multi-process applications," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14, 2014.

[47] SSL Library mbed TLS, https://tls.mbed.org/.

[48] Samsung Electronics, "The KNOX Workspace Technical Details," https://www.samsungknox.com/en/products/knox-workspace/technical.

[49] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "Kaslr is dead: Long live kaslr," in *Engineering Secure Software and Systems*, E. Bodden, M. Payer, and E. Athanasopoulos, Eds. Cham: Springer International Publishing, 2017, pp. 161–176.

[50] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *ArXiv e-prints*, Jan. 2018.

[51] L. McVoy and C. Staelin, "Lmbench: Portable tools for performance analysis," in *ATC '96*, 1996.

[52] A. Kopytov, "SysBench: A System Performance Benchmark," 2004, https://github.com/akopytov/sysbench.

[53] Apache Software Foundation, "Apache HTTP server benchmarking tool," http://httpd.apache.org/docs/2.4/programs/ab.html.

[54] IBM Cloud Video, "Internet connection and recommended encoding settings," https://support.ustream.tv/hc/en-us/articles/207852117-Internet-connection-and-recommended-encoding-settings.

[55] R. Guanciale, H. Nemati, C. Baumann, and M. Dam, "Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures," in *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[56] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *USENIX Security '16*, 2016.

[57] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy*, 2015.

[58] Y. Zhou and D. Feng, "Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing." *IACR Cryptology ePrint Archive*, 2005.

[59] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, 2013.

[60] S. Gueron, "A memory encryption engine suitable for general purpose processors." *IACR Cryptology ePrint Archive*, 2016.

Le Guan is a postdoctoral fellow under supervisor Peng Liu at Penn State University. His research interests rest mainly in the area of system security, particularly in hardware aided security and mobile security.

Chen Cao is currently a post-doc at the Cyber Security Lab in the College of Information Sciences and Technology at the Pennsylvania State University. His research interests include operating system design and implementation, operating system security.

Peng Liu is a Professor of Information Sciences and Technology, founding Director of the Center for Cyber-Security, Information Privacy, and Trust, and founding Director of the Cyber Security Lab at Penn State University. His research interests are in all areas of computer and network security.

Xinyu Xing is an Assistant Professor in the College of Information Sciences and Technology at The Pennsylvania State University. He earned his Ph.D. in Computer Science from Georgia Tech. His research area covers system security, binary analysis and deep learning.

Xinyang Ge is a Senior Research Software Developer Engineer at Microsoft Research Redmond. His research interests lie in the areas of system and security.

Shengzhi Zhang is an Assistant Professor in the School of Computing, Florida Institute of Technology. His research interest includes, but not limited to system security, mobile security and vehicle security.

Meng Yu joined the Department of Computer Science, The University of Texas at San Antonio in July, 2015. His research interests include computer and network security.

Trent Jaeger is a Professor in the Computer Science and Engineering Department at The Pennsylvania State University and the Co-Director of the Systems and Internet Infrastructure Security (SIIS) Lab. Trent's research interests include operating systems security and the application of programming language techniques to security.