SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM

Ahmed M. Azab,[1] Kirk Swidowski,[1]Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning
Samsung KNOX R&D, Samsung Research America
{a.azab, r1.bhutkar, jia.ma, wenbo.s, ruowen.wang, peng.ning}@samsung.com, kirk@swidowski.com

***Abstract*—Previous research on kernel monitoring and protection widely relies on higher privileged system components, such as hardware virtualization extensions, to isolate security tools from potential kernel attacks. These approaches increase both the maintenance effort and the code base size of privileged system components, which consequently increases the risk of having security vulnerabilities. SKEE, which stands for Secure Kernel-level Execution Environment, solves this fundamental problem. SKEE is a novel system that provides an isolated lightweight execution environment at the same privilege level of the kernel. SKEE is designed for commodity ARM platforms. Its main goal is to allow secure monitoring and protection of the kernel without active involvement of higher privileged software.**

SKEE provides a set of novel techniques to guarantee isolation. It creates a protected address space that is not accessible to the kernel, which is challenging to achieve when both the kernel and the isolated environment share the same privilege level. SKEE solves this challenge by preventing the kernel from managing its own memory translation tables. Hence, the kernel is forced to switch to SKEE to modify the system's memory layout. In turn, SKEE verifies that the requested modification does not compromise the isolation of the protected address space. Switching from the OS kernel to SKEE exclusively passes through a well-controlled switch gate. This switch gate is carefully designed so that its execution sequence is atomic and deterministic. These properties combined guarantee that a potentially compromised kernel cannot exploit the switching sequence to compromise the isolation. If the kernel attempts to violate these properties, it will only cause the system to fail without exposing the protected address space.

SKEE exclusively controls access permissions of the entire OS memory. Hence, it prevents attacks that attempt to inject unverified code into the kernel. Moreover, it can be easily extended to intercept other system events in order to support various intrusion detection and integrity verification tools. This paper presents a SKEE prototype that runs on both 32-bit ARMv7 and 64-bit ARMv8 architectures. Performance evaluation results demonstrate that SKEE is a practical solution for real world systems.

[1]These authors contributed equally to this work

NDSS '16, 21-24 February 2016, San Diego, CA, USA
 ISBN 1-891562-41-X
http://dx.doi.org/10.14722/ndss.2016.23009

I. Introduction

Many of the current commodity operating systems, like Linux, Windows, and FreeBSD, rely on monolithic kernels, which store security and access control policies in memory regions that are accessible to their whole code base. Hence, the security of the whole system relies on a large Trusted Computing Base (TCB) that includes the base kernel code in addition to potentially buggy device drivers.

An exploit of a monolithic kernel would allow complete access to the entire system memory and resources. In addition, it can effectively bypass kernel level security protection mechanisms. Recent incidents [1], [2], [5], [28], [32], [53] show that exploiting the OS kernel is a real threat. Hence, there is a need for security tools that provide monitoring and protection of the kernel. These tools have to be properly isolated so that they are protected from potential kernel exploitation.

A. Previous Attempts

Virtualization-based Approaches: A large body of research, such as [8], [14], [23], [24], [30], [31], [34], [37], [43]–[46], uses virtualization to provide the required isolation for security tools that monitor and protect the OS kernel. Nevertheless, virtualization is primarily designed to allow multiple OSes to share the same hardware platform. It is not practical, particularly in real world systems, to exclusively use the virtualization layer for OS kernel monitoring. Hence, the target security tools are in practice running alongside a hypervisor.

The TCB of a typical hypervisor has to be big enough to handle resource allocation and hardware peripheral virtualization. Therefore, commodity hypervisors are already struggling with their security problems. For example, there are 225 and 164 reported vulnerabilities for VMware [16] and Xen [17] respectively by December 2015. Security monitoring and protection requires a sizable code base to intercept OS events and introspect the current OS state. This code base should also be extendable to support various intrusion detection and integrity measurement mechanisms. As a result, hosting security tools inside the hypervisor increases the size of its TCB, which causes fundamental security concerns.

To achieve isolation without relying on the hypervisor, recent research efforts have been mainly exploring three alternatives: Microhypervisors, sandboxing and hardware protection.

Microhypervisor Approaches: A microhypervisor is a thin hypervisor that only focuses on providing isolation. Trustvisor [39] uses a microhypervisor to provide isolation to verified security sensitive workloads. Nova [49] uses microhypervisors

to provide memory protection of the virtualization layer. The basic idea of using microhypervisors is to minimize the TCB to host sensitive code that requires escalated protection. Hence, microhypervisors do not provide a good fit to host security tools that require a relatively large code base. A good solution is needed to provide an extra layer of isolation to host OS kernel security tools without adding more code to the microhypervisor environment.

Sandboxing Approaches: Fides [50], Inktag [27] and AppShield [13] are examples of systems that use sandboxes to isolate security sensitive code from the OS kernel. A sandbox can be used to host kernel monitoring and protection tools. However, these techniques use virtualization to provide the isolation. Hence, they require the hypervisor to be actively involved in managing and scheduling the sandbox, which would also include monitoring and managing some kernel operations. Therefore, these approaches suffer from the same fundamental problem of virtualization, which is increasing the TCB size of the hypervisor.

Hardware Protection Approaches: Intel introduced Software Guard Extensions (SGX) [4], [26], [42], which allows the execution of verified code inside secure enclaves. These enclaves are isolated from the OS kernel. They also run in isolation from each other. SGX enclaves can be used to host security tools without increasing the TCB of higher privileged layers. Nevertheless, there is no similar protection for ARM. Instead, ARM provides TrustZone [6], which is a monolithic secure world that is isolated from the high level OS. Although TZ-RKP [11] and SPROBES [25] proved that TrustZone can be used to monitor the kernel, these approaches suffer from the fundamental problem of increasing the code base and maintenance effort of the high privileged TrustZone layer.

This brief review of previous work testifies that there is a problem in hosting OS kernel security monitoring and protection tools. Using the hypervisor to directly host the security tool or to manage a sandbox adds risk to the hypervisor security. On the other hand, no adequate hardware-based solution is available on ARM. Hence, ARM, which is the most widely used architecture in mobile devices, lacks a proper technique that provides isolation without adding risk to its higher privileged hardware components.

B. Introducing SKEE

This paper presents SKEE, which stands for Secure Kernel-level Execution Environment. SKEE is a lightweight framework that provides a secure isolated execution environment without relying on active involvement of a higher privileged layer. Nonetheless, SKEE achieves the same level of security and isolation required to host security tools that provide monitoring and protection for commodity OS kernels.

SKEE relies on a time sharing model where a CPU is either running in the OS kernel or in the isolated environment at any point of time. For convenience, we refer to this new isolated execution environment as SKEE and to the OS kernel being monitored as "the kernel." We also use the term "context switch" to exclusively refer to the operation of switching back and forth between the kernel and SKEE.

SKEE is designed to run on ARM. It bridges a critical gap in the security solutions available for ARM. SKEE raises the bar of OS security monitoring and protection without adding potential security risks and heavy maintenance cost to secure subsystems like TrustZone or virtualization extensions.

SKEE addresses an intuitive and straightforward security requirement of real world systems. It solves multiple technical challenges to achieve these goals. First of all, SKEE has to be perfectly isolated from the kernel. If an attack succeeds in compromising the kernel, it must not be able to compromise the security tool hosted by SKEE. This isolation is non-trivial to achieve given that both SKEE and the kernel are required to run at the same privilege level.

Second, SKEE is required to expose an interface that switches to the isolated environment so that security critical events are trapped into SKEE for inspection. This requirement adds extra challenge to the isolation mechanism because the context switching entry point is exposed to the potentially compromises kernel. Hence, the switch mechanism must be secure against all software attacks that aim to breach the isolation provided by SKEE.

Finally, SKEE is required to allow the security tool to inspect the kernel state to detect potential attacks. Hence, SKEE must provide a one-way isolation that allows security tools to access kernel memory. It also should prevent the kernel from handling certain events so that they are only handled by the security tool. This requirement is hard to achieve because modern OS kernels are designed to run at the highest privilege level of the system and control the entire system resources.

SKEE solves all these challenges. It provides a unique solution that is compatible with existing hardware platforms without using any special hardware extensions. Moreover, it does not interfere with the operation of other security mechanisms that run in TrustZone, virtualization extensions or even inside the OS kernel itself.

C. SKEE Overview

SKEE uses a set of novel techniques to achieve three key objectives: isolation, secure context switching, and the ability to monitor and protect the kernel, without involving a higher privileged layer.

Isolation: To achieve the required isolation, SKEE uses a two-step solution: *create a protected virtual address space for SKEE* and *restrict the kernel access to the MMU*.

The first step is to create a separate protected virtual address space for SKEE. The memory layout of the whole system is modified so that the memory regions used by SKEE are carved out of the memory ranges accessible to the kernel. This is done by modifying the memory translation tables used by the kernel so that none of the translation entries point to the physical memory regions used by SKEE. To protect this new address space, all memory translation tables must be part of this new protected address space so that they are exclusively accessible to SKEE. Therefore, the kernel cannot modify any of the memory translation tables to tamper the virtual memory access permissions.

The second step is depriving the kernel from controlling certain MMU functions so that it cannot direct the CPU to use alternative memory translation tables other than the ones

protected by SKEE. SKEE adopts a technique similar to the one presented in TZ-RKP [11]. It starts by instrumenting the kernel code to remove certain MMU control instructions, such as the ones that change the location of memory translation tables. SKEE also monitors memory layout changes to guarantee that no other unverified privileged code is allowed to execute.

By enforcing these two steps, the kernel is neither allowed to modify the existing memory translation tables nor change the MMU configurations to use unverified translation tables. As a result, it cannot violate the isolation provided to SKEE, which retains the exclusive access to control the MMU and memory translation tables in its own address space.

Secure Context Switching: SKEE context switching relies on the primitives of switching memory translation tables, which are commonly used in switching the execution between user processes. These operations were designed assuming that the context switch occurs in a higher privilege level, like the kernel to user processes, before the execution jumps to the new context. This is not the case of SKEE because context switching happens at the same privilege level.

To maintain the isolation, SKEE uses novel techniques that force the kernel to go through a designated switch gate to jump to the isolated environment. This gate is designed to enforce a strict execution flow that is both *atomic* and *deterministic*. The former is required to prevent the kernel from gaining control while the protected address space is accessible, while the latter is required to guarantee that switching to SKEE always passes through a designated entry point that contains all proper security checks. As a result, these two properties combined prevent exposing SKEE's address space to the kernel.

Instructions that control the MMU and switch the virtual address space are different between the 32-bit ARMv7 and the 64-bit ARMv8 architectures. Hence, SKEE has different switching mechanisms for each of these architectures.

Kernel Monitoring and Protection: SKEE provides the required capabilities to do effective monitoring and protection of the kernel. SKEE is allowed to access the entire system memory range. It can modify the kernel code to place hooks on certain operations. Moreover, SKEE controls virtual memory access permission so it can selectively protect certain data areas from the kernel. Therefore, the monitoring and protection provided by SKEE is comparable to that provided by virtualization-based isolation.

Prototype Overview: A prototype of SKEE is implemented on two commercial smartphones. The first is the Samsung Galaxy Note4, which uses the Snapdragon APQ8084 32-bit ARMv7 processor by Qualcomm. The second is Samsung Galaxy S6, which uses the Exynos 7420 64-bit ARMv8 processor by Samsung's System LSI. SKEE was subject to rigorous evaluation. The results show it is feasible to implement and it has an acceptable performance overhead.

The performance evaluation also shows that the number of CPU cycles required for switching to and from the isolated environment is in the range of few hundred cycles. This range is comparable to those required to switch the execution from one user process to the other. Hence, a security tool using SKEE can be always extended to support additional monitoring and protection mechanisms. Switching time to and from SKEE

```
if(va & (~0 << (BITS_PER_LONG - ttbcr.n))) {
    ttbr = ttbr1;
}
else {
    ttbr = ttbr0;
}
```

Figure 1. Selecting a TTBR on ARMv7 architecture

is much faster than switching time to and from the ARM TrustZone environment, which may reach up to thousands of CPU cycles [11]. Hence, SKEE is not only lightweight, but it can be also faster than TrustZone based systems, such as the ones presented in TZ-RKP [11] and SPROBES [25].

It is worth noting that SKEE is not designed to replace higher privileged layers, such as TrustZone or virtualization extensions. Although it is technically feasible to use SKEE to load custom secure applications or host security sensitive data, both TrustZone and virtualization extensions are more suited to achieve these objectives using their hardware-based protection. SKEE aims at keeping these layers more secure through reducing their code base and maintenance effort.

D. Summary of Contributions

This paper makes the following technical contributions:

- A lightweight practical solution for ARM platforms to provide kernel monitoring and protection without relying on higher privileged system components.
- Novel techniques to create a protected address space that is isolated from the kernel despite running at the same privilege level.
- Techniques to provide a secure, atomic and deterministic method to switch the context between the kernel and the isolated environment.
- Providing the isolated environment with the required capabilities to do effective kernel monitoring and protection.
- Full prototype implementation and rigorous evaluation of SKEE using popular mobile devices.

This paper is organized as follows: Section II provides background information. Section III discusses threat model, security guarantees and assumptions. Section IV presents SKEE in detail. Section V presents prototype implementation and evaluation. Section VI presents related work. Section VII concludes this paper with some future research directions.

II. Background

SKEE's isolation employs basic MMU operations. This section gives a necessary background on how the MMU controls virtual memory translation on ARM.

Memory Management in 32-bit ARMv7: On ARMv7, controlling the MMU is done through special instructions that move the value of general purpose registers to system management registers of coprocessor 15 (CP15).

Table I. EFFECT OF TTBCR.N ON ADDRESS TRANSLATION ON 32-BIT ARMV7 USING SHORT DESCRIPTOR FORMAT

TTBCR.N value	Starting address of TTBR1 translation
0b000	TTBR1 not used
0b001	0x8000_0000
0b010	0x4000_0000
0b011	0x2000_0000
0b100	0x1000_0000
0b101	0x0800_0000
0b110	0x0400_0000
0b111	0x0200_0000

Memory translation on 32-bit ARMv7 involves three MMU control registers: Translation Table Base Control Register (TTBCR), Translation Table Base Register 0 (TTBR0) and Translation Table Base Register 1 (TTBR1). TTBR0 and TTBR1 point to different sets of memory translation tables. TTBCR chooses which of the two sets is used when translating a particular memory address. TTBCR contains a 3 bit called TTBCR.N that determines the virtual address range translated by each of the two registers as shown in Figure 1.

Memory translation on ARMv7 supports both short descriptor and long descriptor translation table formats. For the sake of clarity, we only focus on the short descriptor translation table format. In this format, the effect of TTBCR.N is shown in Table I. If the value of TTBCR.N is 0, then TTBR1 is not used, otherwise both TTBR0 and TTBR1 are used. TTBR0 memory translation tables are used to map the virtual address range that starts from address 0x0 to the starting address of TTBR1's translation range, which is always smaller than 0x8000_0000 (2GB) as shown in Table I.

Memory Management in 64-bit ARMv8: On ARMv8, a.k.a. AArch64, the 64-bit virtual address range is split into two subranges. The first, which is translated using TTBR0, is at the bottom of the address space. The second, which is translated using TTBR1, is at the top of the address space. A typical use of this arrangement is that the kernel is mapped at the top subrange of the virtual address space using TTBR1 tables, while the user processes is mapped at the bottom subrange of the virtual address space using TTBR0 tables.

On ARMv8, the MMU control registers can be changed by the MSR instruction, which moves the value of general purpose registers to system registers. The MSR instruction can use the Zero Register (XZR) to move the value zero to any of the special registers.

Address Space Identifier (ASID): Memory translation tables also control if a certain virtual memory mapping is either global or non-global using the non-Global (nG) bit in translation table descriptors

A global virtual memory page is available for all processes on the system, so a single cache entry can exist for this page translation in the Translation Lookaside Buffer (TLB).

A non-global virtual memory page is process specific, meaning it is associated with a specific ASID. Hence, multiple TLB entries can exist for the same page translation. The software is expected so switch the ASID when switching between different processes. Only TLB entries that are associated with the current ASID are available to the CPU.

On ARMv7, the current ASID is defined by the Context ID Register (CONTEXTIDR). On ARMv8, the ASID is defined by the translation table base registers. The ASID is used to enhance the performance by eliminating the need of flushing the TLB on every process switch.

Memory Management in Virtualization Layer: ARM's virtualization extensions provide an additional mode of privileged execution to host the hypervisor. This privileged mode is also equipped with an additional memory translation layer, called Second Stage (S2) address translation, which is pointed to by the vttbr register. If S2 memory address translation is enabled, then physical memory access from the guest OS is treated as Intermediate Physical Addresses (IPA) and is translated to actual physical addresses using the S2 memory translation tables. S2 memory address translation is used by the hypervisor to customize the physical address range available to the guest OS.

III. THREAT MODEL, SECURITY GUARANTEES AND ASSUMPTIONS

Threat model: SKEE considers all software attacks against the kernel. It assumes attackers can successfully exploit existing kernel vulnerability. For the sake of presentation, attacks against the kernel are classified into three classes.

The first class aims at modifying, amending, or relocating the kernel executable. SKEE prevents this class of attacks as an essential part of its secure operations. Just using SKEE on a system eliminates the threat of running unverified malicious code in the privileged kernel mode.

The second class aims at exploiting a vulnerability to alter the kernel data or control flow so that existing kernel code shows unexpected malicious behavior. These attacks can cause a wide range of damage to the system. One example of these threats is to escalate the privilege of malicious user processes by modifying the kernel data that defines process credentials. Another example of these threats is return oriented attacks [29], [47] that allow an attacker to run malicious logic using existing kernel code. SKEE provides a safe environment that hosts security tools to detect these exploits. The exact anomaly detection technique or the integrity properties to be measured is determined by the security tool as an orthogonal system and is out of scope of this paper.

The third class aims at compromising kernel monitor and protection tools by compromising the SKEE isolated environment. SKEE guarantees that these attacks can neither compromise the isolation nor bypass the monitoring.

Security Guarantees: SKEE provides two main security guarantees to the isolated environment. First, it prohibits the kernel from modifying the memory layout or access permission of the system. As a result, even if an attack completely compromises the kernel, it would not be able to revoke the access protection of the isolated environment. Second, SKEE guarantees that switching from the potentially compromised kernel to the isolated environment exclusively passes through a strictly controlled switch gate. As a result, the isolated environment can safely inspect input parameters passed from the kernel for potential security threats. For example, SKEE inspects requested changes to the memory layout to confirm that they do not violate the guaranteed isolation.

Given these two security guarantees, SKEE uses its control of the system memory layout to prevent attackers from bypassing the monitor.

Assumptions: SKEE assumes the whole system is loaded securely. Hence, the isolated environment is setup securely during boot up time. This process is straightforward using secure boot. Intuitively, secure boot only guarantees the integrity of the kernel during the boot up process. It cannot guarantee the integrity of the kernel after the system runs and starts to interact with potential attackers.

SKEE also assumes that the kernel supports W⊕X memory mapping (i.e., it does not use memory pages that include both data and code). SKEE assumes that the hardware platform implements the Privileged eXecute Never (PXN) memory access permission as defined by the ARM architecture.

On 32-bit ARMv7 architecture, SKEE requires the kernel to only use `TTBR0` for mapping the OS memory, while leaving `TTBR1` to be exclusively used by SKEE. Moreover, it assumes the lowest 2GB of the virtual address space is exclusively used by non-privileged user space code. These two requirements do not affect the OS functionality because both `TTBR0` and `TTBR1` map the same virtual address range, so only `TTBR0` is sufficient to map the whole system memory. Most commodity OSes use at least 2GB of memory for the user address space. In fact, Linux satisfies both requirements in its default configurations because it relies on a 3GB user space to 1GB kernel space split, which cannot be achieved if `TTBR1` is in use.

On 64-bit ARMv8 architecture, SKEE requires the presence of a memory page at physical address 0x0. It assumes this particular page is exclusively used by SKEE. The same address range is usually used to place the ROM that starts the system boot process. Nevertheless, this requirement is easily fulfilled using virtualization to provide an accessible intermediate physical address 0x0 using S2 memory translation tables as mentioned in Section II. The S2 translation remaps this intermediate physical address to another existing physical address. The OS is forced to translate through the S2 tables, so it always sees this page as address 0x0. Higher privileged software still sees the original physical address 0x0 and can still use it in the booting process. Using virtualization in this case does not conflict with SKEE's objective. Other than setting up the S2 translation tables, the virtualization layer is completely passive and does not interfere in any SKEE related operations.

IV. SKEE Design

This section presents the design of SKEE. The main goal is to provide a lightweight execution environment to enable a security tool to run in isolation from the kernel without active involvement of higher privileged system components, such as TrustZone or virtualization layer.

The basic idea behind SKEE is to create a new self-protected virtual address space that hosts the isolated execution environment. This virtual address space is created as a part of the system boot up sequence. As mentioned is Section III, SKEE assumes the presence of secure boot, which guarantees that the system boots in a known secure state. The secure boot sequence is modified so that it creates two separate address spaces for the kernel and SKEE, as shown in Figure 2. The boot sequence also loads the verified binaries of both the kernel and SKEE in their relevant locations.

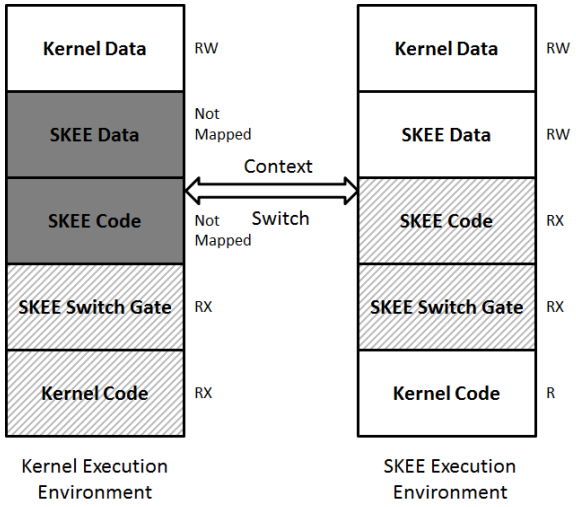

Figure 2. An overview of the SKEE approach

After the secure boot sequence, the system becomes subject to potential attacks. These attacks cannot compromise SKEE, which runs in its own address space. Hence, a security tool hosted by SKEE would be able to continuously monitor the kernel to detect, and possibly eliminate, these attacks.

To securely achieve this objective, there are three main requirements that need to be met; isolation, secure context switching and the ability to monitor and protect the kernel.

In the following, we first present how SKEE achieves the required isolation by preventing the kernel from accessing this protected address space. Then, we present how SKEE achieves secure context switching on both 32-bit ARMv7 and 64-bit ARMv8 architectures. Afterwards, we discuss how SKEE uses these features to achieve the required monitoring and protection of the kernel. Finally, we summarize the security guarantees provided by SKEE.

A. SKEE Isolation

The kernel accesses physical memory through virtual memory mappings defined by memory translation tables, a.k.a. page tables. These mappings also set the access permission corresponding to each translation. The presence of a translation table entry that maps to a physical address is a key precondition to the kernel's ability to access this physical address. If this precondition is not met, this physical address is not accessible to the kernel. Based on this precondition, SKEE uses a two-step solution to prevent the kernel from accessing certain physical memory ranges.

Creating a Protected Address Space: The first step is to have separate address spaces for SKEE and the kernel. As shown in Figure 3, the kernel address space, which is controlled by the kernel's memory translation tables, is instrumented to enforce the following rules: 1) removing all entries that map to either the SKEE environment or the kernel's memory translation tables, 2) mapping the kernel code and the SKEE switch gate

as read-only, 3) restricting all other memory areas, including kernel data and user level memory, from executing privileged code using the PXN bit. Enforcing these rules prevents the kernel from modifying its own code or accessing SKEE's address space. It also prevents the kernel from modifying its own memory translation tables to escape this protection.

On the other hand, SKEE's address space has valid mappings of the entire memory. However, the kernel code is not allowed to execute so that the kernel is prevented from regaining control while the SKEE address space is active. The switch gate is also mapped in the SKEE address space and has execution permission to allow SKEE to jump back to the kernel securely.

Restrict Kernel Access to the MMU: To prevent the kernel from violating the address space separation, it is only allowed to use instrumented memory translation tables. This objective is achieved by restricting the kernel from modifying certain MMU registers. In particular, the kernel is not allowed to change translation table base registers.

SKEE removes certain control instructions from the kernel code and replaces them with hooks that jump to the switch gate. Identifying specific instructions among the kernel binary is straightforward (i.e., without false positives or false negatives) because ARM uses fixed size aligned instruction set.

This technique requires the following four conditions: 1) these instructions are only allowed to execute in the privileged mode, 2) the instrumented kernel is the only privileged code in its own address space, 3) the kernel code is mapped read-only, and 4) the instrumented kernel code is instruments to remove all executable words that match the op codes of these instructions. If all these four conditions are satisfied, then the kernel cannot execute these instructions unless they are emulated by SKEE. The same technique was used by TZ-RKP [11] and SPROPES [25].

After the system is booted, the Linux kernel supports Loadable Kernel Modules (LKM) to be dynamically loaded. LKM code runs in the same privilege level as the kernel. SKEE supports LKMs to be loaded as long as they stick to the W⊕X mapping so that they are not be used to inject unknown instructions to the kernel. LKMs have to be loaded by SKEE because it controls the whole system's memory mappings. Hence, SKEE is able to verify that the LKM code region does not contain any of the privileged instructions that were removed from the kernel.

B. SKEE Secure Context Switching

To allow secure switching between SKEE and the kernel, the switching mechanism prevents the kernel from regaining control while SKEE's address space is accessible.

As shown in Figure 2, context switching is done through a special switch gate. This gate is designed to be *atomic*, *deterministic* and *exclusive*. Having these properties is essential to preserve the isolation between the two environments.

Atomic Execution through the Switch Gate: Before the kernel enters the switch gate, the SKEE address space is not accessible. The switch gate first exposes the SKEE address space and then jumps to SKEE. Since both SKEE and the kernel run at the same privilege level, there is no single point of entry or hardware control to do the switching in one instruction. Instead, a sequence of instructions is used to perform this function. SKEE guarantees that attempts to manipulate this sequence of instructions will not expose its address space to the kernel.

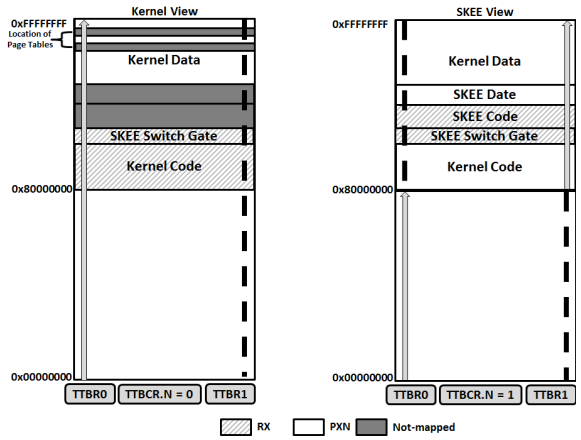

Figure 3. An example of address Space Separation on ARMv7. The vertical arrows show the virtual memory range translated via the corresponding translation table base register. The dashed lines show the memory range that is not addressable via the corresponding translation table base register.

A potentially compromised kernel can attempt to attack the switch gate in two ways: 1) interrupting the gate's execution sequence or 2) jumping to the middle of this sequence. SKEE guarantees that these attacks cannot return execution back to the kernel while SKEE's memory is accessible.

Deterministic Execution Sequence: Although the switch gate is accessible to the kernel, it does not trust the system state or any input from the kernel. Hence, its execution sequence is deterministic in the sense that it has the same behavior regardless of the current system state and input parameters.

Exclusive Access to Switching Functions: The switch gate is the only entry point to SKEE. It relies on instructions that are restricted to execute anywhere else in the kernel.

1) Secure Switching on 32-bit ARMv7: ARMv7 active memory translation tables are defined by the translation table base registers `TTBR0` and `TTBR1`.

To switch between two different address spaces, the corresponding translation table base register has to be updated accordingly. This update can only be done by an `MCR` instruction that moves the value of a general purpose resister (GPR) to either `TTBR0` or `TTBR1`. If the switch gate exposes a single instance of such `MCR` instructions, then a compromised kernel can compromise the isolation by creating a malicious set of page tables, loading the base address of these tables to the general purpose register, and jump to that `MCR` instruction to have an unrestricted address space.

To solve this challenge, the switch gate does not include instructions that update either `TTBR0` or `TTBR1`. Instead, it relies on changing the active translation table register by updating the `TTBCR.N` field. The basic idea is that any non-zero value that gets loaded into `TTBCR.N` will lead to the same system state. Therefore, switching is always deterministic

```
1  /* Start of the SKEE Entry Gate */
2  mrs    r0, cpsr                // Read the status register
3  push   {r0}                    // Save the status register value
4  orr    r0, r0, #0x1c0          // Set the mask interrupts bits
5  msr    cpsr, r0                // load the modified value
6
7  mov    r0, #0x11
8  isb                            // Syncronization barrier
9  mcr    p15, 0, r0, c2, c0, 2   // Modify the TTBCR to activate SKEE
10 isb
11
12 mcr    p15, 0, r0, c8, c7, 0   // TLB invalidate
13 isb
14
15 bl     skee_entry              // Jump to SKEE entry point
16 /* End of the SKEE Entry Gate */
17
18 /* Start of the SKEE Exit Gate */
19 mov    r0, #0
20 isb
21 mcr    p15, 0, r0, c2, c0, 2   // Modify the TTBCR to deactivate SKEE
22 isb
23
24 mcr    p15, 0, r0, c8, c7, 0   // TLB invalidate
25 isb
26
27 pop    {r0}                    // Reload status register value
28 msr    cpsr, r0                // Restore the original status register
29
30 bl     kernel_entry            // Jump back to the kernel
31 /* End of the SKEE Exit Gate */
```

Figure 4. SKEE Switch Gate on ARMv7

even if the kernel attempts to maliciously modify the general purpose registers before jumping to the switch gate.

Figure 3 shows on overview of address space separation on the 32-bit ARMv7 architecture. SKEE enforces two rules on the OS kernel: 1) The kernel only uses `TTBR0` to map the OS memory, while `TTBR1` is dedicated to SKEE, and 2) the virtual memory address range smaller than 0x8000_0000 (2GB) is assigned to user space applications restricted by the Privileged Execute Never (PXN) memory translation flag.

Given these two rules are enforced; the switch gate only needs to modify `TTBCR.N` to switch to and from SKEE. When switching to SKEE, `TTBCR.N` is set to a non-zero value to force translation of all virtual addresses larger than 0x8000_0000 to reference `TTBR1` instead of `TTBR0`. Since both the kernel and SKEE lie in this range, this means that only SKEE address space will be mapped and the kernel address space will be unmapped. Resetting `TTBCR.N` back to 0 will force translation of the entire address space to use `TTBR0`. Hence, the SKEE address space, which is only mapped through `TTBR1`, will be hidden from the kernel.

Figure 4 shows the switch gate on ARM v7. The code of the gate works as follow:

1) ***Lines 2-5:*** Disable interrupts.
2) ***Lines 7-10:*** Load `TTBCR` with a non-zero value to enable `TTBR1` to be used for translating virtual memory above address 0x8000_0000 (2GB).
3) ***Lines 12 and 13:*** Invalidate the TLB so that the new memory translation takes effect.
4) ***Line 15:*** Jumps to SKEE after the isolated environment is activated.

This switch gate is *atomic*, *deterministic* and *exclusive*; hence it guarantees all three SKEE security requirements.

First, the gate execution is deterministic. Steps 1, 3, and 4 do not rely on memory or register values. Hence, the kernel cannot alter the results of their execution. It is worth noting that the addressing layout has to be carefully selected so that an immediate branch instruction (i.e. a one that does not rely on registers) is used on Step 4. Otherwise, the kernel would be able to jump to any instruction in the SKEE code base.

Due to the unique use of the `TTBCR`, the `MCR` instruction in step 2 yields a deterministic result regardless of the value loaded to the GPR. The effective field of this register is `TTBCR.N`, which can be loaded by 8 different values. Nevertheless, all non-zero values of `TTBCR.N` will switch to the SKEE address space and deactivate the kernel address space. As shown in Figure 3, loading `TTBCR.N` with the value 1 makes the address range above 0x8000_0000 (2GB) translated through `TTBR1`. If the kernel loads this field with another non-zero value, it will only extend the range of the memory translated by `TTBR1` and hence cause no threat to the isolation scheme. If `TTBCR.N` is loaded with 0, then `TTBR1` will not be used and the SKEE address space will not be exposed.

In all cases, the memory range below 0x8000_0000 (2GB), which is always translated using `TTBR0`, is mapped with the PXN restriction. Therefore, this memory range does not have to be trusted since it cannot run in the privileged mode or access the SKEE privileged memory area.

`TTBCR` has three other fields: `TTBCR.PD0`, `TTBCR.PD1`, and `TTBCR.EAE`. The first two have no effect on the protection. They can only prohibit translation table walks. The last field switches between the long and the short descriptor address formats. If the wrong value is put into this field, the system is going to crash due to the wrong format of translation tables. Hence, a wrong value in any of these fields can only affect the availability of SKEE, but it does not threaten its protection.

Second, the gate execution is atomic. The control flow cannot change because no branch or return instructions exist until switching is complete. Nevertheless, an attacker can jump to any step in the sequence above. The one interesting step to skip is step 1 as this will not disable interrupts. Skipping step 1 will allow the kernel to set interrupts, such as watchdog bark, to be triggered at any of the following steps. If an interrupt is triggered between steps 2 and 3, the execution will return to the kernel while the SKEE address space is exposed. Although the kernel translation tables will not be in use, the kernel code can still execute at this point relying on existing TLB entries.

To resolve this issue, a security check is added to interrupt handlers of the system to test the value of `TTBCR.N`. If `TTBCR.N` has a non-zero value, then this attack scenario is detected and the system will be stalled to prevent further malicious execution. After step 3, interrupts will only cause the system to stall because the handler, which is part of the kernel code, will be located in non-executable memory. An adversary can neither change the code nor the location of interrupt handlers because SKEE restricts the kernel access to the Vector Base Address Register (`VBAR`). `VBAR` will always point to verified read-only kernel code.

Switching to SKEE is exclusive to the switch gate because it has the only executable opcode that can modify `TTBCR`.

Figure 4 also shows the exit gate from SKEE to the kernel. It executes the same operations, in almost reverse order, so that execution only goes back to the kernel after the SKEE address space is locked down.

```
1  /* Start of the SKEE Entry Gate */
2  mrs    x0, DAIF            // Read interrupt mask bits
3  str    x0, [sp, #-8]!      // Save interrupt mask bits
4  msr    DAIFset, 0x3        // Mask all interrupts
5
6  mrs    x0, ttbr1_el1       // Read existing TTBR1 value
7  str    x0, [sp, #-8]!      // Save existing TTBR1 value
8
9  msr ttbr1_el1, xzr         // Load the value Zero to TTBR1
10 isb
11
12 tlbi vmalle1               // Invalidate the TLB
13 isb
14
15 adr x0, skee_entry         // Jump to SKEE entry point
16 br x0
17 /* End of the SKEE Entry Gate */
```

Figure 5. SKEE Entry Gate on ARMv8

```
1  /* Start of the SKEE Exit Gate */
2  nop                        //no operation
3  nop                        // Fill the page with no operations to
4  nop                        // align the last instruction with the
5  nop                        // bottom of the isolated page boundry
6
7  msr    DAIFset, 0x3        // Mask all interrupts
8
9  ldr x0, [sp, #8]!          // Reload kernel TTBR1 value
10 dsb sy
11 msr ttbr1_el1, x0          // Restore TTBR1 to kernel value
12
13 /*------------Isolated Page Boundry--------------*/
14
15 isb
16 tlbi vmalle1               // Invalidate the TLB
17 isb
18
19
20 ldr x0, [sp, #8]!          // Reload interrupts mask bits
21 msr DAIF, x0                // Restore interrupts mask bits register
22
23 ret
24 /* End of the SKEE Exit Gate */
```

Figure 6. SKEE Exit Gate on ARMv8

The exit gate is part of the switch gate page, which is accessible to the kernel. In line 21, the exit gate exposes another instruction that writes `TTBCR` to the kernel. This instruction is protected the same way as the entry gate. If the kernel directly jumps to this instruction to enable the SKEE address space, the following TLB invalidation will cause the kernel space to be evicted from the TLB and loose execution permission. Hence, the system will fail when it tries to return back to the kernel in line 30. Intuitively, the switch gate is required to use an immediate branch instruction in line 30 to guarantee that the system will be stalled rather than branching to random instructions in the SKEE code base. Moreover, if an interrupt is triggered right after the `TTBCR` write and before the TLB invalidation, then execution will be stalled by the check on the interrupt handler as discussed previously.

2) SKEE Secure Switching on 64-bit ARMv8: `TTBR0` and `TTBR1` are used to map different virtual address ranges on 64-bit ARMv8. `TTBR1` is designated for kernel addressing and `TTBR0` is designated for user space addressing. Restricting the OS to use one translation table base register and leave the other to SKEE requires considerable modifications to the OS, which will negatively impact SKEE's portability.

To solve this problem, SKEE shares `TTBR1` with the kernel. Therefore, switching between the two different address spaces requires the value of `TTBR1` to be modified. This is done by the `MSR` instruction, which moves the value of a general purpose register to a special register, such as `TTBR1`. As mentioned previously, allowing the kernel to change the value of the translation table base registers using a general purpose register does not guarantee deterministic execution.

To solve this challenge, the switch gate uses a special `MSR` encoding to guarantee deterministic change of `TTBR1`. This special encoding relies on the Zero register (`XZR`), which is a special register that is always read as 0.

On ARMv8, `TTBR1` has two fields: `BADDR` and `ASID`. Moving `XZR` to `TTBR1` will set both fields to 0. Setting `BADDR` to 0 means that the active memory translation tables are based at physical address 0x0. As mentioned in Section III, SKEE requires the presence of this physical address to support ARMv8 platforms. If address 0x0 is not part of the physical memory layout available to the OS, then it can be virtualized using S2 translation tables as explained in Section III. The physical page at address 0x0 is used to host the memory translation tables of the SKEE address space and will not be accessible to the kernel's address space.

Figure 5 shows the entry path of the switch gate on ARM v8. The code of the gate works as follow:

1) ***Lines 2-4:*** Disable Interrupts.
2) ***Lines 6-10:*** Save the kernel's `TTBR1` value and load `TTBR1` with 0 using `XZR`.
3) ***Lines 12 and 13:*** Invalidate the TLB so that the new memory translation takes effect.
4) ***Lines 15 and 16:*** Jump to SKEE after the isolated environment is activated.

This switch gate is *atomic*, *deterministic* and *exclusive*; hence it guarantees all three SKEE security requirements.

First, the gate is guaranteed to be deterministic because it absolutely relies on no memory or register values. In particular, the write to `TTBR1` uses `XZR`, which cannot be modified by the kernel. Hence, `TTBR1` will always point to address 0x0.

Second, the gate execution is guaranteed to be atomic using a similar technique to the one used for ARMv7's gate. If an interrupt is received while `TTBR1` points to address 0x0, then it means that a compromised kernel skipped step 1 of the switch gate and the system will be stalled.

Finally, switching to SKEE is exclusive to the switch gate because it has the only executable opcode that can modify `TTBR1`. Nevertheless, there is a key challenge to meet this objective; the switch gate is required to restore the `TTBR1` value upon exiting from SKEE to the kernel. This step requires the presence of the executable opcode of an `MSR` instruction that writes a non-zero value to `TTBR1`. As mentioned previously, if this instruction is exposed to the kernel, then it can compromise SKEE's isolation.

SKEE solves this problem using a novel technique that hides the exit gate before execution jumps to the kernel. This is achieved by placing the instruction that restores `TTBR1` at the end of a physical memory page that belongs to the isolated environment. This page is never exposed to the kernel. The following memory page, which is accessible to both SKEE and the kernel, is responsible for restoring interrupts before returning to the kernel.

Figure 6 shows the exit path of the switch gate on ARM v8. The code of the gate works as follow:

1) ***Lines 2-5:*** Pad the memory with a sequence of no operation (`NOP`) instructions so that the instruction that restores `TTBR1` is pushed to the boundary of the isolated page.
2) ***Line 7:*** Confirms that interrupts are masked.
3) ***Lines 9-11:*** Reload the kernel's `TTBR1`. When this instruction is executed, the isolated page will not be accessible and the program counter will point to the next page that is accessible to both environments.
4) ***Lines 15-17:*** Invalidate the TLB so that any cached translation of SKEE's address space is discarded.
5) ***Lines 20-23:*** Restore interrupts and return to the kernel.

Step 1 is to add padding to so that the instruction that writes `TTBR1` is placed at the boundary of the isolated environment.

Step 2 guarantees that interrupts are disabled. Since the exit gate is exclusively available to the trusted SKEE environment, there is no risk that the execution flow will skip this step and directly jumps to following steps.

Step 3 switches the address space from SKEE to the kernel. When Line 11 is fetched for execution, the program counter will already be pointing to the next instruction, so no fault will be caused due to the removal of the isolated page from the accessible address space. Nevertheless, the address layout of the exit switch gate has to be carefully crafted. If a single instruction is placed on the isolated page after the `TTBR1` value is restored, then a TLB miss may occur and the system would lock up. This is particularly possible in multi-core environments, where other cores can be invalidating the TLB of all cores at any point of time.

Steps 4 and 5 are carried out at the page mapped directly after the isolated page. This page has the same virtual address mapping in the two address spaces. Although this page has no instructions that change `TTBR1`, the context switching effectively takes place at this page. The execution goes to the kernel after `TLB` is invalidated and interrupts are enabled.

The entire ARMv8 switching mechanism does not rely on the value of `TTBR0`. Nonetheless, SKEE is responsible for emulating writes to this register within its address space. SKEE enforces two restrictions on `TTBR0` mappings: 1) All mappings are forced to have the PXN access restriction so they cannot be used to control the MMU, and 2) none of the mappings point to SKEE's physical memory.

3) Using ASID for Faster Context Switching: Both ARMv7 and ARMv8 switch gates mostly rely on basic hardware operations that are not expected to cause large performance overhead. The only exception is TLB invalidation, which forces the CPU to reload all memory translations through page table walks. To avoid this potentially expensive step, SKEE proposes an alternative design that relies on the ASID to protect its address space. This solution requires two changes to the SKEE algorithm.

First, the entire SKEE address space is to be mapped as non-global memory. Therefore, cached TLB entries are only available when a particular ASID is active. The switch gate would still need to be mapped as global memory because it is accessible to both address spaces.

Second, a unique ASID is assigned to SKEE. This unique ASID is never active while the kernel is running. Otherwise, the kernel can access the SKEE address space using existing TLB entries. Likewise, other ASIDs are not used while SKEE is running so that cached entries of the SKEE address space mappings do not leak to the kernel.

On ARMv8, implementing this solution is straightforward because the ASID is assigned along with the translation table register values. In this case, ASID 0 is assigned to SKEE, while all other ASIDs are assigned to the OS.

ARMv8 allows the OS to select which translation table base register defines the active ASID. SKEE will enforce that the active ASID value is associated with `TTBR1`. This selection is done by the `TCR`, which is only accessible to SKEE. The only change required thereafter is for the entry and exit gates to skip the TLB invalidation. All SKEE address space translations will be associated with ASID 0 in the TLB. Upon exiting, SKEE will verify that the restored `TTBR1` value has a non-zero ASID field. Hence, all SKEE cached translations are not accessible to the kernel.

It is worth noting that in our SKEE prototype, the exit gate was mapped as a non-global page. Nevertheless, some ARMv8 implementations might require the exit gate to be mapped as a global page to be able to change `TTBR1` value. In this case, the exit gate should invalidate the global TLB entry corresponding to its own virtual address. This is an extremely fast operation when compared to invalidating the entire TLB.

On ARMv7, the solution is more complex because the ASID is defined by `CONTEXTIDR`. Hence, selecting the active ASID and modifying `TTBCR` cannot be done in an atomic operation. Figure 7 shows the redesigned ARMv7 switch gate that relies on the ASID. ASID 0 is assigned to SKEE and other ASIDs are used by the kernel.

Upon entry, `CONTEXTIDR` is set to 0 before switching `TTBCR` to guarantee that SKEE address space entries are only associated to that ASID. Upon exit, the kernel ASID is only restored after `TTBCR` switches back to the kernel.

The kernel can use the newly exposed instructions that write `CONTEXTIDR` to switch to ASID 0 to expose available TLB entries that map the SKEE address space. To prevent this scenario, a security check is added to interrupt handlers to test the value of `CONTEXTIDR`. If `CONTEXTIDR` has a zero value, then this attack scenario is detected and the system will stall.

The kernel may also skip the step that writes to `CONTEXTIDR` and switch the `TTBCR` directly aiming to let SKEE's address space mappings leak to a different ASID. SKEE uses the security check at lines 19-22 to prevent this because the execution will be halted before jumping to SKEE.

The same security check is added to the exit gate to guarantee that neither writes to `TTBCR` or `CONTEXTIDR` is maliciously used. The two checks at lines 33-36 and lines 42-45 verify `CONTEXTIDR` and `TTBCR` respectively and stall the execution if an unexpected value is detected. Intuitively, firing an interrupt to skip the security checks will not work due to the presence of similar check at the interrupt handler.

```
/* Start of the SKEE Entry Gate */
push   {lr}                    // save the return address
mrs    r0, cpsr                // read the status register
push   {r0}                    // save the status register value
orr    r0, r0, #0x1c0          // set the mask interrupts bits
msr    cpsr, r0                // load the modified value

mrc    p15, 0, r0, c13, c0, 1  // read current CONTEXTIDR
push   {r0}                    // save the CONTEXTIDR value
mov    r0, #0
mcr    p15, 0, r0, c13, c0, 1  // set CONTEXTIDR to 0
isb

mov    r0, #0x11
isb                            // syncronization barrier
mcr    p15, 0, r0, c2, c0, 2   // modify the TTBCR to activate SKEE
isb

L1:
mrc    p15, 0, r0, c13, c0, 1  // read current CONTEXTIDR
cmp    r0, #0                  // compare current CONTEXTIDR with 0
bne    L1                      // loop back if CONTEXTIDR is not 0

bl     skee_entry              // jump to SKEE entry point
/* End of the SKEE Entry Gate */

/* Start of the SKEE Exit Gate */
mov    r0, #0
isb
mcr    p15, 0, r0, c2, c0, 2   // modify the TTBCR to deactivate SKEE
isb

L2:
mrc    p15, 0, r0, c2, c0, 2   // read current TTBCR
cmp    r0, #0                  // compare current TTBCR with 0
bne    L2                      // loop back if TTBCR is not 0

pop    {r0}                    // reload CONTEXTIDR value
mcr    p15, 0, r0, c13, c0, 1  // write original CONTEXTIDR
isb

L3:
mrc    p15, 0, r0, c13, c0, 1  // read current CONTEXTIDR
cmp    r0, #0                  // compare current CONTEXTIDR with 0
beq    L3                      // loop back if CONTEXTIDR is 0

pop    {r0}                    // reload status register value
msr    cpsr, r0                // restore the original status register

pop    {lr}                    // reload the return address
movs   pc, lr                  // jump back to the kernel
/* End of the SKEE Exit Gate */
```

Figure 7. A Faster SKEE Switch Gate on ARMv7

These security checks also eliminate the need for an immediate branch to return to the kernel.

A key security issue of this technique is the presence of the kernel address space translation entries in the TLB while SKEE is executing. The main threat is that execution may maliciously be diverted to the kernel code by exploiting vulnerability in the security tool hosted by SKEE.

These are two possible solutions to prevent this threat: 1) modify the kernel address space to put the kernel code in a non-global memory so the TLB entries are not accessible while SKEE is running, and 2) use existing sandboxing techniques, such as Native Client [54] and MiniBox [36], or Control Flow Integrity (CFI) techniques, such as the CFI enforcement initially proposed by Abadi et. al. [3] and MoCFI [20], to prevent control flow attacks against the SKEE environment.

The first solution will add performance overhead due to increased TLB usage to cache kernel code with multiple ASIDs. The resulting overhead in this case will likely offset the performance enhancement gained by avoiding the TLB invalidation. Hence, it is not a feasible solution from the performance perspective. The second solution is more feasible because the code base of the security tools is smaller than that of the kernel. Hence, implementing sandbox isolation or CFI is feasible. Nevertheless, this solution requires techniques orthogonal to the work presented in here. Implementing and evaluating these techniques is out of scope of this paper.

C. Kernel Monitoring and Protection

To allow effective monitoring and protection of the kernel, SKEE provides the security tool with: 1) the ability to trap kernel critical events, 2) the ability to access kernel memory, and 3) the ability to control kernel memory protection.

SKEE's control of the kernel's virtual address space allows it to force the kernel to trap on certain operations by modifying the access permission of memory regions associated with these operations. For example, all memory translation tables are mapped read-only to the kernel. Hence, the kernel is forced to request from SKEE to update the memory translation tables. Similarly, hooks can be placed to intercept other events, such as modification of security critical data structures.

SKEE can also remove any particular privileged instruction from the kernel code and replace it with a hook that traps to SKEE. These hooks can be placed at kernel critical execution paths such as interrupt handlers or system call handlers.

To conclude, the privilege allowed to SKEE is equivalent to that allowed to virtualization based mechanisms. SKEE has the advantage that it does not increase the size of the TCB of commodity hypervisors. SKEE also can work on ARMv7 systems that do not support virtualization extensions.

D. Security Analysis

Throughout section IV, we discussed in detail how SKEE achieves isolation, secure context switching and kernel monitoring. In this section, we first summarize how these features fulfill the required security guarantees. Afterwards, we discuss how SKEE prevents other possible attack scenarios.

Security Guarantees: As mentioned in Section III, SKEE provides two principal security guarantees. First, it guarantees that the kernel cannot break the isolation. Second, it guarantees that switching from the kernel to the isolated environment cannot expose the address space protection.

Section IV-A shows how SKEE uses the MMU to provide the isolation. The memory layout defined by SKEE prevents the kernel from accessing the isolated environment. Moreover, the entire kernel address space lacks the required privileged instructions to control the MMU to revoke this protection.

Section IV-B shows how all context switching scenarios are atomic, deterministic and exclusive. These features combined guarantee that the isolated environment is only accessible after it takes control of the system. They also guarantee that this only happens at a specific entry point and in specific execution conditions. Hence, the kernel cannot tamper with the context switching process to break the isolation.

Multi-core System Operations: ARM architecture specifications use a separate TLB for each CPU core. Hence, SKEE and the kernel can run simultaneously on different cores. The TLB entries cached on one core that runs SKEE are not available to other cores that might be running the kernel and vice versa. Therefore, SKEE is safe to use in multi-core systems.

If a specific implementation supports a shared TLB, then SKEE will be required to use ASIDs for isolation. As discussed in Section IV-B3, having a separate ASID for SKEE will prevent the kernel from accessing existing TLB entries that map the SKEE address space.

Side Channel Attacks: Due to the lack of hardware protection, SKEE does not provide a guarantee against side-channel attacks. Nevertheless, the effect of these attacks is limited to leaking information about the SKEE environment without the ability to alter its operations or break the isolation.

DMA Attacks: Hardware peripherals are sometimes allowed to bypass the MMU and do Direct Memory Access (DMA) to physical memory. This feature can be used by attackers to read or write arbitrary memory regions. These attacks threaten SKEE because a compromised kernel can reprogram hardware peripherals to directly write to the SKEE address space.

On hardware platforms that support ARM System Memory Management Unit (SMMU) [7], preventing DMA attacks against SKEE is straightforward. SKEE is first required to prevent the kernel from managing the SMMU registers and page tables using the techniques discussed in Section IV-A. Afterwards, SKEE would use the ARM SMMU to restrict DMA access to the isolated environment.

On hardware platforms that do not support ARM SMMU, SKEE needs to further instrument the memory layout so that the kernel cannot access the DMA controller of hardware peripherals. This can be done by remove the mapping of the particular control structure of the target device from the kernel address space. In this case, the exact implementation will differ according to the specifications of the used hardware platform.

Attacks against the Isolated Environment: If the kernel passes a maliciously crafted input that exploits vulnerability in the SKEE framework or in the hosted security tool to hijack SKEE's control flow, then it can use the SKEE code base to break the protection. Nevertheless, the exact same risk faces all kernel security monitoring and protection tools.

In fact, SKEE profoundly enhances the system security in this case. If vulnerability exists in the hosted security tool, then the extent of the attack will be limited to the same privilege level of the kernel. On the other hand, if the same security tool is hosted by the hypervisor or by TrustZone, then such attack would have an even higher impact by compromising these security sensitive system components.

Dynamically Generated Kernel Code: Some kernel modules, such as BSD packet filtering [38], dynamically generate kernel code. These modules pose a threat to SKEE because they require the kernel to have access to memory pages that are writable and executable. Hence, they can be used to dynamically generate executable privileged instructions that allow the kernel to control the MMU.

To solve this problem, SKEE can prevent the kernel from writing to the code pages that contain the dynamically generated code. Instead, the kernel would be required to pass the code to SKEE so that it gets inspected first before being written to the executable memory ranges. SKEE would then confirm that the dynamically generated code does not have any instance of the restricted privileged instructions.

V. Implementation and Evaluation

We implemented two prototypes of SKEE. The first prototype was developed for the 32-bit ARMv7 architecture. It was tested on the Samsung Note4 smartphone, which uses Sanspdragon APQ8084 processor from Qualcomm. The second prototype was developed for the 64-bit ARMv8 architecture. It was tested on both the Samsung Galaxy S6 and the Samsung Galaxy Note5 smartphones, which use the Exynos 7420 processor from Samsung System LSI.

In both prototypes, the kernel is modified so that the SKEE environment is initialized during the boot up sequence. This includes creating a new memory translation tables for SKEE as well as modifying the kernel's memory translation tables to exclude the SKEE address space. This step is trusted because SKEE assumes the presence of secure boot protection.

The kernel is modified to place hooks upon modifying memory translation tables or MMU control registers. The hooks jump to SKEE through the switch gate. The MMU control registers emulated by SKEE include translation table base, context ID and vector base address registers. SKEE does not allow these operations to be carried out by the kernel; Bypassing these hooks will only cause the system to stop functioning properly. It is worth noting that the kernel disables the MMU when the CPU core is coming in and out of the sleep mode. Hence, the sleep/wake up sequence needs to be modified to go through SKEE to guarantee that no attack code can be launched while the MMU is disabled.

SKEE must confirm that translation table updates and control register modifications requested by the kernel do not compromise the address space isolation. The verification technique to be used is out of the scope of this paper. Existing techniques, such as TZ-RKP [11], can be used for this purpose. In order to understand the bare SKEE overhead as well as the overhead with security checks, we ran two groups of experiments. In the first group, SKEE emulates requests received from the kernel without security checking. In the second group, SKEE checks that the emulated requests do not modify the memory layout in a way that compromises the isolation.

A. Overhead of Emulating System Events

In the first group of experiments, we measured the overhead of emulating system events using SKEE, which represents the minimum overhead required to create SKEE's isolated environment on any system. In this group of experiments, SKEE does not do any verification on the emulated system events. It is important to measure the security impact of bare SKEE implementation without security checks or any hosted security tools for two main reasons.

The first is to measure the cost of running security tools inside SKEE versus the cost of running the same security tools without SKEE's protection. Since SKEE runs in the normal world alongside the kernel, the performance impact of executing code inside or outside the isolated environment is the same. Hence, the real performance impact is the time added to enter to and exit from the isolated environment.

The second is that there is a plethora of system monitoring and protection tools. These tools can range from simple boundary checks that virtually add no overhead to complex intrusion detection systems that require extensive processing. Hence, it is important to measure the bare overhead of creating SKEE's isolated environment to be able to extrapolate the performance cost of hosting any security tool inside SKEE.

Table II. SWITCHING TIME

Processor	Average Cycles
ARMv7	868
ARMv7 (No TLB invalidation)	550
ARMv8	813
ARMv8 (No TLB invalidation)	284

The experiments presented in this section were done using both 32-bit Samsung Galaxy Note4 and 64-bit Samsung Galaxy S6 smartphones. We created three custom images for each device: a non-modified Android system, a test system that supports SKEE using TLB invalidation and a test system that supports SKEE using ASID protection. The target OS was Android Lollipop version 5.0, which ran on Linux kernel version 3.10.61.

Overhead of Switching to SKEE: The first experiment is measuring the execution time needed for context switching. In this experiment, we used both the system that uses TLB invalidation and the system that relies on ASID. We used ARM cycle count register (`CCNT`) to measure the full round trip from the kernel to SKEE.

Table II shows the average number of cycles needed to do context switching. We run the same test on both ARMv7 and ARMv8 versions. Each test is repeated twice, one with the full switch gate that includes TLB invalidation and the other using the ASID protection. The purpose is to estimate the performance of a system that might use sandboxing techniques to securely skip the relatively expensive TLB invalidation. It is worth noting that the effect of the TLB invalidation is not limited to the added execution time on the switch gate, but it also has a side-effect on other system operations. Table II also shows that switching the ASID is more expensive in the case of ARMv7 compared to ARMv8. This can be attributed to the multiple steps required to switch the ASID on ARMv7 compared to the single atomic step on ARMv8.

Benchmark Performance Comparison: The second experiment is to use benchmarking tools to evaluate the performance overhead of the SKEE prototype described above. Benchmarking the performance was only done on the system that uses TLB invalidation. We did not evaluate the ASID based protection because it is not complete without adopting a sandboxing mechanism that prevents returning to the kernel to hijack SKEE's control flow. The system that uses TLB invalidation gives a perspective on the worst case performance of a system adopting SKEE.

Table III. SKEE BENCHMARK SCORES ON ARMV7

Benchmark	Original	SKEE	Degradation (%)
CF-Bench	30933	29035	6.14%
Smartbench 2012	5061	5002	1.17%
Linpack	718	739	-2.93%
Quadrant	12893	12552	2.65%
Antutu v5.7	35576	34761	2.29%
Vellamo			
Browser	2465	2500	-1.42%
Metal	1077	1071	0.56%
Geekbench			
Single Core	1083	966	10.8%
Multi Core	3281	2747	16.28%

Multiple benchmark tools were used to compare the performance of SKEE with the original system. Results of the 32-bit ARMv7 are shown in table III. When tested using seven different benchmarking tools, the prototype shows a performance degradation that varies according to the used benchmark. Results of the 64-bit ARMv8 are shown in table IV. It was tested using the same benchmarking tools and the same variation of degradation rates was observed. We attribute this variation to the unexpected system state when the TLB is frequently invalidated. The process of restoring the cached entries using translation table walks can vary according to the state of the system.

Table IV. SKEE BENCHMARK SCORES ON ARMV8

Benchmark	Original	SKEE	Degradation(%)
CF-Bench	75641	66741	11.77%
Smartbench 2012	14030	13377	4.65%
Linpack	1904	1874	1.58%
Quadrant	36891	35595	3.51%
Antutu v5.7	66193	67223	-1.56%
Vellamo			
Browser	3690	3141	14.88%
Metal	2650	2540	4.15%
Geekbench			
Single Core	1453	1235	15.00%
Multi Core	4585	4288	6.48%

Loading Apps Delay: The third experiment evaluated the effect of SKEE on the look and the feel of the device from the perspective of the end user. The experiment measured the time needed to load some Android Apps. The time was measured between the time the App icon is pressed to the time it is fully loaded. We selected a set of gaming apps that require long time to load due to the size of their binaries and the high resolution graphics involved. The apps were selected from the list of the most popular apps on the Android App Store.

Table V. SKEE APP LOAD DELAY ON ARMV7

App	Original	SKEE	Overhead (%)
Temple Run 2	9.31	10.33	10.96%
Hill Climb Racing	3.66	3.71	1.37%
Angry Birds	4.72	4.79	1.48%
Crossy Road	4.81	5.24	8.94%
Subway Surf	5.45	5.95	9.17%

Table VI. SKEE APP LOAD DELAY ON ARMV8

App	Original	SKEE	Overhead(%)
Temple Run 2	6.08	6.58	8.22%
Hill Climb Racing	2.42	2.73	12.81%
Angry Birds	4.12	4.32	4.85%
Crossy Road	3.42	3.80	11.11%
Subway Surf	4.42	4.71	6.34%

Tables V and VI show the result of this experiment on ARMv7 and ARMv8 respectively. The overhead represents the extra time needed to load the app when SKEE is present.

Device Boot up Time: The last experiment measured the overhead added to device boot up time. Booting up is one of SKEE's worst case scenarios because it requires enormous number of memory allocations. In 32-bit ARMv7, the average original system boot up time is 21.35 seconds, while the average boot up time for the SKEE system is 23.10 seconds (8.2% increase). In 64-bit ARMv8, the average original system boot up time is 21.72 second, while the average boot up time for the SKEE system is 24.30 seconds (11.9% increase).

B. Added Overhead of Sample Security Checks

In the second group of experiments, we added security checks to guarantee that the emulated events do not compromise the isolation. The purpose is to measure the overhead of SKEE with a sample framework that satisfies the minimal set of security checks required to protect the isolated environment.

To build the sample security framework, we adopted the same technique presented in TZ-RKP [11]. In this framework, SKEE creates an array that stores the status of every physical frame of the system memory. It uses this array to verify that writes to either translation tables or translation table base registers will not expose SKEE's protected address space.

The experiments in this section were done using a Samsung Galaxy Note5 smartphone, which uses 64-bit ARMv8 Exynos 7420 processor. The target OS was Android 5.1.1, which ran on Linux kernel 3.10.61. In these experiments the system is set with maximum logging and debugging capabilities.

This group of experiments was done using different hardware, Android version and system settings from the experiments presented in Section V-A. Hence, the performance of the original system varies between both groups of experiments. To make sure that results are accurate, we compare the performance of three system images: an original image without SKEE, an image where SKEE is only used to emulate system events withouts any security checks, and finally an image where SKEE hosts the framework that verifies the emulated events do not compromise the isolation.

Table VII. SKEE ADDED SECURITY CHECKS BENCHMARK SCORES

Benchmark	Original	SKEE	Degrad. (%)	SKEE + Sec. Checks	Degrad. (%)
CF-Bench	63273	58903	6.91%	57250	2.81%
Smartbench	15820	15217	3.81%	15104	0.74%
Linpack	1849	1697	8.22%	1560	8.07%
Quadrant	31429	30843	1.86%	29330	4.91%
Antutu v5.7	65242	62866	3.64%	58658	6.69%
Vellamo					
Browser	4659	4256	8.65%	4350	-2.21%
Metal	2158	2139	0.88%	2081	2.71%
Geekbench					
Single Core	1508	1342	11.00%	1340	0.15%
Multi Core	4566	4388	3.90%	4207	4.12%

The same set of benchmark tools used in Section V-A were used to evaluate the performance of SKEE with the newly added security checks. The results are shown in table VII. The fourth column calculates the percentage degradation between bare SKEE and the original system, while the sixth column calculates the percentage degradation between SKEE hosting the security framework and the bare SKEE system.

It can be observed that overhead added by the bare SKEE platform slightly varies from the one reported in table IV. For instance, the Vellamo-browser benchmark reported less overhead in this experiment. We can conclude that the impact of TLB invalidation is not uniform; it varies according to the original system settings. Similar to the results reported in Section V-A, it can be also observed that the measured overhead widely vary according to the used benchmark. Finally, it can be observed that the performance impact of the security framework is within 10%.

We measured the overhead added to the device boot up time. The average original system boot up time is 30.15 seconds. The average boot up time for bare SKEE is 33.80 seconds (11.7% increase), while that of SKEE hosting the security framework is 35.12 seconds (4.9% additional increase).

C. Performance Enhancement

The experiments presented in this paper do not include performance enhancement technique. The relatively high performance overhead is attributed to the large number of page table updates that are emulated by SKEE. Hence, adopting techniques that group page table updates to reduce the number context switching, such as those introduced in TZ-RKP [11], will reduce the performance overhead.

VI. RELATED WORK

Section I cites a plethora of related work that use hypervisors to monitor and protect the OS. Hypervisors can either rely on hardware-assisted or software-based virtualization techniques. The latter is often referred to as para-virtualization.

KVM/ARM [18] is an example of systems that build a hardware-assisted ARM hypervisor, which can be used to host security tools. The key difference between such systems and SKEE is the location of the security monitor. In hardware-assisted virtualization, the hypervisor relies on hardware extensions to provide the required isolation and memory virtualization. Using the same virtualization layer for kernel monitoring increases both the size of the hypervisor's TCB and the hypervisor's interaction with the kernel, which consequently increases the chances of having vulnerabilities. This is specifically a concern in real world systems, where the virtualization layer is used for purposes other than kernel monitoring. SKEE solves this problem by creating an extra layer of indirection that is less privileged than the virtualization layer, yet more privileged than the kernel.

Para-virtualization is theoretically less secure than hardware-assisted virtualization. However, it is more flexible because it allows the hypervisor to be built without monopolizing hardware extensions. ARMVisor [21] is an example of systems that build an ARM hypervisor using para-virtualization.

SKEE adopts two ideas that are used in para-virtualization approaches: 1) creating a separate protected address space for the monitor and 2) preventing the kernel from accessing the MMU. Nevertheless, there are two key differences between the two approaches. First, SKEE's runs at the same privileged layer as the kernel. Para-virtualization techniques, such as ARMVisor, put the kernel in a less privileged layer alongside user space code, which is less secure and harder to implement on real-world systems. The second is SKEE's novel context switching technique, which provides an atomic, deterministic and exclusive switch gate to the isolated environment.

The turtles project [12] and CloudVisor [55] are examples of systems that propose having multiple layers of virtualization on the same system, a.k.a. nested virtualization. This can be used to achieve the objectives of SKEE. Nevertheless, they are both built for the x86 architecture. SKEE is the first system that achieves the same objectives on the ARM architecture.

There are techniques to measure the integrity of the hypervisor, such as HyperSentry [9] and HyperCheck [51], and to protect it from potential attacks, such as HyperSafe [52].

Nevertheless, these approaches cannot eliminate all attacks that target the virtualization layer.

Section I also discussed the main research directions that achieve isolation without relying on the hypervisor, which are microhypervisors, sandboxing and hardware protection.

There are multiple systems that achieve hardware-based isolation on x86, such as Flicker [40], [41], SIM [48], Nested Kernel [19], and SICE [10]. However, they all rely on x86-specific hardware features. Flicker uses Intel Trusted eXecution Technology (TXT). SIM relies on the presence of a `CR3_TARGET_LIST`, which is a feature provided by Intel to allow the guest OS to switch the address spaces without active involvement of the hypervisor. Nested Kernel uses the Write Protection (`WP`) bit of the `CR0` register to prevent the kernel from accessing the isolated environment. When the (`WP`) bit is set, the kernel is prevented from writing to read-only pages. When it is clear, the kernel is allowed to write to any page despite the read-only protection. This bit is used as a gate to switch the access permission for certain parts of the kernel. SICE relies on x86's System Management Mode (SMM). Unfortunately, ARM does not have equivalent features.

In addition to these directions, previous research work proposed using formally verified microkernels to have a secure core [22], [33], [35]. This secure code can be used to host a security tool. However, formal verification is a challenging long process. Adding a security tool to the microkernel will make it a less practical solution.

VirtualGhost [15] suggests a sandbox that relies on compilation time constraints. Nevertheless, the process of using a custom compiler is also challenging and decreases the chance of adopting this solution in real-world systems.

Finally, there are systems that protect the OS from potentially malicious code. The most notable work in this direction is Native Client [54] and Minibox [36]. There is also ARM-Lock [57] and AppCage [56], which achieve the same objective using ARM specific techniques. As mentioned in Section IV, these systems can be used to compliment SKEE in confining its environment to guarantee that it will not jump back to the kernel while the SKEE address space is exposed.

VII. Conclusion

We introduced SKEE, a system that enables ARM platforms to support an isolated execution environment without adding code to the TCB of higher privileged layers. The new environment is designed to provide security monitoring and protection of the OS kernel.

SKEE provides the isolated environment with three key properties: 1) isolation from the kernel, 2) a secure gate to switch the context between the isolated environment and the kernel, and 3) the ability to place hooks to intercept kernel events for security inspection.

We presented a detailed security analysis that proves the SKEE protection is non-bypassable by the kernel. We also presented prototype implementation and evaluation results. The results show that SKEE is a practical solution for real-world systems. The future work will focus on integrating intrusion detection and system monitoring mechanisms to run inside SKEE to detect attacks and take the proper corrective actions.

References

[1] "Android rooting method: Motochopper," http://hexamob.com/how-to-root/motochopper-method.

[2] "How to root my Android device using vRoot," http://http://androidxda.com/download-vroot.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. ACM, 2005, pp. 340–353.

[4] 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, p. 10.

[5] Argp and Karl, "Exploiting UMA, FreeBSD's kernel memory allocator," in *Phrack Magazine*, vol. 0x0d, November 2009.

[6] ARM Ltd, *TrustZone Technology Overview*. [Online]. Available: http://www.arm.com/products/esd/trustzone_home.html

[7] *ARM System Memory Management Unit. Architecture Specification*, ARM Ltd, 2012.

[8] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang, "HIMA: A hypervisor-based integrity measurement agent," in *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC '09)*, 2009, pp. 193–206.

[9] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, "HyperSentry: enabling stealthy in-context measurement of hypervisor integrity," in *Proceedings of the 17th ACM conference on Computer and communications security (CCS '10)*, 2010, pp. 38–49.

[10] A. M. Azab, P. Ning, and X. Zhang, "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*, ser. CCS '11, 2011, pp. 375–388.

[11] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. ACM, 2014, pp. 90–102.

[12] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation of nested virtualization." in *OSDI*, vol. 10, 2010, pp. 423–436.

[13] Y. Cheng, X. Ding, and R. H. Deng, "Efficient virtualization-based application protection against untrusted operating system," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15. ACM, 2015, pp. 345–356.

[14] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure virtual architecture: a safe execution environment for commodity operating systems," in *Proceedings of the 21st ACM SIGOPS symposium on Operating systems principles (SOSP '07)*, 2007, pp. 351–366.

[15] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual ghost: Protecting applications from hostile operating systems," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. ACM, 2014, pp. 81–96.

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

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

[18] C. Dall and J. Nieh, "Kvm/arm: The design and implementation of the linux arm hypervisor," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. ACM, 2014, pp. 333–348.

[19] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. ACM, 2015, pp. 191–206.

[20] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, "MoCFI: A framework to mitigate

control-flow attacks on smartphones," in *Proceedings of the 19th Symposium on Network and Distributed System Security*, 2012.

[21] J.-H. Ding, C.-J. Lin, P.-H. Chang, C.-H. Tsang, W.-C. Hsu, and Y.-C. Chung, "ARMvisor: System virtualization for ARM," in *Proceedings of the Ottawa Linux Symposium (OLS)*, 2012, pp. 93–107.

[22] K. Fisher, "Using formal methods to eliminate exploitable bugs," in *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Aug. 2015.

[23] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: a virtual machine-based platform for trusted computing," in *Proceedings of the 19th ACM symposium on Operating systems principles (SOSP '03)*, 2003, pp. 193–206.

[24] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proceedings of the 10th Network and Distributed Systems Security Symposium*, 2003, pp. 191–206.

[25] X. Ge, H. Vijayakumar, and T. Jaeger, "SPROBES: Enforcing kernel code integrity on the trustzone architecture," in *Proceedings of the 2014 Mobile Security Technologies (MoST) workshop*, 2014.

[26] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2013, p. 11.

[27] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. ACM, 2013, pp. 265–278.

[28] G. Hotz, "towelroot," https://towelroot.com/.

[29] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *Proceedings of the 18th USENIX Security Symposium*, 2009.

[30] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction," in *Proceedings of the 14th ACM conference on Computer and communications security (CCS '07)*, 2007, pp. 128–138.

[31] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Antfarm: tracking processes in a virtual machine environment," in *Proceedings of the annual conference on USENIX '06 Annual Technical Conference (ATEC '06)*, 2006, pp. 1–1.

[32] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *23rd USENIX Security Symposium. USENIX Association*, 2014.

[33] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09)*, 2009, pp. 207–220.

[34] K. Kourai and S. Chiba, "Hyperspector: virtual distributed monitoring environments for secure intrusion detection," in *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments (VEE '05)*, 2005, pp. 197–207.

[35] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter, "L4android: A generic operating system framework for secure smartphones," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '11)*, 2011.

[36] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "Minibox: A two-way sandbox for x86 native code," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Jun. 2014, pp. 409–420.

[37] L. Litty, H. A. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries," in *Proceedings of the 17th USENIX Security Symposium*, 2008, pp. 243–258.

[38] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, ser. USENIX'93, 1993.

[39] J. McCune, Y. Li, N. Qu, A. Datta, V. Gligor, and A. Perrig, "Efficient TCB reduction and attestation," in *the 31st IEEE Symposium on Security and Privacy*, May 2010.

[40] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki, "Flicker: an execution infrastructure for TCB minimization," in *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2008.

[41] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri, "Minimal tcb code execution (extended abstract)," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy (SP'07)*, May 2007.

[42] 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 *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2013, pp. 1–1.

[43] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Proceedings of the 29th IEEE Symposium on Security and Privacy*, 2008, pp. 233–247.

[44] N. L. Petroni Jr. and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM conference on Computer and communications security (CCS '07)*, 2007, pp. 103–115.

[45] J. Rhee, R. Riley, D. Xu, and X. Jiang, "Defeating dynamic data kernel rootkit attacks via VMM-based guest-transparent monitoring," in *Proceedings of the International Conference on Availability, Reliability and Security (ARES '09)*, 2009, pp. 74–81.

[46] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP '07)*, 2007, pp. 335–350.

[47] H. Shacham, "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)*, 2007, pp. 552–561.

[48] M. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-vm monitoring using hardware virtualization," in *Proceedings of the 16th ACM conference on Computer and communications security (CCS '09)*, 2009, pp. 477–487.

[49] U. Steinberg and B. Kauer, "NOVA: a microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European conference on Computer systems (EuroSys'10)*. ACM, 2010, pp. 209–222.

[50] R. Strackx and F. Piessens, "Fides: Selectively hardening software application components against kernel-level or process-level malware," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. ACM, 2012, pp. 2–13.

[51] J. Wang, A. Stavrou, and A. K. Ghosh, "HyperCheck: A hardware-assisted integrity monitor," in *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection (RAID'10)*, September 2010.

[52] Z. Wang and X. Jiang, "HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, May 2010.

[53] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu, "From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. ACM, 2015, pp. 414–425.

[54] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 2009, pp. 79–93.

[55] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. ACM, 2011, pp. 203–216.

[56] Y. Zhou, K. Patel, L. Wu, Z. Wang, and X. Jiang, "Hybrid user-level sandboxing of third-party android apps," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15. ACM, 2015, pp. 19–30.

[57] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, "ARMlock: Hardware-based fault isolation for ARM," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. ACM, 2014, pp. 558–569.