Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World

Ahmed M. Azab[1] Peng Ning[1,2] Jitesh Shah[1] Quan Chen[2]
Rohan Bhutkar[1] Guruprasad Ganesh[1] Jia Ma[1] Wenbo Shen[2]

[1] Samsung KNOX R&D, Samsung Research America
{a.azab, peng.ning, j1.shah, r1.bhutkar, g.ganesh, jia.ma}@samsung.com

[2] Department of Computer Science, NC State University
{pning, qchen10, wshen3}@ncsu.edu

ABSTRACT

TrustZone-based Real-time Kernel Protection (TZ-RKP) is a novel system that provides real-time protection of the OS kernel using the ARM TrustZone secure world. TZ-RKP is more secure than current approaches that use hypervisors to host kernel protection tools. Although hypervisors provide privilege and isolation, they face fundamental security challenges due to their growing complexity and code size.

TZ-RKP puts its security monitor, which represents its entire Trusted Computing Base (TCB), in the TrustZone secure world; a safe isolated environment that is dedicated to security services. Hence, the security monitor is safe from attacks that can potentially compromise the kernel, which runs in the normal world. Using the secure world for kernel protection has been crippled by the lack of control over targets that run in the normal world. TZ-RKP solves this prominent challenge using novel techniques that deprive the normal world from the ability to control certain privileged system functions. These functions are forced to route through the secure world for inspection and approval before being executed. TZ-RKP's control of the normal world is non-bypassable. It can effectively stop attacks that aim at modifying or injecting kernel binaries. It can also stop attacks that involve modifying the system memory layout, e.g, through memory double mapping.

This paper presents the implementation and evaluation of TZ-RKP, which has gone through rigorous and thorough evaluation of effectiveness and performance. It is currently deployed on the latest models of the Samsung Galaxy series smart phones and tablets, which clearly demonstrates that it is a practical real-world system.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

http://dx.doi.org/10.1145/2660267.2660350.

General Terms

Security

Keywords

Integrity Monitoring; ARM TrustZone; Kernel Protection

1. INTRODUCTION

Despite recent advances in systems security, attacks that compromise the OS kernel still pose a real threat [1,5,27,37]. Such attacks can access system sensitive data, hide malicious activities, escalate the privilege of malicious processes, change the OS behavior or simply take control of the system.

Previous Attempts: Traditionally, kernel protection was done using security tools that run in the same address space and privilege level as the kernel. This approach is not secure because an attack compromising the kernel would consequently compromise these security tools as well. To protect the kernel, security protection tools should be properly isolated. Recent efforts to achieve this objective can be divided into two categories: Hypervisor-based approaches and hardware-based approaches.

Hypervisor-based approaches, such as [12,20,23,24,30,31, 35,38,43–45,50,52], use virtualization to provide their security tools with high privilege and isolation. Nevertheless, hypervisors face security challenges of their own. They become more complex because they are required to do more tasks for system management and resource distribution. Hence, they become subject to numerous vulnerabilities [2,3,6,7,48,49, 56]. Although dedicating the whole virtualization layer to hosting security tools will decrease the chances of exploitation by reducing the hypervisor code size, this is not a practical solution because it will deprive the system from using other virtualization capabilities. Furthermore, hardware-based virtualization is not supported on all platforms. For example, the widely-used Qualcomm Snapdragon MSM8974 and APQ8084 ARM processors do not implement the hypervisor extension. Hence, these platforms need an alternative method to host their security protection tools.

Hardware-based approaches use a different type of hardware protection. Realizing the security threats to kernels and hypervisors, popular hardware platforms introduced a new secure and isolated execution environment, which is sometimes called the "secure world." The secure world consists of a thinner, more secure software layer dedicated to

hosting security services. Moreover, it has both limited interface with the normal world and hardware protection against illegal memory access from the normal world kernel or hypervisor. Examples of the secure world include Intel TXT [29], AMD SVM [8] and ARM TrustZone [9].

Hardware-based approaches, such as [13, 41, 55], use the secure world to host kernel security protection tools. However, these systems are crippled by their inability to closely monitor events that happen inside the target kernel. Therefore, they do periodic kernel integrity checking, which has limited effectiveness because it can only detect attacks after they have already happened. Moreover, it can miss the attack altogether, if the traces are properly hidden.

In summary, previous research shows that using the hypervisor for kernel protection has security risks. At the same time, hardware-based isolation lacks a critical capability, which is event-driven monitoring. Hence, there is a trade off between the isolation provided to the security tool and the control it possesses over the system to be measured.

Introducing TZ-RKP: TrustZone-based Real-time Kernel Protection (TZ-RKP) provides the basic architecture required to protect the OS kernel using a security monitor that is entirely located within the secure world of ARM TrustZone. TZ-RKP enjoys the strongest possible isolation from potential attacks. At the same time, it gives the security monitor full control over the target kernel's memory management. Moreover, it can intercept critical events and inspect their impact on security before allowing them to get executed. As a result, TZ-RKP solves the dilemma that previous approaches face because it does not have to trade off between isolation and effectiveness.

Current TZ-RKP design relies on hardware features that are specific to the ARM architecture, which is used by the majority of today's mobile devices. Mobile devices, such as smart phones and tablets, are getting more ubiquitous. They also expose many interfaces and carry sensitive data, hence compromising the kernel of such devices can cause devastating consequences [15]. Having a trustworthy kernel allows the target platform to have a secure foundation that hardens the security of higher level system components.

Technical Challenges: A security protection tool running in the TrustZone secure world has two main technical challenges. First, the normal world, which hosts the target OS, has full control over its own resources, such as its physical memory and its Memory Management Unit (MMU). Intuitively, such control would allows attackers that exploit the normal world to bypass any regular security monitoring technique. For instance, attackers can change the location of interrupt handlers to create invisible hooks that hijack the kernel's control flow [22].

Second, the TrustZone secure world is not capable of intercepting many of the critical events that occur in the normal world, e.g., page fault exceptions and execution of system control instructions. If these events are not trapped by the secure world, their impact on the system security can go unnoticed by the security monitor.

An Overview of TZ-RKP: TZ-RKP achieves two important objectives. First, it completely prevents running unauthorized privileged code on the target system, which is accomplished by preventing modification of the target kernel code, injection of unauthorized code into the kernel, or execution of the user space code in the privileged mode. Second, it prevents kernel data from being directly accessed by user level processes. This includes preventing double mapping of physical memory that contains critical kernel data into user space virtual memory. This is an important step toward complete security protection of the kernel using TrustZone.

In the proposed architecture, the security monitor runs in the TrustZone secure world, while the kernel to be monitored runs in the normal world. For convenience, we refer to the the security monitor as TZ-RKP and to the OS kernel being monitored as "the kernel."

TZ-RKP provides two features that allow for an effective kernel protection: (1) *Event-driven monitoring* of the normal world critical events, and (2) *memory protection* of critical parts of the normal world memory.

The first feature, event-driven monitoring, is enforced by depriving the kernel from its ability to control certain critical functions. Hence, it is forced to route requests to perform these functions through TZ-RKP. These functions include system control instructions and updates to the memory translation tables.

TZ-RKP instruments the kernel so that certain system control instructions are removed from its executable memory, which is the only memory that can execute privileged instructions in the normal world. The removed system control instructions are the ones that allow the normal world to control security critical system state, such as defining the location of memory translation tables and exception handlers. Therefore, the only way to execute these instructions is through emulating them from the secure world. We call this operation *Control Instruction Emulation*.

Memory translation tables, a.k.a. page tables, define the virtual to physical address mapping and the access permissions of virtual memory. TZ-RKP ensures that translation tables cannot be modified by the normal world. Moreover, it modifies the kernel so that requests to update the translation tables are routed through the secure world.

The second feature provided by TZ-RKP is memory protection, which ensures that the target kernel cannot be modified. It also prevents kernel code injection and return-to-user attacks. Memory protection plays an important role to guarantee that monitoring is non-bypassable. It also provides a new method to prevent kernel critical data from being directly accessed by potentially malicious user processes

The core TZ-RKP techniques presented in this paper were first invented and recorded in a provisional patent submitted on August 2012. Afterwards, a patent application that is publicly accessible was submitted on March 2013 [11].

Currently, TZ-RKP is implemented and deployed on recent models of Samsung Galaxy smartphones and tablets, including Samsung Note 3 and Samsung S5 smartphones, protecting millions of these devices. These models are among the most advanced mobile devices on the market today. TZ-RKP is deployed as a part of the TrustZone-base Integrity Measurement Architecture (TIMA), which provides a suite of TrustZone integrity measurement and security services on Samsung smartphones and tablets [47].

TZ-RKP demonstrates that the security provided by TrustZone does not have to be limited to executing isolated applications; it can be extended to provide non-bypassable security protection for the running OS. TZ-RKP has been subject to rigorous evaluation. The results show that it is feasible to implement with minor instrumentation of the kernel, resulting in an acceptable performance overhead.

Summary of Contributions: We make several technical contributions in this paper:

- We provide a complete and feasible solution for kernel protection using TrustZone, which is the most secure component of the ARM architecture.
- We provide techniques that allow the secure world to control certain critical functions of the normal world. This allows non-bypassable event-driven monitoring of the normal world, which solves the main challenge that faced previous approaches.
- We provide techniques to guarantee a non-bypassable memory protection of the normal world's critical memory regions.
- We provide a novel memory protection technique to prevent unauthorized direct access of kernel data.
- We present full implementation and rigorous evaluation of TZ-RKP using advanced mobile devices.

This paper is organized as follows: Section 2 discusses our assumptions and threat model. Section 3 provides required background information. Section 4 presents TZ-RKP in detail. Section 5 presents our implementation. Section 6 presents our experimental evaluation. Section 7 presents related work. Section 8 concludes this paper with some future research directions.

2. THREAT MODEL AND ASSUMPTIONS

Threat model: Threats against the OS kernel are divided into three main categories.

The first category includes attacks that aim at executing unauthorized privileged code inside the normal world. This includes all attacks that aim to: (1) inject malicious code into the kernel, (2) modify privileged code binaries that already exist in memory or (3) escalate the privilege of user space code. A real world example of these attacks is vroot [5], where the adversary modifies a kernel function pointer to jump back into a malicious user space code in the privileged mode. TZ-RKP prevents all these attacks without relying on the target kernel itself.

The second category includes attacks that use malicious double mapping to modify kernel data. These are special class of data attacks that exploit kernel vulnerabilities to generate user space mapping of kernel memory. A real world example of these attacks is motochopper [1]. TZ-RKP prevents this type of attacks, but this protection relies on the kernel to provide information about its own memory usage first. This information is provided before the designated memory areas are used to store data. Hence, the kernel data double mapping protection is effective on all kernel memory ranges before they become a valid target for attackers.

The third category includes another type of data attacks, where an adversary exploits a vulnerability to trick the kernel to directly change one or more critical data fields inside its own memory. A real world example of these attacks is towelroot [27], which uses a pointer manipulation vulnerability to modify the data field that defines process credentials leading to privilege escalation. Attackers may also exploit these vulnerabilities to modify control flow data, such as function pointers or return addresses. In this case, these attacks can escalate to hijack the kernel control flow, such as return oriented attacks [28,51]. Unfortunately, these attacks cannot be currently prevented or detected by TZ-RKP.

Although TZ-RKP cannot prevent the third category of kernel attacks, this does not undermine its security value for two main reasons. First, TZ-RKP's active monitoring and memory protection does not rely on the target kernel. Hence, these attacks cannot bypass TZ-RKP's control of system critical functions, even if they compromise the kernel control flow. In other words, attackers cannot change the state of the MMU, memory translation tables or even switch among different processes without passing through TZ-RKP. These forced hooking points provide a secure base for anomaly detection mechanisms, such as [33], to detect the presence of these vulnerabilities.

Second, many of the vulnerabilities found in the kernel allow limited access to kernel data, which sometimes prevent attackers from accessing certain critical memory areas, such as the kernel stack. This limited access would prevent kernel data attacks from doing effective damage through control flow hijacking. For example, a theoretical return-oriented attack would require a complete manipulation of the kernel stack to allow the attack code to jump to multiple code gadgets. Such manipulation cannot be achieved if the vulnerability only allows the attack to corrupt a single return address or a limited range of the kernel memory. In these cases, TZ-RKP would prevent the damage that can result from having these vulnerabilities exploited by attacks of the first or the second categories.

Assumptions: TZ-RKP assumes an ARM-based architecture that implements the TrustZone extensions. It also assumes that it runs as a part of the secure world, while the target OS runs in the normal world. TZ-RKP also assumes that the whole system is loaded securely, including both the secure and the normal worlds. This process is straightforward using trusted boot. Intuitively, trusted 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.

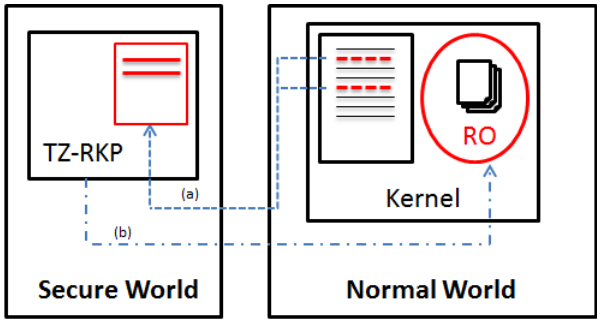
TZ-RKP also assumes that the kernel can function properly using W$\oplus$X memory mapping (i.e., using mutually exclusive memory pages for data and code). Finally, TZ-RKP assumes that the hardware platform implements the Privileged eXecute Never (PXN) memory access permission as defined by the ARM specifications.

3. BACKGROUND

TZ-RKP relies on some hardware features of the ARM architecture. These features include fixed length instructions, control of privileged mode memory, and the TrustZone security extension.

ARMv7 uses memory aligned instructions that have a fixed length of either 16 bits for the Thumb mode or 32 bits for the ARM mode. This is in contrast to other architectures like x86, where the instructions can be located at any offset in memory. Having a fixed-length aligned instruction set allows TZ-RKP to restrict the presence of certain instructions within the normal world kernel.

ARM provides two code execution modes; privileged and unprivileged. It provides control flags that prevent certain virtual memory pages from executing privileged code. It also provides control flags to mark certain memory areas, such as the kernel memory, as privileged. It prevents unprivileged code from accessing privileged memory. Intuitively, the un-



(a) Control instructions and page table update functions are replaced by traps to the secure world

(b) Page tables are mapped read-only so they cannot be directly modified by the kernel

Figure 1: TZ-RKP Basic Components

privileged mode is used to run the user process, while the privileged mode is used to run the kernel. TZ-RKP relies on this access control mechanism to enforce its protection.

TrustZone is a security extension defined by ARM. The basic idea is to logically partition the computing platform into two execution domains: the normal world and the secure world. To facilitate context switch between the two worlds, a special processor mode, known as the monitor mode, is introduced. The monitor mode, which is part of the secure world, is the only entry point from normal world to secure world. Execution jumps from the normal world to the secure world by explicitly issuing the Secure Monitor Call (`SMC`) instruction. The secure world can access the full range of the physical memory and all hardware peripherals. It can also emulate the execution of normal world control instruction. On the other hand, some physical memory ranges and hardware peripherals can be restricted to be only accessed by the secure world. Hence, these secure physical memory and hardware peripherals enjoy full hardware-based protections from attacks that can potentially compromise the normal world.

4. TZ-RKP DESIGN

TZ-RKP can be implemented on any ARM processor that supports TrustZone. The design presented here solely relies on the architectural specifications defined by ARM [10]. TZ-RKP can be customized to monitor any target OS. The current implementation is customized to target Android's Linux Kernel. We use this implementation as an example to clarify our approach.

Figure 1 shows the system architecture. TZ-RKP runs in the TrustZone secure world, while the target kernel runs in the normal world. The target kernel is forced to request TZ-RKP to perform two operations on its behalf: (1) emulating control instructions that change the system state and (2) updating the target OS memory translation tables.

In the following, we present the control instruction emulation. Then, we present how TZ-RKP traps updates to memory translation tables. Afterwards, we discuss how TZ-RKP uses these two features to achieve the required memory protection. Next, we present kernel data double mapping prevention. Finally, we summarize the security guarantees provided by TZ-RKP.

4.1 Control Instruction Emulation

Control instructions change the system state, and hence may impact the system security. Thus, TZ-RKP implements a novel technique to deprive the target kernel from the ability to execute certain instructions so that it is forced to route the execution to TZ-RKP to emulate these instruction.

In the widely used ARMv7 32-bit architecture, control instructions are done by writing to certain coprocessor register, which can only be done by the two privileged ARM instructions `LDC` and `MCR`. In this section, we use this particular architecture to demonstrate our technique. Nevertheless, the same technique works on the 64-bit ARMv8 architecture that uses direct control instructions, rather than coprocessor register writes.

To achieve control instruction emulation, TZ-RKP overcomes a key TrustZone limitation; the secure world is not capable of preventing and intercepting control instruction execution in the the normal world. It is worth noting that this limitation is not specific to TrustZone. Although hypervisor extensions usually allow for event interception, isolated execution environments do not usually provide this feature.

The solution is to remove certain control instruction from the normal world and replace them by hooks so that they can be emulated by TZ-RKP. Nevertheless, providing a non-bypassable guarantee for this solution is not straightforward.

Our technique to make this solution secure is based on three distinct hardware features. First, instructions that change the system state are always privileged and cannot be executed in the unprivileged execution mode. Second, ARM virtual memory system allows restricting certain memory ranges from executing privileged instructions using the Privileged eXecute Never (PXN) access restriction. Therefore, privileged instructions, such as `LDC` an `MCR`, are only allowed to be fetched, i.e., executed, from certain memory areas. Finally, the ARM instruction set requires instructions to incorporate both the opcode and operands in either 16-bit THUMB mode or 32-bit ARM mode aligned memory.

Based on these properties, the technique to solve this problem can be summarized in the following principle:

> Given that (1) only certain memory regions are mapped with the permission to execute privileged instructions, (2) the same memory regions are mapped with read-only permission and (3) the same memory regions do not contain a single memory word that matches the encoding of a particular instruction, then it will be absolutely impossible for the normal world to execute this instruction, because it can neither find this instruction in its privileged executable memory nor it can execute it from the rest of the memory.

All three conditions are achieved by TZ-RKP. It enforces a strict policy that the whole normal world memory, except the memory range that hosts the verified kernel code, is mapped with the PXN flag set. It also enforces a policy that the target kernel is mapped read-only. The target OS memory layout is shown in Figure 2. TZ-RKP uses its control of the normal world memory protection, which will be discussed in Section 4.3, to enforce these policies.

The third condition is satisfied by instrumenting the kernel to replace all the `LDC` and `MCR` instructions that target specific control registers with `SMC` instructions. The instrumentation can be either done by code modification or binary

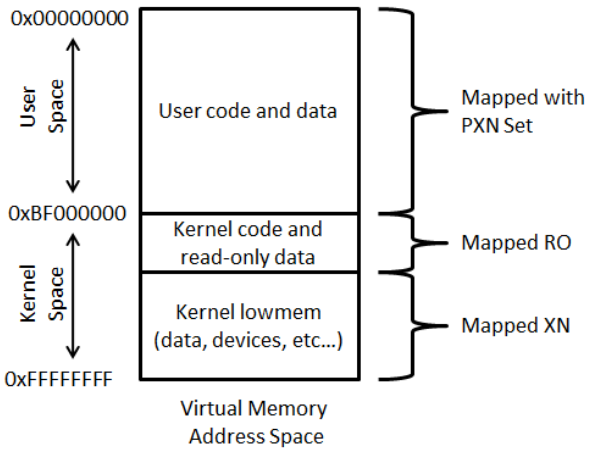


Figure 2: Target OS Virtual Memory Layout

rewriting. When TZ-RKP receives these `SMC` calls from the kernel, it emulates the required instruction by writing to the target control coprocessor registers from the secure side.

Satisfying all three conditions leads to the definitive and unique situation that the target kernel cannot execute certain privileged instructions. Hence, it loses control over certain critical functions. In our case, we select the instructions that manage the normal world MMU and define the location of exception handlers. Nevertheless, the same technique can be extended to monitor other kernel functions.

Security Guarantee: Control instruction emulation is both secure and does not involve complex implementation details.

ARM enforces alignment and fixed size instructions that include both the opcode and the operands within the same word. Hence, each instruction can be represented by a unique set of words. For example, the layout of the `MCR` instruction word incorporates all the parameters that define the target register, which are the coprocessor number the crn, crm, op1 and op2. Hence, it is easy to form a template of all `MCR` and `LDC` instructions that write to a specific coprocessor register. TZ-RKP examines every word of the kernel executable binary and replaces all words that match this template with the `SMC` instruction. This template will not match any other word within the kernel executable code, which eliminates any side effects of this operation.

This technique is secure against control flow hijacking, including return oriented attacks. Since ARM instructions take no operand to define the target register, no other word can be used to replace the write to the intended register even if an attacker completely controls the flow of the kernel.

For this technique to be secure, we have to emulate all the relevant control instructions. On ARMv7, four basic sets of instructions need to be emulated. First, system control registers, such as the `SCTLR`, that can disable the MMU, change the location of the exception handlers or disable the enforcement of instruction alignment. Second, the translation table base registers that define the base address for translation table walks. Third, the domain access control registers, such as the `DACR`, that can disable all the page table protection for select memory ranges. Fourth, the cache maintenance registers, such as `NMRR` and `PRRR`, which may need to be monitored to avoid attacks based on cache poisoning.

4.2 Trapping Translation Table Updates

TZ-RKP traps all updates to the memory translation tables so that it can always keep an up-to-date information about two key aspects of the memory layout: (1) the virtual-to-physical mapping of the normal world memory and (2) the access permission of each memory page.

The main challenge is the absence of a hardware-based control or a single entry point to update the translation tables. Translation tables can be anywhere in the physical memory. Therefore, translation table updates are normal memory writes that can only be controlled by memory access permissions.

The kernel creates a new set of translation tables for each process it creates. When a process is scheduled, it modifies the `TTBR` register to point to the translation tables corresponding to that process. The translation tables can only be accessed by the kernel. Hence, they should be only mapped within the kernel virtual memory range. They should never be mapped or accessed by a user process. Figure 2 shows the typical virtual memory layout of the Linux kernel.

The technique to intercept translation table updates starts by depriving the kernel from its own ability to do these updates. Hence, it is forced to request their modifications from TZ-RKP in the secure world.

This is achieved by modifying the access permissions of the kernel virtual address space so that the memory hosting translation tables become read-only memory. As discussed in Section 4.1, TZ-RKP intercepts writes to the `TTBR` registers. Hence, it detects the physical address of translation tables, before they are actually used. At that point, TZ-RKP modifies the kernel address space mapping to ensure that these new translation tables are mapped read-only. This process starts when the kernel boots up and is repeated with every new process that starts in the normal world.

Now that all translation tables are non-writable by the kernel, they are only modifiable by the secure world. Whenever the kernel needs to modify the translation tables, it sends a request to TZ-RKP. This can be done by either one of two ways. First, the kernel code can be modified to call the `SMC` command instead of writing the translation tables. TZ-RKP would intercept the request and carry it over once it verifies the new translation table entries. Second, the kernel can be instrumented so that the page fault handler is replaced by an `SMC` instruction. Hence, page faults will trap into TZ-RKP, which in turn emulate the ones that originate from writing translation table entries. The second method is described in more detail in Section 5.2.

Security Guarantee: Trapping updates to translation tables is non-bypassable if two requirements are met: 1) the kernel is not able to hide its translation tables from TZ-RKP, and 2) the kernel is not able to modify its translation tables without TZ-RKP's knowledge.

As long as control instruction emulation is non-bypassable, the first requirement will be satisfied because the kernel cannot use its translation tables without updating the `TTBR` hardware registers. The second requirement is satisfied through TZ-RKP's memory protection feature, which will be discussed in detail in Section 4.3.

Trapping translation table updates does not require trust in the kernel even if the implementation relies on the kernel to explicitly request the translation table updates from TZ-RKP. Since the translation tables are mapped read-only and cannot be modified by the kernel, there is no way for

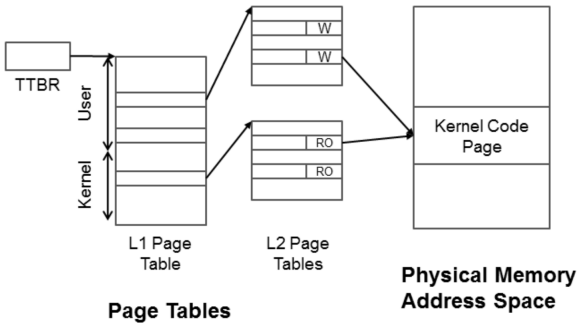


Figure 3: Double mapping of a physical memory area to both user space and kernel space

a compromised kernel to skip these hooks. Moreover, the new values of the translation table entries to be written is inspected by TZ-RKP so verify that they does not violate the security guarantees. A compromised kernel sending malicious requests to update the translation tables will be simply blocked by TZ-RKP.

4.3 Memory Protection

The main objective of memory protection is to guarantee that both control instruction emulation and trapping translation table updates are non-bypassable. The former requires preventing unauthorized writes to the kernel code, while the later requires preventing unauthorized writes to translation tables.

The main challenge of memory protection lies in *double mapping*. Double mapping happens when the same physical memory is mapped to multiple virtual memory addresses. An example of double mapping is shown in Figure 3. To avoid bypassing the memory protection using double mapping, TZ-RKP has to enforce the protection on every virtual mapping of the protected memory. This is specifically hard for dynamic memory ranges, such as translation tables, which can be created and discarded anywhere in the system memory and at anytime. A potentially malicious writable mapping can be created either after or before a certain memory region is used to host translation tables.

To achieve memory protection, TZ-RKP separates the target OS virtual memory into three distinct ranges: the kernel code, user memory and the kernel `lowmem`. The last is a term used to describe the physical memory that is constantly mapped by the kernel. The kernel allocates memory from the lowmem when it needs to create new objects such as translation tables.

The kernel code is the only memory range that has the privileged execution permission and it is mapped read-only. TZ-RKP enforces the strict policy that user space virtual memory ranges are mapped as Privileged eXecute Never (PXN). The kernel lowmem is mapped as non-executable privileged memory. Hence, it cannot be accessed by unprivileged code (i.e., user processes). Whenever TZ-RKP decides to protect a certain memory region (e.g., a translation table), it modifies its lowmem mapping to be read-only.

To protect against double mapping, TZ-RKP records the state of every page of the physical memory. Whenever a new virtual-to-physical mapping is about to be created, TZ-RKP checks the new access permission given to the physical memory page against the stored state to verify that there is no violation to the memory protection. The state of physical frames is stored in an array called the *physmap*. Each entry of the array corresponds to a 4KB physical memory page. Each entry has some status bits and a counter to indicate how many virtual page mappings each physical page has.

The physmap decides which physical pages should be protected. TZ-RKP marks physical pages that are currently used by kernel code or translation tables as protected. Any request to create a writable mapping of these pages is rejected. TZ-RKP also prevents this scenario from happening in the reverse direction. If a page is mapped writable to a user process, TZ-RKP will record this event in the physmap. If TZ-RKP receives a request to use the same page as a translation table, then it will reject the request knowing that this translation table can be modified by the user process that has the previous writable mapping. TZ-RKP also uses a counter to follow the status of pages with multiple mappings. The counters are incremented and decrements as pages get mapped and unmapped.

Security Guarantee: Memory protection relies on the following principle: As long as all virtual-to-physical mappings of a physical memory page only allow for the read-only permission, it will be impossible for this page to be modified by the normal world.

The memory pages protected by TZ-RKP will have only one virtual address mapping, which is the `lowmem`. TZ-RKP modifies the `lowmem` mapping of protected pages to be read-only. Hence, neither kernel code nor translation tables would be writable by the normal world even if the kernel is completely compromised.

As mentioned in Section 4.2, the memory protection guarantees that updates to translation tables are trapped by TZ-RKP. Using the physmap, TZ-RKP inspects the virtual-to-physical mapping created by each translation table update or decides to reject the update, if it violates the security principal stated above.

4.4 Kernel Data Double Mapping Prevention

Kernel data structures are critical to the system security. Maliciously modifying kernel data can lead to wide range of damage from privilege escalation to process hiding [14]. Since TZ-RKP completely protects the kernel code base and prevents "return-to-user" attacks, there are only two possible methods to exploit kernel data. The first is through double mapping the memory hosting kernel data into the address space of the malicious process. The second is to alter the kernel control flow so that it maliciously modifies its own data (e.g., using pointer manipulation or overflow [4]).

In this paper, we focus on defeating the first class of attacks. Double mapping is a real threat to the kernel. For instance, the Motochopper [1] android exploit used an integer overflow to trick the kernel into mapping a huge range of the physical memory into the address space of the attacking process. We leave the second class of kernel attacks to future work.

To prevent malicious double mapping of the kernel data, TZ-RKP should ensure that physical memory pages hosting this data are not mapped to user space processes. They can only be mapped as privileged pages that cannot be accessed by the user space. TZ-RKP enforces this policy using its control of the guest translation table. TZ-RKP rejects all translation table entries that map kernel data to user space.

Nevertheless, there is a big challenge that needs to be solved first. The kernel data is scattered over the physical memory among other memory areas that belong to the user processes, which need to be double mapped so that the system functions correctly.

To solve this challenge, TZ-RKP relies on the target kernel to inform it about the location of its critical data. Modern kernels use allocation techniques, such as slub in Linux, to aggregate data objects together so that fragmentation is minimized. TZ-RKP embeds hooks inside the kernel code so that it gets informed whenever a new memory area is going to be allocated to the kernel. It then marks this memory area in the physmap so that it prevents it from being double mapped to writable memory anywhere in the system.

Security Guarantee: This protection is effective against attacks that use double mapping to exploit kernel data. In our experimental evaluation, implementing this protection stopped the Motochopper kernel exploit.

As discussed in Section 2, this protection has a different threat model than the other techniques. While all other TZ-RKP techniques do not require trust in the kernel, data double mapping prevention requires the kernel to inform TZ-RKP about the allocated data memory areas. This dependency is imminent because the allocation can happen anytime and anywhere in the system memory.

This dependency does not weaken the protection. The kernel is assumed to be secure when it sends the information to TZ-RKP because this happens before the data pages are allocated. TZ-RKP prevents the data from being modified, except by the kernel itself. Thus, it prevents this class of attacks from exploiting the kernel and sending falsified information about memory allocations.

Indeed, this interface is not safe against the second class of attacks, e.g., return oriented attacks. These attacks can directly modify the kernel data despite the presence of our data double mapping protection, hence they are out of the scope of this paper.

4.5 Security Analysis

Throughout section 4, we discussed the security guarantee provided by each TZ-RKP feature. These guarantees are interdependent and compliment each other. In this section, we summarize these guarantees by discussing how they defeat security threats. We categorize security threats by the attack surface they target.

Kernel Attack Surface: The first threat to the kernel is to run a modified binary during system load-time. As mentioned in Section 2, TZ-RKP assumes the presence of trusted boot to defeat that threat.

Trusted boot is also critical to TZ-RKP. It is the only guarantee that the loaded kernel binary is instrumented to remove control instructions, as mentioned in Section 4.1. Once that binary is loaded, the memory protection will guarantee that it cannot be modified.

The security guarantees provided by both control instruction emulation and memory protection are non-bypassable because there is nowhere in the normal world that can execute the emulated instructions or modify the translation tables. These security guarantees are also interdependent. The control instruction emulation relies on the memory protection to guarantee that the kernel will not be modified, while the memory protection relies on the control instruction emulation to detect the presence of new translation table. Hence, TZ-RKP has to be implemented as a complete integrated solution to provide proper protection. Nevertheless, neither of these two security guarantees require trust in the kernel code during system runtime.

Modern kernels usually support extending their code using loadable modules. These modules are not verified by trusted boot because they are loaded after the system starts. TZ-RKP supports these kernel code extensions as long as they are known to the system and they are subject to instrumentation to remove control instructions from their binaries. We rely on an orthogonal system to verify the binaries of loadable kernel modules before they are loaded into memory. This system restricts the modules to be loaded a predefined set of modules. This requirement does not violate any of the common use cases of Android.

Return-to-user Attacks: Another threat that aims at hijacking privileged execution is return-to-user attacks, where a kernel exploit is used to return the execution to user space in the privileged mode. User space code will be then allowed to execute privileged instructions and access privileged data. As discussed in [32], these attacks can have a huge impact on the system security. Moreover, they pose a threat to TZ-RKP because they potentially allow the unverified user space code to execute control instructions, which breaks the non-bypassable monitoring guarantee.

As mentioned in Section 4.3, TZ-RKP stops these attacks using the PXN access restriction of user space memory. Hence, the processor prohibits executing code from this region while running in the privileged mode. Another hardware mechanism that can also be used is the Unprivileged Write Permission implies PL1 Execute Never (UWXN) provided by ARM. UWXN prevents user pages from running privileged code, which can be used to provide additional security restrictions besides the PXN flag.

Kernel Data Attacks: As mentioned in Section 4.4, TZ-RKP prevents direct access of kernel data from user space. User space will have the PXN access restriction so it cannot escalate its privilege to access the privileged kernel data. Moreover, TZ-RKP will prevent kernel data from being double-mapped to user space.

For this particular security guarantee, TZ-RKP uses the kernel to get information about the memory layout. This is acceptable because this type of attacks can only originate from the user space. The philosophy behind using TZ-RKP to provide this protection is to avoid vulnerabilities that usually exist in the huge kernel code base. Such vulnerabilities have been previously exploited [1] to give unauthorized access to kernel data.

DMA Attacks: Hardware peripherals are sometimes allowed to bypass the MMU and do a Direct Memory Access (DMA) to the physical memory. Attackers may use an exploit to trick the kernel into allowing these devices to directly access its memory.

DMA attacks are not a threat to TZ-RKP. The secure world side of TZ-RKP is inherently secure against DMA using the TrustZone protection provided to the secure world. DMA attacks that aim at modifying the kernel binary or the translation tables can be stopped by TZ-RKP using instruction emulation. TZ-RKP needs to further instrument the kernel so that the kernel cannot manage the DMA controller. The exact implementation will differ according to the used hardware platform. For instance, if the DMA controller is managed by a memory mapped register, then TZ-

RKP would make this memory read-only to enforce the interception of DMA mapping. Finally, TZ-RKP would examine these mapping to confirm that none of the hardware devices is allowed access to critical system memory.

Control Flow Attacks: As discussed in Section 2, TZ-RKP cannot prevent all types of kernel data attacks. Some type of kernel data attacks can cause the kernel control flow to change, which may lead to malicious behavior without the change of the kernel code. One example is return oriented attacks [16, 28, 51]. Another example is using a kernel integer overflow or pointer manipulation to trick the kernel code into maliciously modifying its own memory, e.g., [4]. Recent research shows that these attacks are hard to prevent [26,54]. These attacks are considered out of the scope of this paper. Nevertheless, TZ-RKP guarantees that these attacks will not subvert its monitoring framework or load new unauthorized privileged code. Completely eliminating these attacks requires orthogonal techniques, such as [17, 19, 21, 57].

Attacks Against TrustZone: TZ-RKP is hosted by the TrustZone secure world, hence all its security properties, including the non-bypassable monitoring, are conditional to the security of TrustZone. A recent publication [46] showed that one of the TrustZone interfaces can be exploited by an integer overflow vulnerability.

Despite this specific attack, the TrustZone secure world is still the most secure software layer in the whole system because it enjoys hardware protection to its memory from normal world software. Moreover, the secure world has a small controlled interface to the target kernel, which effectively reduces the TrustZone attack surface. This is evident by the fact that exploits that break the TrustZone protection, such as [46], are seldom compared to those that break the kernel, which has a much bigger attack surface and trusted computing base.

5. IMPLEMENTATION

TZ-RKP is implemented and deployed on recent Samsung smartphones and tablets, including Samsung Galaxy Note 3 and Samsung Galaxy S5. In this section, we present the implementation of TZ-RKP on Samsung Galaxy Note 3, which is equipped with Snapdragon quad-core MSM8974 processor from Qualcomm. The target OS is Android Jelly Bean version 4.3, which runs on Linux kernel version 3.4. The kernel uses two-level short descriptor translation tables format as specified by ARM. In the following, we present more details about our implementation.

5.1 System Organization

In our implementation, the target kernel runs in the normal world, while the TrustZone secure world is controlled by a propitiatory Trusted Execution Environment (TEE) that hosts TZ-RKP and other security services. There are three privileged levels of execution in the secure world. The highest is PL2, which hosts the monitor mode that controls the entry point between the normal world and the secure world. The other two are PL1 and PL0, who respectively host the TEE's supervisor code and trusted applications.

Only signed and verified applications are allowed to run in TrustZone. These applications, which are carefully inspected for security, have minimal interface to the normal world. Hence, the attack surface of the secure world as a whole is much smaller than that of the normal world kernel. TZ-RKP is also isolated from other TrustZone applications because it runs in the monitor mode, which is the most secure and privileged level of the secure world and consequently the whole system.

Running TZ-RKP in the monitor mode allows it to read and write control coprocessor registers of the normal world, which is a required feature for critical instruction emulation. Moreover, running TZ-RKP in the monitor mode reduces the performance overhead by avoiding mode switches inside the secure world.

5.2 Kernel Instrumentation

In the current implementation, we directly modify the kernel's source code to place hooks upon writing to critical coprocessor registers and upon modifying translation tables. These hooks execute an `SMC` instruction to jump to the secure world. To differentiate the `SMC` instructions called by TZ-RKP's kernel hooks from those requesting other TrustZone services, we use a command ID that is placed in a general purpose register upon the `SMC` call. Once the execution jumps to the monitor mode, TZ-RKP checks that register value to determine the call type.

As discussed in Section 4, using these hooks does not imply that TZ-RKP security relies on the target kernel because the enforcement does not allow the kernel to either execute critical instructions or to modify memory translation tables without going through TZ-RKP. Moreover, the hook interfaces cannot be exploited by a compromised kernel (e.g., using a man-in-the-middle type of attack) because TZ-RKP inspects the values passed by the kernel and confirms they do not have an impact on the system security before emulating the requested operation.

In addition to code instrumentation, we also use a binary analysis tool to confirm that all critical control coprocessor writes are removed from the target kernel binary and replaced by TZ-RKP hooks. As discussed in Section 4.1, this is a basic requirement for TZ-RKP.

Another method to implement TZ-RKP is to instrument the kernel binary by inserting an `SMC` instruction in place of the page fault exception handler. Hence, every page fault would trap into TZ-RKP, which in turn emulates the ones that result from trying to write the read-only translation table entries. Intuitively, this method is more suitable if the kernel code is not available.

We implemented a proof-of-concept system using binary instrumentation. In that prototype, the protection is initialized by changing the mapping of the first set of translation tables to make them read-only in the same way as the original system, which will be discussed in more details in Section 5.3.

After the system is initialized, the kernel would attempt to normally modify these translation tables, which causes a page fault that jumps into the secure world to be trapped by TZ-RKP. In turn, TZ-RKP inspects the `DFAR` register that stores the virtual address that caused the page fault. If this address matches one of the protected memory translation tables, TZ-RKP first inspects the value to be written to check that it conforms with its security guarantees. Afterwards, it updates the target translation table entry to the new value before returning to the normal world.

To find the value to be written to the memory translation tables, TZ-RKP decodes the instruction that caused the fault to identify the general purpose register that holds that value. The instruction address is stored in the banked

link register (LR) of the normal world's abort mode, which is the execution mode that handles page faults in ARMv7.

Binary instrumenting the kernel faces two challenges that are related to the system performance: 1) There is an extra mode switch that occurs as a result of the data abort and 2) it is hard to modify the kernel behavior to achieve non functional requirements, such as enhancing the system performance, without modifying the kernel's source code. These problems can be solved with more research effort. The main purpose of this prototype is to show that it is feasible to implement TZ-RKP on systems with closed source code.

5.3 Initialization and Basic Operations

Secure system initialization is an essential requirement of TZ-RKP, which needs a guarantee that only the instrumented kernel can be loaded into the normal world. Our implementation achieves this property using trusted boot, which is a part of the TIMA services suite that runs on most recent Samsung mobile devices. The trust chain starts from the boot loader, which verifies the signature of the kernel binary before loading it into memory. Hence, only known kernel binaries are allowed to run on these devices. Indeed, the hardware vendor only signs kernel binaries that are instrumented to comply with TZ-RKP's requirements.

After the instrumented kernel is loaded, it has to trap to TZ-RKP to initialize its MMU. Given that the loaded kernel is deprived from the privilege of setting its own TTBR register, then it has to jump to TZ-RKP to emulate this operation. Upon the first write to the TTBR register, TZ-RKP examines every entry of the translation tables to verify that the corresponding address translation and access permissions do not violate its security guarantees. It verifies that that all translation table entries that map privileged code page are mapped read-only. It also verifies that all user space memory ranges have the PXN flag set and that the translation tables themselves are mapped read-only, so that they cannot be modified by the target kernel. Finally, TZ-RKP updates its own data structures to reflect the current status of the normal world. For instance, it updates the physmap to mark the physical memory pages that host the translation tables as protected, so they are never allowed to have another writable mapping.

These operations are repeated when the TTBR register is updated to point to new translation tables, which occurs whenever a new process is spawned. These steps are also repeated when the translation tables are updated. If the update is in the L1 translation table, it implies that an L2 translation table is either being created or deleted. In both cases, TZ-RKP checks its policies, updates the physmap and changes the memory protection accordingly.

TZ-RKP removes the protection from translation tables after they are discarded. The kernel notifies TZ-RKP when a process is killed and a set of translation tables are no longer in use. This operation does not affect the security of TZ-RKP because it checks that the translation tables are not currently being used (i.e., none of the processor cores TTBR registers point to these tables). Hence, the protection is never removed from an actively running process. Even if an attacker removes the protection from the translation tables of an active process, the translation tables will be protected again once the TTBR register points to them and before they are actually used.

5.4 Performance Enhancement

TZ-RKP is already designed to optimize the execution time of a single trapped event. However, a better performance is achievable when the number of monitored events is reduced. In our implementation, which targets a kernel that uses two-level translation table, evaluation results showed that the highest number of trapped events were updates to the second level (L2) of the translation tables, which frequently happens when new processes and apps are loaded.

TZ-RKP aims at reducing the performance overhead resulting from these frequent traps. It relies on the fact that the majority of translation table updates follow a certain predictable pattern. The motivation is to leverage this pattern to reduce the number of context switches and therefore improve the performance.

This predictable pattern happens when multiple entries of L2 translation tables are modified together to map or unmap contiguous virtual memory ranges. In this case, the kernel copies all the entries to be updated along with their new values into a buffer and passes this buffer to TZ-RKP so they can be written together using a single switch to the TrustZone secure world. We call this operation *grouping* of translation tables updates.

Another predictable pattern happens when a new process is forked, because the kernel copies all the exiting translation table entries of the parent process to the translation tables of the newly forked process. We avoid causing unnecessary overhead in this event by allowing the kernel to fill the translation tables directly without trapping to TZ-RKP. Afterwards, TZ-RKP would start protecting these new translation tables, once the process is scheduled into the system (i.e, once they are pointed to by the ttbr register).

6. EXPERIMENTAL EVALUATION

TZ-RKP passed through rigorous testing and evaluation that includes both validating the effectiveness of its protection and measuring its impact on system performance and power consumption.

Effectiveness Against Attacks: We tested TZ-RKP using the real world exploits shown in Table 1. First, we tested two Android malware: Motochopper and vroot. The former was stopped by the prevention of the kernel data double mapping, as explained in Section 4.4. The later was stopped by the prevention of return-to-user attacks using the PXN access restriction, as explained in Section 4.3

Table 1: Android attacks

Attack	Vulnerability
Motochopper	Data double mapping
vroot	return-to-user
/dev/mem	Write kernel memory
CVE-2013-6432	Write kernel memory

We also wrote our own attack code that exploits two other real-world Linux vulnerabilities: 1) writing to the physical memory using the /dev/mem interface [37], and 2) a pointer manipulation vulnerability that exists in the ping_recvmsg kernel function, which is reported as CVE-2013-6432 [4]. Exploiting this vulnerability allows a user process to trick the kernel into maliciously modifying its own memory.

We use these exploits to request the kernel to write its own code, translation tables and parts of its data. Writing to kernel code failed because it is mapped read-only. We tried writing to the translation tables to change the kernel

memory access permission, which also failed because they are also mapped read-only by TZ-RKP. We also tried to use the TZ-RKP interfaces to send a request to write the translation tables, however it got rejected because the values did not comply with TZ-RKP's policies. Nevertheless, we succeeded to write the kernel data using both exploits because the kernel data double mapping prevention was not effective against these attacks. When exploited by the user space, the kernel is doing the write to its own memory using the `copy_from_user` function without the need of double mapping. As mentioned in Sections 4.4 and 4.5, attacks that trick the kernel control flow to maliciously modifying its own data are out of scope of that protection.

Performance Overhead: Performance is critical to mobile devices due to their limited resources. Indeed, TZ-RKP adds some performance overhead because it adds more processing time to the intercepted events. The execution path of an event intercepted by TZ-RKP includes a switch to the secure world, an emulation of the event, and policy checking. The main anticipated source of overhead is the time required to enter and exit the TrustZone secure world. Other operations are not anticipated to cause big overhead because they mainly rely on a single lookup of the physmap array.

As mentioned previously, TZ-RKP is deployed on multiple mobile phones and tablets. Each of them has been carefully evaluated for performance overhead. For the sake of brevity, we present the performance evaluation on only one of these devices, which is a smart phone equipped with the Qualcomm Snapdragon msm8974 quad-core processor. The target OS is Jelly Bean Android v4.3; kernel v3.4.

We perform a set of experiments to evaluate the performance of TZ-RKP. To measure the performance overhead introduced by TZ-RKP, we measure and compare the performance of an original system against a TZ-RKP system. Both the original and the TZ-RKP systems are special engineering builds that only differ in the presence of the TZ-RKP protection. Hence, the performance measurement results presented in this paper are different from those of the commercial products, which are usually subject to multiple cycles of performance enhancement and tuning before they are commercially released. The TZ-RKP system used in these experiments is our first development version, which includes all the performance enhancements presented in Section 5.4. It is worth noting that it might have been subject to slight changes to enhance the performance or change its properties before it was commercially released. In the rest of this section, we present the result of these experiments.

Overhead of Switching to the Secure World: Our first experiment is to measure the execution time needed for a full context switching to and from the isolated environment. We used ARM cycle count register (`CCNT`) to measure the number of cycles of a full round trip from the normal world to the secure world. The switching time on our target platform was around 2000 cycles.

We also tested multiple other ARM processors and we found out that the switching time varies between several hundred to several thousand cycles. This variation is mainly caused by the way TrustZone memory protection is implemented on each processor. The numbers of cycles is also dependent on the secure world software implementation, such as the way caching is configured on the secure world.

Since TZ-RKP requires frequent switching to the secure world, the variation of the number of cycles required for a single switch would greatly impact the overall system performance. Hence, we recommend that TZ-RKP's performance overhead should be carefully examined before it is implemented in a production environment.

Table 2: Benchmark results *

Benchmark	Original	TZRKP	Overhead
Quadrant	16406.17	16171.39	1.43%
Antutu	34828.67	33919.00	2.61%
GL Frame	4171.83	4165.50	0.39%
GL FPS	43.42	43.33	0.19%
Linpack			
Single Thread	438.19	434.47	0.85%
Multi Thread	1052.31	1041.39	1.43%
RL	12.55	12.74	1.51%
Vellamo			
HTML 5	2159.33	2089.33	3.24%
Metal	1224.00	1130.33	7.65%
CF-Bench	34855.33	32221.33	7.55%
Smartbench	7178.00	7081.67	1.40%
Geekbench	4066.33	3789.00	6.80%

* The presented performance measurement results are different from commercial products.

Benchmark Performance Comparison: Our second experiment is to use benchmarking tools to evaluate the performance overhead of our TZ-RKP implementation.

We used multiple benchmark tools to compare the performance of TZ-RKP with the original system. The averaged results are shown in table 2. When tested using nine different benchmarking tools, TZ-RKP shows a low overhead that ranges between 0.19% to 7.65%. The result is expected because these benchmarks incorporate a thorough evaluation of the overall system performance, which includes CPU, I/O, and memory. TZ-RKP adds overhead to a small portion of these operation. In particular, operations related to memory allocation that involve translation table updates, which are intercepted by TZ-RKP.

App Load Time Delay: Our third experiment is to measure the impact of TZ-RKP on the load time of Android apps. App load time is a critical aspect of the performance of smart phones and tablets because it impacts user experience. We also anticipated that loading an app would be impacted by TZ-RKP because of the high number of memory allocation requests required by this operation.

Measuring the app load time was done using a high definition camera by recording a video for the device display when an app is open. The video was later played back to extract the exact time needed for the app window to fill the screen. We measured the load time of each app twice. The first time incorporates creating a new process for the app, loading the app binary from the disk and drawing the app display for the first time. The second time is faster because the app process usually still exists in memory, so it only involves redrawing the app window to the display. Table 3 shows the result of this experiment. The overhead represents the extra time needed to load the app when TZ-RKP is present.

It is observed that there is a noticeable difference between the time needed to load the app when TZ-RKP is present in both the first and second times; the TZ-RKP overhead in the second load time is less than that of the first load time of the app. When we investigated this issue further, we found that the delay in the first time happens due to multiple operations that include mapping the memory needed to load the binaries of the application and its libraries. In the second

Table 3: App load time (in seconds) *

App	Original		TZ-RKP		Overhead	
	1st	2nd	1st	2nd	1st	2nd
Gallery	1.08	0.74	1.36	0.84	0.28	0.10
Clock	1.20	0.97	1.39	1.04	0.19	0.07
Calendar	1.02	0.74	1.25	0.87	0.23	0.12
Camera	2.07	1.59	2.59	2.01	0.52	0.42
Contacts	1.30	0.57	1.54	0.68	0.24	0.11
Dialer	1.06	0.47	1.29	0.56	0.23	0.09
Message	0.91	0.52	1.08	0.61	0.17	0.09
Music	1.12	0.89	1.28	1.00	0.16	0.11
Setting	0.97	0.79	1.20	0.89	0.23	0.10
Google	1.60	1.30	1.83	1.42	0.23	0.12
Calculator	0.97	0.64	1.12	0.75	0.15	0.11
Downloads	1.04	0.47	1.27	0.50	0.23	0.03

* The presented performance measurement results are different from commercial products.

time, most of the delay happens when Android maps the memory needed for the application process to control the screen display. The same operation is needed in both the first and subsequent app loading.

It is also observed that the camera app has substantially bigger delay when TZ-RKP is present. When we investigated this issue, we found out that most of the delay occurs when the kernel allocates the I/O memory needed for the camera display driver.

Power Consumption Overhead: Our fourth experiment showed that TZ-RKP does not cause a measurable increase in power consumption.

Power consumption was first measured during sleep time. TZ-RKP did not cause overhead because the CPU is in sleep mode and there is no events to be monitored. Afterwards, the powers consumption was measured on various system events that include: boot-up the device, app load time and playback of videos. In all cases, the difference in the power consumption between the original system and the one with TZ-RKP was within the margin of error of the experiment.

Device Bootup Time: We anticipated some delay in the boot up time due to the enormous number of memory allocations required. Nonetheless, system bootup is not very important to Android users because the devices are usually kept in the sleeping mode and rarely rebooted. In our experiment the average original system bootup time is 21.72 second, while the average bootup time for a TZ-RKP equipped system was 24.30 seconds.

Finally, it is worth noting that before TZ-RKP is commercially deployed, the hardware vendor may opt to reduce the performance overhead on certain types of devices by selectively suspending the protection for certain operations or by using other optimization techniques that are out of the scope of this paper.

7. RELATED WORK

The two main research directions that target integrity monitoring are: hypervisor-based and hardware-based approaches. In Section 1, we discussed the problems they face and how TZ-RKP solved them.

Another method to do system protection without relying on the hypervisor is to use a thin privileged software layer, such as microhypervisors [36, 40, 53] or formally verified microkernels [34]. These approaches are effective in isolating sensitive workloads. Nevertheless, microhypervisors monopolize the hardware virtualization extensions, which impact the portability and flexibility of these approaches. Moreover, both hypervisors and microhypervisors rely on the virtualization extension which is not available on all ARM platforms. Indeed, these approaches are more suitable than TZ-RKP for platforms that implement the hypervisor extensions but do not implement the TrustZone extensions.

Formally verified microkernels are currently far from the point of being commercialized because of the challenges they face to formally verify every operation needed to completely isolate a security tool and monitor the kernel integrity using software. Hardware isolation mechanisms, such as TrustZone, solve this problem because the hardware itself provides the protection required for the integrity monitor.

In Section 1, we discussed the main limitations of existing hardware-based solutions. Some research efforts [18, 39, 42, 58] proposed new hardware modifications to achieve event-driven monitoring using hardware-based techniques. Nevertheless, modifying the hardware is a long-term objective that is not achievable for existing systems.

Finally, SPROBES [25], a recent publication, realized the same research problem tackled by this paper. It presents a generic system that was developed in parallel to TZ-RKP to enforce kernel code integrity on TrustZone architecture. Many of the techniques presented in SPROBES overlaps with the TZ-RKP techniques presented in this paper and described in the prior patent application [11]. Nevertheless, we present a real world prototype and implementation of the proposed system. In addition, we presented addition techniques that focus on data integrity and on performance enhancement to make TZ-RKP a real world solution.

8. CONCLUSION

We introduced TZ-RKP, a system that provides kernel real-time protection from within the TrustZone secure world. TZ-RKP runs in the TrustZone secure world. Hence, it is safe from attacks that compromise the kernel, which runs in the normal world. TZ-RKP solves the dilemma that previous approaches face because it does not have to trade off isolation and effectiveness.

TZ-RKP provides non-bypassable event-driven monitoring of the target OS kernel. It uses memory protection to prevent attacks that aim at modifying the running version of the kernel. This paper also introduced the current TZ-RKP implementation and experimental evaluation. Results indicate that TZ-RKP is an effective and practical security solution for real world systems.

Since TZ-RKP completely protects the kernel code base, our future work will focus on preventing attacks that trick the kernel into maliciously modifying its own data. Such attacks can hijack the control flow of the kernel and cause severe damage. Fortunately, these attacks cannot bypass the control of TZ-RKP of the system's critical functions. Hence, it can still inspect the kernel for anomalies or traces of these attacks. Our future work will specifically focus on building a detection mechanisms that runs inside TZ-RKP to detect these attacks and take the proper corrective action.

Acknowledgments

We would like to thank Xun Chen, Michael Grace, Kirk Swidowski, Vinod Ganapathy, and the anonymous reviewers

for their valuable input that helped us improve the quality of the paper.

9. REFERENCES

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

[2] CVE-2007-4993: Xen guest root can escape to domain 0 through pygrub. https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2007-4993.

[3] CVE-2008-2100: VMware buffer overflows in VIX API let local users execute arbitrary code. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-2100.

[4] CVE-2013-6432. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-6432.

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

[6] Vulnerability in xenserver could result in privilege escalation and arbitrary code execution. http://support.citrix.com/article/CTX118766.

[7] Xbox 360 hypervisor privilege escalation vulnerability. http://www.securityfocus.com/archive/1/461489.

[8] Advanced Micro Devices. *AMD64 architecture programmer's manual: Volume 2: System programming*, September 2007.

[9] ARM Ltd. TrustZone. http://www.arm.com/products/processors/technologies/trustzone.php.

[10] ARM Ltd. *ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition*, 2012.

[11] A. M. Azab and P. Ning. Methods, systems, and computer readable medium for active monitoring, memory protection and integrity verification of target devices, Feb. 6 2014. WO Patent App. PCT/US2013/000,074.

[12] 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)*, pages 193–206, 2009.

[13] 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)*, pages 38–49, 2010.

[14] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC '08)*, pages 77–86, 2008.

[15] J. Bickford, R. O'Hare, A. Baliga, V. Ganapathy, and L. Iftode. Rootkits on smart phones: Attacks, implications and opportunities. In *Proceedings of the Eleventh Workshop on Mobile Computing Systems and Applications (HotMobile '10)*, pages 49–54, 2010.

[16] E. Buchanan, E. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS '08)*, pages 27–38, 2008.

[17] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[18] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic. SecureME: a hardware-software approach to full system security. In *Proceedings of the international conference on Supercomputing (ICS '11)*, pages 108–119, 2011.

[19] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[20] 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)*, pages 351–366, 2007.

[21] 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 (NDSS'12)*, 2012.

[22] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. Cloaker: Hardware supported rootkit concealment. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*. IEEE, 2008.

[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)*, pages 193–206, 2003.

[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 (NDSS '03)*, pages 191–206, 2003.

[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] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[27] G. Hotz. towelroot. https://towelroot.com/.

[28] 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.

[29] Intel Corporation. *Trusted eXecution Technology preliminary architecture specification and enabling considerations*, 2006.

[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)*, pages 128–138, 2007.

[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)*, pages 1–1, 2006.

[32] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kGuard: Lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[33] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In *Proceedings of the 39th International Conference on Dependable Systems and Networks (DSN'09)*, 2009.

[34] 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)*, pages 207–220, 2009.

[35] 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)*, pages 197–207, 2005.

[36] 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.

[37] A. Lineberry. Malicious code injection via /dev/mem. *Black Hat Europe*, 2009.

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

[39] Z. Liu, J. Lee, J. Zeng, Y. Wen, Z. Lin, and W. Shi. Cpu transparent protection of os kernel and hypervisor integrity with programmable dram. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*, 2013.

[40] 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.

[41] 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.

[42] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang. Vigilare: Toward snoop-based kernel integrity monitor. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*, 2012.

[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*, pages 233–247, 2008.

[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)*, pages 103–115, 2007.

[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)*, pages 74–81, 2009.

[46] D. Rosenberg. QSEE TrustZone kernel integer over flow vulnerability. In *Black Hat conference*, 2014.

[47] Samsung. White paper: An overview of Samsung KNOX, 2013.

[48] Secunia. Vulnerability report: VMware ESX server 3.x. `http://secunia.com/advisories/product/10757/`.

[49] Secunia. Xen multiple vulnerability report. `http://secunia.com/advisories/44502/`.

[50] 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)*, pages 335–350, 2007.

[51] 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)*, pages 552–561, 2007.

[52] 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)*, pages 477–487, 2009.

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

[54] S. Vogl, J. Pfoh, T. Kittel, and C. Eckert. Persistent data-only malware: Function hooks without code. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[55] 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.

[56] R. Wojtczuk and J. Rutkowska. Xen 0wning trilogy. In *Black Hat conference*, 2008.

[57] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.

[58] V. Zimmer and Y. Rasheed. Hypervisor runtime integrity support. US Patent 20090164770, June 2009.