2020

Hardware-Assisted Security Mechanisms On Arm-Based Multi-Core Processors

Shengye Wan
William & Mary - Arts & Sciences, simonsywan@gmail.com

Hardware-Assisted Security Mechanisms on ARM-Based Multi-Core Processors

Shengye Wan

Nanchang, China

Master of Science, College of William & Mary, 2016
Bachelor of Engineering, Huazhong University of Science and Technology, 2014

A Dissertation presented to the Graduate Faculty
of College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

College of William & Mary
August 2020

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

[illegible]

Shengye Wan

Approved by the Committee, August 2020

[illegible]

Committee Chair
Kun Sun, Adjunct Associate Professor, Computer Science
College of William & Mary

[illegible]

Gang Zhou, Professor, Computer Science
College of William & Mary

[illegible]

Qun Li, Professor, Computer Science
College of William & Mary

[illegible]

Xu Liu, Assistant Professor, Computer Science
College of William & Mary

[illegible]

Ning Zhang, Assistant Professor, Computer Science and Engineering
Washington University in St. Louis

ABSTRACT

During the last decade, Trusted Execution Environment (TEE) provided by ARM TrustZone had become one of the most popular techniques to build security on mobile devices. On a TrustZone-enabled system, the software can execute in either Secure World (trusted) and Normal World (untrusted). Meanwhile, along with the expeditious development of TrustZone technology, the security of TEE is also challenged by dealing with more and more on-board hardware and in-TEE applications. In this dissertation, we explicitly study the security of ARM TrustZone technology with the latest ARM architecture in three aspects.

First, we study the security of the TrustZone-assisted asynchronous introspection. Previously, asynchronous introspection mechanisms have been developed in the secure world to detect security policy violations in the normal world. However, we identify a new normal-world evasion attack that can defeat the asynchronous introspection by removing the attacking traces in parallel from one core when the secure-world checking is performing on another core. As the countermeasure, we propose a trustworthy asynchronous introspection mechanism called SATIN, which can effectively prevent evasion attacks with a minor system overhead by increasing the attackers' evasion time cost and decreasing the defender's inspecting time.

Second, we design an ARM TrustZone-assisted connectivity mechanism, called TZNIC, to enable the secure world's access to network even at the presence of a malicious OS. TZNIC deploys two NIC drivers, one secure-world driver, and one normal-world driver, that multiplex one physical NIC. We utilize the ARM TrustZone high-privilege to protect the secure-world driver, and further resolve several challenges about sharing one set of hardware peripheral between two isolated software environments. The evaluation shows that TZNIC can provide a reliable network channel for the secure world.

Third, we investigate the memory-safety of secure-world trusted applications. Though the existing TrustZone hardware focuses on protecting the application's confidentiality and integrity from malicious accesses of the normal world, there is little the secure world can do when the inside applications contain vulnerabilities and are further exploited by the normal world. To enhance the security of the secure-world application, we propose RusTEE, a TrustZone-based SDK that enables the development of trusted applications in the memory-safe programming language Rust. RusTEE can utilize the built-in security checks of Rust to mitigate all memory-corruption vulnerabilities for trusted applications. Besides, we enhance the trusted application's security by enforcing the memory-safety on its invocations of system-service APIs and cross-world communication channels.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

This dissertation is written with the help of many influential people to me. I express my gratitude to all of them.

First and foremost, I would like to thank my advisor Dr. Kun Sun. Without his kindly encouragement, I would not be able to start pursuing my research career. He always supports me in working on my research interests and advises me with plenty of patience.

Next, I would like to thank Dr. Ning Zhang and Dr. Yue Li. They both share much valuable research experience with me and guide me to walk through the tough time in my early days.

I would also like to thank my committee members Dr. Qun Li, Dr. Xu Liu, and Dr. Gang Zhou, for their insightful suggestions on my research projects.

I thank my dear friends, Jing Tian, Jianhua Sun, Qijue Wang, and Xiao Liu, for their help and companionship in my Ph.D. career.

Finally, I would like to thank my family members, my father, my grandmother, and my sister, who support and love me unconditionally.

I dedicate this dissertation to my dear father, Limin Wan.

LIST OF TABLES

LIST OF FIGURES

Hardware-Assisted Security Mechanisms on ARM-Based Multi-Core Processors

Chapter 1

Introduction

Over the last ten years, ARM TrustZone has been leveraged extensively to provide security protection on the ARM platforms [16, 103, 24, 54, 123]. With the assistance of TrustZone technology, ARM enables system-wide isolation by creating a Trusted Execution Environment (TEE) for security-sensitive code and data, and this isolation is realized via the hardware features that are built-in with the processor as well as the bus interconnect. A typical TrustZone-based system divides all resources across the System-on-Chip, including the hardware components and running software, into two different worlds with different privilege-settings and protections. Among the two worlds, *Normal World* is responsible for running *rich OS* and all the regular applications, named as *Client Applications (CA)*. In contrast, another isolated *Secure World* only executes a small and trusted *secure OS* along with some high-level security management tasks that are developed as *Trusted Applications (TA)*.

Previously, most TrustZone-related researches [16, 24, 109, 117] place trust on the secure-world side and focus on exploiting the secure world to conduct different secure-sensitive tasks. Meanwhile, in the pace of the advancement of TrustZone technology, the security of the secure world now faces several newly rising challenges from the latest hardware features and inside applications. First, as most advanced ARM architecture are multi-core supported, the architecture allows each core to enter its secure world independently, which means the rich OS and the secure OS can run in parallel [18, 69, 58]. Even such simultaneously execution capability certainly increases the performance on the normal-world side, it raises the new race conditions for the security components inside the secure world. For example, when the secure world conducts an introspection on the rich OS in one core, the inspected rich OS is still running in other cores, so it may infer the introspection's execution on other cores or even deceive the introspection by feeding the false result.

Moreover, because the latest ARM TrustZone supports the secure world to obtain a dynamic and ample memory space, the secure OS is no longer limited by the size of its Trusted Computing Base (TCB). Therefore, the secure OS can be developed with more and more functionalities, such as integrating the drivers of the necessary peripherals. However, as most peripherals, such as the Network Interface Card (NIC), is also required by the rich OS, both OSes have the same privilege to operate the same piece of hardware. Moreover, a general-purpose peripheral cannot differentiate the security attributes of an I/O operation, so it treats a secure-world operation as same as the normal-world one. In this case, it is challenging to achieve a balance between two worlds for the peripheral's availabilities while also promising the security of secure-world operations.

Finally, every TA inside the secure world can be risky and even harm the TEE's security. As the recent TA is developed with a variety of functionalities, it is hard to validate each TA's semantic correctness. These TAs could be compiled with the memory-unsafe vulnerabilities, and then get imported into the secure world. Moreover, as the TA collaborates with the corresponding CA that resides in the normal world, the communication channels between the two worlds can be exploited to attack the TA and conquer the entire secure world. Understanding how to assure the memory-safety of every TA and prevent the cross-world attack on TA is a critical and challenging question.

1.1 Problem Statement

In this dissertation, we thoroughly study the latest ARM TrustZone technology and three particular security concerns regarding the ARM multi-core system.

(1) Conducting Secure Asynchronous Introspection on Multi-Core ARM Processors. It is a well-known technology to utilize TEE to inspect the rich OS's kernel's integrity. However, previous solutions are impractical on the ARM single-core platform as the secure world execution freezes the rich OS entirely and lead to unaffordable performance overhead. As the modern ARM architecture introduces the multi-core features, a more acceptable introspection mechanism is utilizing one core for the secure-world introspection while preserving other cores for the normal world tasks. Though such a mechanism maintains both worlds' performances, it also introduces the severe security concern on the secure-world introspection because the malicious rich OS may conduct the evasion attack during the introspection to hide its attacking trace. Such a new rising risk requires a detailed investigation of its practicality, and the secure world may require an enhanced introspection mechanism if such risk is a real threat to the latest ARM platform.

(2) Providing Reliable Network for TrustZone Secure World. Traditionally, the mobile device's network is maintained by the network driver, which is integrated as part of the mobile rich OS (such as Android). However, in recent years, numerous attacking methods have been reported to obtain the root privilege of the rich OS and therefore gain the capability to manipulate the network. For example, 315 new CVEs have been reported on Android in 2019 [36] as the privilege-escalation related vulnerabilities. Allowing attackers to take over the network availability can cause severe consequences, especially for the remote security management applications that are deployed in the secure world, such as remote attestation [74, 66], remote deletion [118, 83], and remote patching [78, 28]. A compromised network channel can lead to loss of access to these services, while the remote attacker can still connect to the phone for stealing user data or even hijacking users' banking transaction [87]. As TrustZone-based applications cannot trust the unreliable normal-world network driver for communicating with the remote server, we should provide a reliable mechanism for the secure world to interact with the network peripherals and therefore maintain the secure-world network availability.

(3) Enhancing the memory-safety of Trusted Applications. Due to the protection of hardware-assisted isolation, it becomes common for TrustZone-based systems [16, 24, 109] to assume the trust of entire TEE, including the trusted applications (TAs) running in the TEE. Also, the functionalities of TEE systems are extended dramatically by installing various TAs in the trusted isolated environment. Though TrustZone technology can assure isolation between TEE and REE, dozens of software-based vulnerabilities in TAs have been reported to compromise the entire TEE system [31, 122, 48]. The term "Trusted Application" only refers to an application that should be trusted to run in TEE, but it does not mean the application is bug-free. The risk of TEE systems being compromised will increase along with the number of TAs installed.

1.2 Contributions

This dissertation proposes three projects that contribute to enhancing the TrustZone secure world's security across the application to OS level. The detailed contributions are listed.

Conducting Secure Asynchronous Introspection on Multi-Core ARM Processors. While performing a systematic study on the security of TrustZone-based asynchronous introspection [109], we propose the idea that traditional introspection strategy can be vulnerable to an evasion attack and then demonstrate our improved asynchronous introspection mechanism. In summary, we make the following contributions.

- we discover a new evasion attack called TZ-Evader against asynchronous inspection on multi-core ARM processors. The attack utilizes the side channel information to infer if any core is running in the secure world and then begins to clean the attacking traces simultaneously on other cores that run in the normal world.

- We develop a high-accurate probing technique called KProber for the normal world to fast probe the running state of all cores. Based on KProber, we implement a proof-of-concept TZ-Evader, which can defeat existing TrustZone-Based asynchronous introspection mechanisms.

- We propose a secure and trustworthy asynchronous introspection mechanism called SATIN to protect mobile devices against TZ-Evader. It wins the race condition over the attacker by minimizing the running time of each introspection round and maximizing the probing delay of TZ-Evader.

Providing Reliable Network for TrustZone Secure World. We design the mechanism for the secure world to reliably conduct I/O tasks on the general-purpose peripherals, which resolves the limitation on the peripheral that cannot tell the privilege differences between the secure world and the normal world. Specifically, we make the following contributions.

- We propose an ARM TrustZone network mechanism TZNIC for reliably sharing one physical network peripheral between two network drivers, inside and outside of the isolated execution environments. The secure-world driver can achieve secure network I/O operations by reliably enforcing the sharing of the normal-world driver's software interfaces.

- By utilizing ARM TrustZone protection on multi-core processors, we design a secure receiving module and a secure transmitting module to reliably receive packets and send responses, respectively, even when the rich OS cannot be trusted. Our mechanism does not require any modification to the rich OS, does not put any trust on the OS component, and is robust on the concurrent read/write challenges.

- We implement a prototype of TZNIC on a development board. The experimental results show that TZNIC can maintain the secure-world network channel with a small system overhead on rich OS. Our system can be deployed on ARM-based processor platforms with the support of a wide range of wired and wireless network devices.

Enhancing the memory-safety of Trusted Applications. We propose a TA-development mechanism, which assists developers to compile TAs in the memory-safe language Rust. By taking advantage of Rust's built-in security, we remove all the memory-unsafe implementation bugs of TA and further enhance its security of sensitive APIs and cross-world behaviors. Particularly, we make contributions in three aspects:

- We propose RusTEE, the first memory-safe trusted application development environment with comprehensive functionalities for TrustZone-assisted systems. By utilizing the built-in security properties and benefits of the Rust programming language, our trusted application environment removes most known memory-unsafe implementation bugs in trusted applications and thus enhance the security of TEE.
- We address two security concerns of the TrustZone-assisted TEE systems, namely, the widely exposed system-service APIs and cross-world communication channels, to enhance the security of Rust-based trusted applications.
- We implement a prototype of RusTEE and evaluate its performance in both a simulation environment and a real development board. Our experimental results show that our system can comply with strictly safe Rust, and it only incurs a minimal overhead. We will open source the system prototype.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows. In Chapter 2, we introduce the background knowledge of ARM TrustZone architecture. In Chapter 3, we present the details of the new evasion attack on the TrustZone-based introspection, and the corresponding countermeasure. In Chapter 4, we propose the TrustZone-based network peripheral mechanism, which enables both the normal world and secure world to apply isolated network drivers on the shared device simultaneously. In Chapter 5, we study the integration of the Rust language and the existing development environment of trusted applications. In Chapter 6, we summarize the related works of TEE-based architecture, introspection technology, TEE-based network management, and Rust-languages, respectively. Finally, we conclude the dissertation's highlight contributions and propose future research directions in Chapter 7.

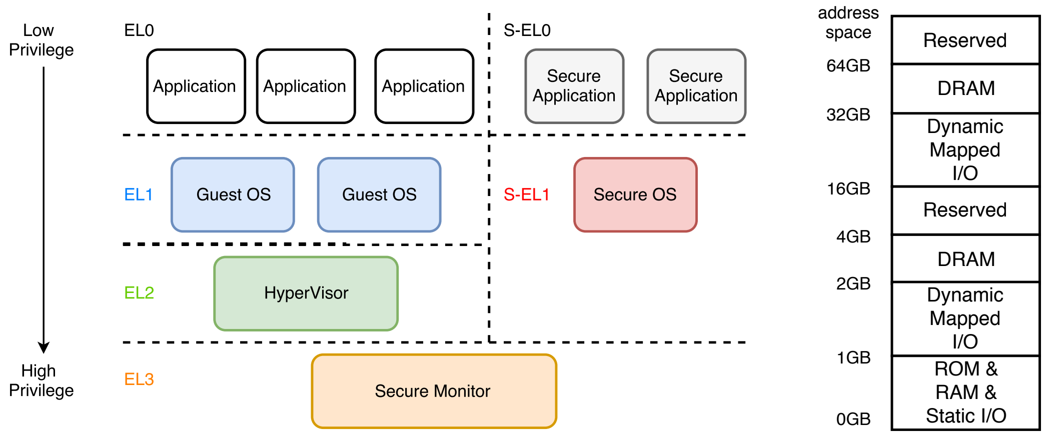
Chapter 2

ARM TrustZone Background

This chapter presents the general background of ARM TrustZone technology. ARM designs and applies the TrustZone technology for its most-advanced Cortex-A chipset family (e.g., ARMv7-A, ARMv8-A) as an efficient and system-wide approach to creating the hardware-level Trusted Execution Environment (TEE). In this chapter, we will focus on the ARMv8-A architecture, which is the latest 64/32-bit ARM architecture and supports execution instructions with 64-bit registers and remains backward compatible with the 32-bit ARMv7 architecture. In the following sections, we will present the TrustZone key features about its security model, memory model, interrupt model, and features related to the multi-core processors.

2.1 Security Model

With ARM architecture, the TrustZone security feature is defined for the system to operate under two environments: the Normal World (non-secure) and the Secure World (secure), as shown in Figure 2.1a. The normal world is accessible to the secure world, but not vice versa. The security setting of each core is achieved via hardware logic in the ARM AMBA bus, and it is independent from the settings of other cores. Each processor core can execute instructions in one of six privileges, where *EL0*, *EL1*, *EL2* are used in the normal world, and *S-EL0*, *S-EL1*, *EL3* are the secure world privileges. Among these privileges,



(a) CPU Security Model [8] (b) Memory Model [4]

Figure 2.1: ARMv8-A Architecture

EL3 is the highest privilege level that only contains a Secure Monitor for controlling the context switch between the secure world and the normal world. In the normal world, the user applications run at EL0, the guest OSes run at EL1, and the hypervisor runs at EL2. In the secure world, the secure applications run in the S-EL0 level, and the secure OS runs in the S-EL1 level. There is no S-EL2 level, so the secure world does not support a hypervisor layer.

2.2 Memory Model

ARMv8-A uses a uniform memory address map to provide a consistent physical address to all shared resources, as shown in Figure 2.1b. The memory can be classified into two main types: *normal memory* and *device memory* [9]. ROM, SRAM, and DRAM belong to the normal memory, which can be configured as either secure memory or non-secure memory [6]. The device memory supports I/O devices, including Static I/O for on-chip peripherals and Dynamic Mapped I/O for general-purpose peripherals (e.g., NIC, mouse, and keyboard). Currently, most general-purpose peripherals treat all read/write access with uniform non-secure privilege, so the normal world and the secure world have the same view and operation privileges on the peripherals' registers [113].

2.3 Interrupt Model

Besides the running application and memory, ARMv8-A also provide the interrupt mechanism with two privileges, namely *normal interrupt* and *secure interrupt.* The secure interrupts are always routed to the secure world no matter which world the CPU core is in [5], while the normal interrupts can be routed to either the normal world or the secure world, depending on specific configuration registers [5]. Since in most cases the device is running normal world tasks and the secure world is asleep, the secure interrupt is a key technique to guarantee the execution of the secure-world components, especially when the normal world is compromised and may decline to invoke the secure world.

The ARM interrupt management framework is responsible for configuring the interrupt routing behavior [11]. Normally, Interrupt Request (IRQ) is configured as a normal world interrupt and Fast Interrupt Request (FIQ) is configured as a secure world interrupt. There are two generic requirements. First, it should be guaranteed to route secure interrupts to be handled by the secure world, even when the current execution is in the normal world. Thus, it protects secure interrupts against potential intervention from non-secure software. Second, it should be able to route the non-secure interrupts to the normal world when current execution is in the secure world. When the non-secure interrupt is configured to be routed to EL3, the secure monitor in EL3 can save the state of software in secure world before handing the interrupt to non-secure software. In this case, the secure world is *preemptive.* When the non-secure interrupt is configured to be routed to the S-EL1 or S-EL0, the secure software can either hand the interrupt to the non-secure software in a preemptive mode, or ignore the interrupt until its running task completes in a *non-preemptive* secure mode. OP-TEE OS [69] is an open-source secure operating system that supports preemptive secure world.

2.4 ARM Multi-Core Processor

All latest ARMv8-A processors (e.g., Cortex-A53, 57, and 72) can be configured with one to four cores within one processor. Furthermore, ARM presents the big.LITTLE heterogeneous design to satisfy different application requirements on system performance and power consumption. Since each core maintains an independent security status, a multi-core platform may run the secure and non-secure software at the same time.

Chapter 3

SATIN: A Secure and Trustworthy Asynchronous Introspection on Multi-Core ARM Processors

3.1 Introduction

Introspection mechanisms have been developed and deployed in a high privileged execution environment to prevent or detect security policy violations in a low privileged execution environment on the host machine [53]. In general, introspection mechanisms can be classified into two categories: *synchronous introspection* for attack prevention [38, 42, 91, 93, 16, 29, 25] and *asynchronous introspection* for attack detection [17, 120, 85, 37, 93, 102]. ARM TrustZone technology is a system-wide security mechanism to provide hardware-level isolation between two execution worlds that share the CPU in a time-sliced fashion, where the secure world has a higher privilege to access the system resources of the normal world such as memory, CPU registers, and peripherals, but not vice versa. To enhance the security of mobile devices, a number of TrustZone-assisted introspection mechanisms have been developed and deployed on millions of mobile devices [93, 16, 29, 25, 102].

Synchronous introspection mechanisms focus on intercepting and mediating security sensitive operations inline by the high privileged execution environment to prevent security policy violations in the low privileged execution environment. For instance, synchronous mechanisms have been developed in the virtual machine manager to ensure memory page protection in virtual machines [38, 42, 91]. Similarly, Samsung's KNOX Real-time Kernel Protection (RKP) mechanism [93, 16] relies on ARM TrustZone technique to intercept certain privileged system functions in the normal world and screen them through the secure world for inspection and approval before being executed.

However, synchronous introspection mechanisms face two main challenges. First, it has to hook up to all security sensitive locations that are potentially exploitable to attackers. Though it is possible to build up a near-complete list based on recently discovered policy violations, it is hard to ensure the completeness of such list. Second, certain implementation bugs, such as write-what-where, allows an attacker to launch *data attacks* bypassing the function checkpoints setup for the synchronous introspection [62, 88]. Once an attacker discovers any vulnerability of synchronous introspection, she can deploy a persistent rootkit to maintain the root access to the normal world OS (rich OS), steal data or mislead user behaviors without being detected by synchronous introspection.

Asynchronous introspection mechanisms can effectively detect those persistent rootkits via analyzing attacking traces of security policy violations from a snapshot of memory along with CPU state information that is periodically or randomly acquired from the low privileged execution environment (e.g. the normal world). Besides simply checking the integrity of the invariant kernel code, a number of proof of concept approaches have been developed to provide a more fine-grained security checking on dynamic kernel data structures after filling the semantic gaps [17, 120, 85, 37]. Unlike the synchronous introspection that requires to intercept all read/write transactions on the target, asynchronous introspection conduct the introspection based on the snapshot of the target, which makes it more effective to introspect the target completely and therefore detect a persistent attack. Meanwhile, as stated in Section 1.1, the TrustZone-based introspection is majorly challenged by securely taking the rich OS's snapshot, especially under the condition that rich OS is not frozen on the multi-core scenario.

One major limitation on applying asynchronous introspection mechanism in practice is that the introspection process may introduce a large system overhead. Particularly, on single core ARM processors, whenever the secure world is performing the security checking, the entire rich OS will be suspended during the memory acquisition and online memory analysis process. Due to this poor usage experience on mobile devices, TrustZone-based asynchronous introspection has not been widely deployed or enabled.

Modern multi-core ARM processors creates new opportunities to deploy a practical asynchronous introspection based on TrustZone without pausing the rich OS. Specifically, the ARM multi-core architecture allows each core to enter its secure world independently, so the rich OS and the secure OS can run in parallel [18, 69, 58]. It is now feasible to make one core or all cores taking turns to perform the asynchronous introspection tasks while leaving other cores to continue the normal world's operations. For example, Samsung KNOX includes a Periodic Kernel Measurement (PKM) mechanism in the secure world to perform periodic asynchronous introspection on a specific core [93].

In this chapter, we reveal a new type of evasion attack that can defeat the asynchronous introspection on multi-core systems by removing the attacking traces concurrently from one core while the security checking is executing on another. Evasion attacks target at defeating asynchronous introspection by predicting precisely the time of next security check and thus removing all attacking evidence to avoid detection [120, 93]. However, on multi-core mobile devices that can run both normal world and secure world concurrently, besides removing the attacking traces before security check, an attacker can also hide its attacking trace right after the start of introspection but before it has the opportunity to examine any malicious bytes. We name this type of evasion attacks as TZ-Evader.

There are two main challenges to be solved when designing a TZ-Evader attack. First, the malicious code running in the normal world needs to know if the asynchronous introspection is running on any core's secure world; however, the ARM TrustZone architecture protects the secure world running information from being accessed by the normal world.

To solve this challenge, we propose to utilize the CPU core's availability as the side channel information to decide if the introspection is running on any core. We develop a user-level prober to stealthily probe the current state of each core. Second, when one core enters the secure world and begins to run the inspection, the malicious normal world needs to detect the core's state changes at an earliest time in order to maximize its evasion capability. To solve this challenge, we propose a kernel-level prober that can accurately monitor the running state changes of all cores. There are two implementation options for deploying the kernel-level prober, either by intercepting the timer interrupt to inject the prober in the rich OS or by manipulating the real-time scheduler of the Linux kernel to add the prober as a high priority process.

We implement a proof-of-concept TZ-Evader attack by integrating the kernel-level prober with traditional persistent rootkit on the ARM Juno r1 development board [14]. We evaluate its effectiveness against the state-of-the-art asynchronous introspection mechanisms, and the experimental results show the new TZ-Evader attack can accurately detect the running of asynchronous introspection and thus conduct a successful evasion attack.

With a deep understanding of the TZ-Evader attack, we propose a secure and trustworthy asynchronous introspection solution called SATIN in the secure world to defeat the TZ-Evader attack. The basic idea is to minimize the running time of each introspection and maximize the probing delay of TZ-Evader at the same time. We propose a number of techniques including *random wake-up time*, *random introspection area*, and *random CPU affinity* to ensure that the asynchronous introspection is always completed before TZ-Evader can hide any attacking traces. We implement a prototype of SATIN on the ARM Juno r1 development board and the experimental results show that it can effectively detect the TZ-Evader attacks with a minor system overhead.

3.2 TZ-Evader: Evasion Attacks on Multi-core Processors

3.2.1 Assumptions and Threat Model

We assume the secure world can be trusted and all the introspection components in the secure world are secure from attacks in the normal world. The asynchronous introspection can run randomly on any core at any time, and it cannot be intercepted by the normal world. We assume the asynchronous introspection does not suspend the rich OS on all cores; otherwise, it will face the same poor user experience problem as that on single-core processors. We assume the rich OS can be compromised and the attacker can bypass the existing synchronous introspection mechanisms to gain root privilege [88, 62]. We assume the attack is an Advanced Persistent Threat (APT), which aims to maintain its presence on the target and makes various effort to remain undetected. For example, a key-logger may collect all user inputs on the keyboard by intercepting a system interrupt, while the hijacking is detectable to the introspection. In this case, whenever the introspection is running, the key-logger should stop the attack and clean its attack trace to camouflage its existence; Meanwhile, for all the other time, it remains in the attacking phase.

3.2.2 New Attack Surface

On multi-core ARM processors, attackers may defeat the existing asynchronous introspection by satisfying two requirements. First, the malicious code in the normal world can detect if one core is entering the secure world. Second, before the core in the secure world can access the attacking traces, the malicious code running on other cores can remove the attacking traces.

3.2.2.1 Probing CPU Core's Running State

Since the normal world cannot directly access any secure world information, we propose to utilize the availability of the shared CPU cores as a side channel information to infer the running state of each core. The main idea is that after the secure world holds one core to perform the introspection, the normal world cannot use that core to run any process. A user-level prober process can be used to conduct this probing task. To trace when the normal world loses the control on a CPU core, the prober process assigns each core with a child-thread, which keeps reporting back the corresponding core's availability. Since the rich OS kernel may migrate one thread task to other cores, especially when one core is paused, we fix the CPU affinity of each thread. Thus, when one core enters the secure world, the attached thread will be paused and cannot be migrated to other cores by the OS scheduler. When one thread is paused, the prober process can detect that the corresponding core enters the secure world.

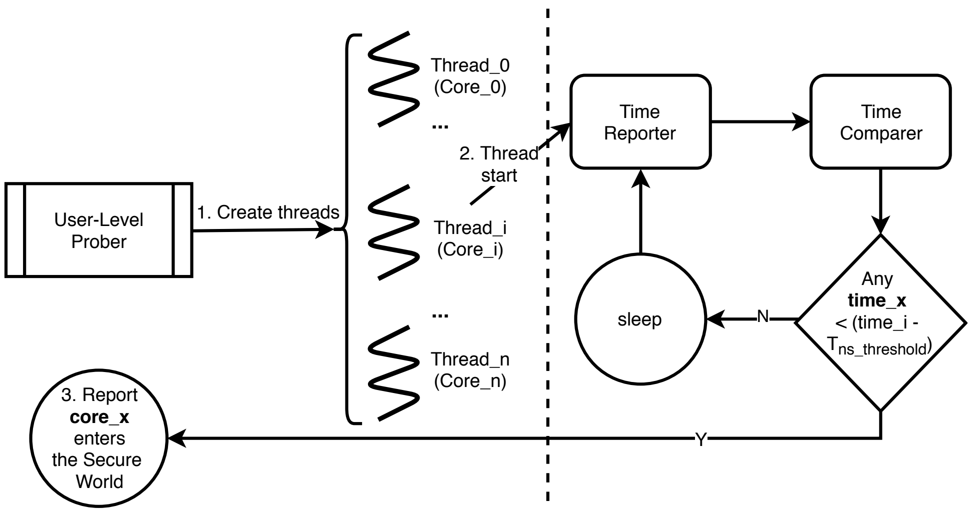


Figure 3.1: User-level Multi-thread Prober

Figure 3.1 shows the multi-thread design of the user-level prober. For a device with n cores, we start a process with n threads, and each thread's CPU affinity is fixed to its corresponding core. Each thread has two components: *Time Reporter* and *Time Comparer*. On core i, the Time Reporter obtains the latest time *time_i* from a shared timer among all CPU cores and then reports the time into a buffer that is readable to all threads. After that, the Time Comparer compares core i's *time_i* with all other cores' latest reported times.

Since each thread reports its latest time independently, even if we can start the Time

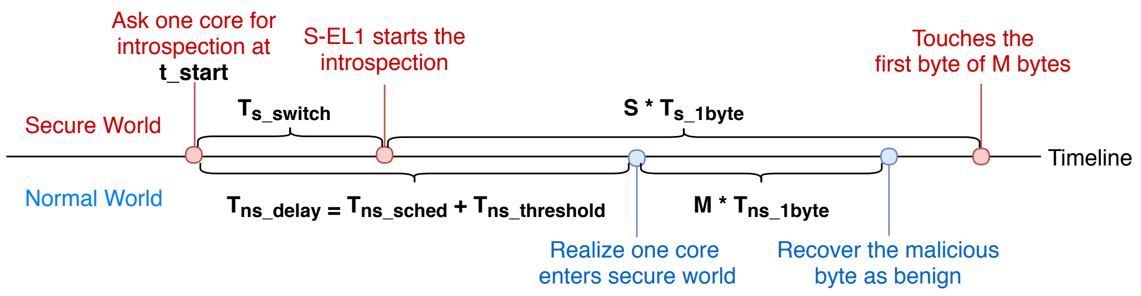

Ask one core for introspection at t_start
S-EL1 starts the introspection
Touches the first byte of M bytes
Secure World
Ts_switch
S * Ts_1byte
Timeline
Normal World
Tns_delay = Tns_sched + Tns_threshold
M * Tns_1byte
Realize one core enters secure world
Recover the malicious byte as benign

Figure 3.2: Race Condition Between Two Worlds on Multi-Core System

Reporters on all cores simultaneously, there exists a time difference when reading those reported time buffers and comparing their values. Meanwhile, since the kernel scheduler manages to provide the "fairness" to all threads, even though we cannot control all threads in a completely synchronized manner, each thread can be executed within a threshold, and the time differences between any two threads have an upper limit. We define this upper limit as $T_{ns\_threshold}$.

If thread i finds any core x's latest $time\_x$ is earlier than its reporting time and the latency is larger than the threshold, namely, $time\_i > (time\_x + T_{ns\_threshold})$, then this thread reports that $core\_x$ switches to the secure world. To remain stealthy, we can afford to have the threads sleep for a time interval. We assume the threads can be scheduled with the average frequency f_{ns_sched}, so between any two consecutive rounds of Time Comparer, there is a delay $T_{ns\_sched} = 1/f_{ns\_sched}$. Finally, the prober needs time $T_{ns\_delay} = T_{ns\_sched} + T_{ns\_threshold}$ to detect that one core enters the secure world.

We implement and evaluate this user-level prober on the ARM Juno development board (see Section 3.3.1). When running one round of probing every 8 seconds, we found the T_{ns_delay} is smaller than 5.97×10^{-3} s while the average time for one core to conduct a kernel integrity check is 8.04×10^{-2} s. We can see the prober is capable of detecting the running of a typical TrustZone-based kernel integrity checking. Since each step of the prober requires no modification with OS kernel privilege, it is stealthy and useful to check if the secure world is ever entered.

3.2.2.2 Race Condition on Multi-core Processors

One state-of-the-art defense mechanism against evasion attacks is to trigger the security checking randomly. However, on multi-core systems, we observe that as long as the normal world is not frozen on all cores, there exist a race condition between the normal world to remove the attacking traces and the secure world to detect the attacking traces. We assume the introspection will check N bytes of critical memory and the normal world needs to clean M bytes of attacking traces.

We formalize the race condition between the evasion attacks and the asynchronous introspection. We suppose the secure world requests to start the asynchronous introspection at the time moment t_start. To switch one core to the secure world for conducting introspection, the ARM processor needs to save the normal world context of this core and then switch to the secure world. We define this context switching period as T_{s_switch}. Thus, the moment that the secure world starts the integrity checking can be as early as $t\_start + T_{s\_switch}$. We define the time for the secure world to inspect one byte as T_{s_1byte}. Since the secure world assumes all N bytes may be vulnerable while it does not know exactly what bytes are modified by the attacker, it may check S bytes before it touches any malicious byte where $0 \leq S \leq N - M$. Therefore, the introspection cannot find any malicious bytes before the time moment $t\_start + T_{s\_switch} + S * T_{s\_1byte}$.

From the attacker's point of view, since she wants to attack the system with the longest duration (e.g., to collect inputted password), she would only stop the attack and hide the attacking trace if she believes there is an introspection. If it can predict the t_start, it can easily hide before the introspection starts. When the t_start is unknown or unpredictable, the attacker need to detect it by probing the secure world's state. We define the probing delay for the attacker to notice one core entering the secure world as T_{ns_delay}. Assume the time for the attacker to recover one malicious byte is T_{ns_1byte}. The total recovering time for the attacker is $T_{ns\_recover} = M * T_{ns\_1byte}$. Then after the moment $t\_start + T_{ns\_delay} + T_{ns\_recover}$, the secure world cannot detect the malicious modification.

The timing relationship between the two worlds is shown in Figure 3.2. Finally, for the attacker that is able to escape from the introspection, the condition 3.1 should be satisfied :

$$(T_{s\_switch} + S * T_{s\_1byte}) > (T_{ns\_delay} + T_{ns\_recover}) \tag{3.1}$$

We can see that the key element for evasion attacks to win the race condition is to minimize the probing delay T_{ns_delay}. Though the user-level prober shows the capability of understanding the secure world state stealthily, its probing stability and accuracy cannot be guaranteed. Particularly, when one core is scheduled with several threads that have the same or higher schedule priority than the probing thread, the prober's T_{ns_sched} is increased. To improve the probing accuracy, we should minimize the detection delay. In the next subsection, we present a new evasion attack called TZ-Evader based on a more accurate kernel-level prober.

3.2.3 TZ-Evader Design

TZ-Evader includes two components: a kernel-level attack module that conducts the persistent attack on the OS and removes the attacking trace accordingly; a fine-grained kernel-level prober module that can accurately and reliably monitor the running state of each core. The kernel-level prober also contains two components Time Reporter and Time Comparer, the same as the user-level prober. After obtaining the root privilege in rich OS, the prober module is loaded. If the prober can receive the time reported from all CPU cores, TZ-Evader believes there is no introspection in the secure world and keeps the attack active on the rich OS. Once the prober module reports that one core may be switched to the secure world, TZ-Evader begins to remove its attacking trace. We provide two implementation options for the kernel-level prober. The first option KProber-I is based on intercepting the timer interrupt to inject the prober in the rich OS,and the second option KProber-II works by manipulating the real-time scheduler of the Linux kernel to add the prober as a high priority process to be scheduled.

3.2.3.1 KProber-I

On ARM processors, each core has its own timers to generate time interrupts. The Time Reporter and Time Comparer are injected into the normal world timer interrupt handler, so as to ensure the prober being executed with the same frequency as the timer interrupts. After this hijacking, for any incoming timer interrupt to core i, the interrupt handler updates the $time\_i$ into its corresponding $buffer\_i$ and compares it with other $n-1$ cores' time reports before resuming the normal timer interrupt handler. Linux kernel is typically configured as the $CONFIG\_NO\_HZ\_IDLE$ mode, which means when the core is not in the IDLE state, the per-core timer raises the timer interrupt for scheduling-clock ticks periodically with the frequency of HZ. For most versions of the Linux kernel, $100 \leq HZ \leq 1000$ [34]. To avoid any core entering the idle mode, KProber-I keeps running a user-level multi-threads program on each core. KProber-I can guarantee to work with a frequency no less than HZ on any core, no matter how many tasks are running on that CPU core. Though this implementation option can achieve the highest time accuracy from the rich OS perspective, it requires to modify the timer interrupt handler, which may introduce extra attacking trace for the defender to detect. In Section 3.2.3.2, we present another implementation without modifying any kernel static area. Moreover, since there are many potentially unknown mechanisms to manipulate the handler, the defender has to scan the entire kernel for detecting all potential preparation traces, which gives KProber-I a larger chance to be recovered as we evaluated in Section 3.3.3.

3.2.3.2 KProber-II

This prober utilizes the Linux's real-time (RT) scheduler to ensure a reliable execution of Time Reporter and Time Comparer. According to the Linux kernel design, RT scheduler has higher scheduling priority than the default Linux CFS scheduler, which is responsible for scheduling most of Linux application threads. Meanwhile, RT scheduler can be used to schedule tasks with higher priority. Therefore, by setting the prober with the highest priority of RT scheduler, KProber-II can protect the reliable execution of Time Reporter and Time Comparer from being affected by either CFS-scheduled threads or low priority RT-scheduled threads.

Theoretically speaking, the timer-interrupt based prober is more stable than the RT scheduler based prober, since the frequency of the RT scheduler relies on the timer interrupt. However, injecting a prober into the interrupt handler demands more engineering efforts than simply increasing the priority of the attacking threads using the real-time scheduler. We present more implementation details in Section 3.3.1.1.

3.3 TZ-Evader Evaluation

3.3.1 TZ-Evader Implementation

We develop a prototype of TZ-Evader on ARM Juno r1 development board [12], which is featured with the ARM big.LITTLE technology that consists of a 4-core Cortex-A53 "LITTLE" processor for maximum power efficiency and a 2-core Cortex-A57 "big" processor to achieve maximum computation performance. The secure monitor running in EL3 is provided by ARM trusted firmware (ARM-TF), and the secure world OS running in S-EL1 is modified based on the Test Secure Payload (TSP) of ARM-TF [11]. We modify the secure timer interrupt handler in the TSP to perform the integrity check over the normal world. The normal world runs OpenEmbedded LAMP OS with kernel version lsk-4.4-armlt in EL1, which is downloaded using the script from Juno Wiki of ARM Community [14].

3.3.1.1 Kernel-Level Prober Implementation

We deploy two types of KProber to probe a specific core or a randomly chosen core. To probe a specific core's running state, we fix one thread of Time Reporter on the targeted core and fix another thread containing Time Reporter and Time Comparer on another core. To probe a random CPU core, we assign each core with one thread that contains Time Reporter and Time Comparer.

To implement the timer interrupt based KProber-I, one key technical issue is to hijack the time interrupt handler. In ARMv8-A architecture, the address of the original timer interrupt address is saved in the *IRQ Exception Vector*, which can be located in the AArch64 Exception Vector Table [10]. The table's starting address is saved in the Vector Based Address Registers $VBAR\_ELi (1 \leq i \leq 3)$. After locating the timer interrupt, we modify its corresponding table entry to redirect it to our hijacking code.

For the real-time scheduler based KProber-II, we use the function $pthread\_setschedparam()$ to schedule the targeted threads with the real-time scheduler. We use the rt-scheduler $SCHED\_FIFO$ with the priority parameter $sched\_get\_priority\_max(SCHED\_FIFO)$ for all KProber-II's threads. After investigating the relationship between thread sleeping and CPU utilization, we set the sleep time $T_{sleep} = 2 \times 10^{-4}\,\mathrm{s}$ and we assume the $T_{ns\_sched} = T_{sleep}$. In the following experiments, we implement Time Reporter with KProber-I and Time Comparer with KProber-II to demonstrate that both techniques can achieve reliable probing results.

3.3.1.2 Sample Kernel-Level Attack

To facilitate the evaluation of TZ-Evader, we implement a kernel-level attack that can hijack the GETTID system call. Successful system hijacking requires modifying an entry of the system call table, and this attack modifies one 8-bytes address of the system call table. Since the system call table is defined as text kernel data, TrustZone-based introspection can detect the GETTID system call is hijacked if the introspection scans and detects any of these 8 bytes is modified. Note there are many other kernel level attacking vectors, we just use GETTID hijacking attack as an example to study the evasion attacks.

3.3.2 TZ-Evader Evaluation

3.3.2.1 Introspection Time Delay

We first evaluate the time delay of the introspection. As we mentioned in the Equation 3.1, TrustZone-based asynchronous introspection suffers two major delays: T_{s_switch} and $s * T_{s\_1byte}$. To evaluate T_{s_switch}, we execute the context switching function of Test Secure Payload Dispatcher 50 times on one A53 core and one A57 core. The result shows for a secure timer interrupt raised at t_start, the time for the dispatcher to pause the normal world and jump to the related timer interrupt on the A53 core or A57 core are similar, ranging from 2.38×10^{-6} s to 3.60×10^{-6} s.

Then we evaluate T_{s_1byte} regarding two different introspection techniques. Traditional hardware-assisted asynchronous kernel introspection takes a snapshot of the kernel [120, 119] and then analyzes the memory copy. Since this copy remains inaccessible by the attacker, the analysis steps after taking the snapshot are not vulnerable to the TOCTTOU attack. Meanwhile, since the secure world and the normal world share the system hardware, TrustZone-based introspection can directly read the normal world OS' kernel from the secure world. After reading the kernel data, it can hash the data and compare the hash value to a pre-calculated authorized value. In our experiment, we measure the time for the secure world to take the snapshot and hash the kernel data. We use *djb*2 [82] as the hash function. Each measurement is repeated 50 times. Table 3.1 shows that directly hashing the kernel's memory is more efficient than capturing and hashing the snapshot. In addition, it consumes less memory than the snapshot approach. Therefore, directly hashing the memory is better than taking snapshot when the asynchronous introspection targets at the static kernel area. We also find that it takes less time to conduct the introspection on the A57 core than the A53 core, since A57 core is more powerful than the A53 core.

Table 3.1: Secure World Introspection Time

Core-Time	**Hash 1-Byte**	**Snapshot 1-byte**
A53-Average	1.07×10^{-8} s	1.08×10^{-8} s
A53-Max	1.14×10^{-8} s	1.57×10^{-8} s
A53-Min	9.23×10^{-9} s	9.24×10^{-9} s
A57-Average	6.71×10^{-9} s	6.75×10^{-9} s
A57-Max	7.50×10^{-9} s	7.83×10^{-9} s
A57-Min	6.67×10^{-9} s	6.67×10^{-9} s

3.3.2.2 Attack Time Delay

We evaluate normal world attack time delay in two aspects, where $T_{ns\_recover}$ is introduced by the the kernel-level attack module, and $T_{ns\_threshold}$ is introduced by the prober module. We repeat the measurement of the recovery time $T_{ns\_recover}$ 50 times on one A53 core and one A57 core. For the A53 core, the average recovering time is 5.80×10^{-3} s. For the A57 core, the average recovering time is 4.96×10^{-3} s.

Then we present the prober's time delay $T_{ns\_threshold}$ when KProber is probing all cores simultaneously. As the prober execution involves all available cores, we present the prober's time delay $T_{ns\_threshold}$ regardless of core types. To observe the variation of the threshold, we execute the KProber with different probing periods. For each probing period, we choose the largest difference calculated by the Time Comparer as the threshold, and we repeat the measurement 50 times. We present the average threshold, maximum threshold, and minimum threshold of the 50 rounds for each time period in Table 3.2.

Table 3.2: Probing Threshold on Multi-Core

Probing Period	**Average**	**Max**	**Min**
8 s	2.61×10^{-4} s	7.76×10^{-4} s	1.07×10^{-4} s
16 s	3.54×10^{-4} s	1.38×10^{-3} s	1.31×10^{-4} s
30 s	4.21×10^{-4} s	8.99×10^{-4} s	2.59×10^{-4} s
120 s	5.26×10^{-4} s	9.49×10^{-4} s	3.18×10^{-4} s
300 s	6.61×10^{-4} s	1.77×10^{-3} s	4.18×10^{-4} s

Based on the experiment results, we find that the average threshold becomes larger along with a longer probing period and the maximum threshold is around 1.8×10^{-3} s. To further understand the variation of the threshold, we investigate the reported time of each thread and identify that, in some rare cases, Time Comparer on *core_i* may get the *time_x* of the *core_x* with an abnormal large delay, which is up to 1.3×10^{-3} s. This cross-core reading delay leads to the large threshold. Meanwhile, a longer probing period increases the occurrence of those rare cases, so the average threshold increases too.

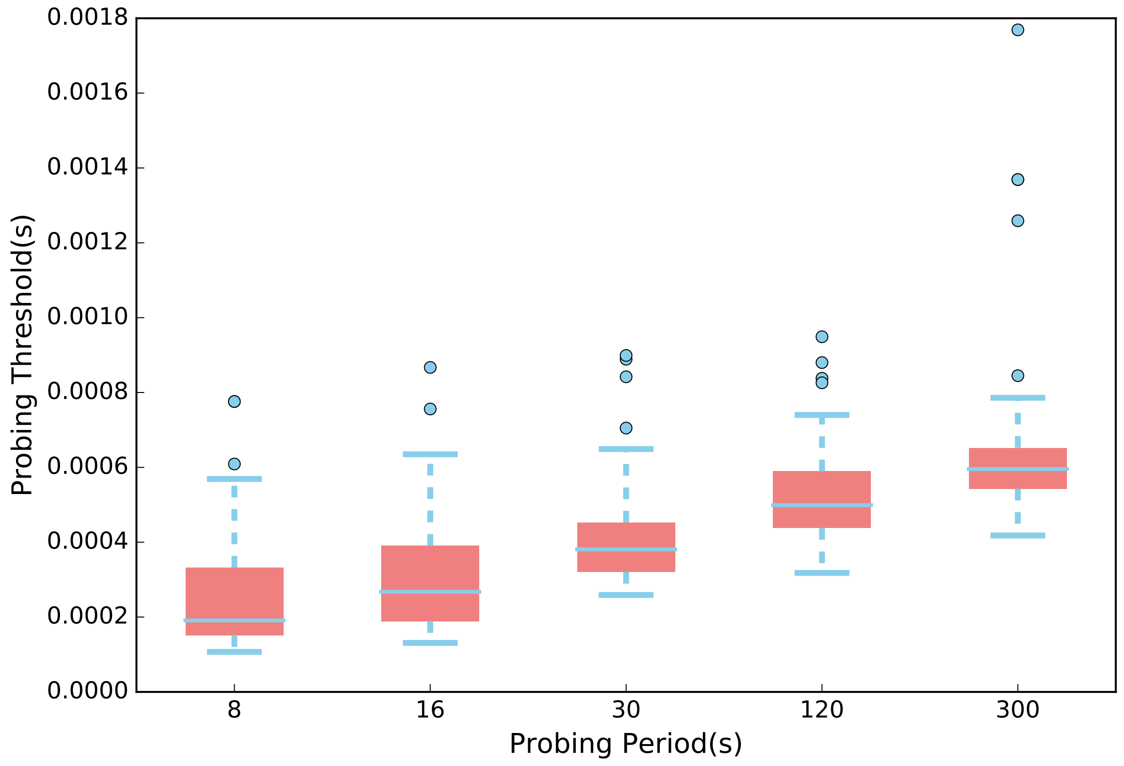


Figure 3.3: KProber Probing Threshold Stability

To present the stability of KProber, we show the variation of the thresholds with different probing periods in Figure 3.3. We can see that even though the KProber's average probing threshold increases with the probing period, the upper whiskers of the thresholds only go up slightly, and only few extreme large outliers are introduced for probing period 300 s, which go over 1×10^{-3} s.

Finally, we also observe that setting introspection with a fixed CPU affinity is easier to be probed than using all cores randomly. As we mentioned in Section 3.3.1.1, we also evaluate the case to use KProber for figuring out one single core's availability. We conduct the experiment with the same setting as presented above. According to our experiment result, the average thresholds to probe the single core only equal to $\sim 1/4$ of the presented threshold for probing all cores, for all five probing periods we evaluated. This means the more cores KProber needs to probe, the larger probing threshold it suffers.

3.3.3 Race Condition Analysis

Based on the experiment results, we have following observations on the race condition between the normal world and the secure world:

1. The switch delays T_{s_switch} for the A53 core and A57 core are similar.
2. The secure world can use more powerful CPU core (i.e., A57 core) for introspection to achieve a shorter T_{s_1byte}.
3. The secure world is easier to be probed if it's CPU affinity is predictable to the normal world.
4. The timing bottleneck of TZ-Evader is the time period for recovering its attacking trace $T_{ns\_recover}$.

According to Equation 3.1, we know the attack can finish its cleaning task after the secure world inspects S memory bytes, where:

$$S > \frac{T_{ns\_sched} + T_{ns\_threshold} + T_{ns\_recover} - T_{s\_switch}}{T_{s\_1byte}} \tag{3.2}$$

Now we consider the worst case for the TZ-Evader: the introspection starts on one A57 core while the TZ-Evader uses on one A53 core to remove its attacking trace. In Section 3.3.2.1, we have $T_{s\_switch} \leq 3.60 \times 10^{-6}$ s. Also, the secure world can inspect the kernel data with the maximum speed $T_{s\_1byte} = 6.67 \times 10^{-9}$ s. The attacker recovers its attacking trace as the lowest efficiency $T_{ns\_recover} = 6.13 \times 10^{-3}$ s, and $T_{ns\_sched} = 2 \times 10^{-4}$ s. In Section 3.3.2.2, we have for the worst case of TZ-Evader, $T_{ns\_threshold} \leq 1.80 \times 10^{-3}$ s. By applying the experimental data into the above Equation 3.2, we have $S \leq 1218351\ bytes$. Supposing the attack bytes appears randomly in the kernel and the introspection checks the entire OS kernel whose size is 11916240 *bytes*, nearly $1 - \frac{1218351}{11916240} \approx 90\%$ of the kernel space is not protected by existing TrustZone-based asynchronous introspection. Therefore, as long as both kernel attack module and prober module do not locate in the 10% scanned area, the system suffers TZ-Evader attack.

3.4 SATIN: Secure Asynchronous Introspection

We propose a secure and trustworthy TrustZone-based asynchronous introspection mechanism named SATIN that can practically inspect rich OS and effectively defeat TZ-Evader attacks. We first discuss two major challenges on developing the secure asynchronous introspection and then present the detailed design of the proposed mechanism. Figure 3.4 shows the architecture of SATIN, which is implemented in the secure world. It consists of two major components, where the integrity checking module performs the integrity checking on rich OS using a divide-and-conquer method to control the checking time of each round and the self activation module is responsible for waking up the secure world with the help of a secure timer and ensuring that the entrance cannot be either predicted or quickly probed by the normal world.

3.4.1 Asynchronous Introspection Challenges

We identify two main challenges on developing secure asynchronous introspection mechanisms against TZ-Evader.

3.4.1.1 Challenge 1: Performance vs. Detection Accuracy

On single core processors, the TrustZone-based introspection introduces unacceptable overhead as the introspection has to suspend the rich OS during the security checking process. The pausing issue leads to poor user experience. For example, a music song

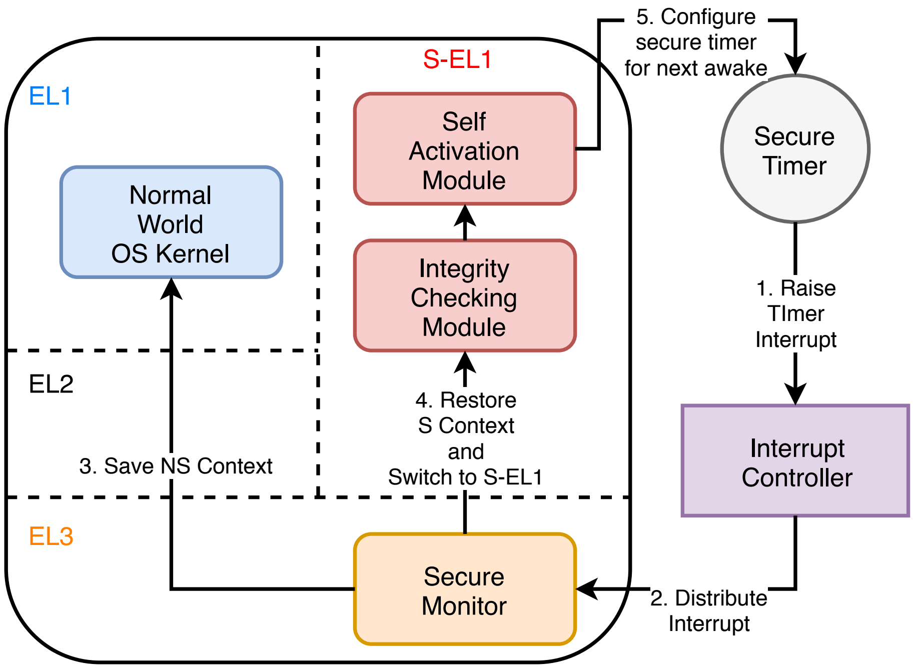


Figure 3.4: SATIN Architecture

played in the normal world becomes incoherent when the CPU enters the secure world. Similarly, on the multi-core processors, we cannot frequently freeze all cores even for a small period of time. For instance, Azab et al. [17] propose an introspection mechanism that requires to freeze all CPU cores on the server every 8 or 16 seconds.It becomes one of the biggest obstacles for asynchronous introspection mechanisms to be widely adopted and deployed, though a better detection accuracy can be achieved by freezing all cores. Meanwhile, even a multi-core platform can execute the introspection and other tasks simultaneously, the execution of the secure world still possesses some shared resources such as the CPU core. It is not well studied how much overhead is introduced by running the introspection in parallel to the rich OS.

3.4.1.2 Challenge 2: Evasion Attack

It is a challenge to defend against evasion attack when designing asynchronous introspection mechanisms [53]. If the attacker can escape ahead of the introspection by predicting or probing the execution of the asynchronous introspection, then the introspection result cannot be trusted [111, 102]. On single core processors, random checking is an effective scheme to defeat evasion attacks. However, on multi-core ARM processors, TZ-Evader can even escape from the random checking on any random core. It is critical to develop a secure asynchronous introspection mechanism to defeat the new evasion attacks.

3.4.2 Integrity Checking Module

To improve the detection rate, we propose to reduce the introspection time for each round by dividing the entire OS kernel into smaller areas and taking turns to check one area in each round. Therefore, it can guarantee to finish one round of security checking right before the malicious normal world can probe it but have not chance to remove any attacking traces. The integrity checking module prepares the hash value of each small area's benign status during booting stage. Then for each round of wake-up, it scans one small area and compares the hash value with the pre-calculated benign one. If the integrity checking module finds any abnormal small area, it can raise an alarm to the server side or the device user. To ensure that the malware cannot remove its traces before we finish checking on one small area, the size of each small area should be smaller than $(T_{ns\_delay} + T_{ns\_recover} - T_{s\_switch})/T_{s\_1byte}$ bytes. We develop a pseudo-random method to select the next small area for introspection. Suppose the set $set_{area} = \bigcup_{i=0}^{m-1} area\_i$ contains all the areas of the OS kernel, and each $area\_i$ satisfies the above size condition. When the secure world starts one round of the introspection, the module randomly picks one area $area\_x$ from set_{area} and then applies $set_{area} = set_{area} - area\_x$. If $set_{area} == NULL$, then SATIN resets $set_{area} = \{area\_0,\ ...,\ area\_(m-1)\}$.

Also, the integrity checking module needs to guarantee its execution is not interrupted by other non-secure parties. According to the latest ARM interrupt routing model [11], the normal world interrupt signal is possible to interrupt the execution of secure world. To prevent the normal world from using interrupts to interfere in the introspection process, SATIN needs to block all interrupts during each round of the introspection. We propose to deploy the integrity checking module into the secure timer interrupt handler, and then set EL3 to not trap the normal world interrupt by configuring $SCR\_EL3.IRQ = 0$. With this setting, even if the normal world interrupt signal can arrives the secure world, their priorities are lower than the integrity checking module.

3.4.3 Self Activation Module

The self activation module is designed to make sure the secure world can invoke the integrity checking module anytime without the help of the normal world. It relies on a secure timer that the normal world cannot access. Each TrustZone-enabled core has an individual secure timer that can only be read or written with the secure world privilege. During the booting time, the self activation module is invoked once on each core to write the next awake time into the secure timer register.

After the trusted booting process, when the timer condition meets, the timer raises an interrupt for the secure world and the secure monitor switches the core from the normal world to the secure world to handle this interrupt. By configuring the secure timer, we can activate the secure world without involving the normal world. Thus, we can prevent the normal world from disturbing the invocation of introspection. When one core enters the secure world via the secure timer interrupt, SATIN first performs the introspection on one small area and then sets the awake time for the next round of introspection. The self activation module decides the next awake time by $time\_x$, which is set to a base period time t_p (e.g., 8s, 16s, etc.) plus a random deviation t_d (e.g., a random time from $-t_p$ to t_p). By applying the random deviation with the next awake time, the interval between two consecutive rounds of introspection is among $[0, 2 * t_p]$, which means at any moment the introspection could start to scan and the attacker has to keep probing all cores. In addition, the random deviation can effectively minimize the exposure of any recognized patterns on the invocation of the asynchronous introspection to the normal OS. We set $t_p = T_{goal}/m$, where T_{goal} is the time period to guarantee that all the target areas can be scanned at least once.

3.4.4 Multi-Core Collaboration

To increase the checking accuracy, we propose to choose a random core for conducting the introspection task. This design choice is based on the observation that if only one core is used for asynchronous introspection, the malicious normal world can achieve a better probing accuracy than that when all cores are randomly chosen to conduct introspection, as mentioned in Section 3.3.2.2.

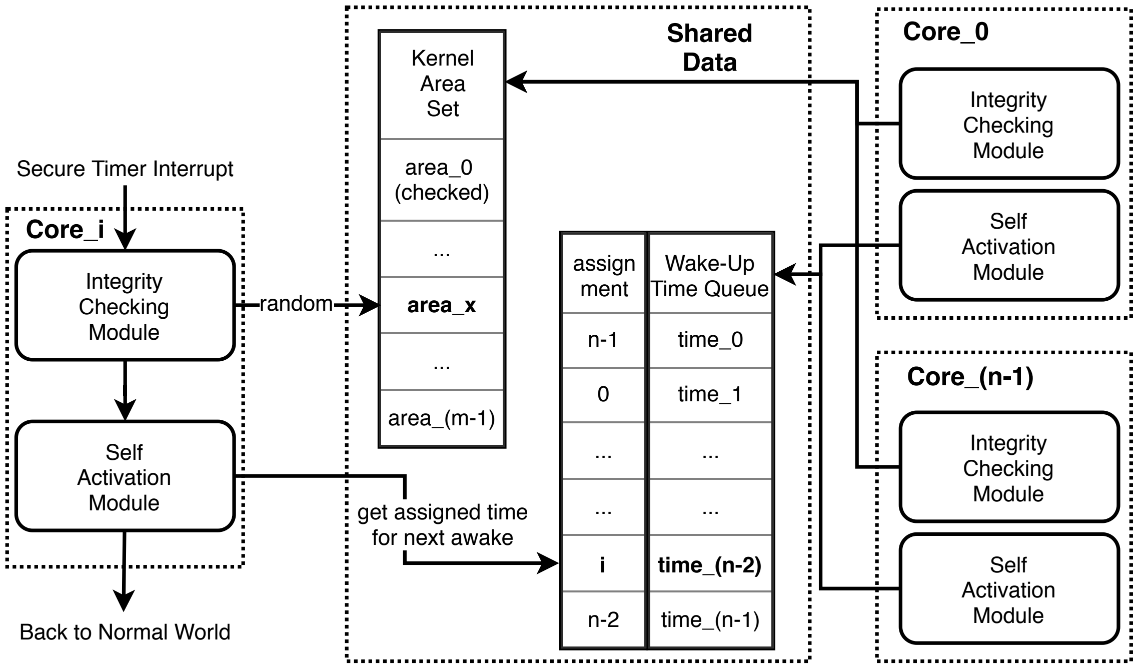


Figure 3.5: Multi-Core Collaboration of SATIN

Figure 3.5 illustrates the collaborative introspection of SATIN on the multi-core architecture. When any core i wakes up for the introspection, it randomly takes one kernel area from the shared Kernel Area Set set_{area} and inspects this area. Later, other cores are not going to inspect this area repeatedly since core i removes the area it chooses from the set. If there is no more area available, the set is refilled with all areas again. Next, core i obtains the next wake-up time from a wake-up time queue and configures it's secure timer accordingly, where the wake-up time queue is responsible to coordinate all cores that wake up in a random sequence.

Coordinating all cores to wake up in a random sequence is also a challenged task. ARMv8-A architecture does not provide a solution for one core to directly read or write the timer of another core. In this case, an intuitive design is to use the cross-core interrupt to notify all cores on serving the introspection in turn. ARMv8-A allows one core to generate a secure interrupt to forcibly switching another core into the secure world, so after core i finishes one round of introspection, it can switch another core j into secure world and then core j sets the secure timer for the next round introspection. However, the switch of core j can also be probed by the normal world so this method may leak the wake up sequence to the normal world, which can defeat the benefits from randomly waking up cores.

To protect the wake-up pattern from the normal world, SATIN does not apply the cross-core interrupt mechanism, and instead coordinates all cores via the secure memory. SATIN stores the wake-up time of each core in the wake-up time queue and requires each wake-up core to check the queue to get next wake up time. For the devices with n cores, the wake-up time queue contains n wake-up time slots. Each time value is determined by the rule discussed in 3.4.3. The initial wake-up sequence of all CPU cores is set during the trusted booting process. Then when any core enters the self activation module, it picks the next time based on a random assignment. Once all core extract their assigned slots from the queue, the self activation module refreshes the queue with n newly generated time values and newly generated random assignment. By flushing the wake-up time queue and re-assigning the time slots to each core, we can use all cores for kernel introspection while prevent the normal world from knowing the awake core or awake time for the next round of introspection. Finally, all cores can coordinate to make sure each core inspects a random memory area at a random wake-up time.

3.5 SATIN Evaluation

3.5.1 SATIN Implementation

To verify the effectiveness of SATIN on defeating TZ-Evader, we develop a prototype of SATIN on the ARM Juno r1 development board with the same configuration as mentioned in Section 3.3.1. We modify the Test Secure Payload in S-EL1 to implement the major modules of SATIN.

3.5.1.1 Self Activation Module

When the self activation module is called on one core, the module stops the secure timer by updating the register $CNTPS\_CTL\_EL1$. Then it takes the assigned wake-up time from the Wake-Up Time Queue and writes the value to comparer register $CNTPS\_CVAL\_EL1$. After that, the module restarts the secure timer for that core. For any core in the processor, when the shared physical timer counter register $CNTPCT\_EL0$ becomes equal to or greater than the per-core register $CNTPS\_CVAL\_EL1$, the core raises a secure timer interrupt.

3.5.1.2 Integrity Checking Module

As we calculated in Section 3.3.3, for each area of the checking module, its size must be smaller than 1218351 bytes. Also, we implement the integrity checking module to guarantee that each section of the normal world OS's *System.map* only belongs to one area for introspection. Thus, we divide the normal world's kernel into 19 areas according to the System.map. Among these areas, the largest one contains 876616 bytes and the smallest one contains 431360 bytes. During the booting time, SATIN hashes these 19 areas and then saves these hash values into an authorized hash table stored in the secure world.

3.5.2 SATIN Evaluation

We evaluate the performance of SATIN in two aspects, namely, the effectiveness on defeating TZ-Evader and the triggered extra system overhead.

3.5.2.1 Defeating TZ-Evader

SATIN can effectively and securely defeat TZ-Evader attacks. First, the introspection module is deployed in the secure world, so we can protect the introspection module from being compromised by malware in the normal world. We assume the hardware-assisted TrustZone technique can be trusted to protect the secure world. Second, it can prevent malware from removing its traces *before* the invocation of each introspection, since the normal world cannot accurately predict or intercept the invocation of introspection operations. Third, it can detect malware that uses race condition to remove its traces *during* the introspection. Because we divide the entire large introspection area into smaller areas, we can finish the introspection of one small area even before the malware detects the entrance of one core into the secure world and then begins to remove the attacking trace. In addition, it is user-friendly. The introspection does not require to fully freeze the rich OS in the normal world. On multi-core processors, since not all cores are forced to enter the secure world at the same time, the rich OS can continue to run on the remaining cores when one core conducts the introspection on one core.

In our introspection mechanism, every m rounds of the introspection can guarantee scanning the entire OS kernel once and the average time between two rounds is t_p. Within the time period $m*(t_p)+\sum_{i=0}^{m-1} size_{area\_i} * T_{s\_1byte}$, it can successfully catch the malicious memory bytes within the checked areas. In our experiment, the entire time is approximately $152\,\mathsf{s}$.

To validate the detection results, we execute TZ-Evader in the normal world while running SATIN simultaneously in the secure world. We set the probing thresholds of KProber as $1.8 \times 10^{-3}\,\mathsf{s}$. TZ-Evader maliciously modifies one system call handler which resides in the $area\_14$ of the integrity checking module. SATIN conducts 190 rounds of introspection to examine the entire kernel 10 times. KProber can faithfully report all 190 rounds of introspection without any false negative or false positive. Among these rounds, SATIN checks $area\_14$ 10 times and correctly detects the hijacked handler all the time. The average time between two consecutive checks for $area\_14$ is $141s$. In the meanwhile, TZ-Evader attempts to attack during these 10 checks but all the recovery efforts fail since the memory cleaning occurs later than the introspection.

3.5.2.2 SATIN Overhead

We use UnixBench [98] to evaluate the performance overhead on normal world operations when enabling our TrustZone-based asynchronous introspection. Figure 3.6 shows the normalized performance degradation when we use the self activation module to wake up the secure world across all cores of the device compared to the case where the self activation module is not enabled.

Since our experiment platform consists of 6 cores (i.e., 4 A53 cores and 2 A57 cores), we measure the overhead using two sets of experiments: executing each benchmark program once (1-task) and invoking 6 copies of the same benchmark simultaneously (6-task). In general, activating the introspection incurs 0.711% and 0.848% performance degradation in the 1-task and 6-task cases, respectively. This is reasonable as there is an increasing chance for SATIN to interrupt the normal world when more cores are utilized simultaneously. We also notice that the two tasks `file copy 256B` and `context switching` experience the largest overhead: 3.556% and 3.912%. The reason is that the test program happens to stay right at the random-selected core for the secure world more times than other cases. We believe this level of performance overhead is acceptable especially when the normal world is not suspended for even one nanosecond on multi-core systems.

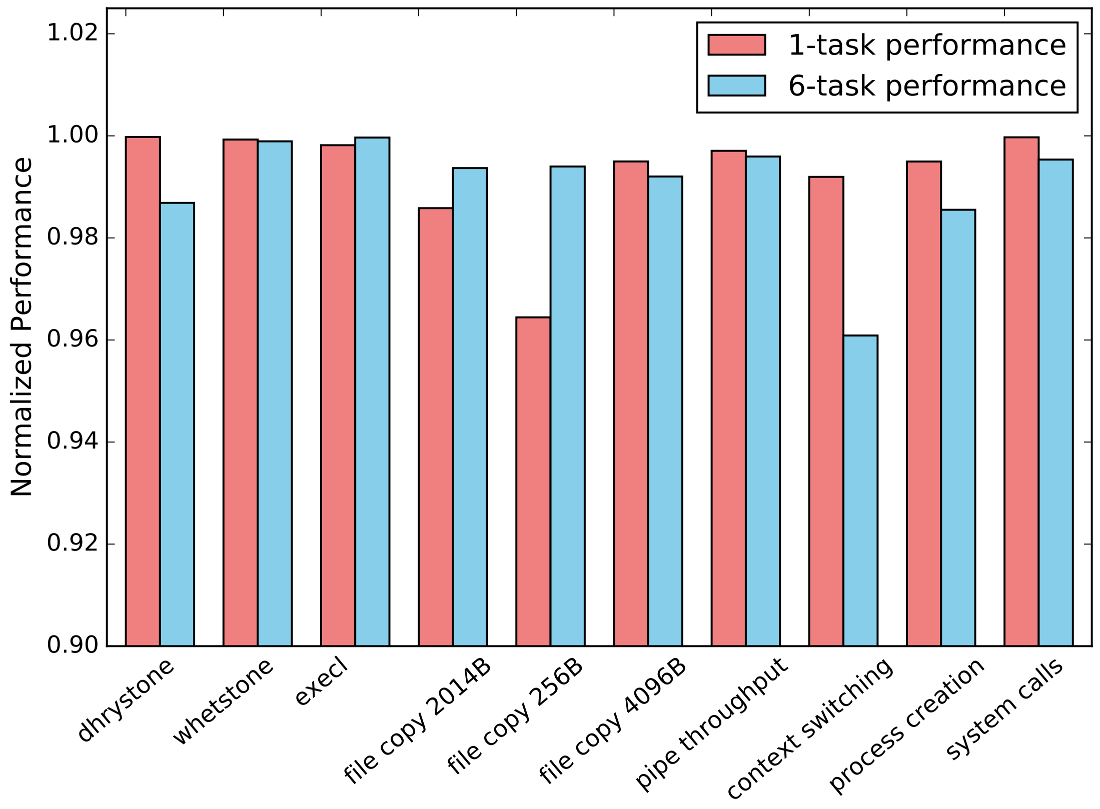


Figure 3.6: SATIN Overhead

3.5.3 SATIN Security Analysis

3.5.3.1 SATIN's Security

SATIN's security is based on the feature that its control flow cannot be affected by the attackers, which can be realized by utilizing TrustZone. Also, even considering recent TrustZone-related side-channel attacks [124], since they are focused on attacking TrustZone's confidentiality instead of control flow, SATIN is secure.

3.5.3.2 SATIN Capability

SATIN is capable of detecting advanced persistent attacks that leave attacking traces during an extended period, even if they apply TZ-Evader to hide their traces. Similarly, SATIN can detect evasion attacks utilizing other side channels. Finally, SATIN can reduce attack efficiency and maximize the chances to detect APT attacks.

3.6 Work Conclusion

In this chapter, we propose a trustworthy and practical TrustZone-based asynchronous introspection mechanism for ARM multi-core platform. We first show that on multi-core systems, even if the secure world uses a random core to inspect the rich OS kernel at random time point as previous asynchronous introspection solutions do, the malware in the normal world can still escape from the security checking by utilizing the race condition between the detector running on one core and the malicious evader running on other cores at the same time. We identify this new type of evasion attack as TZ-Evader and conduct a systematic study on it. We develop a proof-of-concept TZ-Evader attack that uses an accurate kernel-level prober to defeat the existing asynchronous introspection. Finally, we develop a secure TrustZone-based asynchronous introspection mechanism called SATIN on multi-core ARM processors to defeat the TZ-Evader attacks. We implement a prototype of SATIN on ARM Juno r1 development board and the experimental results show that SATIN can effectively prevent evasion attacks on multi-core systems with a minor overhead.

Chapter 4

TZNIC: Towards Providing Reliable Network for ARM TrustZone-Based Application

4.1 Introduction

Personal mobile devices have become one of the most important devices to meet people's daily needs on entertainment, productivity, information access, and financial asset management. According to a 2019 mobile usage report [115], people are spending almost 3 hours per day on average with their phones. Moreover, there has been a growing dependence on network connectivity in almost all mobile device applications. From 2018 to 2019, the network usage of mobile devices increased 47% [30]. Also, Cisco [30] forecasts that mobile data traffic can increase seven-fold between 2017 and 2022, which indicates the rapidly growing users' dependency on the mobile network.

As we stated in Section 1.1, the rich OS is not trusted to provide the reliable network service for security-sensitive components. Meanwhile, on ARM-based mobile devices, utilizing TrustZone technology is one of the most popular design options to fight against the compromised rich OS [109, 26, 61, 117, 116, 66]. Previous works have already presented some TrustZone-based network peripheral management (cellular, Wi-Fi, etc.) under the threat model that rich OS cannot be trusted. For example, to prevent the rich OS from turning on the network stealthily, SeCloack [61] protects the corresponding peripherals by moving them into the secure world and then blocks the rich OS access. Brasser et al. [26] present a similar idea to use the secure world to validate the network peripheral has been turned off. Though these existing works provide thorough investigations under their corresponding threat models, they share two key assumptions that may not generally apply to all the devices. First, the mobile device is equipped with an additional TrustZone controller such as Central Security Unit (CSU) [81] to protect the network peripherals. Second, the rich OS and the user are OK with complete loss of the peripheral access, and thus the network access.

However, the above prerequisites can be too strict for many real-world scenarios. First of all, the CSU is only a device-specific controller presented on a limited number of evaluation boards, such as the i.MX family development board, and is not equipped on the majority of the commercial mobile devices or some other development board like ARM Juno [7]. Furthermore, even if the CSU is not a concern, preventing the rich OS from accessing network peripheral completely is often not an acceptable solution for the general public. In the case where the manufacturer places a remote patch service in the secure world, this service needs to listen on the network channel all the time since the remote network packet may arrive at any time. In this case, if we adopt the previous solutions to only allow the secure world for accessing the network peripheral and handling the network tasks, then the rich OS will suffer unaffordable network-related overhead. A straightforward solution for this problem is providing two network peripherals for the normal world and the secure world separately, while it's infeasible for some mobile devices, especially considering the limited hardware space of the device. In this chapter, we are trying to answer the following question:

On most mobile devices, there is only one network peripheral (e.g., NIC) *for each connection type, what would be the best practical design to provide a reliable network for ARM TrustZone secure world without impacting the rich OS?*

To answer the above question, we design and develop a mobile network framework called TZNIC to provide a reliable network channel for the TrustZone secure world. We propose to deploy two network drivers, a complete network driver in the normal world and a customized slim driver in the secure world, for multiplexing the shared physical NIC to serve two worlds separately. The normal-world network driver can promise the rich OS's network performance while the secure-world driver can deliver the goals of reliably receiving and sending network traffic for the TrustZone-based applications. To build a practical solution with minimal impact to the normal world, TZNIC makes zero modification on the rich OS, which means the rich OS's network driver is unmodified and can conduct the peripheral initialization, provide the software interface for input and output, and handle the network traffic as normal. Meanwhile, since one physical NIC can only connect to one driver's software interface (i.e., descriptor ring buffers and SKB buffers) and TZNIC allows the normal-world driver to provide the interface, the secure-world driver is responsible to reuse all normal-world driver's software interface and interact with the peripheral.

While the two-drivers design brings plenty of benefits for both the normal world and secure world, such design faces a fundamental challenge that is there exists a semantic gap between two isolated worlds. Specifically, we need a sophisticated mechanism in the secure world to understand where does the normal-world driver saves all the network-related packets, and the mechanism should not rely on any trust or assistance from the rich OS. The key idea to solve this challenge is that the secure-world driver reconstructs all the semantic information via directly reading the physical NIC's registers, and then locates the software interfaces of the normal-world driver in the memory. By taking advantage of the higher privilege of the ARM TrustZone secure world, TZNIC can promise that even rich OS privileged attacker cannot hide either NIC registers or memory data from the secure world. In this circumstance, TZNIC makes the secure-world driver fills the semantic gap and forcing the shared software interface without making any change to the normal-world driver or the rich OS.

When relying on the software interface of the normal-world driver to receive packets, TZNIC can make both drivers read the incoming buffer at the same time while it faces another challenge: the rich OS may deliberately delete the packets, whose arrival time is unpredictable to the secure world. To solve this challenge without incurring large overhead to the normal world, we use one core to frequently inspect the received packets from the secure world and keep other cores running in the normal world. To further increase the possibility for the secure-world driver to read the received network packets under the race condition, we make efforts in two aspects. First, the remote server should send out packets in a loss-tolerant format (such as UDP packet) multiple times to increase the chances for the secure world to successfully receive the packet. Second, for any received packet, we save the packet into the secure-only memory to further prevent normal world touch. As we latterly evaluated, such enhancements provide TZNIC a positive opportunity to win the race condition even facing the challenge from the powerful normal-world attacker.

TZNIC also provides a transmitting module to send secure-world network packets. Dissimilar the receiving process that can grant the concurrent operation for both drivers, if we allow both normal-world and secure-world drivers to write in parallel to the shared buffers for sending packets, then it may cause concurrent-write issue and crash the rich OS. We solve this problem by making the normal-world driver yield its packet sending priority for a short time to the secure-world driver. We develop mechanisms to properly save the transmission context (i.e., the shared descriptors and on-peripheral registers) for the normal-world driver before the secure-world driver sends packets, and restore all the saved context after secure transmission.

We implement a prototype of TZNIC on Juno r1 development board [7] and perform an evaluation on system overhead and power consumption. The experimental results show that TZNIC can provide reliable network I/O for the secure world even when the rich OS is compromised. Moreover, our solution is transparent to the rich OS and incurs a small overhead.

4.2 Work Background: DMA-Based NICs

Modern NICs handle network packets via either programmed memory mapped input/output (MMIO) or direct memory access (DMA). In MMIO, the NIC provides on-peripheral device memory and then waits for the CPU to exchange packets via device memory. When using DMA, the NIC directly reads and writes the network packet into the normal RAM memory, without involving the CPU. Most modern NICs are DMA-capable since DMA-based operations remove the burden of the CPU on handling the packets and thus dramatically increase the system performance.

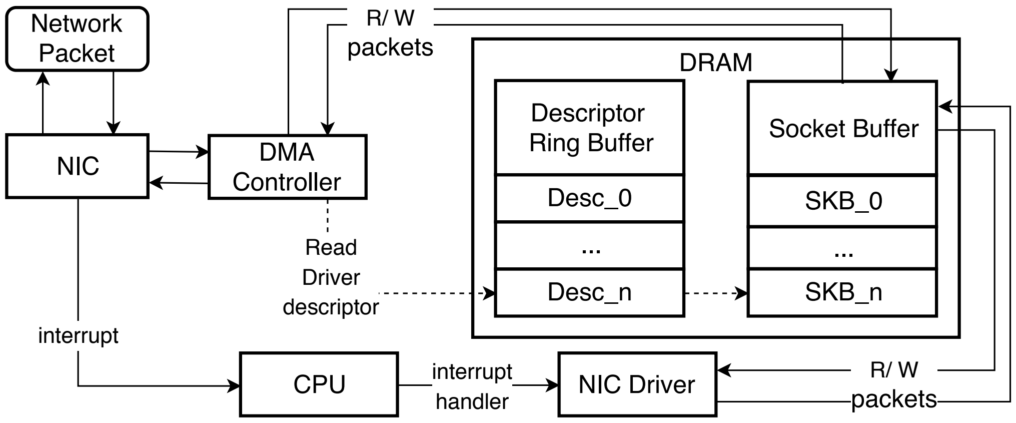


Figure 4.1: DMA-Based NIC Workflow

A typical workflow for the packet transmission between the DMA-capable NIC and the network driver is shown in Figure 4.1. It involves three main components: NIC (with the assistance of DMA Controller), NIC driver, and DMA software interface (Descriptor Ring Buffer and Socket Buffer). During the boot up period, the NIC driver allocates multiple ring-buffer queues to store the descriptors. The descriptor mechanism is designed for coordinating the driver and the NIC to handle packets asynchronously. Each descriptor points to a socket buffer (SKB) that is allocated to store the packet data. When the NIC is about to save a latest received (RX) packet in memory, it checks the latest RX descriptor ring buffer $Desc\_n$ to get the available SKB_n. Then it saves packet to the SKB (via DMA Controller) and raises an interrupt to notify the CPU. To send out a transmitting (TX) packet, the driver first notifies the NIC that there are available packets linked to certain TX descriptors. Then, the NIC retrieves the packets accordingly and sends them out.

4.3 TZNIC Overview

4.3.1 Threat Model and Assumptions

We assume TrustZone can be trusted to protect the software running in the secure world. On multi-core platforms, secure and non-secure software may run at the same time on different cores. Also, during the system boot-up, a trusted boot can ensure the integrity of the kernel images being loaded in the secure world and the normal world. We assume each mobile device has a unique asymmetric key pair and whoever communicates with the secure-world applications can obtain the related public key to protect their communication. We assume there is only one DMA-based network interface available on mobile devices.

An attacker can gain the OS kernel privilege in the normal world via remote attacks, but she cannot physically access the device. We assume the attacker maintains a remote communication channel (e.g., via a network interface) with the compromised mobile device to achieve persistent attacks. In other words, one network interface is always enabled on the mobile device. Meanwhile, the attacker is capable to conduct any OS-privilege attack to make the network unavailable for the services protected within TrustZone secure world.

4.3.2 Key Idea and Challenges

As TrustZone-based applications cannot trust the unreliable normal-world network driver for communicating with the remote server, we make several attempts to deliver dependable network availability by the secure world itself.

The first attempt is to only deploy a secure driver in the secure world to monitor all network packets and identify the packets containing secure-world traffic. However, since most mobile network traffics are targeted to the rich OS and its upon applications, the secure driver needs to forward all those heavy network traffics to the normal world, and this huge context switching overhead renders this solution impractical.

The second attempt is to deploy one complete non-secure driver in the normal world and one complete secure driver in the secure world. The secure driver has a higher privilege than the non-secure driver on the usage of the shared network interface. Therefore, when the secure driver tries to receive any network packet, the other driver is suspended. Since the secure world does not know the arriving time of the received packet, it has to frequently suspend the non-secure driver to check the receiving packets in a timely manner and leads to expensive overhead on the normal network services. Therefore, it is not a practical solution to deploy two independently complete network drivers for controlling one shared network interface simultaneously.

Based on the above two failed attempts, we propose a solution that deploys a complete network driver in the normal world and multiplexes the normal-world driver's software interface to a slim network driver in the secure world. The normal-world driver is fully unmodified and therefore is responsible for initializing the physical NIC device and providing all software interfaces such as descriptor buffers and packet buffers in the normal world. The secure-world driver runs simultaneously with the normal-world driver on multi-core processors and utilize the software interface provided by the normal-world driver to communicate with the remote server. To multiplex the software interface of the normal-world driver, we have to solve three major challenges.

Challenge-1: Filling semantic gap. Since the rich OS in the normal world cannot be trusted, the secure-world driver has to figure out how to use the normal world's network interface without getting assistance from the rich OS. In other words, the secure-world driver needs to fill the semantic gap on locating the critical data structures such as the normal driver's descriptor queue and SKB buffers by itself.

Challenge-2: Resisting interference from the normal world. On a multi-core system, when two network drivers share the same buffer set, two drivers may read or write the same buffer simultaneously. Therefore, when the normal-world driver reads out the packets first from the RX interface, those packets cannot be read by the secure-world driver in the secure world. More severely, a malicious rich OS may deliberately delete the security-sensitive packets from the shared network software interfaces. Our mechanism should be able to resist all those inferences from the normal world.

Challenge-3: Being transparent to rich OS. It contains two requirements. First, the solution should not require any changes on the rich OS. Second, it should have a minimal performance impact on the rich OS.

4.3.3 Our Solution

An overview of TZNIC architecture is shown in Figure 4.2. A normal-world NIC driver in the rich OS initializes the NIC and provides software interfaces for receiving and sending network packets in the normal world. Meanwhile, we deploy a secure-world driver with three components: Sec-RX, Sec-TX, and Sec-Buffer. The secure-world driver extracts the normal-world driver's software interface information from NIC on-peripheral registers. As normal-world attackers cannot disguise the registers from the secure world's view, TZNIC is guaranteed to acquire the correct value of registers. Moreover, we present the mechanism to trusted understand the normal-world driver's semantic information based on these registers, which resolves *Challenge-1.*

After the system boots and the normal-world driver initializes the NIC, all cores run in the normal world by default. Since the arriving time of a remote packet is usually unpredictable to the secure world, TZNIC periodically wakes up the secure-world driver via a secure timer, which raises secure interrupts to switch one CPU core into the secure world. After that, the receiving module Sec-RX runs in parallel to the normal-world NIC driver on reading the received packets that are saved into the normal-world driver's receiving interface (RX buffers). As the RX part of *Challenge-2*, the normal world may

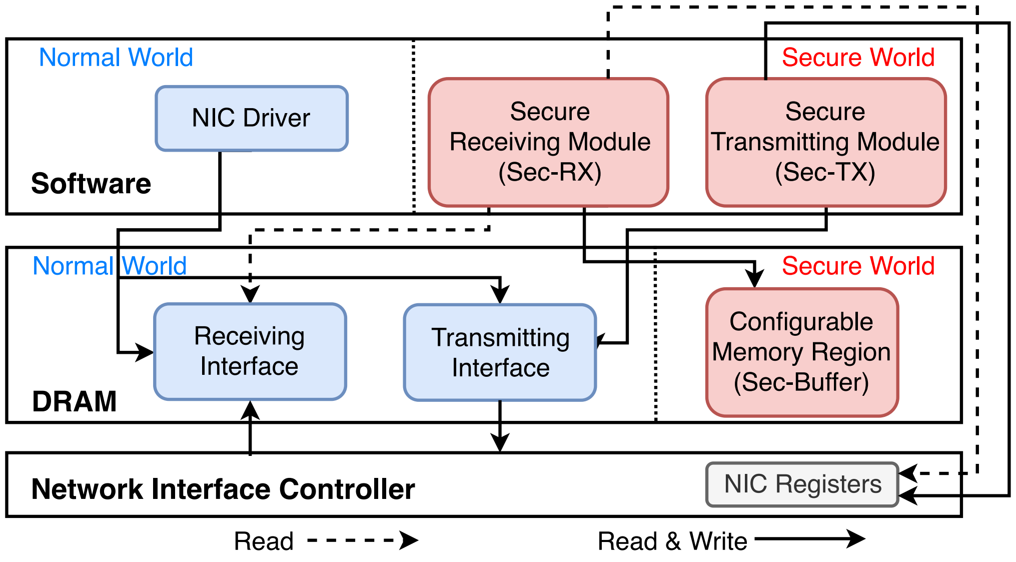


Figure 4.2: TZNIC Architecture

discard the secure-world packets benignly or maliciously before Sec-RX reads the packet. Sec-RX requires the packet sender to send the loss-tolerant packets multiple times for increasing the chance to identify the packets. After identifying a secure-world packet, Sec-RX makes a copy to a secure world DRAM region called Sec-Buffer so the normal world cannot touch the packet anymore.

When the secure world needs to send a response to the remote server, the secure transmitting module Sec-TX utilizes the normal-world TX descriptors, which is part of the normal-world transmitting interface, to send out secure world packets. Unlike Sec-RX that only involves concurrent-read with the normal-world driver, Sec-TX needs to write on the same descriptor of the normal-world driver, which may incur concurrent-write issues on the same descriptor. To tackle this challenge as the TX part of *Challenge-2*, Sec-TX pauses the normal world on all cores for a very short period before sending its secure packet. Our solution makes the sending of secure packets transparent to the rich OS. As we presented, the entire processes of Sec-RX and Sec-TX are working without modifying or requiring the collaboration of the normal world side, and therefore resolves *Challenge-3*.

4.4 TZNIC Design

TZNIC is a TrustZone-based reliable network mechanism that utilizes the shared physical network device to receive and transmit the network traffic for the secure-world services. In the following section, we will introduce the detail design of its modules for receiving packets, sending packets, switching two worlds' status, and the entire workflow.

4.4.1 Secure Receiving Module

The secure receiving module Sec-RX is designed to extract the NIC RX packets received in the normal world and filtering out remote server's packets into Sec-Buffer without any support from the normal world. One design goal of Sec-RX is to have a minimal impact on the normal-world driver and the rich OS. On single-core ARM processors, when the system enters the secure world, the rich OS is frozen until the system switches back. Fortunately, modern multi-core ARM processors allow the execution of Sec-RX on one core and the normal-world driver on other cores at the same time.

Since the normal OS cannot be trusted, the secure receiving module has to extract the packets from normal-world software interface by itself. Sec-RX uses the high-privilege provided by the secure world to first access the registers of the shared NIC and then deduce the descriptors' information based on the registers. Due to the NIC requires a fixed structure to understand the descriptor of different rich OS or their drivers, for any given NIC, the descriptor stores the socket buffer information in the required format and such format cannot be dynamically updated. Based on this feature, once Sec-RX locates the normal-world drivers' descriptors, the module can find out the socket buffer address by reading every descriptor. Finally, it uses the high-privilege again to read the packet saved in the corresponding memory address. The details of extracting such semantic information are presented in Section 4.5.1.2. To split out the secure-world receiving packets from all received packets, Sec-RX allows the secure-world applications to register their servers' IP addresses as the white list and then it keeps reading each packet's sender IP and checking if any packet's IP address matches any specific server. If there is a match, Sec-RX copies the packet to a pre-allocated secure memory region Sec-Buffer. Since Sec-Buffer is not accessible to the normal world, the rich OS cannot delete or modify a packet once Sec-RX has retrieved the packet into Sec-Buffer.

As both worlds have the same level of privilege to read from the receiving buffers in the normal world, the normal-world driver may discard packets from its buffers after the packets are readout. Besides, the rich OS may have chances to delete secure-world packets from the memory before Sec-RX reads them out. Due to these race conditions, secure-world packets may be dropped by the normal-world driver or the malicious OS before the secure world wakes up and copy those packets into the secure memory. As the countermeasure of the race condition, the remote server of a secure-world application should send multiple copies of its packets until it receives a response. Also, the server's packets should be loss-tolerant for the secure world, which means any packet received by Sec-RX should be independent and good enough to inform the secure world about the following tasks. When the secure-world application receives the first packet and if the service requires to receive a large amount of data (e.g., security patch) from the server, TZNIC can suspend the normal world on all cores to stop the race condition from the rich OS. The suspension technique is presented in Section 4.4.2.

Sec-RX keeps reading packets from the receiving buffers in the normal world until it extracts a valid remote request or the one-time polling time reaches an upper limit RX_max. The upper bound RX_max is configurable to determine how many newly arriving packets Sec-RX can monitor in each round of polling. Generally speaking, the maximum threshold should be long enough for Sec-RX to successfully capture at least one received packet (we presented the theoretical analysis for deciding this time in Section 4.5.2.3).

A malicious rich OS may manipulate the receiving packets in the normal world to pass illegitimate packets into the secure world. To protect against fake packets, the remote server and each mobile device share a public/private key pair to authenticate each command packets.

4.4.2 Secure Transmitting Module

The secure packet transmitting module Sec-TX is responsible for sending data back from the secure world to the remote server. Sec-TX extracts the normal-world driver's TX interface information as Sec-RX does. To avoid both drivers write on the same descriptor for sending packets and further triggers concurrency problem, the secure world directly takes full control of the NIC for a short time by suspending the rich OS on all cores. Without the cooperation of the normal-world driver in the normal world, the secure world needs to pause the rich OS, save the normal world driver TX data, send the secure-world related packets, and finally recover the normal world driver TX data.

To pause the normal world OS on all cores, Sec-TX issues a *Software Generated Interrupt (SGI)* to each core, which is then trapped into the secure world for executing its corresponding interrupt handler. The handler makes each core keep waiting in the secure world until the packet sending finishes. In other words, the Sec-TX initially running on one core is responsible for switching all other cores into the secure world and then back to the normal world. When Sec-TX is interacting with the shared NIC, since there is no core available for the normal world, the rich OS cannot disturb the execution of Sec-TX.

After all cores enter the secure world, Sec-TX saves all the content in the original TX descriptor ring buffers of the NIC driver. Then it sets the preserved memory region Sec-Buffer as the normal-world memory to save the pending transmitting packets. This step is necessary. Without this security change, Sec-Buffer cannot be accessed by the NIC that is a normal-world peripheral. To send a packet, Sec-TX generates the packets as normal network packets and encrypts the content with the private key of the mobile device. Then it uses the NIC registers to notice the peripheral for sending packets saved in TZNIC buffers.

When the packet transmission is done, Sec-TX removes all the packet data in Sec-Buffer and sets Sec-Buffer back as secure memory. Meanwhile, Sec-TX needs to recover both TX ring buffers and NIC registers back to the states before the normal world is paused; otherwise, any mismatch may incur errors or even crash the NIC. Finally, Sec-TX releases all cores to the rich OS and normal-world driver resumes working as normal. This mechanism guarantees that a normal-world driver does not need to conduct any extra operation to interact with the NIC while the secure world can send the packets at any time.

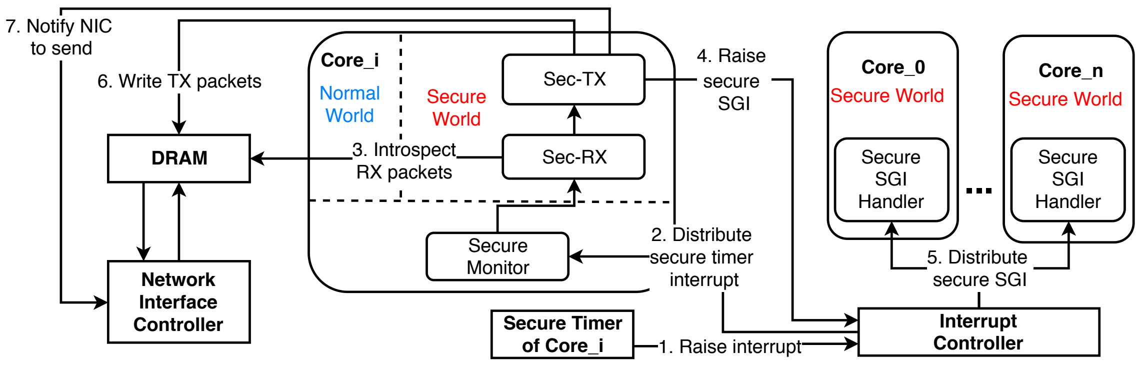


Figure 4.3: TZNIC Workflow

4.4.3 Secure Switching

A secure boot ensures that the software modules in the secure world and the rich OS in the normal world can be booted up after verifying the integrity of their image files. Besides, two interrupt handlers are registered to ensure one core entering the secure world after a timeout and force all other cores entering the secure world, respectively. First, each core has a separate secure timer to trigger *Private Peripheral Interrupt (PPI)* for itself [5]. Thus, a secure timer PPI raised by one core is delivered to this core only, so the secure timer interrupt handler can guarantee to switch this core into the secure world. Before TZNIC falls asleep, one core's secure timer is configured with a fixed time gap RX_sleep to make sure the TZNIC will wake up in the future to receive and respond to the network traffic. During the boot-up, the secure timer is initialized for raising the first interrupt.

Second, the latest ARM processors provide the SGI mechanism for the software to raise interrupts and thus achieving inter-core communication across the multi-core architecture.

The interrupt controller pre-defines the interrupt IDs for the CPU to generate both normal world SGIs and secure world SGIs. After registering the SGI handler with a specific secure SGI ID, a single CPU core within the secure world can generate the SGI to interrupt any other core or multiple other cores at the same time. Therefore, we configure the SGI handler to switch all CPU cores into the secure world and pause the normal world until the initiating core finishes its packet sending tasks.

4.4.4 TZNIC Workflow

We present a complete workflow of TZNIC in Figure 4.3. As the first step, TZNIC uses a secure timer to raise an interrupt after it expires every *RX_sleep* seconds. Then, the Interrupt Controller sends the secure timer interrupt to the corresponding core. After the secure monitor switches the core's security state from non-secure to secure, TZNIC invokes Sec-RX to read the received packets from DRAM memory in the normal world in step 3. If Sec-RX does not receive any command from the remote server, the core sets its secure timer for the next wake-up time and then switches back to the normal world. Otherwise, it decrypts the received packed and pass to the upon application in the secure world. In case that a response needs to be sent back, Sec-RX invokes the Sec-TX to send out the response packets. Sec-TX raises a secure SGI to trap all other cores into the secure world and thus pauses the normal world in step 4 and step 5. After all cores enter the secure world, Sec-TX generates the packets and directly leverages the NIC registers and descriptors to send the response packets in steps 6 and 7. Finally, after configuring the secure timer for the next round of wake up, Sec-TX exits the secure world and restores the normal OS on all cores. The entire workflow does not need any cooperation from the normal world.

4.5 TZNIC Evaluation

4.5.1 System Implementation

We implement a prototype of TZNIC on an ARM Juno r1 development board [7], which uses Marvell 88e8057-a0-nnb2c000 PCI-E Gigabit Ethernet Controller (Yukon-II NIC) as it's network controller. The ARM CoreLink TZC-400 TrustZone Address Space Controller [6] is used to manage the address space as either normal or secure. The TZC-400 supports up to eight separate memory regions with different security settings. A CoreLink GIC-400 Generic Interrupt Controller [3] manages the normal and secure interrupts following the ARM generic interrupt practice [5].

The normal world runs OpenEmbedded LAMP OS with Linux kernel version lsk-4.4-armlt in EL1. The OS is deployed with the network driver *sky2* (version 1.30) [49] to work with the Yukon-II NIC. The secure monitor running in EL3 is based on ARM trusted firmware (ARM-TF) [11]. The secure OS running in S-EL1 is modified based on the Test Secure Payload (TSP) of ARM-TF.

4.5.1.1 Secure World Initialization

We modify the secure timer interrupt handler of the secure OS to wake up the secure world and invoke the Sec-RX. The timer is configured by operating the per-core register $CNTPS\_CVAL\_EL1$. To pause the normal world, we register a secure-SGI handler at the booting process. GIC-400 supports up to 16 types (ID_0 - ID_{15}) of SGI, where ID_8 - ID_{15} are designed for generating secure interrupts [5]. We choose the secure-SGI ID_{10} for the Sec-TX to pause the normal world. The interrupt handler holds the core within the secure world and waits until the calling core releasing the core back to the normal world.

Besides the interrupt handler registration, TZNIC reconfigures the memory space for receiving and transmitting packets. We generate the page table entries for all normal world DRAM physical addresses so Sec-RX can read any descriptor or packet that saved in the normal world memory. Meanwhile, TZNIC reserves 0x00200000 bytes of the DRAM as the secure memory at the booting stage and sets it as an isolated memory region to serve as Sec-Buffer. The reserved memory is configured as readable, writable, and non-executable for secure world access only.

4.5.1.2 Sec-RX Implementation

An overview of the DMA-based NIC RX workflow is shown in Figure 4.4. To recover the runtime semantics of the normal world, the secure world would retrieves various RX descriptor ring buffer information from the NIC registers, including the starting address RX_Start_Addr, the ring size RX_Size, the current offset of the descriptor that the driver is handling RX_Tail, and the latest buffer NIC just updated RX_Head. When a packet arrives, the NIC gets the latest available descriptor offset saved in RX_Head, extracts the SKB address and then saves the packet into corresponding memory. Finally, NIC moves the offset RX_Head forward. The driver always handles the packet at RX_Tail and forwards this register to tell NIC which buffers have been read and are free to use in the future. NIC achieves the transmitting tasks with similar logic, while the only difference is that the network driver is responsible to update TX_Head and NIC will make TX_Tail catches the head register. The registers RX_Head and TX_Tail are read-only to the software and can only be written by the NIC.

To locate and monitor all normal-world driver's software interfaces such as the descriptors and RX packets in the normal world, we extract the RX descriptor ring buffer information from the NIC registers. The registers can be accessed by applying the offset, which is the sum of three parts: register base address $Y2\_B8\_PREF\_REGS$, the number of the queue, and the register offset. The ring buffer's starting address is set in two registers, where $PREF\_UNIT\_ADDR\_LO$ saves the lower 32 bits of the address and $PREF\_UNIT\_ADDR\_HI$ saves the higher 32 bits. By getting the complete address and corresponding offset, we can map each descriptor unit to the normal-world driver's descriptor structure $sky2\_rx\_le$. For each located descriptor, we identify the structure's attribute $\_\_le32$ $addr$, which refers to the SKB address of the real packets. Note the at-

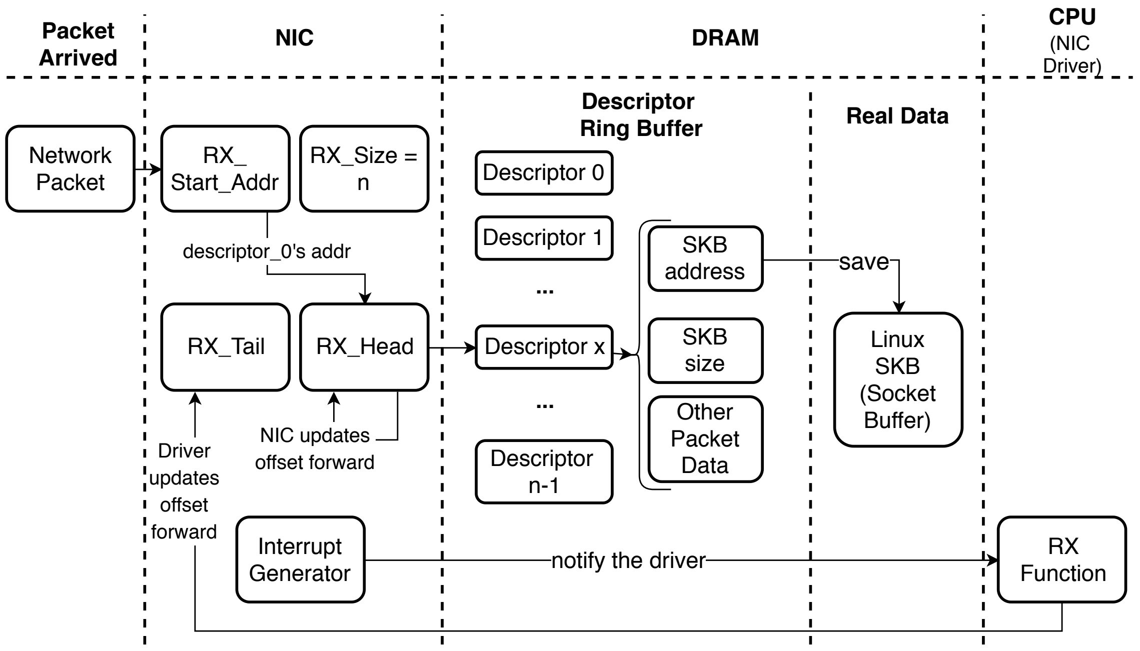


Figure 4.4: DMA-Based NIC RX Workflow

tribute *addr* saves the DMA address (physical address), so the Sec-RX needs to translate the address into the secure world virtual address. Finally, we can read the packet content at the virtual address of *addr*. We implement the remote server to send commands to mobile devices as UDP packets and their entire payloads are encrypted with the RSA algorithm.

4.5.1.3 Sec-TX Implementation

When Sec-TX is called, it first generates the secure-SGI interrupts to suspend the normal world on all cores. Sec-TX then writes the register $GICD\_SGIR$ with value $0x100000a$, to generate the SGI-10 and distribute the interrupt to all other cores.

After the rich OS is paused, Sec-TX reuses the normal-world driver's TX descriptor ring buffers for sending out the packet, and points the descriptor to the packets saved in Sec-Buffer. Sec-TX sets the memory region Sec-Buffer as normal world memory by configuring TZC-400 so the NIC can read packets out normally. After the NIC sends out the packet, Sec-TX needs to recover both TX descriptor buffers and NIC registers back to the states. It is easy to recover the normal world ring buffers since Sec-TX can write normal memory spaces.

However, it is not trivial to restore the TX registers to their original values. When the rich OS is paused, it has a local copy of the ring buffer registers, for example, both registers TX_Head and TX_Tail have the value x. Later, if Sec-TX sends out n packets then both registers will be forwarded at the position $x + n$. To recover the values, Sec-TX needs to recover both registers as n. Nonetheless, only register TX_Head is writable to the software while Sec-TX cannot directly write to the on-peripheral read-only register TX_Tail. To properly recover TX_Tail's value, TX exploits the feature of the ring buffer and always keeps the number of the sending packets n equals to the size of the TX queue. With this transmitting setting, Sec-TX can make the register value rolls back to the original position x because according to the ring buffer design, $x + n == x$. As such, both NIC registers and TX ring buffer are set back to their original states before the normal world NIC driver is resumed.

4.5.2 System Evaluation

Our evaluation aims to answer three key questions for TZNIC, namely, the overhead on the TCB size, the communication stability of TZNIC, and the performance overhead introduced on the rich OS.

4.5.2.1 Size of TCB

The original NIC driver (sky2-1.3) contains 5707 LOC. The slim TZNIC secure-world driver includes 722 LOC for both Sec-RX and Sec-TX and other 341 LOC for supporting the secure interrupt handlers, so its total size is 1063 LOC, which is only 18.63% of the original driver. From the memory perspective, the original driver uses one RX and one TX queue with 1024 buffers, and each buffer requires 8 bytes for the descriptor. Since TZNIC shares those descriptor buffers with the original driver, it can save 16 KB in total.

Moreover, since one SKB size varies from 64 to 65535 *bytes*, when deploying a complete NIC driver in the secure world, it may consume up to 128 MB. As the comparison, TZNIC only needs to allocate 2 MB static buffers in total for saving RX/TX packets and normal-world driver's TX context information.

4.5.2.2 Communication Connectivity

To study the connectivity of the receiving module Sec-RX, we deploy the python tool Scapy [23] on the remote server to send UDP packets to the device. We consider 100 sent packets as a test round and we evaluated 100 rounds for each scenario to calculate the packet received ratio. Sec-RX tries to intercept these UDP packets before they are read out and removed by the normal world. Meanwhile, we use the benchmark $iPerf$ [39] as the normal-world application to receive the UDP packets from the same sender with sending configuration in comparison. We first evaluate the reliability of Sec-RX under the scenario without race condition, which means the rich OS is benign and the software interface provides enough buffers for saving the received UDP packets. In this circumstance, $iPerf$ can receive 100% packets without any loss, and Sec-RX can also receive 100% of the packets when the module is wake up.

To evaluate the scenario that normal world raises the race condition on the received packets, we test the extreme case that the malicious rich OS attempts to utilize all the normal-world computation power to delete the incoming packets and therefore interfere with the secure world network availability. We deploy a kernel-level attacking program for such interruption-purpose. As the baseline of the attacking performance, the attacking program can promise to fully block the benchmark $iPerf$ from receiving any packet, which means the attack is strong enough to make any normal world application unavailable from the network perspective. Under such disturbance, Sec-RX still can receive 67% of the packets from the remote server on average, with the minimum rate as 22% and the maximum receiving rate as 92%. As the packets are repeatedly sent and loss-tolerant, receives any copy is enough for the secure-world applications.

When Sec-TX pauses the rich OS for sending out its packets, the availability of Sec-TX can be assured all the time no matter if the rich OS is malicious or busy.

4.5.2.3 Sec-RX Overhead

In this subsection, we measure the performance impact of Sec-RX on the rich OS. Since Sec-RX's system overhead varies with its awake time and awake period, we first present a theoretical analysis of their impacts. We then show the specific overhead introduced by Sec-RX when it's awake. Moreover, we study the extra power energy consumed by Sec-RX.

Theoretical Analysis. Assuming a real-world device has been turned on for the total period T and Sec-RX has been executed with the time period T_RX. If we define the device performance without our mechanism to be 100% and the degraded performance with Sec-RX as $Perf\_down$, we have the overall performance $Perf\_over$ as follows.

$$Perf\_over = \frac{T\_RX * Perf\_down + (T - T\_RX) * 100\%}{T} \tag{4.1}$$

Since Sec-RX sleeps with a fixed period RX_sleep, we have $T\_RX = T * \frac{RX\_awake}{RX\_awake + RX\_sleep}$. When we set the maximal wake up time RX_max to RX_awake, the worst performance case happens as follows.

$$Perf\_over = \frac{RX\_max * Perf\_down + RX\_sleep * 100\%}{RX\_max + RX\_sleep} \tag{4.2}$$

Next, we present how to adjust the time conditions RX_max and RX_sleep to satisfy the performance requirement. When TZNIC is set to achieve a target performance $Perf\_target$, we have the relationship between RX_max and RX_sleep as follows.

$$\frac{RX\_max}{RX\_sleep} = \frac{100\% - Perf\_target}{Perf\_target - Perf\_down} \tag{4.3}$$

As the Equation 4.3 indicates, TZNIC can maintain the rich OS with a stable overall performance $Perf\_target$ by properly tuning the time conditions RX_sleep and RX_max.

As a numberic example, assume we set the target performance $Perf\_target = 95\%$, and we choose the worst degradation performance $Perf\_down \approx 65\%$ as we evaluated in Section 4.5.2.3, then we know by setting $RX\_sleep = 7 * RX\_max$, the rich OS's overall performance is promised to be equal or higher than 95%.

Rich OS Overhead. We first use the tool *iPerf* to evaluate the communication overhead on the normal world. Our experiment shows that Sec-RX only introduces negligible overhead no matter the NIC is reading or sending packets in the normal world. This result is reasonable since Sec-RX only reads data from the memory using one core for a short period of time, while all other cores are still running the normal network operations in the normal world.

Next, we use the benchmark UnixBench [99] to evaluate the overall computation overhead in the normal world. We run the benchmark in two scenarios, with and without Sec-RX running on one core, using two sets of tests including `1-task` to conduct each performance test with one copy and `6-task` to conduct six copies simultaneously on all 6 cores in our mobile device. Figure 4.5 illustrates the experimental results on the performance overhead caused by Sec-RX.

In general, Sec-RX introduces 16.67% performance degradation on the 1-task tests and 23.54% on the 6-task tests. All the 6-task tests suffer more degradation than the 1-task tests since the 6-task tests are supposed to utilize 6 cores simultaneously while the test loses one core when Sec-RX is running. We observe that Sec-RX affects most benchmark tests with a stable overhead from 11.113% to 18.124% while the `execl` and `file copy` tests suffer the degradation from 29.517% to 38.610%. The reason is that most test sets cannot utilize all CPU resources even for the 6-task cases, but since both the `execl` and `file copy` performance results rely more on the CPU resources, these two sets are affected more by losing even one core. Noted that the presented performance degradation represents the $Perf\_down$ in Equation 4.2, which is only incurred when Sec-RX is awake, while most time Sec-RX is asleep so the performance is not affected.

Power Consumption Overhead. Sec-RX may incur extra power consumption since

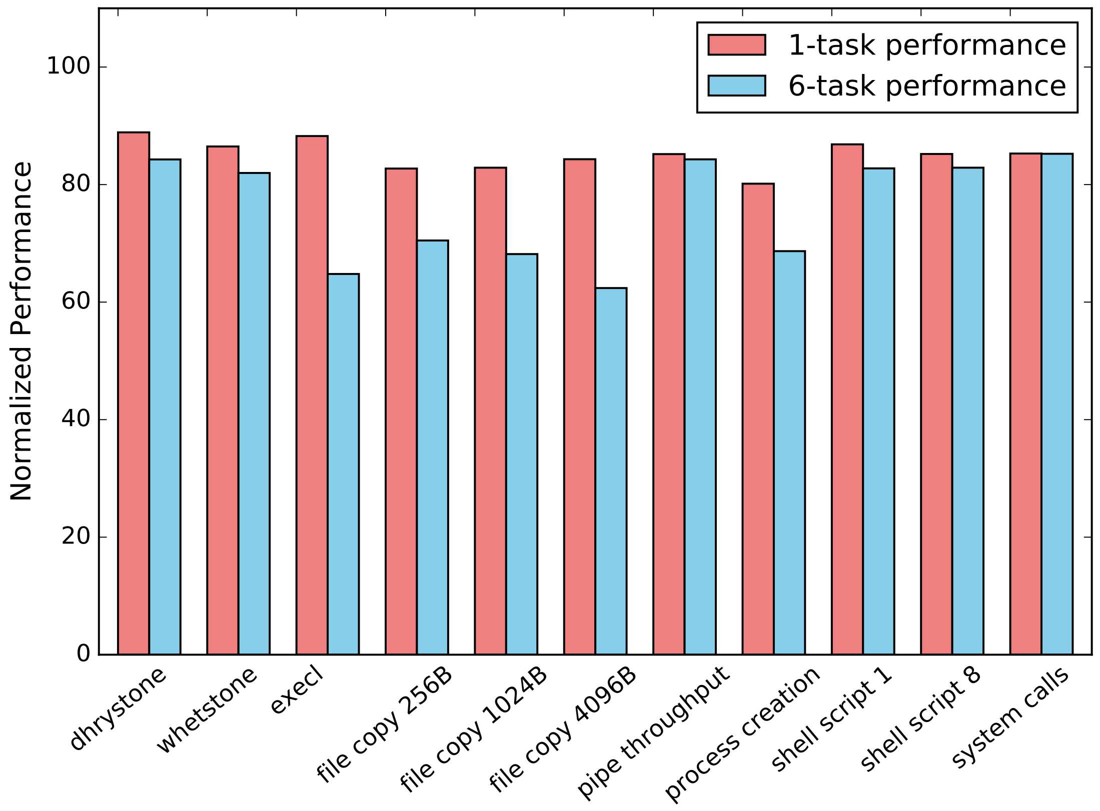


Figure 4.5: Performance Overhead on Normal World

it's running on an isolated core from the normal world. We hook up a Uniwood Energy Usage Monitor [108] to monitor the power consumption with the precision of $0.001W$. We measure the power under five scenarios and record the readings twice per second. Each scenario runs for a period of 2 minutes for 100 times.

The power consumption measured in each scenario is presented in Table 4.1. `Idle` means that the device runs only an idle normal world OS. `RX` represent the cases that the device runs the normal world only while the normal world OS is executing the $iPerf$ receiving tasks with the maximum bandwidth (94 Mbps). This scenario reflect the extra power consumption introduced by the NIC. `1-task` and `6-task` show the power consumption when the normal world runs the Dhrystone test set of UnixBench. `6-task` tends to use all CPU resources, so it consumes the maximum power in all scenarios. Finally, `Sec-RX` means that the device runs an idle normal world while executing Sec-RX at the same time. It shows that the extra power consumption introduced by the module is similar to `1-task`, since both cases use one CPU core consistently. In summary, Sec-RX introduces less than $0.3W$ power overhead to the device when Sec-RX is awake.

Table 4.1: Power Consumption of Sec-RX

Device Status	Average (W)	Min (W)	Max (W)
Idle	19.499	19.497	19.520
RX	19.602	19.586	19.613
1-task	19.601	19.585	19.606
6-task	20.356	20.350	20.360
Sec-RX	19.713	19.702	19.724

Similar to the performance overhead, the power consumption overhead is only introduced when Sec-RX is running. By setting $RX\_sleep = 7 * RX\_max$, the maximum extra power consumption overhead in one hour can be calculated as $1\ hr * \frac{RX\_max}{RX\_max+RX\_sleep} * 0.3\ W = 0.0375\ Wh$. As the baseline data, Samsung S9+ is equipped with a battery of 13.475 Wh [92].

4.5.3 Sec-TX Overhead

Table 4.2: Sec-TX Execution Time

Stages	Average (s)	Max (s)	Min (s)
Switching_Time	5.38×10^{-3}	5.54×10^{-3}	5.12×10^{-3}
One_Ring_Buffer	2.66×10^{-3}	2.84×10^{-3}	2.5×10^{-3}

Since Sec-TX suspends the rich OS when sending packets, we present the suspension time as the TX overhead. The time used by Sec-TX depends on the amount of data that needs to be sent to the remote server, which can be divided into two parts as we listed in Table 4.2. The first part is `switching_time`, which contains all the context switching operations Sec-TX needs to handle at each round. This part includes the time for saving and restoring the normal world TX ring buffers, updating the TZC-400 to prepare the normal world memory for storing packet data, and raising the SGI to pause all other CPU cores. The second part `One_Ring_buffer` reflects the time Sec-TX used for sending data for an entire ring buffer, whose size is 128 packets with our configuration. Recall that in Section 4.5.1.3 we explained that to correctly recover the NIC registers, Sec-TX must send packets with the total number in size of one or multiple ring buffers, depending on the size of packets, for the buffer to wrap around. This part can be affected by multiple factors such as network bandwidth, packet size, and ring buffer size, etc. In this case, we only present an empirical experiment result based on our experiment environment that sets packet size as 64 bytes and the network bandwidth as 100 Mbps. The average time on sending one ring with these conditions is $2.66 \times 10^{-3}\,\mathsf{s}$. The total time for switching and sending is $8.04 \times 10^{-3}\,\mathsf{s}$.

Table 4.3: Porting TZNIC to Other NIC Models

Type	**Brand**	**Model**	**Chipset**	**Driver**	**DMA-Based**	**Other Chipsets**
Wired NIC	Intel	EXPI9301CTBLK	Intel 82574L	intel / e1000e	Yes	82571, ich9lan, pch_lpt, and other 10 chipsets, total 13
Wired NIC	StarTech	ST1000BT32	RTL8110SC	realtek / r8169.c	Yes	RTL8100e, RTL8168cp, RTL8402, and other 32 chipsets, total 35
Wired NIC	Syba	SD-PEX24041	RTL8111F	realtek / r8169.c	Yes	driver has been covered above
Wired NIC	Realtek	RT8111C-PCIE-NIC	RTL8111C	realtek / r8169.c	Yes	driver has been covered above
Wired NIC	D-Link	DGE-530T	DGE-530T	marvell / skge.c	Yes	3Com 3C940, D-Link DGE-530T, and other 11 chipsets, total 13
Wireless NIC	Intel	7260.HMWG.R	Intel 7260	intel / iwlwifi / cfg / 7000.c	Yes	Intel 7260, Intel 3160, Intel 3168, Intel 7265, Intel 7265D, total 5
Wireless NIC	TP-Link	Archer T6E	BCM4352	broadcom / b43	Yes	BCM4306, BCM4311, BCM4318, and other 8 chipsets, total 11
Wireless NIC	Asus	PCE-AC56	BCM4352	broadcom / b43	Yes	driver has been covered above
Wireless NIC	StartTech	300 Mbps N PCI-E	Ralink-RT5392	ralink / rt2x00 / rt2800lib.c	No. Only TX data is sent via DMA	
Wireless NIC	FebSmart	N600	Atheros 802.11n	ath / ath9k	No. Only TX data is sent via DMA	

4.5.4 Portability of TZNIC

TZNIC architecture has three particular requirements, namely, multi-core processors, a high-privileged operating mode, and a DMA-based network peripheral. We show that all three requirements can be satisfied in a wide range of mobile devices. First, most ARM-based processors modern processors (e.g., A53, A57, etc.) are designed with the capability to be integrated with multiple cores. Any device equipped with more than one core is qualified to execute the secure-world and normal-world drivers simultaneously. Second, on ARM processors, the TrustZone technique has been widely integrated to provide an isolated execution environment for protecting the integrity and providing the high-privilege vision for TZNIC to inspect the on-peripheral registers and normal-world driver's software interfaces.

Third, we conduct a study to confirm that most modern network peripherals work as DMA-based NICs and the result is presented in Table 4.3. We first identify the top 5 popular wired and top 5 wireless network interface cards according to a list of best sellers in computer networking cards provide by Amazon [2]. For the top 5 most popular wired and wireless network NIC model, we find the related `Driver` information based on their `Chipset`. The column `DMA Packets` shows if the driver and corresponding hardware chipset are worked as DMA-based NIC or not. If yes, then TZNIC can be designed to cooperate with the corresponding NIC. Finally, since one driver may support more than one chipset so as long as the driver works as DMA logic, the chipsets in `Other Chipset` also can be extended with TZNIC. Since we cannot afford to buy all listed network devices, we only check the NIC models whose drivers are open-sourced and supported in the latest Linux kernel downloaded from Github [105] with the git-tag v4.18-rc7. Fortunately, all the drivers of `wired NIC` can be found in the directory of `drivers/net/ethernet` while the drivers of `wireless NIC` can be found in the directory of `drivers/net/wireless`. When a NIC brand has more than one model as the top 5 popular models, we only choose the most popular model for the brand. We skip the NICs that do not provide official documentation on their Linux driver support. We find that 8 of the 10 network devices support to work as DMA-based peripherals, and their drivers can cover more than 70 chipsets in total to work as the DMA-based NIC. Furthermore, for those two networking cards that are not working as DMA-based NIC, we find that their manufactures provide other products in DMA fashion, which means these branches also have alternative chipsets that can be integrated with TZNIC. For example, even though the driver `ath9k` of Atheros is not a DMA-based driver, another Atheros's driver `ath5k` that is working for Atheros 802.11a/bg Chipset can cooperate with the hardware as DMA-based operation [80]. Also, even the driver `rt2800lib.c` is not a DMA-related driver, another Ralink driver `rt73usb.c` that resides on the folder `<ralink/rt2x00/>` can provide DMA operation for both RX and TX data.

4.6 Work Conclusion

In this chapter, we develop TZNIC, a TrustZone-based network mechanism that utilizes a single physical network interface controller (NIC) shared between the normal world and the secure world on multi-core ARM processors. By properly exploiting the TrustZone architecture and features of DMA-enabled modern NIC, TZNIC can ensure filling the semantic gaps and then conduct receiving/transmitting network tasks simultaneously with the rich OS, while still being self-sufficient and transparent to the rich OS. TZNIC has a small trusted computing base (TCB) in the secure world. It requires no changes to the existing mobile operating systems, so it is promising to port our system to different brands of mobile devices. The experimental results show that TZNIC can achieve reliable side-band management on multi-core ARM processors with minimal system overhead.

Chapter 5

RusTEE: Developing Memory-Safe ARM TrustZone Applications

5.1 Introduction

Among the reported TrustZone-related vulnerabilities, most of them are caused by memory corruption of the memory-unsafe TAs [27]. Due to two architectural features of TAs, namely, conducting the cross-world communication with the REE and invoking kernel-privileged system-service APIs, TAs could be manipulated by REE-side attackers to compromise the entire TEE system. Researchers propose to move the execution of TAs from the TEE to the REE and thus prevent one vulnerable TA from corrupting other TAs or the Trusted OS [103, 24, 29]. Though these solutions can effectively mitigate the risk of vulnerable TAs, they will inevitably introduce non-negligible overhead over the system.

Recently, many programming languages focus effort on enhancing their memory-safety, and several new languages are proposed with memory-safety as one of the goals, such as Rust and Go. Meanwhile, researchers have applied the memory-safe languages from upper application layer (e.g., Intel SGX Enclave programs [110]) to lower system layer (e.g., embedded system OSes [64, 65]). One precondition to the engineering effort to rewrite the code base in these memory-safe languages is relatively small, so that developers can afford to convert the existing software into the memory-safe style. Meanwhile, since ARM TrustZone is proposed to protect a limited number of small security tasks, TAs become another ideal target to be rewritten in the memory-safe language.

In this work, we propose a mechanism called RusTEE to build TrustZone-assisted applications in the memory-safe style, using Rust [75] as the programming language. The basic idea is to leverage newly emerging memory-safe languages and provide a Rust-based Software Development Kit (SDK) on compiling memory-safe TAs to prevent against memory-corruption vulnerabilities. Specifically, we resolve several challenges to develop a TA with Rust. The first challenge is that none of TrustZone-assisted TEE system and associated ARM platform has been recognized as the official support target to the Rust. Therefore, we need to integrate all the Rust fundamental support such as the standard library into the TA development. Second, TAs are required to invoke the APIs of different system services, which are typically implemented as the kernel-privileged libraries. Since some low-level libraries require specific ARM assembly instructions that are not supported in Rust, it is impractical to rewrite all the libraries in Rust. Inspired by a recent work Rust-SGX [110], we solve this challenge by providing a binding layer between the Rust application and C system. The binding provides all the necessary interfaces for the TA dependent libraries while also enforcing the Rust's memory-safe standard on the bounded interfaces. Third, we resolve a TA-specific challenge, i.e., providing a secure cross-world communication channel for the TA in the TEE world to communicate with the software in the REE world. The security of the cross-world communication is ensured by regulating the TA's usage on any shared parameters between the two worlds.

After systematically studying the architectural specification of TrustZone-assisted systems, we successfully import Rust into TA development environment, and further apply multiple security enhancements to reliably invoke system-service APIs and securely conduct the cross-world communication. We develop a prototype of RusTEE based on an open-source project OP-TEE OS [69] and provide a variety of examples to demonstrate the functionalities and efficiency of RusTEE. We have open sourced the RusTEE prototype along with the memory-safe TA examples. The system evaluation has been conducted on multiple ARM platforms, including the AArch64 simulation and a real-world development board Juno r1 [12]. According to our experimental results, RusTEE only introduces 1% performance overhead on average on the evaluated examples. Moreover, RusTEE enables the TAs to be integrated with millions of existing Rust libraries, noticeably extending the functionalities of the TAs in the TEE.

5.2 Work Background

5.2.1 GlobalPlatform TEE Specification

ARM website [13] recognizes *GlobalPlatform TEE Specification (aka, GPD specification)* [46] as a widely used TEE architecture on the latest ARM processors. The GPD specification defines a clear security boundary for TrustZone-assisted TEE systems by providing a completed set of software definitions between REE and TEE. Currently, multiple real-world TEE systems, such as Linaro OP-TEE [69] and Trustonic Application Protection Solution [106], apply the design of GPD specification into their implementations.

According to the GPD specification, an REE hosts the rich OS (e.g., Android, Linux) in association with the user-privileged applications. While most applications are deployed and used entirely in REE as normal applications, some security-sensitive applications can enable the TrustZone protection on their sensitive operations. A security-sensitive application divides itself into two components, an REE-side component called *Client Application (CA)* and a TEE-side component called *Trusted Application (TA)*. The CA supports most non-sensitive functionalities like user interactions; however, neither the counterpart TA nor the TEE trusts the CA. Meanwhile, all sensitive operations are isolated as the TA, which usually runs on a Trusted OS inside the TEE. By leveraging TrustZone hardware-assisted isolation, the confidentiality and integrity of TAs are protected from the untrusted REE. The entire GlobalPlatform Architecture for a TrustZone-assisted device is shown in Figure 5.1.

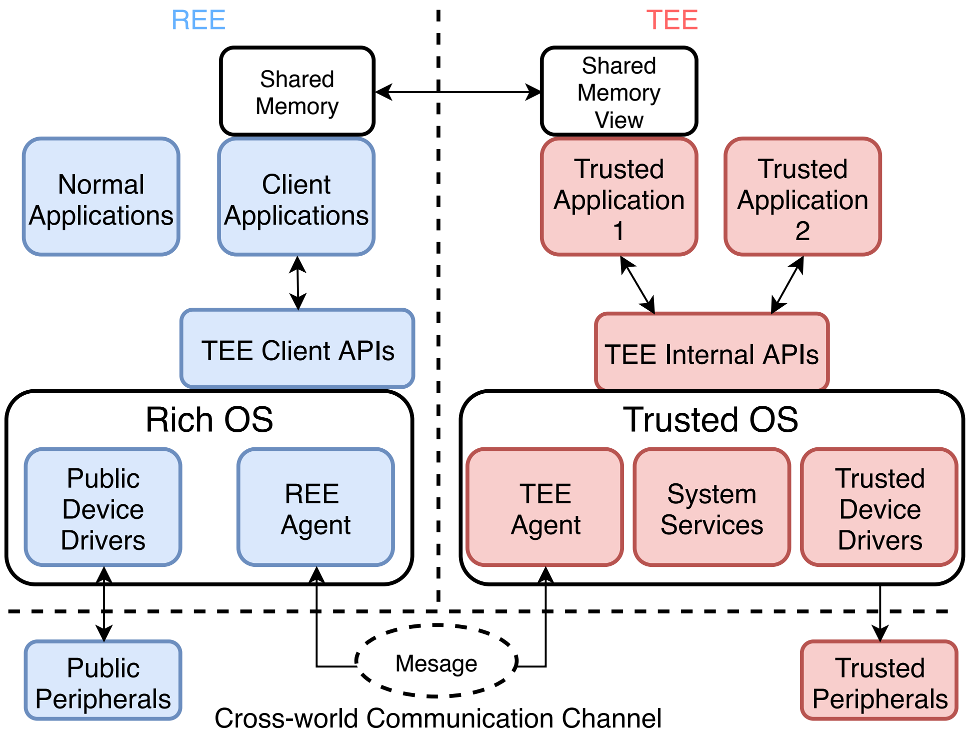


Figure 5.1: GlobalPlatform TEE Architecture

Since the CA and the TA run in two isolated environments, they perform cross-world communication in reliance upon an *REE Agent* and a *TEE Agent* for passing a command or exchanging the data. To request the trusted execution of a TA, the CA calls the *TEE Client APIs* [44] to ask the REE agent to send out the *Message* and build up the cross-world communication channel with a specific TA. Once the TEE Agent receives the Message, it initializes the corresponding TA to respond to incoming REE-side commands. The related responding APIs are defined as Cross-world Communication Channel APIs that belong to *TEE Internal APIs* [45]. To exchange data between two environments, the CA first allocates the communication memory called *Shared Memory* in the REE and then shares the memory with the corresponding TA. Since the TEE has a higher privilege on accessing the REE's memory, the TA can also operate on the shared memory in parallel with the CA.

Besides the communication functions, GlobalPlatform also defines its TEE Internal

APIs to provide essential *System Services*, such as cryptography-related operation, secure storage, and big-number calculation. Since all TEE Internal APIs are provided to all TAs for calling directly, TAs are not required further to implement their own functionalities for these security services. Moreover, many of the GPD TEE Internal APIs are involved with dedicated memory-related operations, which should be thoroughly inspected before running them inside TEE.

5.2.2 Rust

Rust [75] is a programming language designed to achieve both reliability and efficiency. To achieve reliability in two distinct aspects, namely, memory-safety and thread-safety, Rust provides the following mechanisms: (1) claiming the *ownership* of each data object; (2) automatically checking the read/write permissions (*mutability*) of each object; (3) enforcing the *lifetime* managements on all objects; (4) forbidding unsafe typecasting (*type-safety*); (5) disabling dangerous *raw pointer operations* like pointer aliasing or dangling pointers. During the program compilation, if the code violates any Rust's security criteria, the Rust compiler raises errors and generates error messages to help developers correct their code accordingly. Besides improving the code security, Rust brings other benefits such as the highly efficient parallelization, the developer-friendly compiling messages, and thousands of *crates* (similar to the libraries in C language) for supporting different development requirements.

Rust-safe vs. Rust-unsafe. Though Rust is designed to achieve strict security criteria by default, to guarantee any program can indeed be written in Rust, it also provides the keyword `unsafe` [104] for developers to inject memory-unsafe code segments. Rust provides this unsafe option for two primary reasons: 1) allowing developers to develop some "special" functions the cannot pass the compiler's default inspection; and 2) allowing the code to interact with system/hardware components directly. A segment marked as unsafe can bypass the Rust built-in check and therefore may conduct vulnerable behaviors, such as writing on an immutable variable, conducting a non-standard typecasting, or using raw pointers directly. A typical scenario of using unsafe code segment in Rust happens when the Rust code has to invoke the C-based functions, which is defined as *Foreign Function Interface (FFI)* in Rust. Coming with the advantages of extended capabilities, unsafe Rust also introduces security risks. Several related works [104, 21, 22, 107] have revealed that unsafe Rust can introduce potential security risks.

5.3 Motivation and Challenges

5.3.1 Motivation

Over the past decades, more than one hundred vulnerabilities have been reported for TrustZone-assisted TEE systems [32, 33, 31]. Among these reported vulnerabilities, most of them are software-related, which means the vulnerabilities can get exploited even if the device enables and configures TrustZone hardware components appropriately. Recently, Cerdeira et al. [27] provide a systematized summary about the vulnerabilities of existing TEE systems, and they summarize the software-related vulnerabilities in two categories, namely *implementation issues* and *architectural issues.* The implementation issues refer to the bugs triggered by specific implementation details of one TEE system, such as lacking proper security checks on the sensitive variables. Meanwhile, architectural issues include shared deficiencies or design flaws among different TEE systems, regardless of systems' implementation details.

In order to mitigate software-related vulnerabilities on TrustZone-assisted TEE systems, one critical and challenging task is enhancing the security of TAs. Nowadays, commercial TEE systems integrate more and more TA functionalities into the TEE, excessively increasing the total size and semantic complexity of the TEE. With such a large number of complicated TAs, it is impractical for the TEE system's administrator to conduct either artificial or automatic validation on each TA's correctness. Consequently, TAs may get imported into the TEE with potential implementation issues, such as conducting sensitive operations without appropriate validations. Moreover, when TAs are developed in memory-unsafe languages like C language, these implementation issues are difficult to be fully reviewed since a memory-unsafe language can perform dangerous memory operations and cause implementation issues with many possibilities.

Besides introducing implementation issues, TA is also the critical component of two TrustZone-specific architectural issues. First, the TA's capability of invoking kernel-privileged system services can be abused to attack the TrustZone-assisted TEE system and even lead to a compromised TEE. To support the incremental functionalities of TAs, Trusted OSes deploy many system services and expose wide interfaces to TAs; however, there is no security regulation on the interactions between TAs and the Trusted OSes. Therefore, if the vulnerable TAs can be manipulated to invoke system interfaces maliciously, the entire mobile system may be compromised as well. How to govern the interface between the Trusted OS and TAs is an essential architectural challenge when deploying TEE systems. Second, most TEE systems allow TAs to accept input from the REE via the cross-world communication channel. However, since the REE is untrusted and may be fully controlled by attackers, the cross-world communication channel expands the attack surface of the TEE system.

In real-world scenarios, when both the implementation and architectural issues exist in a single TA, they may be exploited together and lead to severe consequences. For instance, a recently reported vulnerability CVE-2018-14491 [48] utilizes a vulnerable One-Time-Password TA for executing arbitrary code on Samsung S5 smart phones. Similar security issues have been reported in other CVEs such as CVE-2015-6639 [32] and CVE-2016-2431 [33]. Motivated by resolving both implementation and architectural issues, we propose to implement TAs in a strict memory-safe style and further mitigate the identified issues of TAs. In the following section, we present three particular challenges and our basic ideas for solving them.

5.3.2 Challenges

The primary object of RusTEE is to provide a secure mechanism that assists developers in building TAs with a memory-safe regulation. Specifically, there are three main challenges we need to resolve to build the required secure TAs.

Challenge-1: Tackling memory corruptions in TA. One fundamental attribute of a secure TA is that the TA does not contain any memory-unsafe implementation issues. In other words, our method should ensure to remove memory corruptions from TAs, such as Use-After-Free or Data Race. To address this problem, we propose to write TAs in the memory-safe programming language Rust.

Challenge-2: Providing secure system-service APIs. Unlike some TEE architecture (e.g., SGX) that can provide multiple hardware-enforced-isolated enclaves, the TrustZone-assisted TEE system only deploys one shared Trusted OS for executing all TAs. Therefore, any compromised TA may utilize the widely provided system-service APIs to attack the shared Trusted OS and compromise all other TAs. In order to eliminate the side-effect of exposing wide APIs to TAs, we provide a binding solution that enforces the Rust's memory-safety on the existing unsafe APIs to prevent TAs form misusing any kernel-privileged TEE system services.

Challenge-3: Building protection on cross-world communication. As an architectural feature of TEE systems, the cross-world communication channel is a must to support the collaboration between TEE and REE. However, this channel also provides another vehicle for the REE-side attackers to manipulate TAs' behavior, especially considering that the communication channel is connected via the untrusted REE's memory. To enhance the security of the cross-world communication channel, we redesign the cross-communication interfaces of TA, which conduct security checks on the passed-in parameters and limit the use cases of untrusted parameters.

5.4 RusTEE Design

In this section, we first present the threat model and overview architecture of RusTEE. Then we elaborate on the detailed security enhancements of RusTEE for resolving TA's security challenges.

5.4.1 Assumptions and Threat Model

We assume the device is equipped with ARM TrustZone technology, and the technology is can provide the hardware-enforced isolation. We assume all TEE system's software components, including the secure monitor, Trusted OS, and all TEE kernel-privileged libraries, are implemented in compliance with the GlobalPlatform TEE specification. In this case, TAs use the GlobalPlatform-defined (GPD-defined) APIs to interact with system services and the cross-world communication channel. We also assume these system components are well written, so there is no insecure flaw in Trusted OS or lower level software. As such, we focus on protecting the memory-safety of TAs that run above Trusted OS. Finally, we assume the TA developers are benign while he or she may still program a TA in a vulnerable way, which is a common scenario recognized in the recent CVEs [32, 33, 31].

5.4.2 Overview

We present the overview architecture of RusTEE in Figure 5.2. The main idea of RusTEE is serving as a Rust-based TA SDK in the TEE. The SDK supports most general development requirements, such as operating primitive data-types, in the strict Rust-safe style by providing Rust standard library and associated essential components to TA developers. With the assistance of the Rust compiler's built-in security checks, RusTEE ensures the TA's source code is free of known memory-corruption bugs and therefore mitigates *Challenge-1.* Since the major challenges for porting Rust standard library into ARM platforms are implementation-related, we will introduce them later in the Implementation Section 5.5.1.

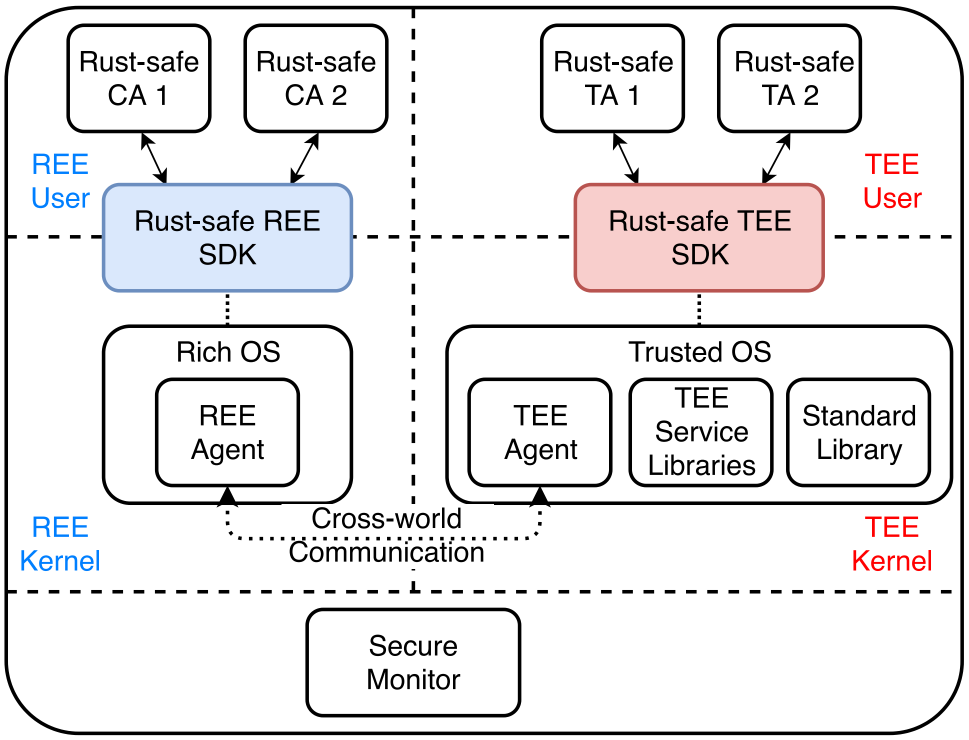


Figure 5.2: RusTEE Architecture

Besides performing general-purpose operations, a TA also needs to invoke functions of particular TEE's system services, which are out of the scope of Rust standard library. Therefore, RusTEE integrates the extra libraries into SDK to support these requirements. There are two design options for shipping a Rust-based SDK with additional libraries. The first option is rewriting all the requested libraries in Rust. The other option is building up the Rust-based SDK based on full-fledged C-based libraries, and further providing a trustworthy binding between Rust and C components. Though the first option offers better independence and memory-safety, it faces two non-trivial challenges when implemented on the ARM TrustZone-assisted platforms. The first challenge is that some TEE's system services involve the TrustZone-specific operations (e.g., reading a secure timer), while these operations can only be implemented with the explicit essential ARM instructions that are unavailable in the Rust's standard supports. Another challenge is that for some TEE's system services (e.g., cryptography), the C-based libraries have better performance than the Rust ones. In consideration of these challenges, we propose to provide the SDK as the binding solution. After systematized studying all critical data structure and function definitions of these additionally involved libraries, RusTEE converts all the interfaces into Rust-safe style to resolve the *Challenge-2*.

Meanwhile, TAs used to face the challenge of handling the commands and parameters that are passed-in via the cross-world communication channel, since these data are generated by the untrusted REE. By carefully reviewing the calling convention of existing cross-world communications, we redesign the connection interface between the TEE's system communication component (TEE Agent) and TAs. The redesigned communication interfaces promise that all parameters are used under secure standards and therefore handle *Challenge-3*.

Finally, RusTEE provides the REE-side SDK, which follows a similar scheme of the TEE-side SDK, as a complementary component to regulate the behaviors of CAs. Note that the security of TA does not depend on whether the REE utilizes the REE-side SDK or not, and the REE-side SDK is provided only in the case that benign CA developers want to improve a CA's memory-safety. In the following section, we focus on presenting our methodology for mitigating the architectural issues for the Rust-based TAs, particularly, securing the widely exposed system-service APIs (hereinafter referred to as "service APIs") and cross-world communication channel.

5.4.3 Secure System-service APIs

In the design of RusTEE, the Trusted OS implements TEE's system services as the C-based libraries for the best practicality, and the OS provides C-based service APIs to the upper-layer applications. To make these APIs available for Rust-based TAs, RusTEE should reliably convert these C-based interfaces into the Rust-based interfaces. We call this conversion as the binding solution. To bridge the semantic gap between Rust and C language, Rust officially provides a standard crate `std::libc`, which matches all data types and structures that are shared by two languages, such as `c_int` and `c_char`. Also, Rust provides the *Foreign Function Interface (FFI)* mechanism to allow Rust-based programs for invoking C-based functions in the Rust-unsafe way. By utilizing these two Rust's components, we can straightforwardly convert the C-based interfaces as the Rust-unsafe interfaces via FFI mechanism, and allow the upper-level TAs to interact with the low-level APIs via the parameters that are matched by `std::libc`.

However, the FFI-based bindings are not memory-safe for TAs to invoke. As we explained in the Background Section 5.2.2, since Rust's built-in security checks ignore any code segment marked as FFI, the bonded APIs can still contain memory-unsafe vulnerabilities. To ensure the security of these bindings, RusTEE applies multiple security-enhancements on the service APIs. In this subsection, we first introduce four general principles that are adapted as the enhancements for all bonded C-based service APIs. Then we present two particular binding principles that we propose for protecting GPD-defined service APIs.

Secure C-based APIs. As one close-related work of RusTEE, Rust-SGX [110] provides a secure binding for Intel SGX between Rust enclave applications and C-based SDK. More importantly, the authors conclude two common challenges for providing binding between Rust and C worlds, which are providing safe memory access of C/C++ objects and raw-bytes. The first challenge is introduced for achieving the type-safety in Rust. Ideally, every type in the Rust program has a precise definition for providing clear semantics about types' use cases. Moreover, an explicit type definition can describe all the legitimate scenarios for casting one type to another. However, in C-based libraries, many complicated data types can only refer to a pointer type `void`, and the pointers can be dangerously accessed with the wrong interfaces when the developer uses them carelessly or confused. The second challenge happens when C-based libraries access the memory chunks directly based on their pointer and length, which is considering as unsafe and not-allowed in Rust. Such pointer/length combinations frequently appear in C-based libraries.

To resolve these two challenges, Rust-SGX defines four principles, which notated as `Bytes`, `ContiguousMemory`, `Sanitizable[T]`, and $\texttt{Handle}_{\tau}$. These four notations can regulate how to convert the challenging C-style APIs into Rust-safe style. Specifically, $Handle_T$ maps each C-based unsafe pointer into specific secure type in Rust. $Bytes$ constructs the concrete memory in the format of arrays for securing memory accesses. The rest two notations $ContiguousMemory$ and $Sanitizable[T]$ are provided for handling the conversion between other unsafe C-based types `T` and the proposed $Bytes$. Moreover, Rust-SGX provides solid formalization to prove the four notations' security with the system defined in CCured [79].

Since RusTEE shares a similar binding solution as Rust-SGX, we adopt all four principles proposed in Rust-SGX. For example, we provide a specific handler for each critical data type. We further realize the other three security principles to bind the service APIs securely. Similar to the solution of Rust-SGX, the realizations of these principles require manual effort to review all libraries' critical data structures and understand the associated memory utilization. To the best of our knowledge, there is no automatic mechanism that can promise a perfect conversion from C-based APIs to Rust-based ones. Hence, we claim such a manual process is acceptable and has the most reliable security-promise for the bonded APIs.

Secure GPD-defined APIs. After thoroughly reviewing all APIs defined in the GPD specification, we identified two additional issues besides the four principles proposed by Rust-SGX. The first issue is that some TEE Internal APIs have complicated dependency-checks. For example, an `API-a` may only be allowed to be invoked when the `API-b` returns a specific `value-c` as the running result. To avoid the case that the developer misses any dependency-check, we enforce every depending API (e.g., `API-a`) to conduct such check automatically, and therefore promise the function of `API-a` is only executed when the required condition is met. For any case that the dependency-check fails, GPD specification defines the invocation on API should be interrupted, and we relay the unexpected status to the Rust error-handling process.

The second issue is that some GPD-defined services require multiple APIs to work in a specific sequence, especially for memory allocation and release. However, TAs can be programmed to invoke these APIs in the wrong order, or even missing some critical steps. To avoid the TA misuses any memory object, we enforce the *Resource-Acquisition-Is-Initialization (RAII)* [101] standard on such APIs. According to the RAII standard, any data structure, named as `struct` in Rust, should be promised with a correct initialization. Moreover, when the developer finished the task on the `struct`, the data structure should provide the correct function to free the resource as well. By enforcing the RAII standard on critical data structures, the memory-related APIs are promised to get execution in the correct sequence.

```
1  /* Implement the details of the structure to enforce security principles */
2  impl OperationHandle {
3      fn allocate(algo:AlgorithmId,
4                  mode: OperationMode,
5                  max_key_size: usize) -> Result<Self> {
6          match unsafe { raw::TEE_AllocateOperation(...) }
7          {
8              /* Check the allocation result automatically */
9              raw::TEE_SUCCESS => Ok(Self::from_raw(raw_handle)),
10             code => Err(Error::from_raw_error(code)),
11         }
12     }
13     ...
14 }
15
16 /* Enforce the resource release with the assistance of the language's type security */
17 impl Drop for OperationHandle {
18     fn drop(&mut self) {
19         ...
20         unsafe { raw::TEE_FreeOperation(self.handle()); }
21         ...
22     }
23 }
```

Listing 1: A Redesigned Encryption-related Data Structure

We present an example for applying our GPD-specific principles in List 1, which is a redesigned Rust-based data structure `OperationHandle` used in TEE's encryption-related operations. As shown in line #9 and line #10, when the structure is allocated, the TA can only move forward if the allocation's return value is `raw::TEE_SUCCESS`, while all the other return values are forwarded to `Err` handler. In this case, as long as developers utilize the redesigned API `OperationHandle::allocate` to acquire the data, the API is promised to check any "potential dangerous return value" and avoid the first issue. Furthermore, when the TA finishes using the allocated data structure, the data is freed automatically because the Rust compiler would execute the function `Drop` (from line #17 to line #23) by default. Therefore, the redesigned `struct OperationHandle` is protected from the second issue.

5.4.4 Secure Cross-world Communication

As an architectural feature of TrustZone-assisted TEE systems, the cross-world communication channel supports the TEE-side TAs to work coordinately with the REE-side CAs. According to the GPD specification, four key data structures are defined and used across the entire CA/TA cooperation process, namely *Context*, *Session*, *Command* and *Parameter*. Starting from the beginning, the CA is required to register its Context in the TEE, without requesting any specific TA to collaborate. Next, the same CA needs to set up a connected Session between it and a specific TA, and this Session is only valid under a registered Context. Once the Session has been correctly set up, the CA can make the following requests to the TA via passing different Commands. If any Command requires the usage of cross-world shared memory (e.g., sharing the plaintext/ciphertext across REE and TEE worlds), the Command can be passed with at most four pairs of Parameters. Each Parameter can represent either a numeric value or a memory chunk. For the entire process of a cross-world communication, we identify three security issues of these four data structures and propose the corresponding security enhancements.

Secure Context's and Session's Lifetime. One premise of successful communication is that the two fundamental data structures, namely the Context and the Session, are correctly initialized. However, this prerequisite can get challenged in several ways with the GPD specification. According to the GPD specification, these two structures are referred to as unsafe raw pointers, and the caller function has no way to tell whether the callee structure is correctly initialized or not. Moreover, a wrongly used structure may lead to a compromised communication scenario. For example, a C-based CA can get manipulated to connect its Session with another malicious CA's Context without getting any error. In such a case, any further operation may get exposed or even manipulated by the malicious CA. To protect the usages on these two structures, we redesign the Context and Session structure as Rust type-safe structures, which can promise the structures are always adequately initialized before use. Furthermore, We take advantage of Rust's `Drop` function to promise these two structures' resources are released as the GPD-defined serialization and, hence, promise the corresponding data is erased after use.

Secure Parameter's Type-safety. We discovered two security issues of the communication data Parameter. First, the Parameter is defined as type-unsafe in the GPD specification, because TAs access Parameters without a clearly defined type. In this case, a TA can use a numerical Parameter as a memory pointer, or vice versa. To provide Parameter as type-safe, we convert all existing Parameter use cases into two specific Rust-safe data types, namely `int` and `slice`, to pass the numerical value and memory chunks, respectively. With the enforced type definition, any misusing will get detected during the compilation stage. The use of `slice` can also regulate CA's behavior to share the memory chunks. Previously, REE allocates all memory buffers for a Parameter. Then REE shares the memory region with the TEE by providing the corresponding memory's raw pointer and size. This memory-sharing process is unsafe since the attacker can manipulate the pointer and size to mislead the TA to access the memory out-of-scope. By converting the Parameter as Rust `slice`, the memory pointer and associated size are guaranteed to get a securely typecasting, which can prevent TAs from further being manipulated to access the wrong memory region.

Secure Parameter's Mutability. Another security concern of Parameter is that a TA may access the TA with incorrect read/write permissions. The GPD specification defines three permissions of Parameters as `input`, `output`, and `inout`, and the Parameters are supposed to get accessed as read-only, write-only, and read/write, respectively. However, a GPD-defined communication channel provides these permissions as independent flags from the corresponding variables, which makes these permissions easily violated. For example, even a memory chunk is designed as a read-only `input` Parameter, a TA can still write on this Parameter as long as the developer does not manually check the Parameter's permission. In Rust, all the read and write permissions are managed via the *mutability* feature by default. By taking advantage of the mutability, RusTEE enforces the permission-check for every Parameter and therefore prevents future violations.

5.5 RusTEE Evaluation

5.5.1 System Implementation

We develop the prototype of RusTEE based on the project OP-TEE [69], which is one of the most well-known open-sourced TEE projects for ARM platforms. OP-TEE implements its Trusted OS and associated software interfaces in compliance with the GPD specification. Currently, the OP-TEE project is available for many ARM TrustZone-assisted devices [71], including the simulation environment QEMU [89], and experimental development boards such as HiKey family [1], Raspberry Pi 3 [90], and Juno [12]. In the following section, we present our modifications to the OP-TEE project for two aspects, namely porting Rust into OP-TEE and binding OP-TEE's Internal APIs (including service APIs and cross-world communication APIs). Meanwhile, we implement the REE-side SDK and rewrite all OP-TEE official C-based examples in Rust. Our rewritten examples demonstrate RusTEE's practicality. Note that we already release RusTEE as an open-source project on GitHub[1], and the latest version supports building both TA and CA in the Rust-safe style. Moreover, RusTEE is configurable to build applications for two most popular ARM architectures: AArch32 and AArch64.

[1]For the anonymous-review purpose, the project's link will be placed here after the review.

5.5.1.1 Porting Rust into OP-TEE

Though Rust officially provides the compilation-support on multiple platforms, none of the OP-TEE-supported platforms is recognized by Rust yet. Moreover, in order to balance the functionalities and Trusted Computing Base (TCB) size of TEE, OP-TEE redesigns its basic library `libutil`, which makes it unmatched to the Rust official crate `std::libc`. To resolve these challenges, we first modify the Rust fundamental components `compiler-builtins` and `rust/libstd` to add OP-TEE as the supported targets, which can be further configured based on the architectural features of `arm` (AArch32) or `aarch64` (AArch64). Furthermore, we manually inspect the OP-TEE's basic library `libutil` and match it with the `libc` crate. As the `libutil` does not fully implement all featured functions presented in `libc`, the matching process is realized as a best-effort solution by acceptably sacrificing some functionalities. For example, due to the implementation limitation, a TA runs in OP-TEE OS is implemented as a single-thread task, and the kernel does not provide any multi-threading management. In this case, whenever a Rust program invokes the thread-related operations, we raise panic messages for these operations to remind the developers.

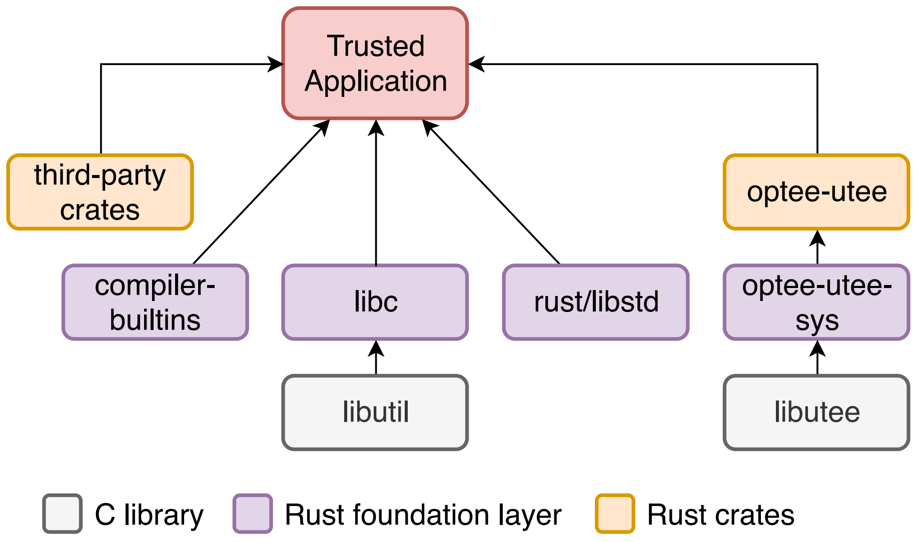


Figure 5.3: Porting Rust into OP-TEE

Besides the three discussed components of Rust's foundation layer, we also provide one extra component `optee-utee-sys` to bind OP-TEE's specific library `libutee` for providing functionalities of all Internal APIs. We further wrap the raw component `optee-utee-sys` as a safe Rust crate `optee-utee`. The details of this binding can be found in Section 5.5.1.2. By integrating all the foundation components along with `optee-utee`, RusTEE provides the comprehensive functions for the TA developers to program a TA in Rust-safe style. Finally, RusTEE also supports developers to import trusted third-party Rust crates into the TA development. The entire implementation structure is presented in Figure 5.3.

5.5.1.2 Binding OP-TEE'S TEE Internal APIs

GlobalPlatform TEE Internal Core API Specification [45] defines six types of the necessary APIs for TA development. The first type *Trusted Core Framework API* defines the APIs that provide basic OS functionalities for all kinds of TAs, such as memory management, system-information retrieving, and cross-world communications. For example, each TA should call the same set of APIs to construct and maintain the communication channel with the REE. Moreover, in the current implementation of cross-world communication, we label two operations, `Parameter::as_value` and `Parameter::as_memref`, as `unsafe` operations because OP-TEE's `Parameter` are implemented as unsafe from Rust's ownership and thread-safety perspective. Specifically, whenever a TA receives the data in the shared memory, the CA and REE still have the privilege to modify the Parameter, so there exists a potential concurrent issue for using shared Parameters. Currently, these two operations are the only two exceptions that can appear in the TA source code as `unsafe` segment. Note that the `unsafe` labels here do not mean any memory vulnerability is actually introduced, while they are more to syntactical definitions to alert the developers. For example, whenever the TA is supposed to use any passed-in data array exclusively, it should copy the data from the unsafe Parameters into a safe array, and then conduct rest operations reliably.

The second type is *Trusted Storage API for Data and Keys*, which provides reli-

Table 5.1: RusTEE Component's LOC

Component	**Lines of Code**
TEE	
Trusted Core Framework API	2076
Trusted Storage API	544
Cryptographic Operation API	672
Time API	52
TEE Arithmetical API	258
REE	
Client API	687
Examples	
Rewritten OP-TEE Examples	1964
Newly Added Examples	2105
Total	8358

able storing for security-sensitive structures, and mostly applied on the cryptography keys' materials. Thirdly, *Cryptographic Operation API* defines the APIs for extensive cryptographic-related tasks such as generating the key, conducting synchronous/asynchronous encryption, and hashing calculations. Next, *Time API* can return the trusted time for TAs, where the time can be selected from different perspectives such as per-TA time, Trusted OS's unified time, or even REE's Rich OS's time. Moreover, *TEE Arithmetical API* are the essential functions that majorly serve for calculating big numbers and primes. Lastly, *Peripheral and Event API* is designed to allow TAs to interact with the hardware peripherals. Most of the peripheral-APIs are platform-specific as different platforms can equip a variety of peripherals. Since OP-TEE OS only implements the first five types of APIs, our prototype binds all of the implemented APIs, and we list the Lines-of-Code (LOC) of each type in Table 5.1.

5.5.1.3 REE and Examples

Besides the TEE-side SDK, we also implement the crate `optee-teec` as the REE-side SDK, which integrates the Rust standard library and other GPD-defined Client API-related libraries to support building secure CAs. Presently, OP-TEE provides six examples to demonstrate the CA/TA workflow in several aspects, such as basic communication functionalities, secure storage, and cryptography-related tasks. To prove the practicality of RusTEE, we completely migrate these six examples by rewriting them in Rust. Moreover, we provide six more examples to present RusTEE's capabilities of interacting with all types of TEE Internal APIs. Finally, we provide one additional example for exhibiting the case that integrates third-party Rust crate `Serde` into TA development. The detailed examples and corresponding performance evaluation are presented in the Evaluation Section 5.5.2. The latest project's LOC[2], which includes both worlds' SDK and examples, are summarized in Table 5.1.

5.5.2 System Evaluation

In this sub-section, we present the performance evaluation of RusTEE. Compared to the previous TA-development mechanisms, our mechanism introduces performance overhead in two aspects: the general overhead of changing programming language and specific overhead of API-related enhancements. First, since RusTEE replaces the previous programming language C with Rust, RusTEE may introduce the overhead because of using the new language. Though some existing benchmarks already presented the difference between these two languages on the x86 platform, we notice their performances vary a lot on ARM devices. Therefore, we present the language-wise difference between C and Rust for ARM devices specifically. We implement four benchmark programs in both languages and evaluate the programs' performances on the ARM-based Juno r1 [12] development board. Furthermore, we re-run the benchmark on the emulator environment QEMU [89] with the same ARM architecture to validate the observation.

Besides the differences in programming languages, RusTEE may introduce extra overhead because it performs multiple security enhancements on the TEE Internal APIs. Since the overhead of invoking APIs is tightly coupled with the real-world use cases, we evaluate

[2]The LOC are counted at the time of this dissertation is written and may change in the future version of the open-source project.

this overhead based on five real-world TAs provided by OP-TEE [70]. We rewrite each TA in Rust and then compare our rewritten TA with OP-TEE's C-based TA. The difference between the two TAs' execution time can indicate the overhead of corresponding APIs.

5.5.2.1 Language-wise Overhead

To present the fundamental difference between languages C and Rust on ARM devices, we evaluate them with four benchmark cases of the open-source programming language benchmark-set [47]. The benchmark-set provides dozens of cases in different languages for evaluating their computation efficiencies on x86 devices. However, it is non-trivial to migrate all benchmark programs on ARM devices because many programs rely on the libraries that are not supported by either C or Rust compiler on ARM platforms. Moreover, as OP-TEE OS only provides limited functionalities in the TEE, TAs are not capable of integrating any benchmark's program completely. After manually reviewing the benchmark-set, we select four cases that can get compiled and executed on ARM platforms stably for both languages. We implement the benchmark programs in the REE to get the support of the Rich OS, which equips the Linux kernel in our implementation.

Among the evaluated cases, case `n-body` models the orbits of Jovian planets as a double-precision simulation; case `fasta` generates and rewrites DNA sequences; case `fannkuch-redux` performs the indexed-access to tiny integer-sequence with the approximated time complexity $n * \log n$; case `spectral-norm` resolves the mathematical challenge [114] that requires to calculate the spectral norm of an infinite matrix. Currently, the benchmark-set already provides detailed performance of C and Rust about each case on x86 platforms, including their execution time, memory space, and CPU utilization. Also, every case can get accomplished with different algorithms.

Since previous coders and researchers already evaluated the thorough performances of two languages on x86 platforms, our experiment focuses on presenting the performances' variations after benchmark programs are migrated from x86 platforms to ARM platforms. We assume an algorithm of one case is executed as 100% time on x86 platforms, and we normalize the execution time of this algorithm on the Juno board accordingly. For each benchmark case, we evaluate all algorithms that can get compiled with both languages' ARM compiler. After collecting all algorithms' results for one case, we calculate the average value of the normalized execution time, and we present the final result in Figure 5.4.

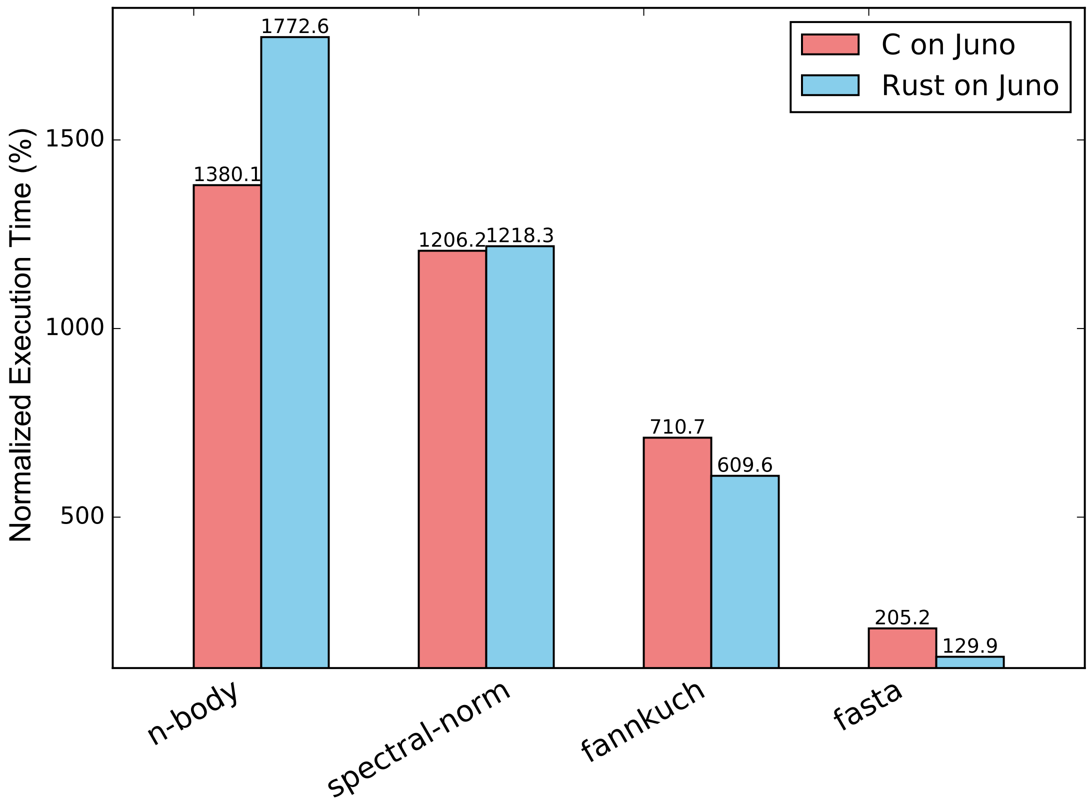


Figure 5.4: C vs. Rust Performance on Juno

According to our experiment, all the benchmark programs run slower on the Juno board than the x86 platform. The numerical difference can be introduced because of the different hardware configuration (i.e., CPU cores and total memory space). Specifically, for the first two cases `n-body` and `spectral-norm`, C language performs relatively better than Rust after normalization, while the other two cases present the contrast observation. Meanwhile, for all evaluated cases, the normalized differences between the two languages are less than 40%.

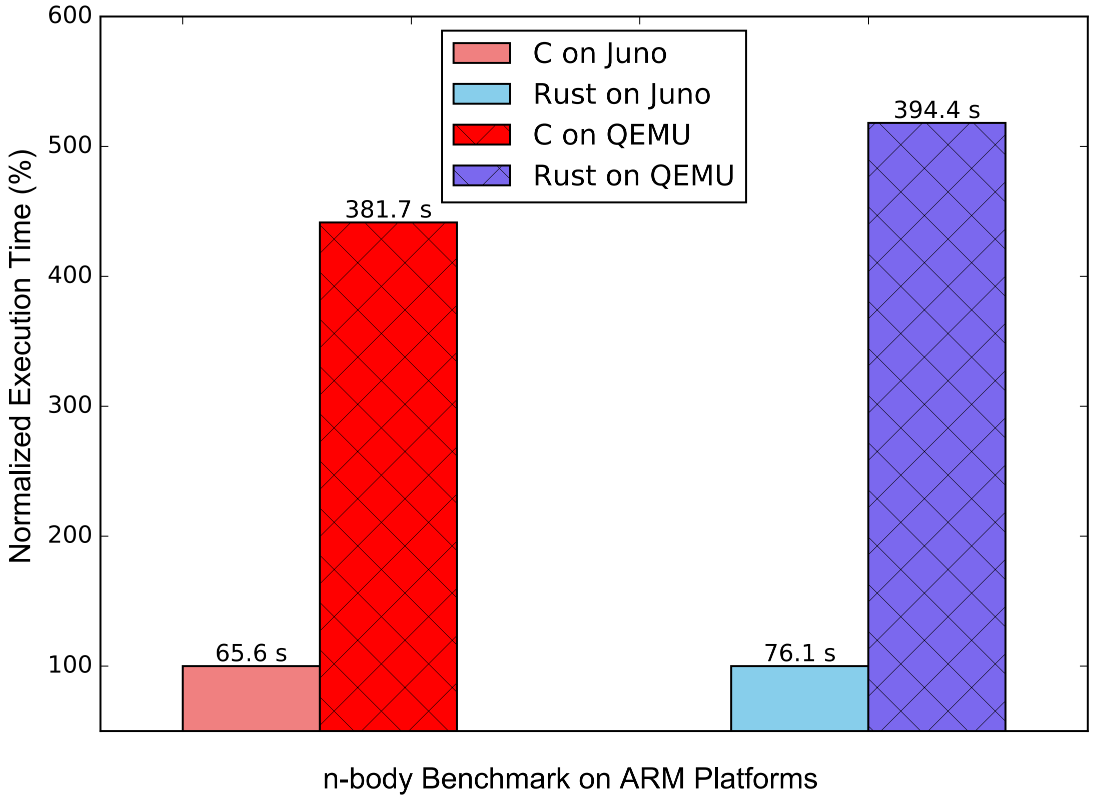


Figure 5.5: QEMU vs. Juno Performance

Different Platforms Evaluation. To validate the performance we evaluated on Juno is representative across different ARM devices, we provide an extra evaluation of the emulation environment QEMU. We re-implement the benchmark `n-body` in two languages on QEMU, and then evaluate the performances as presented in Figure 5.5. We assume the execution time of Juno board's programs are 100%, and then normalize the time of QEMU's programs accordingly. As the experiment shows, comparing to the Juno board, the emulator introduces around 3.5 times extra overhead for both C and Rust languages. Meanwhile, the extra overhead is introduced with a similar ratio for two languages, which means the relative difference between C and Rust stays at the same level on both Juno and QEMU. In conclusion, we claim that the language-wise difference we evaluate in Figure 5.4

is representative of the ARM architecture. Also, the evaluations on either development board or emulation environment present the same pattern of the difference.

5.5.2.2 Enhanced APIs' Overhead

To present the overall overhead of enhancing APIs, we evaluate TAs' performance in five real-world cases to invoke different types of APIs. For each case, we use the same CA to invoke two TAs compiled in C and Rust, respectively. Meanwhile, both C-based and Rust-based TAs are programmed to execute the same task with the same algorithm, while the major difference is that all Rust-based TEE Internal APIs are enhanced by RusTEE. Among the five cases, case `Secure_Storage` provides the functionalities for reading, creating, and deleting the secure-storage objects. We use the time of creating an empty secure-storage object to represent related tasks efficiency; case `Random` generates a 16-bytes random number; case `Hotp` generates ten HMAC-based one-time passwords according to the RFC4226 algorithm [77]; case `Aes` conducts the AES-128 encryption with CTR mode on a 4096-bytes plaintext; case `Acipher` conducts RSA Public-Key Cryptography Standards (PKCS) encryption with the 1024-bits key and the 100-bytes plaintext.

We evaluate each case 10,000 times in total, and we calculate the cases' average execution time with the data set that excludes 10% data outliers (5% largest and 5% smallest data). The comparison of C-based TA and Rust-based TA is presented in Figure 5.6. For each case, we labeled the average execution time above the corresponding TA's bar. As the baseline data, the average context switch time (without conducting any task in TEE) is 676 μs for both C and Rust case, with a negligible variation. We consider the C-based TA's execution time as 100% and then normalizing the Rust-based TA's data accordingly. As the figure presents, for the five evaluated cases, RusTEE only introduces the performance degradation from 0.27% to 3.08%, and four of the five cases are affected with less than 1% overhead.

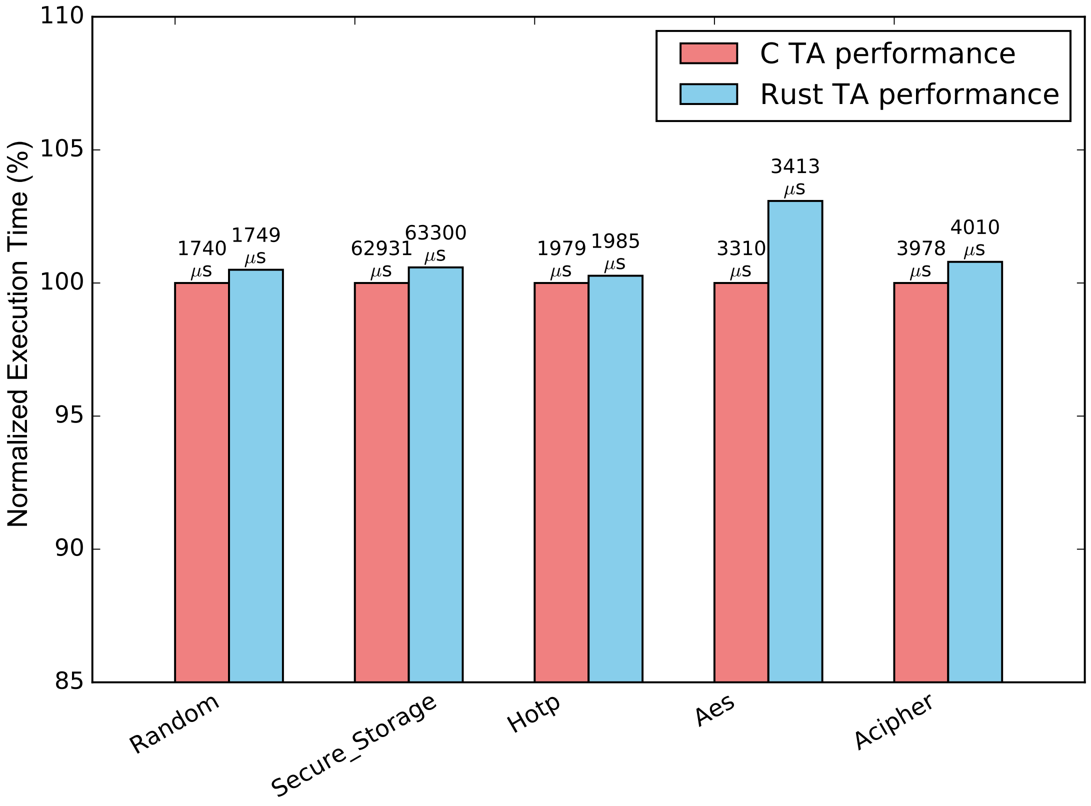


Figure 5.6: Performances of C-based TAs vs. Rust-based TAs

5.6 Work Conclusion

In this work, we presented RusTEE, a Rust-based TrustZone application SDK, which assists developers to compile the TA with the enforced memory-safety features. The TA relies on the language-wise benefit of Rust to mitigate the previously reported implementation issues. Furthermore. RusTEE redesigns the system-services APIs and cross-world communication channel of TA to resolve two architectural issues of TrustZone-assisted TEE systems. We implement RusTEE based on the existing C-based SDK OP-TEE, and evaluate the mechanism on multiple platforms that include both emulators and development boards. According to our evaluation, RusTEE introduces slight performance overhead while significantly increases the application's memory-safety in multiple aspects. Finally, we open-source the entire RusTEE with various examples.

Chapter 6

Related Work

This chapter summarizes the related works of the dissertation. Specifically, we first present the related works of other high-privileged architectures such as TrustZone, SGX, and SMM. Then we present the related works for the three proposed works, which include the related introspection mechanisms, related TrustZone-based network managements, and related memory-safe systems that are developed in Rust.

6.1 High-Privileged Operating Modes

Recently, high privileged operating modes than ring 0 have been widely supported in both x86 and ARM processors to isolate secure sensitive code from rich OS. Moreover, many hardware manufactures provide the hardware-level solutions for creating the hardware-assisted trusted execution environment (TEE) [121]. Based on the ARM TrustZone technology, several works [123, 54, 122, 124] are proposed to investigate and enhance the security of the TrustZone secure world. Meanwhile, TrustZone has been utilized to enhance the security of applications running in the normal world against a malicious rich OS [95, 103, 29]. Santos et al. [95] propose to run the security-sensitive piece of the normal world .NET apps within the secure world. TrustICE [103] provides the solution to allocate the isolated environment for any normal world application, and Cho et al. [29] extend this idea for isolating both normal world application and the hypervisor.

Besides the ARM hardware architecture, Both Intel and AMD support System Management Mode (SMM) in their x86 processors to execute the code with a higher privilege than that running in the Protected mode [52]. SICE [18] introduced the SMM-based isolated environment for x86 multi-core platforms. SICE can provide the remote attestation for the user to verify the integrity of the kernel within its isolated environment. Based on SMM, it is plausible to port our secure asynchronous introspection on X86 multi-core processors. In recent research works, Intel's Software Guard Extensions (SGX) technique has been used for secure communication in x86-based systems [57, 96, 15, 20]; however, the SGX enclave runs as a user-level process instead of a high-privileged mode like TrustZone.

6.2 Introspection Mechanisms

6.2.1 Asynchronous Introspection

Asynchronous introspection mechanisms [91, 85, 50, 38, 37, 68, 86, 55, 120, 59] have been popularly deployed to protect OS kernel integrity. OSck [50] executes a verifier process alongside the target kernel and periodically scans the memory to identify any policy violation. SigGraph [68] proposes to use the graph-based signature to scan the kernel data structure instance and detect the rootkits that are capable of manipulating the data structures. Specialized security tools have been constructed for running on a trusted virtual machine (VM) to detect any security violation on a target VM [38, 42, 91].

Zhang et al. [125] first propose the concept of using an isolated device as the integrity monitor. Then, Copilot [85] utilizes a PCI add-in card to periodically verify the hash checksum of the kernel static data. Later, several system management mode (SMM) based introspection mechanisms have been proposed [120, 17, 119, 59], where HyperCheck [120] and SPECTRE [119] employ the SMM to outsource the snapshot of the kernel to a remote server and conduct the introspection on the server side. HyperSentry [17] performs the kernel measurement locally by periodically triggering the host's SMM via an out-of-band channel. Among SMM-based security mechanisms, multi-core platforms are only briefly mentioned in [17] on freezing all cores during the SMM-based measurement task. The authors of HyperCheck [120] mention that it could be extended on multi-core processors; however, there is no detailed design about it. Several introspection mechanisms are proposed based on other hardware components which can check the kernel transparently [37, 100]. Ether [37] proposed an Intel-VT [51] based kernel analyzer to analyze the software within the virtual machine. LO-PHI [100] transparently examines the kernel memory snapshots without exposing any software-based artifacts by using additional hardware sensors and actuators.

6.2.2 Synchronous Introspection

A number of synchronous introspection mechanisms [84, 97, 112, 43, 16, 60, 76] have been proposed to work on different architectures too. On ARM processors, SPROBES [43] and TZ-RKP [16] are two TrustZone-based synchronous introspection mechanisms proposed recently. SPROBES [43] injects special code into the security-sensitive kernel handlers so it can dynamically check these handlers in the secure world and provide the real-time protection for the normal world. TZ-RKP [16] achieves a similar security goal but focuses on monitoring the data integrity and optimizing the rich OS's performance. Besides utilizing existing hardware-features of the ARM processor, customized hardware has been developed to snoop the memory bus and monitor the security-related writes to the kernel area [60, 73, 76].

6.3 TrustZone-Based Network Connection

ARM TrustZone has been adopted to protect the network services on mobile devices. For instance, TrustZone-based remote attestation enables secure message exchanges between the secure world and the server [74, 66]. A challenge on mobile phones is that there is usually a single NIC that is shared between the normal world and the secure world. One solution is to use the network NIC drive in the normal world to send the encrypted packets created by the secure world, and use the TrustZone to verify and harden the network driver in the normal world. TZ-RKP [16] protects the normal world kernel including the normal world network driver. Li et al. [67] propose building up a trusted path from the normal world network driver to the secure world. A complete secure network driver can be implemented in the secure world [72, 103]. In single-core ARM systems, it must suspend the normal world including its network driver to transmit secure packets. Even on the multi-core platform, such a complete secure-world driver solution requires sophisticated collaboration from the normal world to yield the NIC to the secure world. In TZNIC, both the normal world and the secure world can share the same physical NIC by running two network drivers on different CPU cores with the same driver interface.

Another line of research focuses on the management of the network peripherals instead of utilization. Santos. et al. [94, 35] proposes the idea about utilizing the TrustZone to restrict the peripheral usage via Trust Leases. Brasser. et al. [26] presents the idea to control the normal-world network driver with the secure privilege in the restrict area, for example, a classroom when students are taking tests. SeCloak [61] controls the peripheral's availability by configuring it's security attributes to make sure the peripheral is successfully turned on or off. Unlike these works with the focus on peripherals' ON/OFF regulation, TZNIC's scope include not only the management but also the utilization of these peripherals.

6.3.1 Rust-assisted Systems

In past years, Rust language has become an attractive programming language for developers who have an interest in enhancing application security. As a memory-safe language, Rust's safety has been formally proved in RustBelt [56] in 2017. Meanwhile, lines of works [64, 65, 63, 19, 110] have been proposed to adapt Rust into the development of traditional C/C++ based systems. For example, TockOS [64] presents the idea to write a complete embedded system OS in Rust. Moreover, Rust has been integrated with TEE development [110, 41, 40]. For Intel SGX, Wang et al. [110] propose the open-source project Rust-SGX to deliver the Rust-based SDK for SGX enclave developers, and Fortanix Rust EDP [41] has implemented a similar idea. Regarding the TrustZone technology, RustZone [40] first demonstrates the possibilities to migrate Rust into TrustZone TA development, while lacking a thorough analysis of the security for each component insides TAs. To the best of our knowledge, RusTEE is the first work that presents the complete development kit set for TrustZone TA developers and provides the default features to compile TAs in Rust-safe style.

Chapter 7

Conclusion and Future Research Directions

In this dissertation, we systematically study the security of the latest ARM architecture and associated TrustZone technology. Specifically, we conduct the research on the following three topics.

First, we propose the idea that when the secure and normal worlds are run simultaneously on the multi-core platform, the normal world can present a new-form evasion attack named as TZ-Evader on the secure-world asynchronous introspection. The idea is evaluated with detailed timing data on the ARM development board that equipped with the latest ARMv8 multi-core architecture. Furthermore, we present the corresponding countermeasure SATIN for securely inspecting the rich OS kernel while being strong enough to defend the proposed attack.

Second, we present TZNIC, a TrustZone-assisted network mechanism that allows both the normal and secure world OS to share one physical network peripheral. We propose the design that the secure world can reliably retrieve the on-peripheral registers' values and deduce the normal-world driver's information with these values. TZNIC further extends this idea to fill the semantic gap between two worlds, and allow the secure world to reuse the normal-world network driver without relying upon any cooperation from the rich OS.

Moreover, since TZNIC requires zero modification on the normal-world side, it presents excellent compatibility with the existing TrustZone-based systems.

Finally, we propose RusTEE, a Rust-based TrustZone application SDK, which assists developers to compile the TA with the enforced memory-safety features. The TA relies on the language-wise benefit of Rust to mitigate the previously reported implementation issues. Furthermore. RusTEE redesigns the system-services APIs and cross-world communication channel of TA to resolve two architectural issues of TrustZone-assisted TEE systems. With the enhanced security on both implementation details and architectural features, RusTEE notably enhances the memory-safety of TAs and mitigates many memory-unsafe vulnerabilities reported for the previous TAs.

While the dissertation has explored and evaluated the works on the three specific TrustZone-based security issues, many related topics can be further exploited as listed.

- TZ-Evader presents the capability of a normal world OS to defeat the asynchronous introspection of the secure world with the limitation that the attacker needs to study the context-switch threshold before performing the evasion attack. However, this threshold can vary case-by-case with different hardware configurations. In future work, we plan to investigate more methods for the normal-world OS to probe the secure world execution and raise the race condition accordingly. Meanwhile, a systematic study can be performed for the secure world to analyze if any other service besides the asynchronous introspection is affected by the proposed evasion attack.

- Currently, TZNIC provides the best-effort solution to preserve the full functionalities of the normal-world peripheral's driver and only implement the smallest size of the driver inside the secure world. In this case, the presented TZNIC mechanism suffers the Denial-of-Service attack under the circumstances that the normal world can sacrifice its peripheral availabilities. We consider it is an interesting topic to investigate another design philosophy, which is moving parts of the driver, such as the peripheral initialization and I/O buffer recycling, into the secure world. Such design can provide the secure world with better availability-control on the peripherals since the software interfaces are claimed and managed by the secure world directly. Meanwhile, the normal world will face network-based performance degradation by losing these functions. By evaluating both strategies, we may achieve an ideal balance between the secure world's security and the normal world's performance.

- We believe that RusTEE can bring extra benefits for the mobile manufacturers to quickly review the security of third-party TAs. Before RusTEE, the manufacture can only verify a TA's security via manual inspection, which requires the verifier to understand the complicated logic inside TA. In this case, if the SDK is opened to third-party developers, many human efforts will be introduced to verify the third-party TAs. Moreover, such manual verification only provides the security promise based on personal experience, without any formal proof.After adapting RusTEE into the manufacturer's SDK, the manufacturer will have a straightforward and reliable verification method, which is checking if the TA's source code contains any `unsafe` segment or calls any untrusted crate. We consider providing the automatic verification script for checking third-party TAs in future work.

Bibliography

[1] 96 Boards. HiKey Website, accessed in February 2020. `https://www.96boards.org/product/hikey/`.

[2] Amazon. Best sellers in internal computer networking cards. `https://www.amazon.com/Best-Sellers-Computers-Accessories-Internal-Computer-Networking-Cards/zgbs/pc/13983711`, Accessed in June 2018.

[3] ARM. Corelink gic-400 generic interrupt controller. `https://static.docs.arm.com/ddi0471/a/DDI0471A_gic400_r0p0_trm.pdf`, 2011.

[4] ARM. Principles of arm memory maps white paper. `http://infocenter.arm.com/help/topic/com.arm.doc.den0001c/DEN0001C_principles_of_arm_memory_maps.pdf`, 2012.

[5] ARM. Arm generic interrupt controller architecture version 2.0. `http://docs-api-peg.northeurope.cloudapp.azure.com/assets/ihi0048/b/IHI0048B_b_gic_architecture_specification.pdf`, 2013.

[6] ARM. Arm corelink tzc-400 trustzone address space controller. `https://static.docs.arm.com/100325/0001/arm_corelink_tzc400_trustzone_address_space_controller_trm_100325_0001_02_en.pdf`, 2015.

[7] ARM. Juno arm development platform soc technical reference manual, revision: r1p0. https://www.arm.com/files/pdf/DDI0515D1a_juno_arm_development_platform_soc_trm.pdf, 2015.

[8] ARM. Programmer's guide for armv8-a. http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/DEN0024A_v8_architecture_PG.pdf, 2015.

[9] ARM. Armv8-a memory systems version 1.0. https://static.docs.arm.com/100941/0100/armv8_a_memory_systems_100941_0100_en.pdf, 2016.

[10] ARM. Arm exception table, 2018. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/CHDEEDDC.html.

[11] ARM. Arm trusted firmware, 2018. https://github.com/ARM-software/arm-trusted-firmware.

[12] ARM. Juno Arm Development Platform, 2018. https://developer.arm.com/products/system-design/development-boards/juno-development-board.

[13] ARM. Arm security technology: Building a secure system using trustzone® technology. https://developer.arm.com/ip-products/security-ip/trustzone, accessed in February 2020.

[14] ARM Community. Arm linaro instruction, 2018. https://community.arm.com/dev-platforms/w/docs/303/juno.

[15] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, et al. Scone: Secure linux containers with intel sgx. In *OSDI*, volume 16, pages 689–703, 2016.

[16] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds:

Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 90–102. ACM, 2014.

[17] Ahmed M Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 38–49. ACM, 2010.

[18] Ahmed M Azab, Peng Ning, and Xiaolan Zhang. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 375–388. ACM, 2011.

[19] Abhiram Balasubramanian, Marek S Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. System programming in rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 156–161, 2017.

[20] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.

[21] Ariel Ben-Yehuda. Can mutate in match-arm using a closure. rust issue #27282. `https://github.com/rust-lang/rust/issues/27282`, 2015.

[22] Christophe Biocca. std vec intoiter as_mut_slice borrows &self, returns &mut of contents. rust issue #39465. `https://github.com/rust-lang/rust/issues/39465`, 2017.

[23] Philippe Biondi and the Scapy community. Scapy's documentation. `http://scapy.readthedocs.io/en/latest/index.html`, 2018.

[24] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *NDSS*, 2019.

[25] Ferdinand Brasser, Daeyoung Kim, Christopher Liebchen, Vinod Ganapathy, Liviu Iftode, and Ahmad-Reza Sadeghi. Regulating arm trustzone devices in restricted spaces. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, pages 413–425, 2016.

[26] Ferdinand Brasser, Daeyoung Kim, Christopher Liebchen, Vinod Ganapathy, Liviu Iftode, and Ahmad-Reza Sadeghi. Regulating arm trustzone devices in restricted spaces. In *ACM MobiSys*, 2016.

[27] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA*, pages 18–20, 2020.

[28] Yaohui Chen, Yuping Li, Long Lu, Yueh-Hsun Lin, Hayawardh Vijayakumar, Zhi Wang, and Xinming Ou. Instaguard: Instantly deployable hot-patches for vulnerable system programs on android. In *NDSS'18*, 2018.

[29] Yeongpil Cho, Jun-Bum Shin, Donghyun Kwon, MyungJoo Ham, Yuna Kim, and Yunheung Paek. Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices. In *USENIX Annual Technical Conference*, pages 565–578, 2016.

[30] Cisco. Cisco visual networking index: Forecast and trends, 2017–2022. `https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.pdf`, 2019.

[31] Common Vulnerabilities and Exposures. CVE Search Results for TrustZone, accessed in February 2020. `https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=trustzone`.

[32] Common Vulnerabilities and Exposures. CVE-2015-6639, accessed in May 2020. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6639`.

[33] Common Vulnerabilities and Exposures. CVE-2016-2431, accessed in May 2020. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2431`.

[34] Corbet. How fast should hz be?, 2005. `https://lwn.net/Articles/145973/`.

[35] Miguel B Costa, Nuno O Duarte, Nuno Santos, and Paulo Ferreira. Trubi: A system for dynamically constraining mobile devices within restrictive usage scenarios. In *ACM MobiHoc*, 2017.

[36] CVE Details. Android cve details. `https://www.cvedetails.com/product/19997/Google-Android.html`, 2019.

[37] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62. ACM, 2008.

[38] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 297–312. IEEE, 2011.

[39] Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, Kaustubh Prabhu, Mark Ashley, Aaron Brown, Aeneas Jaissle, Susant Sahani, Bruce Simpson, and Brian Tierney. iperf benchmark. `https://iperf.fr`, 2018.

[40] Eric Evenchick. Rustzone: Writing trusted applications in rust. https://github.com/ericevenchick/rustzone, 2018.

[41] Fortanix. The Fortanix Rust Enclave Development Platform, accessed in February 2020. https://edp.fortanix.com/.

[42] Yangchun Fu and Zhiqiang Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 586–600. IEEE, 2012.

[43] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. Sprobes: Enforcing kernel code integrity on the trustzone architecture. *arXiv preprint arXiv:1410.7747*, 2014.

[44] GlobalPlatform. Tee client api specification v1.0. https://globalplatform.org/specs-library/tee-client-api-specification/, 2010.

[45] GlobalPlatform. Tee internal core api specification v1.2.1. https://globalplatform.org/specs-library/tee-internal-core-api-specification-v1-2/, 2019.

[46] GlobalPlatform. Tee management framework including asn.1 profile v1.0.1. https://globalplatform.org/specs-library/tee-management-framework-including-asn1-profile/, 2019.

[47] Gouy, Isaac. The Computer Language Benchmarks Game, accessed in February 2020. https://benchmarksgame-team.pages.debian.net/benchmarksgame/.

[48] Joffrey Guilbon. Attacking the arm's trustzone. https://blog.quarkslab.com/attacking-the-arms-trustzone.html, 2018.

[49] Stephen Hemminger. Sky2 driver source code. https://elixir.bootlin.com/linux/v4.17-rc4/source/drivers/net/ethernet/marvell/sky2.c, 2005.

[50] Owen S Hofmann, Alan M Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. Ensuring operating system kernel integrity with osck. In *ACM SIGARCH Computer Architecture News*, volume 39-1, pages 279–290. ACM, 2011.

[51] Intel. Intel virtualization technology, 2018. `https://www.intel.com/content/www/us/en/data-center/new-center-of-possibility.html`.

[52] Intel. System management mode (smm). `https://en.wikipedia.org/wiki/System_Management_Mode`, accessed in Oct. 2018.

[53] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E Porter, and Radu Sion. Sok: Introspections on trust and the semantic gap. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 605–620. IEEE, 2014.

[54] Jin Soo Jang, Sunjune Kong, Minsu Kim, Daegyeong Kim, and Brent Byunghoon Kang. Secret: Secure channel between rich execution environment and trusted execution environment. In *NDSS*, 2015.

[55] Xuxian Jiang, Xinyuan Wang, and Dongyan 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*, pages 128–138. ACM, 2007.

[56] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.

[57] Seongmin Kim, Youjung Shin, Jaehyung Ha, Taesoo Kim, and Dongsu Han. A first step towards leveraging commodity trusted execution environments for network applications. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, page 7. ACM, 2015.

[58] M. A. Kinsy, S. Khadka, M. Isakov, and A. Farrukh. Hermes: Secure heterogeneous multicore architecture design. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 14–20, May 2017.

[59] Kevin Leach, Chad Spensky, Westley Weimer, and Fengwei Zhang. Towards transparent introspection. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 248–259. IEEE, 2016.

[60] Hojoon Lee, Hyungon Moon, Ingoo Heo, Daehee Jang, Jinsoo Jang, Kihwan Kim, Yunheung Paek, and Brent Kang. Ki-mon arm: A hardware-assisted event-triggered monitoring platform for mutable kernel object. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[61] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. Secloak: Arm trustzone-based mobile peripheral control. In *MobiSys*. ACM, 2018.

[62] Lev Aronsky. Knoxout-bypassing samsung knox, 2016. `http://media.wix.com/ugd/4e84e6_668d564cc447434a9a8fda3c13a63f6a.pdf`.

[63] Amit Levy, Michael P Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. Ownership is theft: Experiences building an embedded os in rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, pages 21–26, 2015.

[64] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 234–251, 2017.

[65] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. The case for writing a kernel in rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, pages 1–7, 2017.

[66] Wenhao Li, Haibo Li, Haibo Chen, and Yubin Xia. Adattester: Secure online mobile advertisement attestation using trustzone. In *MobiSys*. ACM, 2015.

[67] Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tieyan Li. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*. ACM, 2014.

[68] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Ndss*, 2011.

[69] Linaro. Optee secure os, 2018. `https://github.com/OP-TEE/optee\_os`.

[70] Linaro. OP-TEE Sample Applications, accessed in February 2020. `https://github.com/linaro-swg/optee_examples`.

[71] Linaro. Optee device, accessed in February 2020. `https://optee.readthedocs.io/en/latest/building/index.html`.

[72] Dongtao Liu and Landon P Cox. Veriui: Attested login for mobile devices. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, page 7. ACM, 2014.

[73] Ziyi Liu, JongHyuk Lee, Junyuan Zeng, Yuanfeng Wen, Zhiqiang Lin, and Weidong Shi. *Cpu transparent protection of os kernel and hypervisor integrity with programmable dram.* ACM, 2013.

[74] Samsung Electronics Co. Ltd. Get started with knox attestation. `https://seap.samsung.com/tutorial/get-started-knox-attestation`, 2018.

[75] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[76] Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. Vigilare: toward snoop-based kernel integrity monitor. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 28–37. ACM, 2012.

[77] et al. M'Raihi. Rfc4226: Hotp: An hmac-based one-time password algorithm. `https://tools.ietf.org/html/rfc4226`, accessed in February 2020.

[78] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. Patchdroid: Scalable third-party security patches for android devices. In *ACSAC'13*, 2013.

[79] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[80] Minh-Son Nguyen and Quan Le-Trung. Integration of atheros ath5k device driver in wireless ad-hoc router. In *Advanced Technologies for Communications (ATC), 2013 International Conference on*. IEEE, 2013.

[81] NXP. Applications processor security reference manual for i.mx 6solo-lite. `https://www.nxp.com/webapp/sps/download/mod_download.jsp?colCode=IMX6DQ6SDLSRM`, 2013.

[82] Ozan (oz) Yigit. Hash functions, 2018. `http://www.cse.yorku.ca/~oz/hash.html`.

[83] K. Park, G. I. Ma, J. H. Yi, Y. Cho, S. Cho, and S. Park. Smartphone remote lock and wipe system with integrity checking of sms notification. In *ICCE*, 2011.

[84] Bryan D Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 233–247. IEEE, 2008.

[85] Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, pages 179–194. San Diego, USA, 2004.

[86] Nick L Petroni Jr and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 103–115. ACM, 2007.

[87] Check Point. Cyber attack trends: 2019 mid-year report. `https://www.checkpoint.com/downloads/resources/cyber-attack-trends-mid-year-report-2019.pdf`, 2019.

[88] Project Zero. Lifting the (hyper) visor: Bypassing samsung's real-time kernel protection, 2017. `https://googleprojectzero.blogspot.com/2017/02/lifting-hyper-visor-bypassing-samsungs.html`.

[89] QEMU. QEMU Website, accessed in February 2020. `https://www.qemu.org/`.

[90] Raspberry Pi. Raspberry Pi Serial Products, accessed in February 2020. `https://www.raspberrypi.org/products/`.

[91] Alireza Saberi, Yangchun Fu, and Zhiqiang Lin. Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[92] Samsung. Samsung s9+ specifications. `https://www.samsung.com/us/smartphones/galaxy-s9/specs/`, 2018.

[93] Samsung Electronics Co. Ltd. White paper: An overview of the samsung knox platform. `https://kp-cdn.samsungknox.com/6ee7dbf222f5eabeafea9d15e3986f09.pdf`.

[94] Nuno Santos, Nuno O Duarte, Miguel B Costa, and Paulo Ferreira. A case for enforcing app-specific constraints to mobile devices by using trust leases. In *HotOS*, 2015.

[95] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using arm trustzone to build a trusted language runtime for mobile applications. *ACM SIGARCH Computer Architecture News*, 42(1):67–80, 2014.

[96] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 38–54. IEEE, 2015.

[97] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *ACM SIGOPS Operating Systems Review*, volume 41-6, pages 335–350. ACM, 2007.

[98] Ben Smith, Rick Grehan, Tom Yager, and DC Niemi. Byte-unixbench: A unix benchmark suite, 2011.

[99] Ben Smith, Rick Grehan, Tom Yager, and DC Niemi. Byte-unixbench: A unix benchmark suite, 2011.

[100] Chad Spensky, Hongyi Hu, and Kevin Leach. Lo-phi: Low-observable physical host instrumentation for malware analysis. In *NDSS*, 2016.

[101] Bjarne Stroustrup. Why doesn't c++ provide a "finally" construct? `http://www.stroustrup.com/bs_faq2.html`, accessed in May 2020.

[102] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Sushil Jajodia. Trustdump: Reliable memory acquisition on smartphones. In *In Proc. European Symposium on Research in Computer Security*, 2014.

[103] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*, pages 367–378. IEEE, 2015.

[104] The Rust Programming Language Core Team. Unsafe rust. `https://doc.rust-lang.org/nomicon/README.html`, accessed in February 2020.

[105] Torvalds. Github linux kernel. `https://github.com/torvalds/linux`, 2018.

[106] Trustonic. Not just droning on! the rise of kinibi-m. `https://www.trustonic.com/news/blog/not-just-droning-rise-kinibi-m/`.

[107] Aaron Turon. Abstraction without overhead: Traits in rust. `https://blog.rust-lang.org/2015/05/11/traits.html`, 2015.

[108] Uniwood. Uniwood energy usage monitor. `https://www.amazon.ca/Uniwood-Energy-Monitor-Electricity-Outlet/dp/B078JJBW3P`, accessed in Jan. 2018.

[109] Shengye Wan, Jianhua Sun, Kun Sun, Ning Zhang, and Qi Li. Satin: A secure and trustworthy asynchronous introspection on multi-core arm processors. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 289–301. IEEE, 2019.

[110] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. Towards memory

safe enclave programming with rust-sgx. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2333–2350, 2019.

[111] Jiang Wang, Kun Sun, and Angelos Stavrou. A dependability analysis of hardware-assisted polling integrity checking systems. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.

[112] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 545–554. ACM, 2009.

[113] ARM Developer Website. Accessing memory-mapped peripherals. `https://developer.arm.com/products/software-development-tools/ds-5-development-studio/resources/tutorials/accessing-memory-mapped-peripherals`, 2018.

[114] Wolfram Research, Inc. Hundred-Dollar, Hundred-Digit Challenge Problems, accessed in February 2020. `http://mathworld.wolfram.com/Hundred-DollarHundred-DigitChallengeProblems.html`.

[115] Yoram Wurmser. Us time spent with mobile 2019. `https://www.emarketer.com/content/us-time-spent-with-mobile-2019`, 2019.

[116] Kailiang Ying, Amit Ahlawat, Bilal Alsharifi, Yuexin Jiang, Priyank Thavai, and Wenliang Du. Truz-droid: Integrating trustzone with mobile operating system. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 14–27. ACM, 2018.

[117] Kailiang Ying, Priyank Thavai, and Wenliang Du. Truz-view: Developing trustzone user interface for mobile os using delegation integration model. In *Proceed-*

ings of the Ninth ACM Conference on Data and Application Security and Privacy, pages 1–12. ACM, 2019.

[118] Xingjie Yu, Zhan Wang, Kun Sun, Wen Tao Zhu, Neng Gao, and Jiwu Jing. Remotely wiping sensitive data on stolen smartphones. In *ASIACCS*, 2014.

[119] Fengwei Zhang, Kevin Leach, Kun Sun, and Angelos Stavrou. Spectre: A dependable introspection framework via system management mode. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2013.

[120] Fengwei Zhang, Jiang Wang, Kun Sun, and Angelos Stavrou. Hypercheck: A hardware-assisted integrity monitor. *IEEE Transactions on Dependable and Secure Computing*, 11(4):332–344, 2014.

[121] Fengwei Zhang and Hongwei Zhang. Sok: A study of using hardware-assisted isolated execution environments for security. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, page 3. ACM, 2016.

[122] Ning Zhang, He Sun, Kun Sun, Wenjing Lou, and Y Thomas Hou. Cachekit: Evading memory introspection using cache incoherence. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 337–352. IEEE, 2016.

[123] Ning Zhang, Kun Sun, Wenjing Lou, and Y Thomas Hou. Case: Cache-assisted secure execution on arm processors. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 72–90. IEEE, 2016.

[124] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. Truspy: Cache side-channel information leakage from the secure world on arm devices. *IACR Cryptology ePrint Archive*, 2016:980, 2016.

[125] Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ronald Perez, and Reiner Sailer. Secure coprocessor-based intrusion detection. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 239–242. ACM, 2002.

VITA

Shengye Wan

Shengye Wan is a Ph.D. candidate in the Computer Science Department at the College of William and Mary, under the supervision of Dr. Kun Sun. Before that, he received his Master's degree in the Computer Science Department at the College of William and Mary in 2016 and Bachelor of Engineering degree from Software Engineering Department of Huazhong University of Science and Technology in 2014. His research interests lie in computer and network security with a focus on the trusted execution environment on multi-core mobile devices.