Hardware-Assisted On-Demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices

Yeongpil Cho, *Seoul National University;* Junbum Shin, *Samsung Electronics;* Donghyun Kwon, *Seoul National University;* MyungJoo Ham and Yuna Kim, *Samsung Electronics;* Yunheung Paek, *Seoul National University*

https://www.usenix.org/conference/atc16/technical-sessions/presentation/cho

This paper is included in the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16).

June 22–24, 2016 • Denver, CO, USA

978-1-931971-30-0

Open access to the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16) is sponsored by USENIX.

Hardware-Assisted On-Demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices

Yeongpil Cho[1] , Junbum Shin[2] , Donghyun Kwon[1]
MyungJoo Ham[2] , Yuna Kim[2] , Yunheung Paek[1]
[1]*Seoul National University* [2]*Samsung Electronics*

Abstract

As more and more mobile applications need to run security critical codes (SCCs) for secure transactions and critical information handling, the demand for a Trusted Execution Environment (TEE) to ensure safe execution of SCCs is rapidly escalating. Although a number of studies have implemented TEEs using TrustZone or hypervisors and have evinced the effectiveness in terms of security, they face major challenges when considering deployment in mobile devices. TrustZone-based approaches bloat the TCB of the system as they must increase the code base size of the most privileged software. Hypervisor-based approaches incur performance overhead on mobile devices that are already suffering from resource restrictions.

To alleviate these problems, in this paper, we propose a hybrid approach that utilizes both TrustZone and a hypervisor. Our approach basically implements a TEE using a hypervisor, while mitigating performance overhead by activating the hypervisor only when the TEE is demanded by SCCs. This scheme, called on-demand hypervisor activation, has been efficiently and securely implemented by leveraging the memory protection capability of TrustZone. We have implemented and experimented our system with real world applications. The results show that our system can successfully protect SCCs without any noticeable delay ($< 100\ \mu$s), while limiting the overhead increase due to our hypervisor during its hibernation near 0 %.

1 Introduction

With a plethora of mobile devices, an extensive range of mobile applications providing convenience are emerging into our lives. However, as mobile devices increasingly offer more sophisticated services, security and sensitivity of the data they handle has become a critical issue [31]. Mobile payment applications nowadays, for example, enable customers to purchase diverse products regardless of place or time. For this, the applications must be authorized to process sensitive data, such as credit card information and personal identification numbers. Accordingly, a number of recent attacks on mobile devices for monetary gain have mostly aimed at achieving unlawful access to sensitive data in such applications. To fend off these attacks, many engineers have introduced the notion of *privilege separation* in the development of their applications by utilizing trusted execution environments (TEEs) on mobile devices. The key objective of privilege separation is to minimize the attack surface of sensitive data by limiting the data accessibility only to the trusted parts of an application, called *security critical code* (SCC) [36, 37, 33], that will be partitioned away from the rest of the application at code development time, and deployed exclusively for secure data transactions in the TEE at runtime.

To provide TEEs for privilege separation on mobile devices, a growing amount of work [23, 48, 45, 44] leverages TrustZone [2], which is a hardware-based security extension installed in ARM processors. TrustZone maintains two separate execution environments, the normal world and the secure world. The normal world is reserved for common OSes and untrusted applications. These typically have rich functionality but are prone to potential attacks due to the existence of exploitable vulnerabilities [13, 61]. Whereas in the secure world, a trusted minimal OS is installed to establish a TEE and provide an individual secure execution environment for each SCC at runtime. Unfortunately, this approach, relying on TrustZone to protect SCCs, faces a major challenge in terms of security. In TrustZone, as the secure world is undoubtedly the trusted computing base (TCB) of the entire system due to its highest privilege level, it must maintain integrity to ensure the safety of the system. However, to support the growing number of versatile SCCs, functional extensions of the trusted OS are inevitable, which increases the size and complexity of the

code base of the trusted OS. Recall that, even carefully designed and engineered code contains bugs and vulnerabilities in proportion to its size [38]. Hence, hosting more SCCs in the secure world may open more doors for attackers to compromise the trusted OS by exploiting its vulnerabilities, which in turn may jeopardize the safety not only of the secure world but also of the entire system. In the real world, as a result, mobile device vendors, such as Samsung, usually have a tendency of being reluctant to render the TrustZone-based secure world freely accessible to public developers. Instead, they have only accepted a few OEM applications that have passed thorough in-house testing.

An alternate way to provide a TEE would be to use a hypervisor which, as the most privileged software layer, is responsible for monitoring and controlling the behavior of the OS layer below it. As long as it is carefully designed, the hypervisor can provide these security and isolation guarantees even when the OS is compromised. Therefore, several studies have relied on hypervisors to implement a TEE for SCC protection and have demonstrated the safety and feasibility of their approaches [36, 62]. However, it must be noted [12, 40, 43] that a hypervisor, running as an extra software layer for virtualization in the system, inevitably suffers from non-negligible performance degradation. This performance overhead may particulary be of great concern in mobile devices which are mostly restricted by severe resource constraints. In fact, the performance concern has been considered to be one of the primary reasons that impedes a wide adoption of existing hypervisor-based approaches in such small resource-stringent devices.

Based on our observations on the problems of previous approaches using either TrustZone or a hypervisor, we have developed a new hybrid approach that attempts to take advantages of both a hypervisor and TrustZone in a way to attain safe, yet efficient SCC execution on mobile devices. To limit the extension of the secure world in TrustZone-based approaches, our approach uses a hypervisor to implement an additional TEE in the normal world alongside the original TEE in the secure world. This *virtualization* scheme enables application developers to implement and distribute their SCCs without the security concern for the secure world corresponding to the system TCB. To tackle the performance concern of other hypervisor-based approaches, we have devised a scheme, called *on-demand hypervisor activation*, which activates our hypervisor only when a TEE must be established for SCC executions. In reality, SCCs are executed occasionally just by a handful of special security applications installed in the system such as DRM and certificate managements. Also an earlier study [14] revealed that even for a given security application, SCC often accounts for a small portion of the entire application. All these support our assertion that our hypervisor should be deactivated for most of the time while the system is up and running. Therefore, as being compared to other approaches which maintain their hypervisors persistently at all times, our solution will suffer from much less virtualization overhead.

To confirm the feasibility of our hybrid approach, we have designed a protection system, named *On-demand Software Protection* (OSP). OSP relies on a hypervisor to meet security requirements for ensuring safe executions of SCCs by implementing an additional TEE in the normal world while suppressing the TCB bloating of the secure world. Therefore, mobile device vendors can allow public developers to install and exeucte their SCCs in the TEE without a large amount of verification efforts. OSP also meets the stringent performance requirements of mobile devices by adopting an on-demand hypervisor activation scheme. In our design, we use TrustZone to enforce memory protection when our hypervisor is deactivated. While the hypervisor is active and running on the machine, OSP checks if there are any SCCs currently being executed by an application. As soon as it finds that no SCC is running, it deactivates its hypervisor and simultaneously orders TrustZone to protect the current states of the deactivated hypervisor as well as all the SCCs that were protected by the hypervisor. TrustZone internally maintains a secure enclave that is not accessible to any other software including the OS kernel. Therefore, every critical information about the hypervisor and SCCs will be safely protected while the hypervisor is in hibernation. Later when an application is about to invoke one of the SCCs, OSP removes the protection of TrustZone, and wakes up the hypervisor by reactivating it with its original states that were protected by TrustZone. Then the activated hypervisor soon reconstructs the TEE where the newly invoked SCC will be securely executed.

To evaluate OSP, we have implemented a prototype of OSP on a development board for Exynos 5422, an ARM-based application processor (AP) platform adopted by commercial mobile devices like Samsung Galaxy S5. In implementation of the OSP prototype, we have only utilized the existing hardware features available in most ARM APs; thus, we believe that OSP is deployable on COTS-devices as well. In order to evaluate its feasibility, we have ported some Android applications to OSP: e.g., Chromium web browser and a file encryption application. The results revealed that OSP was able to ensure secure executions of all SCCs in our system.

The rest of this paper is structured as follows. Section 2 provides background information. Section 3 discusses our threat model and assumptions. Section 4 describes the design and Section 5 introduces implementation details of the OSP prototype. Section 6 presents the evaluation of the experimental results and Section 7 discusses

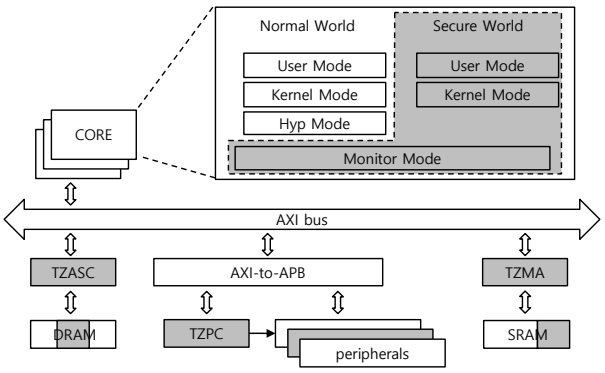

Figure 1: Components of ARM TrustZone

remaining issues. Finally, Section 8 shows related works and Section 9 concludes this paper.

2 Background

This section provides a summary of the security and virtualization extensions supported by ARM.

2.1 Security extensions

TrustZone enables the system to operate in both the secure and normal world in a time-sliced fashion. To separate the two worlds and ensure the confidentiality and integrity of the secure world, diverse extensions are integrated across the system, as depicted in Figure 1. First, the secure and normal world have their own processor modes and system configuration registers and are therefore allowed to build individual software stacks for the OS and applications even if they share a single physical system. To coordinate and arbitrate between the two worlds, the most privileged processor mode, called the monitor mode, is added alongside the existing processor modes. Both the secure and the normal world are able to enter the monitor mode by issuing a secure monitor call (SMC) instruction.

TrustZone includes an extension for secure interrupts as well, which is only visible and delivered to the secure world. The ARM architecture is equipped with a GIC [8] to control system-wide interrupts in a manner similar to APIC of Intel. GIC provides 16 software generated interrupts (SGI) that can be delivered to every core or only to specific cores as inter-processor interrupts (IPI). The security extension of GIC allows us to designate some of the SGIs as secure interrupts such that they can be used to pass signals secretly. GIC also enables secure SGIs using FIQ signals, instead of IRQ signals, in order to increase the priority of the interrupts.

The NS-bit in the secure configuration register (SCR) indicates whether a processor is executing in the normal world or the secure world. This bit is also propagated across the entire system by being attached to system bus transactions, so that scattered TrustZone components are able to manage access to resources, such as memory and peripherals, out of the CPU cores. For example, TrustZone includes TZMA [4] and TZASC [7], respectively located in front of the SRAM and DRAM. They partition the address spaces corresponding to SRAM and DRAM into several regions, each of which is assigned to the secure world or the normal world, and prevent access to the secure world regions from the normal world. TrustZone also adds the TZPC [5], which enforces a similar security policy with regard to peripherals. This way, the secure world can configure and access peripherals in an explicit manner.

2.2 Virtualization extensions

Similar to VT-x [39] of Intel and SVM [29] of AMD, ARM introduced hardware virtualization extensions [6] that allow hypervisors to efficiently manage guest OSes. To empower hypervisors to configure the entire system, ARM supports a privileged processor mode known as the hyp mode [6], which is beneath the kernel mode in the hierarchy of processor modes as described in Figure 1. Hypervisors running in the hyp mode are able to configure fundamental system resources, such as the exception vector table, counter and timer, with a variety of control registers only accessible in the hyp mode. In particular, hypervisors can configure and deploy the extended page tables underneath the primary page tables managed by guests. By assigning various access-permission flags in the extended page tables, hypervisors are able to exclusively enforce access-control policies for all address spaces of guests. Along with the hyp mode, a hypervisor call (HVC) instruction is added for communication between hypervisors and guests.

The ARM virtualization extensions include a system MMU [3] as well. If the system MMU is enabled, each peripheral is given its own page table. Configuring those page tables according to guests, hypervisors can dynamically change the address spaces of peripherals. This facilitates device virtualization without the intervention of hypervisors, thereby improving hypervisors in terms of their performance and porting effort. The system MMU, moreover, is effective at preventing DMA attacks of misconfigured peripherals by limiting the accessible address space of each peripheral.

3 Threat model and Assumptions

In this section, we describe the threat model and assumptions pertaining to the implementation and design of OSP.

Threat model. We assume that our adversaries can exploit vulnerabilities to gain full control over the rich OS. In other words, they can freely perform arbitrary memory reads, memory writes, and code executions in the address spaces of the OS kernel and applications. With this capability, they may attempt to access the address spaces of SCCs in order to steal confidential contents revealed during runtime. They may also try to acquire the binary files of SCCs so that they could extract statically stored secrets with reverse-engineering or determine the core algorithms of SCCs through binary analysis.

In addition, as we cannot fully trust application developers and their products, SCCs could be abused to tamper with and/or eavesdrop on other SCCs and their sensitive data. Malicious SCCs may also attempt to subvert a TEE by making arbitrary system calls with crafted parameters.

Assumptions. We assume that in the secure world, carefully verified software is preinstalled and dynamic software installation is not allowed. The built-in software of the secure world, including the minimal OS and OEM applications, is trusted and will be intactly loaded with a secure boot mechanism such as AEGIS [50] or UEFI [56]. Therefore, we do not take into account any attacks originating within the secure world. We also do not consider denial-of-service (DoS) attacks. Memory attacks, such as cold boot attacks [26] and bus monitoring attacks [49, 54] are beyond the scope of our adversary model as well. Similarly, hardware attacks, such as physical side-channel and JTAG attacks are not considered in this work.

4 Design

OSP creates a TEE alongside Trustzone, which provides a security and efficient protection mechanism. This TEE can be used by mobile device vendors to provide a way for application developers to protect their SCCs. In this section, we present the details of the design of OSP and explain how it achieves this goal.

4.1 Design objectives

In order to secure SCCs, we deliberately design OSP while seeking to accomplish the following objectives.

Practical mechanism. Opening the secure world for SCC protection causes a security concern about TCB bloating from an increased code base. Therefore, OSP should arrange a TEE on the exterior of the secure world, thereby enabling application developers to protect their SCCs without reducing the level of security of the secure world. In addition, as we consider resource constrained mobile devices, OSP should incur negligible performance overhead when maintaining the TEE.

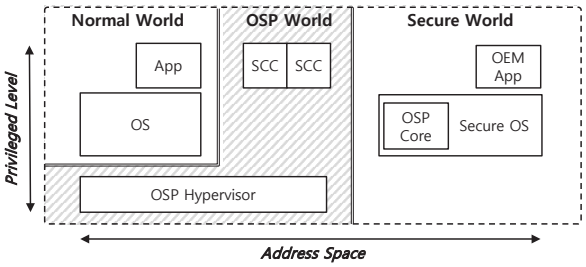

Figure 2: Overview of OSP. OSP consists of the OSP hypervisor, which protects and manages the SCCs, and the OSP core, which controls and configures OSP overall.

Runtime protection. To protect the confidentiality and integrity of SCCs, OSP should provide each SCC an individual execution environment, which is isolated from the OS kernel and other SCCs. As SCCs are not trusted entities, each SCC should be able to communicate with other SCCs and the OS kernel only when allowed by OSP.

Secure provisioning. SCC binaries often encompass secrets, such as key values and core algorithms, which developers want to protect. Therefore, SCC binaries should not be exposed to attackers during their distribution so as to ensure the confidentiality of those secrets.

4.2 Overall Design

Figure 2 depicts the overall design of OSP. OSP defines the OSP world alongside the normal world and the secure world. As the OSP world is completely separated from both worlds, OSP can securely provide an additional TEE to SCCs while keeping the secure world compact. OSP consists of two software components: the OSP core and the OSP hypervisor. As the TCB of the entire OSP system, the OSP core, located in the secure world, is responsible for initializing OSP during the system boot sequence and for deploying and controlling the OSP hypervisor at runtime. The OSP hypervisor, the de facto TCB of SCCs, plays a vital role in the functionality of OSP. It protects the OSP world by blocking unauthorized accesses of the normal world; it also creates a TEE in the OSP world, thereby providing isolated execution environments for SCCs.

Although the OSP hypervisor is a fundamental component in OSP for the runtime protection of SCCs, it may incur non-negligible performance impacts due to virtualization overheads. Therefore, to minimize such overheads, OSP activates its hypervisor only while a protection service is required, i.e., when one or more SCCs are running. Moreover, the OSP core expands the secure world enough to cover the entire OSP world to protect it from invasions by the normal world when the OSP hypervisor is no longer active.

Function Name	Parameter	Call-site	Description
Management interfaces			
SCC_register	scc_file_name, ptr_external_handler	app	Registers an SCC with a specification. Upon success, returns the SCC's number.
SCC_unregister	scc_num	app	Unregisters an SCC.
SCC_parameter_add	ptr_scc_param_spec, param_flag, ptr_param, length	app	Add a parameter to a parameter specification.
SCC_invoke	scc_num, entry_func, ptr_scc_param_spec, arg0...arg3	app	Invokes an SCC with a parameter specification. Upon finish, returns a return value.
SCC_ret_to_scc	scc_num, return_value	app	Return to an SCC with a return value
Service interfaces			
OSP_save	ptr_data, length	SCC	Save data on secure storage. Upon success, returns the storage number.
OSP_load	storage_num, ptr_buffer, length	SCC	Loads the data for a storage number.
OSP_delete	storage_num	SCC	Deletes the data for a storage number.
OSP_encrypt	ptr_data, ptr_buffer, length	SCC	Encrypt data
OSP_decrypt	ptr_data, ptr_buffer, length	SCC	Decrypt data
OSP_signing	ptr_data, length, private_key, signature	SCC	Sign data with a given private key
OSP_verification	ptr_data, length, public_key, signature	SCC	Verify data with a given public key
OSP_external_handler	cmd, arg0...arg3	SCC	Call the external handler with parameters. Upon finish, returns a return value

Table 1: The management and service interfaces of OSP

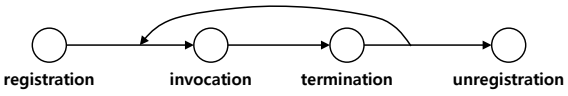

Figure 3: The lifecycle model of an SCC

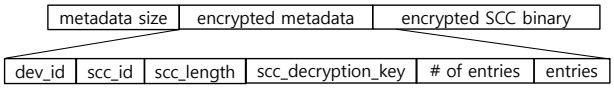

Figure 4: The format of an SCC file

4.3 Development of SCCs

To protect sensitive data using SCCs, developers need to develop their applications while being conscious of the concept of privilege separation. Developers should handle sensitive data only in SCCs and should transmit the data to the remainder of their applications after encrypting it to prevent exposure. For the sake of minimizing the attack surface, we highly recommend that developers ensure that their SCCs are self-contained to prevent internal states from being exposed outside of the SCCs during execution. However, SCCs may sometimes want to outsource certain functions, such as network or file system access, to enrich their functionality. OSP supports such cases by letting developers implement external handlers that can process outsourced requests in their applications running on the rich OS and allowing SCCs to call those handlers.

Application developers should design SCCs considering the lifecycle model of an SCC, depicted in Figure 3. They can implement SCCs using the following interfaces that are offered in the form of a static or dynamic library. We describe the details in Table 1.

Management interface. Using the management interface, a developer can include an SCC into her application as if using a dynamic library. To begin with, we assume that there is a prebuilt SCC file (§4.4). The developer should initially call `SCC_register` with the name of the SCC file and, if needed, the address of an external handler located in the application. An SCC number is then given after registration, which is used to specify an SCC in the later invocation and unregistration processes. To invoke the registered SCC, the developer should prepare a parameter specification by gathering the properties of the parameters that are to be passed, each of which consists of a start address, a length and flags. In particular, the flags specify when a parameter will be marshalled; the `input` and the `output` flags indicate that the corresponding parameter will be marshalled when the SCC is invoked and returned, respectively, and the `shared` flag means that the corresponding parameters do not need to be marshalled because they are shared between the application and the SCC. At this point, the developer can invoke the SCC with the specified parameters and can continue to invoke it unless the SCC is unregistered.

Service interface. The current implementation of OSP provides secure storage and cryptographic services, which allows SCCs to protect passwords and cryptographic keys. Expanding the capabilities of SCCs by adding new services offered by the OSP is left for future work (refer to Section 7). In the current OSP, instead, SCCs can outsource some operations that are not supported by OSP, i.e., memory management, networking, file system, to an external handler, which would be appointed during the SCC registration step, located in the application. However, as external handlers may be po-

tentially vulnerable, developers are responsible for verifying returned results to defeat unintended attacks such as Iago attacks [16].

4.4 Provision of SCCs

SCCs can be distributed in various ways. In this paper, we assume a scenario where application developers distribute their SCCs along with their applications. Application developers initially need to have a developer ID (e.g., a developers private key) that can identify them individually. Because developers are allowed to distribute more than two SCCs, they should choose a unique SCC ID for distinguishing each SCC. To maintain integrity and confidentiality, SCCs must be distributed in an encrypted form. Figure 4 describes the distribution file format of an SCC, which is made up of metadata and an encrypted SCC binary. The metadata consists of two noted IDs and a key for decrypting the encrypted SCC binary. In addition, the metadata should contain a list of entry functions that are allowed for applications to invoke; thus preventing non-designated internal functions from being called directly. Lastly, the metadata is encrypted asymmetrically with the public key of OSP to protect the contents by sealing.

4.5 Execution of SCCs

At runtime, once the OSP hypervisor receives a registration request with an encrypted SCC file, the hypervisor copies contents of the file to the OSP world and performs a series of decryptions and parsing. It first decrypts the metadata of the SCC with the private key of OSP and parses that to extract developer and SCC IDs and the decryption key. The hypervisor subsequently decrypts the encrypted SCC binary and begins to load the decrypted contents onto the OSP world. It prepares an empty extended page table and maps the address space of code, data and stack of the SCC to the page table. Then, it finalizes the registration step by returning the number of the SCC to the caller application.

An invocation request for the SCC is delivered to the OSP hypervisor with a parameter specification and an entry functions number. In the beginning, parameters that are documented on the parameter specification are marshalled according to the details of the specification. Next, the hypervisor maps the parameters to the page table of the SCC. The hypervisor masks unrelated interrupts to the SCC, preventing the OS kernel from interrupting the execution of the SCC. The hypervisor, moreover, applies the prepared page table to the system to reflect the SCCs own address space. Finally, it checks the correctness of the passed entry functions number and it transfers control to the corresponding entry function. At the same time, the hypervisor instigates a timeout to prevent a buggy or malicious SCC from seizing the system for too long. After a while, if the SCC finishes its work, the OSP hypervisor instantly restores the states relevant to the caller application and returns the control to the application. A similar procedure is conducted when the SCC calls its external handler while it is being executed. In this case, however, it is required for the application to issue `OSP_ret_to_scc` in order to resume the SCC.

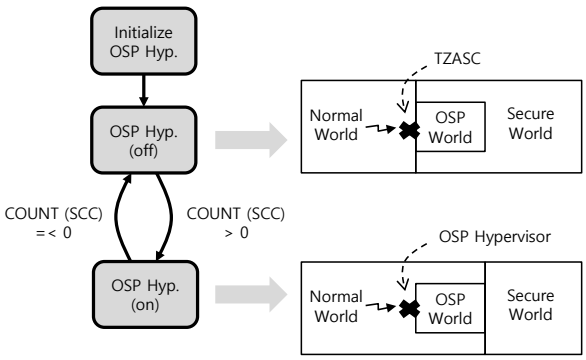

Figure 5: Two protection mechanisms for the OSP world

The execution environment of the SCC is maintained until the application unregisters the SCC explicitly. If an unregistration request is issued, the OSP hypervisor completely clears every relevant state of the SCC, such as the page table and the contents of the heap and stack regions. To use the same SCC from that time, the application must register it again.

4.6 On-demand activation of the OSP hypervisor

As noted above, the OSP hypervisor is activated by the OSP core only while SCCs are running in the OSP world. The OSP core finishes the default configuration of the OSP hypervisor during the boot sequence. This process includes the creation of the default extended page table that identically maps the entire address space of the normal world, although the OSP core does not enable extended paging at this point. However, considering that the OSP hypervisor depends on extended paging, to protect the OSP world while the hypervisor is deactivated, introducing another mechanism is inevitable.

For this purpose, OSP capitalizes on TZASC which, as a hardware component of TrustZone, allows dynamically setting the address space of the secure world. While the extended paging is disabled, as described in Figure 5, the OSP core includes the OSP world in the secure world using TZASC, thereby preventing malicious accesses originating from normal world software. Note that OSP creates its TEE in the normal world rather than

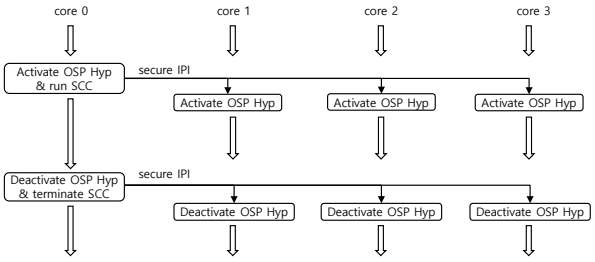

Figure 6: On-demand activation of the OSP hypervisor in multi-core environments

```
Procedure ACTIVATE_OSP_HYP
  Enable the extended paging
  Send secure IPIs to other cores to enable the extended paging, too
  Reduce the secure world to reveal the OSP world using TZASC
End

Procedure DEACTIVATE_OSP_HYP
  Expand the secure world to cover the OSP world using TZASC
  Clean and invalidate cache entries of the OSP world
  Disable the extended paging
  Send secure IPIs to other cores to disable the extended paging, too
End
```

Figure 7: On-demand activation and deactivation routines of the OSP core

the secure world. Therefore, this configuration is cleared when SCCs are invoked; from this point on, the OSP hypervisor protects the OSP world from the normal world by activating extended paging. When an SCC is terminated, the OSP hypervisor checks if any other SCC is still running. When all SCCs are terminated, the OSP core disables extended paging to reduce the performance degradation caused by extended paging. This, however, renders security-critical data stored in the OSP world vulnerable to untrusted software in the normal world because they are now accessible to anyone with control over the normal world. To address this problem, before disabling extended paging, the OSP core reconfigures TZASC so that the secure world once again engulfs the OSP world.

4.6.1 Multi-core support

OSP supports multi-core environments; it allows several SCCs to run concurrently in different cores. However, this does not mean that the OSP hypervisor can be individually activated in each core, even though each core has its own MMU and control registers. Because there is only one TZASC within the system, located between the system bus and main memory, when a core configures TZASC, the effect is not limited to that core. For example, when in a core, if the OSP core configures TZASC to pull the OSP world from the secure world and activates the OSP hypervisor, the OSP world will immediately be exposed to normal world software on all of the other cores. In addition, another severe problem will arise when the OSP hypervisor is deactivated. Let us assume that, in a core, the OSP core deactivates the OSP hypervisor and reconfigures TZASC to include the OSP world in the secure world. At this point, however, the OSP hypervisor is still activated in the other cores, a permission violation for the secure world will be provoked (at least due to address translations by extended paging). Consequently, OSP must synchronize the hypervisor activation state of every core, as in Figure 6.

4.6.2 Activation and deactivation routines

Figure 7 summarizes the routines of the OSP core for activating and deactivating the OSP hypervisor at runtime. If the activation routine is initiated, the routine initially enables extended paging and sends secure IPIs to other cores, so that they enable extended paging as well, to activate the OSP hypervisor. This must be done before removing the protection of TZASC in order to prevent untrusted software from accessing the OSP world. We can control extended paging using the Hyp Configuration Register (HCR), which is not accessible in the normal world. The HCR register consists of a number of configuration bits; in particular, we can enable and disable extended paging by setting and clearing the VM-bit. In addition, the HCR register contains the TDC-bit. This bit makes OSP enable cache memory while SCCs run even if address translation of the kernel space is disabled. After activating the OSP hypervisor, the routine controls TZASC to reveal the OSP world from the secure world. TZASC can be controlled using memory mapped registers similar to most components of ARM. As explained in Section 2.1, TZASC manages regions as a unit of permission enforcement. We can control these regions with two primary registers, the Region Setup Register and Region Attributes Register. The former one controls the base address of each region. The latter one plays a more important role; it determines the size and permission[1]. Particularly, this register has an enable bit, so that we can enable and disable the corresponding region by toggling the bit.

The deactivation routine is performed in the reverse order of the activation routine. First, it configures TZASC to cover the OSP world with the secure world. After configuring TZASC, the routine cleans and invalidates every cache entry corresponding to the OSP world. Otherwise, some states of the OSP world remaining in the cache memory may be exposed to untrusted software in the normal world. Finally, the routine deactivates the OSP hy-

[1] A permission can be configured as a combination of secure read, secure write, non-secure read and non-secure write flags.

pervisor by disabling extended paging. As the last step, it sends secure IPIs to the other cores so that they can also disable extended paging. Each core disables extended paging as soon as it receives the IPI.

4.7 Interface implementation

As noted in Section 4.3, OSP provides two types of interfaces, called the management interface and the service interface. In this subsection, we explain how OSP implements these interfaces.

Management interface. In general terms, the normal world software is intended to communicate with a hypervisor using the HVC instruction. However, this option is not available in OSP considering the dynamic activation state of the OSP hypervisor. Thus, OSP would have to provide two duplicate management interfaces which are implemented based on the SMC and the HVC instructions, and the normal world software would need to choose the proper interface each time depending on whether or not the OSP hypervisor is hibernating in the secure world.

To avoid this complication, OSP implements the management interface using only the SMC instruction. For this, the activation routine of the OSP core, introduced in Section 4.6.2, sets the TSC-bit of the HCR register to 1, thus trapping future executions of the SMC instruction into the OSP hypervisor. After which, if a normal world software executes SMC instructions, the OSP hypervisor first analyzes whether the SMC instruction is intended for the interface of OSP by parsing it. If so, the hypervisor performs a management operation according to the request, and if not, it is transferred to the OSP core to be handled in the secure world.

Service interface. It is fairly known that the supervisor call (SVC) instruction is used to implement system calls of the kernel. Moreover, (unprivileged) applications are not allowed to execute the SMC or the HVC instructions on ARM. Accordingly, OSP enables SCCs to use the service interface that is implemented based on the SVC instruction. For this, the activation routine of the OSP core sets the TGE-bit of the HCR register. By doing so, all executions of the SVC instructions are trapped into the OSP hypervisor; thus, the hypervisor can receive and handle service requests of SCCs.

5 Implementation

In this section, we explain implementation details which were not presented in the earlier sections.

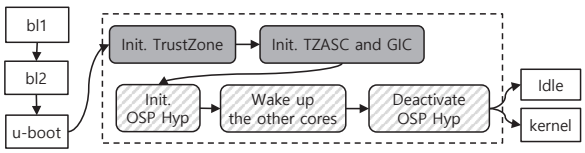

Figure 8: Boot sequence of OSP

5.1 OSP Hypervisor

The structure of our OSP hypervisor is somewhat related to KVM/ARM [21], an open-source hypervisor found in the mainline Linux kernel, in the sense that our hypervisor borrows several key implementation mechanisms regarding virtualization. However, in comparison to KVM/ARM, the OSP hypervisor has a simple structure with a small code base because it only needs to support a single guest OS. As a result, it can run with much lower overhead than the general-purpose KVM hypervisors. For example, the OSP hypervisor requires only a quarter to half of the CPU cycles (1,119 cycles) required by KVM/ARM (from 2,112 to 4,917 cycles) for world-switching latency (round trip from the kernel to the hypervisor).

For simplicity, we statically place the OSP hypervisor on the top 128 MB of the physical memory address that is reserved for the OSP world. Such static deployment may reduce the available physical memory of the kernel, but this problem could be mitigated by making the OSP hypervisor use the memory management service of the kernel. This task is left for future work.

5.2 Boot Sequence of OSP

Figure 8 illustrates the modified boot sequence used when launching OSP. We assume that each bootloader verifies the integrity of the succeeding bootloader using a secure boot mechanism so that we can trust the code and initial states of the OSP software components. The OSP core should start while running in the kernel mode of the secure world to access the privileged system control registers related to TrustZone and virtualization. First, it enables SMC instructions and sets the SMC call handlers for OSP. It also initializes TZASC and GIC. Next, it prepares the OSP hypervisor and the TEE by initializing virtualization features, such as the extended page tables and programming interfaces. In a multi-core environment, as each core has an independent execution environment, the OSP core wakes the other cores and initializes them as well. The OSP core finally deactivates itself by executing a secure monitor call and transfers control to the kernel. The remaining cores, apart from the primary one, jump to the idling code, a.k.a. a boot monitor.

6 Evaluation

In this section, we evaluate OSP by analyzing its performance overhead and security. Experiments were conducted on ODROID-XU3 Lite [27], which has an Exynos-5422 SoC with an ARM Cortex-A15 1.8 GHz quad-core processor and 2 GB of DRAM, used on a variant of a Samsung Galaxy S5 smartphone. The OS is Android 4.4.2 with Linux Kernel 3.10.

To obtain accurate results, we leave the device idle and let it cool down between experimental trials. Without such measures, the processor would be throttled by heat, ultimately leading to an inaccurate evaluation. Note that mobile processors can overheat within seconds if they are fully utilized. A cooling fin and fan were attached onto the target experiment board as additional countermeasures.

6.1 Performance impact

To investigate the performance impact on the system, we tested the three following cases:

- **baseline**: with a bare Android without OSP
- **hyp_on**: with OSP while the OSP hypervisor is activated
- **hyp_off**: with OSP while the OSP hypervisor is deactivated

Note that the performance of **hyp_on** represents the performance of applications when SCCs are running in other cores. We show the results normalized by the values of the **baseline**.

We experimented with popular mobile benchmarks: AnTuTu, BaseMark and Geekbench. We also experimented with other synthetic workload benchmarks with various categories: CPU and memory (Vellamo-Machine, CF-bench), JavaScript for web browsers (Vellamo-Browser), file system throughput (IOZone), graphics throughput (GFXBench) and kernel system calls (lmbench).

Figure 9 shows the experimental results. In the figure, higher values represent shorter latency times or higher throughput, where 1 represents the performance of the **baseline**. In addition, geomean indicates the normalized geometric mean values of all benchmark results.

When the OSP hypervisor is activated, performance is degraded mainly due to a high TLB miss penalty caused by complicated address translations. Therefore, memory-intensive tasks show somewhat higher overhead than computation-intensive tasks in **hyp_on**. Note that most test cases, with the exception of "DVFS Off" of AnTuTu, allow the kernel to freely adjust the CPU frequency, as in most mobile devices. Also, note that "DVFS Off" represents the case in which the frequency of the CPU is fixed at 1.2GHz such that we can eliminate the possibility of performance throttling, which could be caused by overheating or by the variances induced by DVFS itself. These results suggest that such effects do not arise.

The slowdown in **hyp_off** is near 0 % (mean = 1.003); on the other hand **hyp_on** shows visible degradation (mean = 1.066). These results reflect the effectiveness of on-demand activation for reducing hypervisor-induced overhead. Overall, OSP virtually does not incur any slowdown for the system when no SCCs are running.

6.2 World switching latency

We investigated the latency of a single round of world switch between the normal world and the OSP world. The latency depends on whether or not the OSP hypervisor is activated. Our results show that the latency is only 550 cycles when the OSP hypervisor is activated. In contrast, the world switching latency is 127,453 cycles (71 μs at 1.8 GHz), which includes 11,191 cycles to set up the OSP world in the OSP core, 550 cycles to enter and exit the OSP world, 31,450 cycles to clean and invalidate the cache memory, and 68,329 to verify the configuration of the system MMU to defeat DMA attacks. This amount of latency is likely to be tolerable in a commercial device. Note that the latency of OSP is comparable to the context switch latency of ARM processors of previous generations [22, 1].

6.3 Application benchmarks

To investigate the feasibility of OSP, we ported two applications, the Chromium web browser and a file encryptor. As a result, we confirmed that (1) OSP incurs no noticeable delay when SCCs are called infrequently, and (2) on-demand activation makes OSP effective against hypervisor overhead.

Chromium web browser Modern web browsers internally provide an autocomplete function for user convenience, which adds IDs and passwords to a login form using saved values. However, this function introduces a risk that the saved list of IDs and passwords can be exposed to untrusted software. Therefore, the list in the autocomplete function must be secured.

We conducted an experiment on the Chromium web browser for Android, version 46.0.2469.0. If the browser finds a login form, it provides the autocomplete functionality by using the `LoginDatabase` class, including `AddLogin`, `UpdateLogin` and `GetLogins` members. However, as, in the target version of the browser, the `LoginDatabase` class saves and secures IDs and passwords by simply encoding them with UTF-8, we modified it to encrypt IDs and passwords before saving them

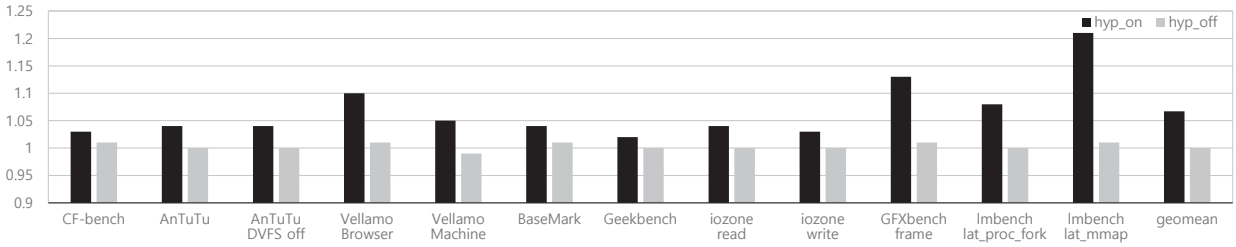

Figure 9: Performance results of OSP relative to the baseline

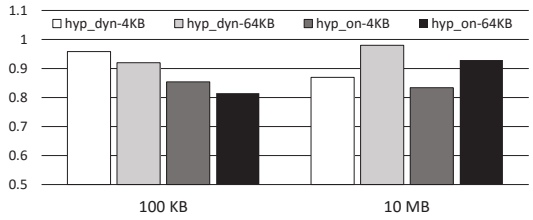

Figure 10: Performance results of the encryption SCC for 100 KB and 10 MB files relative to the baseline

(**baseline**). To evaluate OSP, instead of that, we modified the `LoginDatabase` class to save IDs and passwords after encrypting them with our SCC. Moreover, according to the lifecycle model of an SCC in Figure 3, we inserted the registration and unregistration routines of the SCC into the constructor and destructor of the `LoginDatabase`.

After making such modifications , we visited the login page of Facebook. In Chromium, the page load function and the autocomplete function run in separate threads. Therefore, we measured the page loading time and the autocompletion time separately. The experiment was repeated 100 times. The page load time averaged at 995.7 ms for both the **baseline** function and that with an SCC. Regarding the autocompletion time, the trial with an SCC averaged at 0.101 ms, which is 20.4 times slower than the **baseline**. However, when combined with the page load time, the difference in the autocompletion time is negligible.

File encryptor Many recent applications encrypt their sensitive data, such as chat logs and private pictures, for data protection purposes. In the figure, a higher value represents a longer execution time. A significant issue associated with such an approach is the method used to protect their encryption key. To address this, we implemented an SCC which provides AES-256 encryption and decryption functions and linked it to our own file encryptor using JNI. The file encryptor reads a file in chunks of 4 KB and 64 KB and encrypts each chunk with the SCC.

Figure 10 shows experimental results, consisting of the results of 100 separate executions of different input file sizes. In comparison to the **baseline**, **hyp_dyn** (enabling the on-demand feature), shows a degraded performance level due to activation overhead, proportional to the number of invocations of the SCC. We investigate the impact of on-demand activation by comparing **hyp_dyn** to **hyp_on** (disabling the on-demand feature). As a result, **hyp_dyn** is more efficient than **hyp_on** despite the accumulated overhead from on-demand activations. This is likely due to the fact that **hyp_on** incurs performance overhead caused by the hypervisor while the host application completes file operations between the SCC calls.

6.4 Security analysis

6.4.1 TCB size

In OSP, the OSP hypervisor is the TCB of SCCs and the OSP core residing in the secure world belongs to the TCB of the entire system. To estimate the safety of OSP in terms of TCB size, we measured the number of source lines of our OSP prototype with the SLOCCount tool [57]. The OSP hypervisor consists of $<$ 3,000 C SLOC and $<$ 500 assembly SLOC. The OSP core has $<$ 700 C SLOC and $<$ 100 assembly SLOC. In conclusion, OSP has as small a TCB as in previous works [47, 36, 62].

6.4.2 On-demand activation

After the system is turned on, OSP undergoes various state transitions. We analyzed the security of OSP as follows according to its states.

Initialization. The initialization of OSP is carried out as part of the boot sequence described in Section 5.2. We guarantee its safety under two assumptions: first, there must be no exploitable vulnerabilities in the code of OSP; second, a secure boot mechanism must be implemented. Therefore, we can be sure that the loaded OSP code and initial states are intact and that the boot stage of OSP is completed with certain known good states.

Hyp_off. While the OSP hypervisor is not deployed, the OSP core temporarily covers the OSP world with the secure world by configuring TZASC. All code and data residing in the OSP world are isolated from the normal world by the TrustZone components that are scattered across the system so that malicious memory accesses from both untrusted software in the normal world and misconfigured peripherals are completely prevented.
Hyp_on. While the OSP hypervisor is deployed, the OSP world strictly belongs to the normal world within the concept of TrustZone, which splits all system resources into the normal and secure worlds. Nevertheless, the OSP world is still secure from untrusted software, as the OSP hypervisor is capable of blocking unallowed memory accesses to the world by means of extended paging. DMA attacks are also thwarted by examining the mapping tables of the system mmu so as to prevent the OSP world from being exposed to peripherals.

6.4.3 Malicious SCC

As OSP allows application developers to deploy their SCCs without thorough verification or examination, it is reasonable to postulate that there could be SCCs built with malicious intention. Malicious SCCs might attempt to tamper with the normal world software or other SCCs. In OSP, however, because each SCC is strongly isolated, the attack surface is minimized, leaving few windows for malicious SCCs to compromise the hypervisor. They may try to abuse the service interface of OSP with crafted parameters to compromise the OSP hypervisor. To defeat such approaches, we may need to execute each SCC in a sandbox [59, 33]. Unfortunately, as the current OSP prototype does not contain such defense mechanisms, there is a possibility that the OSP hypervisor could be compromised. Nevertheless, as the OSP core, being located in the secure world, still has full control over the OSP hypervisor, it can ensure the integrity of the OSP hypervisor by using TrustZone-based solutions [9, 25].

7 Future work

Trusted I/O path. Although the currently implemented OSP does not offer a trusted I/O path, it is a desired feature for application developers. With this feature, SCCs could directly interact with users without going through the vulnerable OS kernel. Fortunately, as explained in 2.1, TrustZone includes various components that are capable of isolating interrupts and bus transactions between the CPU cores and peripherals. They are sufficient to facilitate the implementation of a trusted I/O path [32, 51]. Similar to academic works, off-the-shelf mobile devices are known to depend on TrustZone to implement a trusted path on fingerprint sensors or other components.

Therefore, it would be reasonable for OSP to reley on the TrustZone-based trusted I/O path to provide features for SCCs rather than to develop its own trusted I/O path. Therefore, OSP initially needs to establish a secure channel between the OSP world and the secure world. The OSP core can do this by creating and passing a session key to the OSP hypervisor early in the boot stage. The OSP hypervisor can then offer a trusted I/O path, which is implemented in TrustZone, to SCCs through the OSP core without the intervention of the OS kernel.
Kernel-mode SCC. In this paper, we have assumed that SCCs are executed in the user mode. However, we can also consider other SCCs running in the kernel mode. For example, if there are SCCs that are intended to monitor the integrity of the kernel, they must run in the kernel mode to execute privileged instructions or to access privileged data structures such as page tables. We believe that kernel-mode SCCs are difficult to protect in TrustZone-based solutions due to the high security risk involved in permitting privilege instructions. On the other hand, we deem that OSP could cover such SCCs as well. To provide isolated execution environments, OSP depends on the OSP hypervisor working beneath the OS kernel; therefore, we can improve the OSP hypervisor to mediate and verify behaviors of kernel-mode SCCs that may corrupt the system.
Other Future Work. ARM introduced the big.LITTLE architecture, which leverages big (high-performance) cores or little (low-performance) cores depending on the performance requirements of tasks, thereby improving the power efficiency. The current prototype of OSP has yet to support this technique. However, as it becomes more popular on mobile devices, it will be necessary to upgrade OSP to support it.

8 Related work

In this section, we compare OSP with existing solutions attempting to protect software.
TrustZone-based solutions TrustZone, originated by ARM, has been spotlighted as a secure and lightweight solution to protect SCCs. Recently, it was also adopted in the x86 architecture by AMD. To enhance its usability, TLR [45, 44] ported the .NET framework inside TrustZone so that it enables SCCs programmed with the .NET bytecode to execute in the secure world. However, its monolithic design in which all SCCs are sharing a single world will increase the attack surface in proportion to the number of installed SCCs. TrustICE [52] addressed this problem by providing each SCC with a separated execution environment, called ICE, in the normal world. Thus, in that work, third-party developers are permitted to pro-

tect their SCCs. This work achieves a similar objective as OSP, but unlike OSP, TrustICE was designed to create isolated execution environments based on the kernel mode, thus using the same privilege level with the untrusted OS kernel. Consequently, it faces limitations in supporting multi-core environments. While an ICE runs in a core, the other cores must be suspended until the ICE is terminated, to prevent the OS kernel from accessing the ICE.

Hypervisor-based solutions Many studies have attempted to provide isolated execution environments by leveraging hypervisors. TERRA [24] and Proxos [55] attempted to provide each application with its own operating system. Overshadow [17] and SP^3 [58] protected application data from being exposed to untrusted OSes by encrypting the data transparently.

As frequent encryption operations incur significant performance overhead in the system, numerous studies have constructed isolated execution environments using elaborate access-control mechanisms based on the extended paging technique. InkTag [28], AppShield [18] and AppSec [43] concentrated on shielding all applications from an untrusted OS. On the other hand, TrustVisor [36], MiniBox [33] and Wimpy Kernel [62] focused on protecting security critical portions of applications (SCCs in this paper) rather than all applications. These solutions attempt to reduce the code size of their hypervisors in order to reduce the size of their TCB as well. Our OSP can be considered similar to these solutions in the sense that OSP protects a small portion of an application using a lightweight hypervisor, minimizing the code size of its hypervisor.

However, all of the aforementioned techniques place a burden on the system through persistent computational overhead for their hypervisors to maintain virtualization. Curtailing such overhead by dynamically activating hypervisors was originally proposed in earlier studies [35, 41], in which the technique was used for efficient OS maintenance based on a hypervisor. However, these techniques are inadequate for software protection as their designs do not consider security constraints. Meanwhile, P-MAPS [42] and another study [60] adopted aforementioned techniques for security purposes. However, in comparison with OSP, their implementations are not lightweight because they rely on time-consuming cryptographic functions of TPM for on-demand functionality. In particular, P-MAPS has world-switching latency of 300 ms, which is clearly noticeable to users.

Hardware-based Approaches AEGIS [50], Bastion [15], SecureME [19] and XOMOS [34] provide secure execution environments. However, they are not compatible with conventional systems because they require new architectural features.

Flicker [37], based on TPM and DRTM of Intel x86, supports on-demand protections for SCCs. Similar to P-MAPS, Flicker incurs a world switching latency problem owing to its dependency of TPM. SICE [10] and Secure Switch [53] create isolated execution environments with an additional CPU mode known as the system management mode (SMM). However, SICE shows a few seconds of latency when entering the isolated execution environment, and Secure Switch can only build secure environments in a specific type of memory, i.e., SMRAM, which is physically limited.

Intel recently proposed Software Guard Extension (SGX) [30], containing a new set of special instructions for creating isolated execution environments. SGX is secure against various memory attacks [20] including cold-boot attacks [26] and bus monitoring attacks [49, 54]. Although it is not yet deployed in commercial products, HAVEN [11] and VC3 [46] demonstrated its effectiveness through real-world scenarios in a cloud system. Unfortunately, available of this technique is limitedly to the Intel x86 architecture.

9 Conclusion

In this paper, we have proposed OSP, a TrustZone-hypervisor hybrid protection system, which aims to provide isolated computing environments for SCCs in an efficient and secure manner. OSP reduces the virtualization overhead by leveraging the on-demand hypervisor activation scheme that is efficiently carried out with assistance of TrustZone. To measure the performance of OSP on a mobile device, we performed a set of experiments with ODROID-XU3-Lite using the mobile processor adopted by latest commercial smartphones. Our evaluations have shown that OSP achieves very low performance overhead during hypervisor hibernation (near 0 %) and efficiently protects SCCs with low activation latency ($< 100\ \mu$s).

Acknowledgments. We thank our shepherd Theodore Ts'o, the reviewers and Kang G. Shin for their insightful remarks that improved the paper. This work was partly supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No. R0190-16-2010, Development on the SW/HW modules of Processor Monitor for System Intrusion Detection), the National Research Foundation of Korea(NRF) grant funded by the Korea government (MSIP) (No. 2014R1A2A1A10051792), the MSIP(Ministry of Science, ICT and Future Planning), Korea, under the ITRC(Information Technology Research Center) support program (IITP-2016-R0992-16-1006) supervised by the IITP(Institute for Information & communications Technology Promotion), Inter-University Semiconductor Research Center (ISRC) and the Brain Korea 21 Plus Project in 2016.

References

[1] Performance Measurement on ARM. http://www.pengutronix.de/development/kernel/arm-benchmarks-20100729_en.html.

[2] ALVES, T., AND FELTON, D. Trustzone: Integrated hardware and software security. *ARM white paper* (2004).

[3] ARM. System memory management unit (smmu). http://www.arm.com/products/system-ip/controllers/system-mmu.php.

[4] ARM. Primecell infrastructure amba3 trustzone memory adapter (bp141). In *ARM DTO 0017A* (2004).

[5] ARM. Primecell infrastructure amba3 trustzone protection controller (bp147). In *ARM DTO 0015A* (2004).

[6] ARM. Architecture reference manual armv7-a and armv7-r edition. In *DDI 0406C* (2009).

[7] ARM. Trustzone address space controller (tzc-380). In *ARM DDI 0431B* (2010).

[8] ARM. Generic interrupt controller architecture version 2.0. In *ARM IHI 0048B* (2013).

[9] AZAB, A. M., NING, P., SHAH, J., CHEN, Q., BHUTKAR, R., GANESH, G., MA, J., AND SHEN, W. 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* (2014).

[10] AZAB, A. M., NING, P., AND ZHANG, X. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)* (2011).

[11] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with haven. In *Proceedings of 11th USENIX Symposium on Operating Systems Design and Implementation* (2014).

[12] BHARGAVA, R., SEREBRIN, B., SPADINI, F., AND MANNE, S. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (2008).

[13] BICKFORD, J., O'HARE, R., BALIGA, A., GANAPATHY, V., AND IFTODE, L. Rootkits on smart phones: attacks, implications and opportunities. In *Proceedings of the eleventh workshop on mobile computing systems & applications* (2010).

[14] BRUMLEY, D., AND SONG, D. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium* (2004).

[15] CHAMPAGNE, D., AND LEE, R. B. Scalable architectural support for trusted software. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on* (2010).

[16] CHECKOWAY, S., AND SHACHAM, H. Iago attacks: why the system call api is a bad untrusted rpc interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013).

[17] CHEN, X., GARFINKEL, T., LEWIS, E. C., SUBRAHMANYAM, P., WALDSPURGER, C. A., BONEH, D., DWOSKIN, J., AND PORTS, D. R. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (2008).

[18] CHENG, Y., DING, X., AND DENG, R. Appshield: Protecting applications against untrusted operating system. *Singaport Management University Technical Report, SMU-SIS-13* (2013).

[19] CHHABRA, S., ROGERS, B., SOLIHIN, Y., AND PRVULOVIC, M. Secureme: a hardware-software approach to full system security. In *Proceedings of the international conference on Supercomputing* (2011).

[20] COLP, P. J., ZHANG, J., GLEESON, J., SUNEJA, S., DE LARA, E., RAJ, H., SAROIU, S., AND WOLMAN, A. Protecting data on smartphones and tablets from memory attacks. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2015).

[21] DALL, C., AND NIEH, J. 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 (ASPLOS)* (2014).

[22] DAVID, F. M., CARLYLE, J. C., AND CAMPBELL, R. H. Context switch overheads for linux on arm platforms. In *Proceedings of the 2007 Workshop on Experimental Computer Science* (2007).

[23] DEVRIENT, G. . Mobicore. http://www.gi-de.com/en/trends_and_insights/mobicore/trusted-mobile-services.jsp.

[24] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (2003).

[25] GE, X., VIJAYAKUMAR, H., AND JAEGER, T. Sprobes: Enforcing kernel code integrity on the trustzone architecture.

[26] HALDERMAN, J. A., SCHOEN, S. D., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J. A., FELDMAN, A. J., APPELBAUM, J., AND FELTEN, E. W. Lest we remember: Cold boot attacks on encryption keys. In *Proceedings of the 17th Conference on Security Symposium* (2008).

[27] HARDKERNEL. Odroid. http://www.hardkernel.com.

[28] HOFMANN, O. S., KIM, S., DUNN, A. M., LEE, M. Z., AND WITCHEL, E. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013).

[29] INC, A. Secure virtual machine architecture reference manual, 2005.

[30] INTEL. Software guard extensions programming reference.

[31] LABS, M. Threats report. http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2015.pdf, 2015.

[32] LI, W., MA, M., HAN, J., XIA, Y., ZANG, B., CHU, C.-K., AND LI, T. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems* (2014).

[33] LI, Y., MCCUNE, J., NEWSOME, J., PERRIG, A., BAKER, B., AND DREWRY, W. Minibox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference (ATC)* (2014).

[34] LIE, D., THEKKATH, C. A., AND HOROWITZ, M. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (2003).

[35] LOWELL, D. E., SAITO, Y., AND SAMBERG, E. J. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems* (2004).

[36] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., AND PERRIG, A. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy* (2010).

[37] MCCUNE, J. M., PARNO, B., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems* (2008).

[38] MISRA, S. C., AND BHAVSAR, V. C. Relationships between selected software measures and latent bug-density: Guidelines for improving quality. In *Computational Science and Its ApplicationsICCSA 2003*. 2003.

[39] NEIGER, G., SANTONI, A., LEUNG, F., RODGERS, D., AND UHLIG, R. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal* (2006).

[40] NORDHOLZ, J., VETTER, J., PETER, M., JUNKER-PETSCHICK, M., AND DANISEVSKIS, J. Xnpro: Low-impact hypervisor-based execution prevention on arm. In *Proceedings of the 5th International Workshop on Trustworthy Embedded Devices* (2015).

[41] OMOTE, Y., SHINAGAWA, T., AND KATO, K. Improving agility and elasticity in bare-metal clouds. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (2015).

[42] RAVI, S., ULHAS, W., AND PRASHANT, D. Dynamic software application protection. *Intel Technology Journal* (2009).

[43] REN, J., QI, Y., DAI, Y., WANG, X., AND SHI, Y. Appsec: A safe execution environment for security sensitive applications. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2015).

[44] SANTOS, N., RAJ, H., SAROIU, S., AND WOLMAN, A. Trusted language runtime (tlr): enabling trusted applications on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications* (2011).

[45] SANTOS, N., RAJ, H., SAROIU, S., AND WOLMAN, A. Using arm trustzone to build a trusted language runtime for mobile applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2014).

[46] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. Vc3: Trustworthy data analytics in the cloud using sgx. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015).

[47] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles* (2007).

[48] SIERRAWARE. Open virtualization - arm trustzone and arm hypervisor open source software. `http://www.sierraware.com`.

[49] SOLUTIONS, E. Analysis tools for ddr1, ddr2, ddr3, embedded ddr and fully buffered dimm modules. `http://www.epnsolutions.net/ddr.html`, 2014.

[50] SUH, G. E., CLARKE, D., GASSEND, B., VAN DIJK, M., AND DEVADAS, S. Aegis: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing* (2003).

[51] SUN, H., SUN, K., WANG, Y., AND JING, J. Trustotp: Transforming smartphones into secure one-time password tokens. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015).

[52] SUN, H., SUN, K., WANG, Y., JING, J., AND WANG, H. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on* (2015).

[53] SUN, K., WANG, J., ZHANG, F., AND STAVROU, A. Secureswitch: Bios-assisted isolation and switch between trusted and untrusted commodity oses. In *Network and Distributed System Security Symposium (NDSS)* (2012).

[54] SYSTEM, F. Ddr2 800 bus analysis probe. `http://www.futureplus.com/download/datasheet/fs2334_ds.pdf`, 2006.

[55] TA-MIN, R., LITTY, L., AND LIE, D. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI)* (2006).

[56] UNIFIED, E. Inc. unified extensible firmware interface specification, 2014.

[57] WHEELER, D. A. Sloccount. `http://www.dwheeler.com/sloccount`, 20015.

[58] YANG, J., AND SHIN, K. G. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (2008).

[59] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy* (2009).

[60] YEFREMOV, D., AND IAKOVENKO, P. An approach to on the fly activation and deactivation of virtualization-based security systems. In *Proceedings of the Spring/Summer Young Researchers Colloquium on Software Engineering* (2010).

[61] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012).

[62] ZHOU, Z., YU, M., AND GLIGOR, V. Dancing with giants: Wimpy kernels for on-demand isolated i/o. In *Proceedings of the IEEE Symposium on Security and Privacy* (2014).