Position Paper: Challenges Towards Securing Hardware-assisted Execution Environments

Zhenyu Ning, Fengwei Zhang, Weisong Shi
Department of Computer Science
Wayne State University
Detroit, Michigan, USA, 48098
{zhenyu.ning,fengwei,weisong}@wayne.edu

Weidong Shi
Department of Computer Science
University of Houston
Houston, Texas, USA, 77204
wshi3@uh.edu

ABSTRACT

A Trusted Execution Environment (TEE) provides an isolated environment for sensitive workloads. However, the code running in the TEE may contain vulnerabilities that could be exploited by attackers and further leveraged to corrupt the TEE. The increasing buggy code inside the TEE concerns the security of the entire TEE. In this position paper, we present the challenges towards securing trusted execution environments and potential mitigation.

ACM Reference format:
Zhenyu Ning, Fengwei Zhang, Weisong Shi and Weidong Shi. 2017. Position Paper: Challenges Towards Securing Hardware-assisted Execution Environments. In *Proceedings of HASP '17, Toronto, ON, Canada, June 25, 2017,* 8 pages.
https://doi.org/http://dx.doi.org/10.1145/3092627.3092633

1 INTRODUCTION

Recently, Trusted Execution Environments (TEEs) have been widely adopted in commodity systems for enhancing the security of software execution. This approach runs the security sensitive workloads in a trusted environment and all the running states of the workloads are guaranteed to be isolated from the potentially infected environment (e.g., the OS or hypervisor). The examples of TEE include but not limited to: Intel Software Guard eXtensions (SGX) [6, 27, 46], AMD Memory Encryption Technologies [21], ARM TrustZone Technology [7], x86 System Management Mode [30], AMD Platform Secure Processor [5], and Intel Management Engine (ME) [53]. Although these well-designed and hardware-assisted TEEs provide secure execution environments, the code running in them could be buggy, which leads that the "trusted" execution environments (TEEs) are not trustworthy. While the argument is that the code base of a workload in a TEE is small enough so that the risk of having vulnerable code is low; however, due to the increasing complexity of the software and proliferation of using TEEs in commodity systems, the developers keep increasing the size of the code in TEEs (e.g., OS running in TrustZone [8], hypervisor is deployed in SMM [12], Linux containers running in SGX [9]). The large code base of workloads in a TEE inevitably creates vulnerabilities that can be exploited by attackers. Even the security design and implementation of TEEs is flawless (e.g., perfect isolation and secure architectural design), we cannot prevent attacks that are due to the deployed buggy code. Even worse, the security features of TEEs might help the attackers. Leveraging these security features, attackers can implement higher level stealthy rootkits, which is extremely difficult to be detected by the existing defense tools. For example, the anti-virus tools running in the OS are not able to detect malicious code in an Enclave created by Intel SGX because the running memory in the Enclave is encrypted. SMM-based rootkits [1] have been used by National Security Agency as stealthy cyber weapons. Additionally, ring -3 rootkits [71] have been demonstrated by using Intel ME. Therefore, running the untrusted code in trusted execution environments raises a big security concern. Moreover, this can generate a series of research challenges since existing defense mechanisms can not be applied directly. The main objective of this paper is to present this problem, discuss the research challenges, and provide potential directions to address them.

Problem Statement: Trusted Execution Environments have been introduced in different platforms for securing software execution, but achieving security not only depends on technologies of execution enthronements themselves (e.g., small TCB, strong isolation), but also relies on the executed code. Current state-of-the-art trusted execution environments research lacks of frameworks to verify the executed code, defenses within the trusted environments, methods detecting compromised TEEs, and mitigation plans for TEE-attack responses.

We consider the following **Research Challenges (RCs)** for further securing trusted execution environments.

- **RC1: Hunting Bugs in TEE's code**.

  The software running in a TEE can contain text-book vulnerabilities such as buffer overflows since this software can be written by careless programmers or third-party developers. If we have the source code of the software (e.g., SGX applications), we might be able to use existing static analysis or bug checking tools to identify vulnerabilities and minimize the number of bugs in the software. That means, we need to modify these tools to make it work for particular environment-specific applications (e.g., SMM or TrustZone code). Additionally, in some cases, we might not have the source code, instead of having a binary image of a TEE (e.g., SMM code). Hunting bugs in binary images is very difficult and time consuming. Furthermore, other TEE's image such as ME's code might not be available due to the hardware protections from vendors [66].
- **RC2: Protecting Mechanisms within TEEs**.

HASP '17, June 25, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-5266-6/17/06…$15.00
https://doi.org/http://dx.doi.org/10.1145/3092627.3092633

It is impractical for software to have perfect code without any bug, and analysis systems (e.g., RC1) cannot guarantee that they identify all vulnerabilities. Therefore, there is a need for us develop further defense mechanisms within TEEs. In the normal environment (e.g., OS), we have a series of defense mechanisms such as Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR). However, these defense mechanisms are missing in the current TEEs, on the contrary, we consider these environments are more secure than the OS environment. TEEs are normally have dedicated resources, limited software libraries, and they are hardware-specific, but recent efforts show that more defense mechanisms have been adding into TEEs (ASLR in Intel SGX [60]).

- **RC3: Detecting a Compromised TEE**.

  As mentioned, a TEE might be compromised due to its text-book vulnerabilities or other attacks. However, detecting a compromised TEE is challenging because the memory content of the TEE is either encrypted or inaccessible from outside. For example, Intel SGX encrypts its code and data in enclaves; SMM and TrustZone code is not accessible by the system software (e.g., OS). On one hand, because of these "protection" features, TEEs can achieve a strong security guarantee. On the other hand, after compromising a TEE, attackers can implement undetectable stealthy rootkits (e.g., SMM-based keyloggers [24]) in it. While providing a TEE as a strong isolated environment, having approaches to detect a compromised TEE is a challenging task.
- **RC4: Patching and Rejuvenation of TEEs**.

  After detecting a compromised TEE, it is critical to mitigate the attack and restore to a healthy state in the TEE. However, if the TEE is compromised, it is likely that the system software is malicious, too. Thus the patching process running in the system software cannot be trusted. Moreover, the TEEs normally have a high-privilege and is hardware-specific. It is challenging to develop methods to quickly restore the compromised TEE from a clean copy and patch the vulnerable images.

The rest of the paper organized as follows. Section 2 explains explain different trusted execution environments. Section 3 surveys existing TEE-based systems and attacks against each TEE. Section 4 presents the challenges towards securing trusted execution environments and their potential solutions. Lastly, Section 5 concludes the position paper with our visions.

2 BACKGROUND

In this section, we explain different Trusted Execution Environments. We category them into three types: 1) Ring 3 TEEs implemented via memory encryption; 2) ring -2 TEEs implemented via memory restriction; and 3) ring -3 TEEs implemented via co-processors. Next, we describe these three types of TEEs using the real world technologies.

2.1 Ring 3 TEEs via Memory Encryption

2.1.1 Intel Software Guard Extensions. In 2013, Intel presented three introduction papers on Software Guard eXtensions (SGX) [6, 27, 46]. Intel SGX is a set of instructions and mechanisms for memory accesses added to Intel architecture processors. These extensions allow an application to instantiate a protected container, referred to as an enclave. An enclave could be used as a TEE, which provides confidentiality and integrity even without trusting the BIOS, firmware, hypervisors, and OSes. Some of the researchers consider SGX as a new generation of TXT [20, 54]. Intel SGX is the latest iteration for trustworthy computing, and all future Intel processors will have this feature and use it as a TEE for addressing security problems. However, researchers raised security concerns about it. Recently, Costan and Devadas [20] published an extensive study on SGX. They analyzed the security features of SGX and raised concerns such as cache timing attacks and software side-channel attacks. Additionally, SGX tutorial slides from ISCA 2015 [31] mentioned that SGX does not protect against software side-channel attacks including using performance counters. Jain et al. [33] developed OpenSGX, an open-source platform that emulates Intel SGX hardware components at the instruction level by modifying QEMU.

2.1.2 AMD Memory Encryption Technologies. Recently, AMD introduced two new x86 features in ISCA 2016 and USENIX Security 2016 tutorials [39, 40]. One feature is called Secure Memory Encryption (SME), which defines a new approach for main memory encryption. The other is called Secure Encrypted Virtualization (SEV), which integrates with existing AMD-V virtualization architecture to support encrypted virtual machines. These features provide the ability to selectively encrypt some or all of system memory as well as the ability to run encrypted virtual machines, isolated from the hypervisor. AMD SME is a competitive technology with Intel SGX, and they provide ring 3 TEEs via memory encryption. Besides ring 3 TEEs, AMD memory encryption technologies can provide other system-level TEEs (e.g., hypervisor-level, ring -1). The SEV technology can encrypt a virtual machine, and the OS running in the VM can be a TEE. AMD SME and SEV are upcoming coming technologies that will be supported in near future AMD chipsets.

2.2 Ring -2 TEEs via Memory Restriction

x86 System Management Mode and ARM TrustZone Technology create TEEs via *memory restriction*. Specifically, they use hardware (e.g., memory management unit) to setup access permissions of memory regions for the execution space, so the normal system software cannot access the execution space. Note that TEEs via memory restriction share the CPU with the normal system software in a time-slice fashion.

2.2.1 X86 System Management Mode. System Management Mode (SMM) [29] is a mode of execution similar to Real and Protected modes available on x86 platforms (Intel started to use SMM in its Pentium processors since the early 90s). It provides a hardware-assisted isolated execution environment for implementing platform-specific system control functions such as power management. It is initialized by the Basic Input/Output System (BIOS). SMM is triggered by asserting the System Management Interrupt (SMI) pin on the CPU. This pin can be asserted in a variety of ways, which include

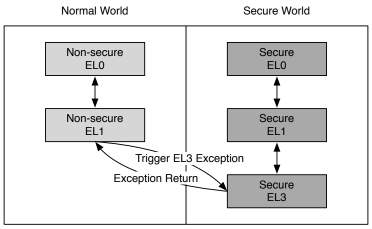

Figure 1: Processor Modes with ARM TrustZone

writing to a hardware port or generating Message Signaled Interrupts with a PCI device. Next, the CPU saves its state to a special region of memory called System Management RAM (SMRAM). Then, it atomically executes the SMI handler stored in SMRAM. SMRAM cannot be addressed by the other modes of execution. The requests for addresses in SMRAM are instead forwarded to video memory by default. This caveat, therefore, allows SMRAM to be used as a secure storage. The SMI handler is loaded into SMRAM by the BIOS at boot time. The SMI handler has unrestricted access to the physical address space and can run privileged instructions (For this reason, SMM is often referred to as *ring -2*.) The `RSM` instruction forces the CPU to exit from SMM and resume execution in the previous mode.

2.2.2 ARM TrustZone Technology. ARM TrustZone technology [7] is a hardware feature that creates an isolated execution environment since ARMv6 around 2002 [14]. Similar to other hardware isolation technologies, it provides two environments or worlds. The Trust Execution Environment (TEE) is called the secure world, and the Rich Execution Environment (REE) is called the normal world. To ensure the complete isolation between the secure world and the normal world, TrustZone provides security extensions for hardware components including CPU, memory, and peripherals.

The CPU on a TrustZone-enabled ARM platform has two security modes: Secure mode and normal mode. Figure 1 shows the processor modes in a TrustZone-enabled ARM platform. Each processor mode has its own memory access region and privilege. The code running in the normal mode cannot access the memory in the secure mode, while the program executed in the secure world can access the memory in normal mode. The secure and normal modes can be identified by reading the NS bit in the Secure Configuration Register (SCR), which can only be modified in the secure mode. As shown in Figure 1, ARM involves different Exception Levels (EL) to indicate different privileges in ARMv8 architecture, and lower EL owns lower privilege. The EL3, which is the highest EL, serves as a gatekeeper managing the switches between the normal mode and the secure mode. The normal mode can trigger an EL3 exception by calling a Secure Monitor Call (SMC) instruction or triggering secure interrupts, to switch to the secure mode, and the secure mode uses the Exception Return (ERET) instruction to switch back to the normal mode.

TrustZone uses Memory Management Unit mechanism to support virtual memory address spaces in both the secure and normal worlds. The same virtual address space in the two worlds is mapped to different physical regions. There are two types of hardware interrupts: Interrupt Request (IRQ) and Fast Interrupt Request (FIQ). Both of them can be configured as secure interrupt by configuring the IRQ bit and FIQ bit in SCR, respectively. The secure interrupt is directly routed to the secure EL3 ignoring the configuration of the normal world. ARM recommend that the IRQ is used as the interrupt source of the normal world and the FIQ is used as secure interrupt.

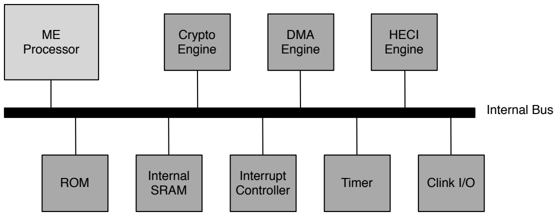

Figure 2: Architecture of Intel ME

2.3 Ring -3 TEEs via Co-Processors

2.3.1 Intel Management Engine. The Intel Management Engine (ME) is a micro-computer embedded inside of all recent Intel processors, and it exists on Intel products including servers, workstations, desktops, tablets, and smart phones [53]. Intel introduced ME as an embedded processor in 2007. At that time, its main function was to support Intel Active Management Technology (AMT), and Intel AMT is the first application running in the ME. Recently, Intel started to use ME as a Trusted Execution Environmental (TEE) for executing security-sensitive applications. According to the latest ME book [53] written by an Intel Architect working on ME, a few security applications have been or will be implemented in ME including enhanced privacy identification, protected audio video path, identity protection technology, and boot guard.

Figure 2 shows the hardware architecture of ME. From the figure we can see that ME is like a computer; it contains a processor, cryptography engine, Direct Memory Access (DMA) engine, Host-Embedded Communication Interface (HECI) engine, Read-Only Memory (ROM), internal Static Random-Access Memory (SRAM), a timer, and other I/O devices. ME executes the instructions on the processor, and it has code and data caches to reduce the number of accesses to the internal SRAM. The internal SRAM is used to store the firmware code and runtime data. Besides the internal SRAM, ME also uses some Dynamic Random-Access Memory (DRAM) from the main system's memory (i.e., host memory). This DRAM serves a role as the disk; the memory pages of code/data that are not currently used by ME processor will be evicted from SRAM and swapped out to DRAM in the host memory. The region of DRAM is reserved by the BIOS when the system boots. This DRAM is dedicated for ME use and the operating system cannot access it. However, the design of ME does not trust the BIOS and it assumes the host can access the reserved DRAM region.

2.3.2 AMD Platform Secure Processor. Although ME is for Intel processors, we can find similar technologies on AMD platforms. AMD Secure Processor [5] (also called Platform Security Processor or PSP) is a dedicated processor embedded inside of the main AMD CPU. It works with ARM TrustZone technology and ring -2 Trusted Execution Environments (TEE) to enable running third-party trusted applications. AMD Secure Processor is a hardware-based technology which enables secure boot up from BIOS level into the TEE. Trusted third-party applications are able to leverage industry-standard APIs to take advantage of the TEE's secure execution environment. Another example is System Management Unit (SMU) [45]. The SMU is a subcomponent of the Northbridge that is responsible for a variety of system and power management tasks during boot and runtime. The SMU contains a processor to assist [4]. Since AMD integrated Northbridge into the CPU, the SMU processor is an embedded processor inside of the CPU, which is same as Intel ME.

3 TEE-BASED SYSTEMS

TEE-based solutions are introduced in a variety of modern systems including cloud platforms (servers and clusters), endpoints (desktops and mobile devices), and edge nodes [63] (routers and gateways). In this section, we survey the applications and systems that leverage TEEs in ARM and x86 architectures.

3.1 SGX-based Systems and Attacks

Previous SGX-based systems such as Haven [13] ported system libraries and a library OS into an SGX enclave, which forms a large TCB. Arnautov et al. [9] proposed SCONE, a secure container mechanism for Docker that uses SGX to protect container processes from external attacks. Hunt et al. [28] developed Ryoan, an SGX-based distributed sandbox that enables users to keep their data secret in data-processing services. Schuster et al. [58] developed VC3, an SGX-based trusted execution environment to execute MapReduce computation in clouds. Karande et al. [41] secures the system logs with SGX. Shih et al. [64] leverages SGX to isolate the states of Network Function Virtualization (NFV) applications.

Schwarz et al. [59] attacks the SGX enclave via cache side channels, and demonstrates that the private key in the RSA implementation of mbedTLS can be extracted within five minutes. Other than RSA decryption, Ferdinand [16] also demonstrates a more efficient attack on the human genome indexing via SGX cache-based information leakage. AsyncShock [74] shows that the thread scheduling can be controlled by the attack, and the thread manipulation can be further used to exploit synchronization bugs inside SGX enclaves. SGX-Shield [60] provides secure address space layout randomization support for SGX programs. T-SGX [65] fight against the controlled-channel attack and ensures that the page fault will not be leaked.

3.2 SMM-based Systems

In recent years, SMM-based research has appeared in the security literature. For instance, SMM can be used to check the integrity of higher level software (e.g., hypervisor and OS). HyperGuard [55], HyperCheck [82], and HyperSentry [11] are integrity monitoring systems based on SMM. Moreover, National Science Foundation funded a project about using SMM for runtime Integrity checking last year [2]. SICE [12] presents a trusted execution environment for executing sensitive workloads via SMM on AMD platforms. SPECTRE [79] uses SMM to introspect the live memory of a system for malware detection. Another use of SMM is to reliably acquire system physical memory for forensic analysis [51, 73]. IOCheck [77, 81] secures the configurations and firmware of I/O devices at runtime. HRA [36] uses SMM for secure resource accounting in the cloud environment even when the hypervisor is compromised. MalT [78] progresses towards stealthy debugging by leveraging SMM to transparently debug software on bare metal. TrustLogin [80] protects user credentials especially passwords from theft in an untrusted environment. HOPS [42] uses SMM to create low-artifact process introspection techniques. As we can see that an array of SMM-based systems have been presented, and there is a need for us to develop novel techniques to secure the code of these systems.

Modern computers lock the SMRAM in the BIOS so that SMRAM is inaccessible from any other CPU modes after booting. Wojtczuk and Rutkowska demonstrated bypassing the SMRAM lock through memory reclaiming [55] or cache poisoning [76]. The memory reclaiming attack can be addressed by locking the remapping registers and Top of Low Usable DRAM (TOLUD) register. The cache poisoning attack forces the CPU to execute instructions from the cache instead of SMRAM by manipulating the Memory Type Range Register (MTRR). Duflot also independently discovered this architectural vulnerability [23], but it has been fixed by Intel adding SMRR [29]. Furthermore, Duflot et al. [22] listed some design issues of SMM, but they can be fixed by correct configurations in BIOS and careful implementation of the SMI handler. Wojtczuk and Kallenberg [75] presented an SMM attack by manipulating UEFI boot script that allows attackers to bypass the SMM lock and modify the SMI handler with ring 0 privilege. The UEFI boot script is a data structure interpreted by UEFI firmware during S3 resume. When the boot script executes, system registers like BIOS_NTL (SPI flash write protection) or TSEG (SMM protection from DMA) are not set so that attackers can force an S3 sleep to take control of SMM. Butterworth et al. [18] demonstrated a buffer overflow vulnerability in the BIOS updating process in SMM, but this is not an architectural vulnerability and is specific to that particular BIOS version.

3.3 TrustZone-based Systems and Attacks

Mobile devices have been increased dramatically in past few years, security became one of the major concerns of the users. ARM introduced TrustZone Technology and researchers used it to build an array of systems for enhancing the security of mobile devices. TrustDump [69] provides reliable memory acquisition by leveraging TrustZone. It uses a non-maskable secure interrupt to switch to the trust domain and introspects the memory of normal domain from trust domain. TZ-RKP [10] runs in the secure world and protects the normal OS kernel by event-driven monitoring. Sprobes [25] provides an instrumentation mechanism to introspect the normal OS from the secure world and guarantees the kernel code integrity. SeCReT [35] is a framework that enables a secure communication channel between the normal world and the secure world. TrustICE [70] provides a trusted and isolated computing environment for executing sensitive workloads. TrustOPT [68] presents a secure one-time password tokens by using ARM TrustZone technology on mobile devices. AdAttester [43] proposes a verifiable mobile ad framework that secures online mobile advertisement attestation using TrustZone. [15] suggest to use TrustZone to regulate the peripherals of devices (e.g., cameras) in restricted spaces. fTPM [50] is a firmware version of TPM 2.0 that implemented in ARM TrustZone. PrivateZone [34] uses TrustZone to create a private execution environment that is isolated from both the Rich Execution Environment and TEE.

C-FLAT [3] fights against control-flow hijacking via runtime control-flow verification in TrustZone.

Qualcomm's use Secure Channel Manager (SCM) to interact with Qualcomm's Secure Execution Environment (QSEE) via SMC instruction, and [52] leverages this interface and exploits an integer overflow vulnerability to write arbitrary secure memory. Next, they rewrite the SMC handler table with this approach and gain arbitrary TrustZone code execution. [62] use `ret2user` to gain root privilege, and also a vulnerability of unchecked bound to write one byte to almost any physical address, which finally leads to arbitrary payloads to be executed in TEE. ARMageddon [44] uses `Prime+Probe` cache attack to leak the information from secure world to normal world, and makes monitoring TrustZone code execution in normal world feasible.

3.4 ME-based Systems and Attacks

Intel uses ME as a TEE to execute security sensitive operations [53]. In 2009, Tereshkin and Wojtczuk [71] demonstrated that they can implement ring -3 rootkits in ME by injecting the malicious code into the Intel Active Management Technology (AMT). DAGGER [67] bypasses the ME isolation using a similar technique in [71], but it hooks the ME firmware function `memset` because it is invoked more often. Skochinsky [66] discovers that the ME firmware on the SPI flash uses Huffman encoding to prevent reverse engineering for implementing rootkits. Recently, Intel disclosed an AMT vulnerability in ME (CVE-2017-5689 or INTEL-SA-00075 [32]). This bug allows attackers to remotely gain administrative control over Intel machines without entering a password [49], and this remote hacking flaw resides in Intel chips for seven years [26].

4 CHALLENGES AND DIRECTIONS

In this section, we detail the challenges for securing the hardware trusted execution environments. Moreover, we provide directions that might be able to address these challenges.

4.1 Hunting Bugs in TEE's code

The software running in a TEE might contain text-book vulnerabilities that can be easily exploited by attackers. Kallenberg and Kovah [37] found that "millions of BIOSes" are easy to be compromised because the known vulnerabilities of SMM code are never patched in the BIOS. Butterworth et al. [18] demonstrated a buffer overflow vulnerability in the BIOS updating process in SMM. Di [62] found vulnerabilities that are able to execute arbitrarily code in TrustZone code. Additionally, an array of SGX-based systems have been developed [9, 13, 28, 41, 58, 64], these SGX-based applications inevitably contain vulnerabilities due to their large code bases. There is a need for us to develop effective and efficient frameworks to find the vulnerabilities in the code/images running in the TEEs and reduce the chance of having vulnerable codes before attackers exploit it.

However, existing solutions of bug hunting can not be applied directly because the TEE's code requires particular environments (e.g., SMM and TrustZone) for execution. If we have the source code of the TEE software (e.g., SGX applications), we might be able to modify existing static analysis or bug checking tools to identity bugs and minimize the number of vulnerabilities. However, if we do not have the source code but with a TEE's image, hunting bugs in binary images is very time consuming and require heavy reverse-engineering efforts. Furthermore, other TEE's image such ME code might not be obtained due to hardware vendor's protection mechanism [66].

Therefore, there is a need to develop a framework to check the TEE's code before it runs in the high-privileged, isolated, and trusted environment. For instance, we can use symbolic execution (e.g., S2E platform) to analyze the SMI handler code and TrustZone firmware. Symbolic execution can help explore the execution paths of SMI handlers and TrustZone images, and discover the paths that lead to known exploitation. Since S2E can directly work on binaries on both x86 and ARM architectures, it can analyze commercial SMI handlers and TrustZone code without knowing the source code. Note that Bazhaniuk et al [47]. proposed using the similar way for analyzing SMI handler for the BIOS security. However, they only target on detecting the out-call functions (i.e., calling functions outside of the protected memory, SMRAM, that is controlled by attackers) in the SMI handler [56]. Moreover, not only targeting on SMI handler code on x86, we can apply the approach to the TrustZone firmware on ARM architecture. Furthermore, we can modify S2E plugins to work with other vulnerabilities such as buffer overflows and poisonous pointers [48] to help us validate the inputs from the untrusted environment.

4.2 Protecting Mechanisms within TEEs

It is impractical to have perfect code running in the TEEs, and the attackers will eventually find a vulnerability and exploit it at some point. However, existing TEEs lack of defense mechanisms in the execution environments. For instance, the code running in SMM shares a single address space and paging is disabled [29]. Applications running in the SGX enclaves do not have basic protection mechanisms such as ASLR. In the normal computing environment (e.g., OS), we have an array of system-level defense mechanisms such as non-executable stack, data execution prevention, and address space randomization. However, these defense mechanisms are missing in the TEEs. Since we consider these environments are more secure than the normal OS, these basic system defense mechanisms are needed for securing the environment.

One of the defenses is to diversify TEE's environment. This increases the difficulty and cost for attackers to successfully exploit the bugs. With this approach, we can create dynamic TEEs. According to the Intel manual [29], system management mode has a very simple addressing mechanism. It disables the paging and works directly on physical addresses. When the system boots up, the BIOS initializes SMM and loads the SMI handler code to a physical memory address at SMM_Base + `0x8000`. SMM_Base represents the beginning of the SMRAM. Typically, the BIOS vendors set the SMM_base as `0xa0000` and this memory region overlaps with the VGA memory. We can randomize the base address of SMRAM for every boot or reboot by modifying the BIOS code. Specifically, we can randomly setup the SMRAM address in the BIOS for every reset signal. Note that the reset signal can be caused by a variety of power state changes including cold boot, warm boot, wake up from S3 (i.e., suspend to RAM), and so on. By randomizing the base address of SMRAM, it increases the difficulty for attackers to dump the SMRAM for exploitation or reverse engineering. Additionally, we can randomize the saved states in SMRAM, instead of at the fix location, SMM_base + 0xFC00. Then, SMM attacks such as [48] would not work since it requires to overwrite the SMM_base register at the save states area.

The execution environment of TrustZone is more complex than that of SMM. TrustZone has its own page tables and operates with memory management unit enabled. To diversify the execution environment of TrustZone, we can first randomize the location of the TrustZone firmware code. Within the TrustZone, it can support and run a Secure OS. We can implement the Address Space Layout Randomize (ASLR) technique on the Secure OS of the ARM trusted firmware. This addition can reduce the success rate of exploitation on attacks that leverage buffer overflows or return-oriented programming [61]. We may start this research direction with Coreboot [19] for the TEE like SMM, and Trusted Firmware [8] for the TrustZone.

Additionally, the randomization is needed in ring 3 TEEs (e.g., Intel SGX) as well. SGX-Shield [60] provides secure address space layout randomization support for SGX programs. Moreover, we can periodically or randomly instantiate an SGX enclave, and move the security sensitive workloads from one enclave to another. In this case, the associated states of the enclave are on the move so attacks depending on static information (e.g., memory addresses) might not work anymore.

4.3 Detecting a Compromised TEE

In practice, a TEE might be compromised due to the vulnerabilities in the code. However, detecting a compromised TEE is a very challenging problem because TEEs run at a high-privilege memory space that inaccessible from the system software (ring -2 TEEs) or use encrypted memory that their contents are mysterious without the key (ring 3 TEEs). For example, Intel SGX encrypts its code and data in enclaves; SMM and TrustZone code is not accessible by the system software (e.g., OS). Because of these "security protection" features, a TEE can achieve a strong security guarantee. However, after compromising a TEE, attackers can implement undetectable advanced rootkits in it.

Embleton et al. [24] use SMM to implement a chipset-level keylogger and a network backdoor capable of directly interacting with the network card to send logged keystrokes to a remote machine via network packets. Schiffman and Kaplana [57] further demonstrated that with USB keyboards instead of PS/2 ones. Other SMM-based attacks focus on achieving stealthy rootkits [1, 17]. For instance, the National Security Agency (NSA) uses SMM to build an array of rootkits including DEITYBOUNCE for Dell and IRONCHEF for HP Proliant servers [1]. Several attacks [67, 71] have been demonstrated using ME to implement advanced stealthy rootkits. Tereshkin and Wojtczuk [71] injects malicious code into the Intel Active Management Technology (AMT) to implement ME ring -3 rootkits. DAGGER [67] is a DMA-based keylogger implemented in ME, and it captures keystrokes very early in the platform boot process, which enables DAGGER to capture harddisk encryption passwords. While proving a TEE as a strong isolated computing environment, having a method to detect a compromised TEE is a challenging task.

One potential approach to detect a compromised TEE is to use the performance implications, timing, or other side-channel information. For instance, we might be able to detect compromised SMM or TrustZone via the timing side-channel information. The intuition is that ring -2 TEEs share the main CPU with the system software in a time-slice fashion. This approach would not work for ring -3 TEEs since they run on separated processors, not the main CPU. Normally, SMM or TrustZone is invoked very few times or the execution times of them have some specific patterns for normal system operations. If we see a system that dramatically changes its execution pattern (staying in SMM or TrustZone too long for sending out the sensitive memory pages via network packets) or invokes ring -2 TEEs very frequently (e.g., SMM-based keyloggers [24] generating SMIs for each keystroke), the system is more likely compromised. To detect the SMI invocation and its execution time, [72, 82] have implemented a tool called SMI Detector. The idea behind this tool is that the SMI invocations suspend all cores in the CPU, the SMI Detector can measure the missing time. A similar tool can work on TrustZone since it also shares the CPU in a time-slice fashion. Note that using the side-channels based approach for detecting compromised TEEs might not work for all the cases (e.g., timing side channel does not work for ring -3 TEEs). Other side channels including power consumption, cache access patterns, network traffic patterns can be considered for other cases.

4.4 Patching and Rejuvenation of TEEs

This subsection talks about the challenges on how to mitigate attackers from a compromised TEE and patch it to a good state. One simple approach is to use the system software to update the compromised TEE. However, if the TEE is compromised, it is likely that the system software is malicious, too. Thus the patching process running in the system software cannot be trusted. To ensure the restoring process is not tampered, we have to rely on a *Trust Base*. However, having such as a *Trust Base* is a challenging task.

For ring -2 TEEs, we might be able to use firmware as the *Trust Base*. This updating process works for some real world attacks such as Incursions (CERT VU#631788) [37]. In this attack, adversaries are able to bypass the isolation and get into SMM to run arbitrary code, the BIOS firmware is still protected by the `Write Enable` bit in the BIOS Control register (`BIOS_CNTL`) [29]. As long as the attackers cannot flash the BIOS firmware, the system can perform a quick restart to destruct the SMRAM and re-initialize the compromised SMI handler. In this case, the update process of the compromised TEE from the firmware works. However, it is possible that attackers can bypass the write protection and reflash the firmware. For instance, Wojtczuk and Kallenberg [75] presented an SMM attack by manipulating UEFI boot script that allows attackers to bypass the BIOS write protection lock and modify the SMI handler with ring 0 privilege (CERT VU#976132). Moreover, Speed Racer [38] described a race condition that allows an attacker to subvert the firmware flash protection mechanism. In these attacking scenarios, how can we restore the SMI handler to a clean state if the firmware can not be trusted? If we assume the BIOS, SMM, and system software are all compromised, we need to rely on a component that does not have them in the Trusted Computing Base (TCB). One potential solution is the ring -3 TEEs such as Intel ME and AMD PSP. However, how to update ring -3 TEEs is another challenging task.

5 CONCLUSIONS

Existing trusted execution environments focus on reducing TCB and achieving strong isolation by leveraging hardware support. However, other threats such as buggy code running in a TEE raise security concerns. The goal of this position paper is to draw the attention to the system security community about the challenges for achieving a more secure execution environment. We also provide our visions and potential directions for addressing the challenges.

6 ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments that improved the paper. This work is partly supported by the National Science Foundation grant DGE-1433817 and the North Atlantic Treaty Organization grant G110696. Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government.

REFERENCES

[1] 2014. NSA's ANT Division Catalog of Exploits for Nearly Every Major Software/Hardware/Firmware. http://Leaksource.wordpress.com. (2014).

[2] 2015. TWC: Small: System Infrastructure for SMM-based Runtime Integrity Measurement. https://nsf.gov/awardsearch/showAward?AWD_ID=1528185. (August 2015).

[3] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. C-FLAT: Control-flow attestation for embedded systems software. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*.

[4] Advanced Micro Devices, Inc. 2015. BIOS and Kernel Developer's Guide (BKDG) for AMD Family 16h Models 30h-3Fh Processors. http://support.amd.com/TechDocs/52740_16h_Models_30h-3Fh_BKDG.pdf. (March 2015).

[5] AMD TATS BIOS Development Group. 2013. AMD Security and Server Innovation. http://www.uefi.org/sites/default/files/resources/UEFI_PlugFest_AMD_Security_and_Server_innovation_AMD_March_2013.pdf. (2013).

[6] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*.

[7] ARM. 2009. ARM Security Technology - Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. (2009).

[8] ARM. 2016. ARM Trusted Firmware. https://github.com/ARM-software/arm-trusted-firmware. (2016).

[9] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L Stillwell, David Goltzsche, Dave Eyers, Rudiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of The 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*.

[10] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*.

[11] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. 2010. HyperSentry: Enabling Stealthy In-Context Measurement of Hypervisor Integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*. 12.

[12] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. 2011. SICE: A Hardware-level Strongly Isolated Computing Environment for x86 Multi-core Platforms. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*. 14.

[13] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*.

[14] David Brash. 2002. ARM White Paper, The ARM Architecture Version 6 (ARMv6). http://lars.nocrew.org/computers/processors/ARM/ARMv6.pdf. (January 2002).

[15] Ferdinand Brasser, Daeyoung Kim, Christopher Liebchen, Vinod Ganapathy, Liviu Iftode, and Ahmad-Reza Sadeghi. 2016. Regulating ARM TrustZone Devices in Restricted Spaces. In *Proceedings of The 14th ACM International Conference on Mobile Systems, Applications and Services (MobiSys'16)*.

[16] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. (2017). http://arxiv.org/abs/1702.07521

[17] BSDaemon, coideloko, and D0nAnd0n. 2008. System Management Mode Hack: Using SMM for 'Other Purposes'. *Phrack Magazine* (2008). Issue 65.

[18] John Butterworth, Corey Kallenberg, and Xeno Kovah. 2013. BIOS Chronomancy: Fixing the Core Root of Trust for Measurement. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)*.

[19] Coreboot. 2011. Open-Source BIOS. http://www.coreboot.org/. (2011).

[20] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. https://eprint.iacr.org/2016/086.pdf. (2016).

[21] Jeremy Powell David Kaplan and Tom Woller. 2016. AMD Memory Encryption, White Paper. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf. (April 2016).

[22] Loic Duflot, Olivier Levillain, Benjamin Morin, and Olivier Grumelard. System Management Mode Design and Security Issues. http://www.ssi.gouv.fr/IMG/pdf/IT_Defense_2010_final.pdf. (????).

[23] Loic Duflot, Olivier Levillain, Benjamin Morin, and Olivier Grumelard. 2009. Getting into the SMRAM: SMM Reloaded, In Proceedings of the 12th CanSecWest Conference (CanSecWest'09). *CanSecWest, Vancouver, Canada* (2009).

[24] Shawn Embleton, Sherri Sparks, and Cliff Zou. 2008. SMM rootkits: A New Breed of OS Independent Malware. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Networks (SecureComm'08)*.

[25] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. 2014. SPROBES: Enforcing Kernel Code Integrity on the TrustZone Architecture. In *Proceedings of The 3rd IEEE Mobile Security Technologies Workshop (MoST)*.

[26] Dan Goodin. 2017. The hijacking flaw that lurked in Intel chips is worse than anyone thought. https://arstechnica.com/security/2017/05/the-hijacking-flaw-that-lurked-in-intel-chips-is-worse-than-anyone-thought/. (May 2017). Accessed 05/10/2017.

[27] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Carlos Rozas, Vinay Phegade, and Juan del Cuvillo. 2013. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*.

[28] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proceedings of 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[29] Intel. 2009. 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. (2009). http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[30] Intel. 2014. 64 and IA-32 Architectures Software Developer's Manual: Chapter 34. (2014).

[31] Intel. 2015. ISCA 2015 SGX Tutorial. https://software.intel.com/sites/default/files/332680-002.pdf. (2015).

[32] Intel Security Group. 2017. INTEL-SA-00075. https://security-center.intel.com/advisory.aspx?intelid=INTEL-SA-00075&languageid=en-fr. (May 2017). Accessed 05/10/2017.

[33] Prerit Jain, Soham Desai, Seongmin Kim, Ming-Wei Shih, JaeHyuk Lee, Changho Choi, Youjung Shin, Taesoo Kim, Brent B. Kang, and Dongsu Han. 2016. OpenSGX: An Open Platform for SGX Research. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS'16)*. San Diego, CA.

[34] Jinsoo Jang, Changho Choi, Jaehyuk Lee, Nohyun Kwak, Seongman Lee, Yeseul Choi, and Brent Byunghoon Kang. 2016. PrivateZone: Providing a Private Execution Environment using ARM TrustZone. *IEEE Transactions on Dependable and Secure Computing* (2016).

[35] Jin Soo Jang, Sunjune Kong, Minsu Kim, Daegyeong Kim, and Brent Byunghoon Kang. 2015. SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment. In *Proceedings of 22nd Network and Distributed System Security Symposium (NDSSÂŠ15)*.

[36] Seongwook Jin, Jinho Seol, Jaehyuk Huh, and Seungryoul Maeng. 2015. Hardware-Assisted Secure Resource Accounting under a Vulnerable Hypervisor. In *Proceedings of 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environment (VEE'15)*.

[37] Corey Kallenberg and Xeno Kovah. 2015. How Many Million BIOSes Would you Like to Infect? http://conference.hitb.org/hitbsecconf2015ams/wp-content/uploads/2015/02/D1T1-Xeno-Kovah-and-Corey-Kallenberg-How-Many-Million-BIOSes-Would-You-Like-to-Infect.pdf. (2015).

[38] Corey Kallenberg and Rafal Wojtczuk. 2014. Speed Racer: Exploiting an Intel Flash Protection Race Condition. https://events.ccc.de/congress/2014/Fahrplan/system/attachments/2565/original/speed_racer_whitepaper.pdf. (2014).

[39] David Kaplan. 2016. AMD x86 Memory Encryption Technologies, USENIX Security Tutorial 2016. https://www.usenix.org/conference/usenixsecurity16/

technical-sessions/presentation/kaplan. (2016).

[40] David Kaplan, Tom Woller, and Jeremy Powell. 2016. AMD Memory Encryption Tutorial, ISCA 2016. https://sites.google.com/site/metisca2016/. (2016).

[41] Vishal Karande, Erick Buaman, Zhiqiang Lin, and Latifur Khan. 2017. SGX-Log : Securing System Logs With SGX. In *Proceedings of the 12th ACM on Asia Conference on Computer and Communications Security (AsiaCCS'17)*.

[42] Kevin Leach, Chad Spensky, Westley Weimer, and Fengwei Zhang. 2016. Towards Transparent Introspection. In *Proceedings of 23rd IEEE Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*.

[43] Wenhao Li, Haibo Li, Haibo Chen, and Yubin Xia. 2015. AdAttester: Secure Online Mobile Advertisement Attestation Using TrustZone. In *Proceedings of The 13th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys'15)*.

[44] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache attacks on mobile devices. In *Proceedings of 25th USENIX Security Symposium (USENIX Security'16)*.

[45] Rudolf Marek. 2014. AMD x86 SMU firmware analysis - Do you care about Matroshka processors? https://events.ccc.de/congress/2014/Fahrplan/system/attachments/2503/original/ccc-final.pdf. (2014).

[46] Frank Mckeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*.

[47] Oleksandr Bazhaniuk and John Loucaides and Lee Rosenbaum and Mark R. Tuttle and Vincent Zimmer. 2015. Symbolic Execution for BIOS Security. In *Proceedings of 9th USENIX Workshop on Offensive Technologies (WOOT'15)*.

[48] Oleksandr Bazhaniuk, Yuriy Bulygin, Andrew Furtak, Mikhail Gorobets, John Loucaides, Alexander Matrosov, Mickey Shkatov. 2015. A New Class of Vulnerabilities in SMI Handlers. http://www.c7zero.info/stuff/ANewClassOfVulnInSMIHandlers_csw2015.pdf. (2015).

[49] Carlos Perez. 2017. Tenable Blog: Rediscovering the Intel AMT Vulnerability. https://www.tenable.com/blog/rediscovering-the-intel-amt-vulnerability. (May 2017). Accessed 05/10/2017.

[50] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. 2016. fTPM: A Software-only Implementation of a TPM Chip. In *Proceedings of The 25th USENIX Security Symposium (UsenixSecurity'16)*.

[51] Alessandro Reina, Aristide Fattori, Fabio Pagani, Lorenzo Cavallaro, and Danilo Bruschi. 2012. When Hardware Meets Software: A Bulletproof Solution to Forensic Memory Acquisition. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'12)*.

[52] Dan Rosenberg. 2014. Reflections on trusting trustzone. *BlackHat USA* (2014).

[53] Xiaoyu Ruan. 2014. *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress.

[54] Joanna Rutkowska. 2015. Intel x86 Considered Harmful. http://blog.invisiblethings.org/papers/2015/x86_harmful.pdf. (October 2015).

[55] Joanna Rutkowska and Rafal Wojtczuk. 2008. Preventing and Detecting Xen Hypervisor Subversions. http://www.invisiblethingslab.com/resources/bh08/part2-full.pdf. (2008).

[56] Ilia Safonov and Alex Matrosov. 2016. Excite project: all the truth about symbolic execution for BIOS security. http://2016.zeronights.org/program/9. (2016).

[57] Joshua Schiffman and David Kaplan. 2014. The SMM Rootkit Revisited: Fun with USB. In *Proceedings of 9th International Conference on Availability, Reliability and Security (ARES'14)*.

[58] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*.

[59] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Proceedings of 14th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'17)*.

[60] Jaebaek Seo, Byounyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Proceedings of 24th Network and Distributed System Security Symposium (NDSS'17)*.

[61] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS'07)*.

[62] D Shen. 2015. Attacking your trusted core: Exploiting trustzone on android. *Black Hat USA* (2015).

[63] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5 (Oct 2016), 637–646. https://doi.org/10.1109/JIOT.2016.2579198

[64] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. 2016. S-NFV: Securing NFV states by using SGX. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*.

[65] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of 24th Network and Distributed System Security Symposium (NDSS'17)*.

[66] Igor Skochinsky. 2014. Intel ME Secrets: Hidden code in your chipset and how to discover what exactly it does. https://recon.cx/2014/slides/Recon%202014%20Skochinsky.pdf. (2014).

[67] Patrick Stewin and Iurii Bystrov. 2012. Understanding DMA Malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'12)*.

[68] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. 2015. TrustOTP: Transforming Smartphones into Secure One-Time Password Tokens. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS'15)*.

[69] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Sushil Jajodia. 2014. TrustDump: Reliable Memory Acquisition on Smartphones. In *Proceedings of The 18th European Symposium on Research in Computer Security (ESORICS'14)*.

[70] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. 2015. TrustICE: Hardware-assisted Isolated Computing Environments on Mobile Devices. In *Proceedings of The 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'15)*.

[71] Alexander Tereshkin and Rafal Wojtczuk. 2009. Introducing Ring -3 Rootkits. http://invisiblethingslab.com/itl/Resources.html. (2009).

[72] J. Wang, K Sun, and A. Stavrou. 2012. A Dependability Analysis of Hardware-Assisted Polling Integrity Checking Systems. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'12)*.

[73] Jiang Wang, Fengwei Zhang, Kun Sun, and Angelos Stavrou. 2011. Firmware-assisted Memory Acquisition and Analysis Tools for Digital Forensic. In *Proceedings of the 6th International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE '11)*.

[74] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *Proceedings of The 21st European Symposium on Research in Computer Security (ESORICS'16)*.

[75] Rafal Wojtczuk and Corey Kallenberg. 2014. Attacking UEFI Boot Script. 31st Chaos Communication Congress (31C3), http://events.ccc.de/congress/2014/Fahrplan/system/attachments/2566/original/venamis_whitepaper.pdf. (2014).

[76] Rafal Wojtczuk and Joanna Rutkowska. 2009. Attacking SMM Memory via Intel CPU Cache Poisoning. (2009). http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf

[77] Fengwei Zhang. 2013. IOCheck: A Framework to Enhance the Security of I/O Devices at Runtime. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*.

[78] Fengwei Zhang, Kevin Leach, Angelos Stavrou, Haining Wang, and Kun Sun. 2015. Using Hardware Features for Increased Debugging Transparency. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*.

[79] Fengwei Zhang, Kevin Leach, Kun Sun, and Angelos Stavrou. 2013. SPECTRE: A Dependable Introspection Framework via System Management Mode. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*.

[80] Fengwei Zhang, Kevin Leach, Haining Wang, and Angelos Stavrou. 2015. TrustLogin: Securing Password-Login on Commodity Operating Systems. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (AsiaCCS'15)*.

[81] Fengwei Zhang, Haining Wang, Kevin Leach, and Angelos Stavrou. 2014. A Framework to Secure Peripherals at Runtime. In *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS'14)*.

[82] Fengwei Zhang, Jiang Wang, Kun Sun, and Angelos Stavrou. 2014. HyperCheck: A Hardware-assisted Integrity Monitor. In *IEEE Transactions on Dependable and Secure Computing (TDSC'14)*.