國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master's Thesis

基於 KVM Hypervisor 虛擬化 Arm TrustZone

Virtualizing Arm TrustZone on a KVM-based Hypervisor

林俊諺

Chun-Yen Lin

指導教授: 黎士瑋 博士

Advisor: Shih-Wei Li, Ph.D.

中華民國 113 年 11 月

November, 2024

doi:doi:10.6342/NTU202404660

Acknowledgements

謝謝我的指導教授黎士瑋博士，您在我就讀研究所期間對我的指導，以及對我的研究的建議，幫助我完成這篇論文，同時也感謝實驗室的所有成員，有你們一起在同個研究領域專研，使我更加振奮。

doi:doi:10.6342/NTU202404660

摘要

ARM TrustZone 是一種硬體安全技術，能夠將中央處理器 (CPU) 的執行環境切割成普通世界以及安全世界，使較為機密的操作 (如加密、身份驗證) 能夠在安全世界中執行，從而與較易受到攻擊的普通世界隔離。雖然 TrustZone 已廣泛部署在實體硬體上，但仍不提供虛擬化技術的支援，無法直接地於虛擬機器 (VM) 中使用，因此在現在最多人使用的 Linux KVM Hypervisor 上，執行虛擬機器的使用者無法利用 TrustZone 功能來保護他們的系統。為了解決這項限制，我們擴充了 KVM 以支援 TrustZone 的虛擬化，並將其提供給虛擬機器使用。我們利用攔截並模擬技術，來將敏感指令重新導向至 KVM 並模擬它們。同時我們開發了一項新技術，例外層級多工 (Exception-Level Multiplexing)，這項新穎的技術可安全地讓 TrustZone 軟體在虛擬機器環境中利用現有的 Arm 硬體執行。此外，我們基於目前 QEMU 中的 TrustZone 硬體模板，創建了一個虛擬的安全記憶體區域，並將安全 IO 映射到該區域。我們的虛擬化 TrustZone 支援 OP-TEE，OP-TEE 是一項需要在 Arm TrustZone 上的執行可信執行環境 (TEE)，並允許 OP-TEE 能夠在虛擬機器中利用我們的虛擬 TrustZone 中執行包含可信應用 (TAs)、TEE 核心以及安全監視器等 TrustZone 的軟體。最後，我們測量了效能，我們在使用 KVM 虛擬化的 TrustZone 上執行 OP-TEE 的安全應用，比在 QEMU 虛擬機器上執行的 OP-TEE 有約莫十倍的效能優化。

關鍵字： KVM、TrustZone、虛擬化、Arm

doi:doi:10.6342/NTU202404660

Abstract

Arm TrustZone technology provides two distinct CPU execution environments: the Normal and the Secure World. Arm enforces resource isolation of the two worlds, ensuring that security-critical operations, such as encryption and authentication, can be executed in the Secure world and thus isolated from the potentially compromised Normal world that hosts a comprehensive software environment. Although TrustZone is widely deployed on physical hardware, it is unavailable to virtual machines (VMs). Notably, users who run VMs on the popular Linux KVM hypervisor cannot leverage TrustZone features to secure their systems. We have extended KVM to expose a virtual TrustZone to VMs to address this limitation. We leverage trap-and-emulate to virtualize sensitive TrustZone operations while introducing exception-level multiplexing, a novel technique that safely enables native execution of TrustZone software on the existing Arm hardware in the VM environment. Our implementation builds on the current TrustZone hardware abstraction in QEMU that exposes a virtualized secure memory and IO devices. Our resulting KVM pro-

doi:doi:10.6342/NTU202404660

totype supports OP-TEE, a de-facto open-source TEE implementation for Arm TrustZone, allowing a comprehensive software environment for OP-TEE that encompasses Trusted Application (TAs), TEE kernel, and the security monitor to execute in a virtualized Trust-Zone on a VM. Performance evaluation of the OP-TEE prototyped running on a virtualized TrustZone in the VM on our KVM prototype shows that it outperforms OP-TEE running on a QEMU-hosted VM by 10 times.

Keywords: KVM, TrustZone, Virtualization, Arm

doi:doi:10.6342/NTU202404660

Contents

doi:doi:10.6342/NTU202404660

doi:doi:10.6342/NTU202404660

List of Figures

doi:doi:10.6342/NTU202404660

List of Tables

doi:doi:10.6342/NTU202404660

List of Listings

doi:doi:10.6342/NTU202404660

Chapter 1 Introduction

TrustZone [20] is a security extension for Arm System-On-Chips (SoCs) to enable two execution environments: a Trusted Execution Environment (TEE) to run a secure TEE kernel that manages trusted applications (TAs) and a Rich Execution Environment (REE) to host a commodity OS, such as Linux, and their applications. For instance, systems deployed in practice host security-critical TAs, such as those for encryption and storage services and key management in isolated TEEs. TrustZone imposes hardware-level isolation mechanisms and protects TEEs' resources (e.g., memory and I/O device accesses) from potential interferences from the REE, providing a robust layer of security.

A TrustZone-enabled system integrates a software-based secure monitor to context-switch between the REE and TEE. Arm extends processors to provide a higher-privileged CPU mode, called monitor mode (EL3), to run the secure monitor and isolate it from the TEEs and REEs, which operate on less privileged kernel (EL1) and user (EL0) modes. This hierarchical structure ensures that the secure monitor maintains complete control over world switches while preventing potential compromises from lower privilege levels from affecting the system's security foundation.

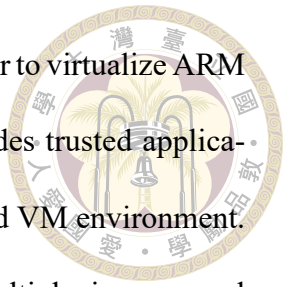
Arm processors have been widely adopted in deployments from mobile and embedded devices to the recent surge of Arm-based personal computers, automobiles, and cloud

doi:doi:10.6342/NTU202404660

servers [5; 28]. To address the growing demands of virtualization in modern computing infrastructure, Arm introduced Virtualization Extensions (VE). Commodity hypervisors, like Linux/KVM [2], have been extended to leverage Arm VE to simplify their design and run unmodified operating systems in virtual machines (VMs) natively to achieve good performance. However, none of the existing Arm-based commodity hypervisors expose TrustZone to VMs. This makes it impossible for VM users to capitalize on TrustZone's features to protect their systems. Moreover, virtualized TrustZone support is crucial in streamlining TEEs' software development and testing workflows by removing the requirement for physical TrustZone-enabled hardware. QEMU offers software-based emulation of TrustZone features. However, its reliance on binary translation prevents TrustZone software from executing natively and introduces substantial performance overhead. Given the growing prevalence of Arm-based hardware, enabling native code execution within virtualized TrustZone environments is imperative for efficient development and testing.

Previous work [24] extended Xen to support virtualized TrustZone. vTZ [24] modifies the secure monitor to enable virtualized TrustZone using physical hardware. However, this approach hinders adoption in real-world scenarios. The primary challenge stems from Arm device vendors' common practice of implementing secure monitors within proprietary firmware, which users are typically prevented from modifying. This vendor lock-in effectively prevents customized vTZ firmware from being deployed. Furthermore, vTZ's applicability with the more widely adopted KVM hypervisor was unexplored, raising questions about its broader compatibility. In addition, vTZ does not support a complete TrustZone stack. It focuses exclusively on supporting TEE kernels and TAs within VMs, preventing users from deploying TrustZone's secure monitors to the virtualized environment.

doi:doi:10.6342/NTU202404660



This work proposes a new approach to extend the KVM hypervisor to virtualize ARM TrustZone, enabling a complete TrustZone software stack that includes trusted applications, a TEE kernel, and a secure monitor to execute in a de-privileged VM environment. To retain performance efficiency, we introduced Exception Level multiplexing, a novel technique that multiplexes software executing in virtual EL3 and EL1 in the TrustZone onto the physical EL1 in the normal world. This allows most of the TrustZone software to execute natively in the VM. For instructions that require hypervisor intervention, such as those that access sensitive EL3 registers that are non-existent in EL1 or trigger world switches, we adopted trap-and-emulate to ensure these sensitive instructions are trapped and handled by KVM. Furthermore, we built on QEMU's existing hardware abstraction for the virtualized TrustZone environment to expose virtualized secure memory and I/O devices, simplifying development efforts.

Our KVM prototype supports the de-facto open-source TEE implementation for Arm, OP-TEE-based [1] environment. We allow the TEE kernel and TAs from OP-TEE and the integrated secure monitor from Arm Trusted Firmware-A (TF-A) [21] in the virtualized TrustZone within a VM. Our implementation preserves TrustZone's safety requirements without relying on physical hardware features. Performance evaluation of the KVM prototype shows the TAs from OP-TEE running in the VM outperform the counterparts running on a QEMU-hosted VM by 10 times.

doi:doi:10.6342/NTU202404660

Chapter 2 Background

2.1 Arm TrustZone

Arm TrustZone [20] is a hardware-based security technology that enables a system to isolate sensitive operations from the rest of the system. Initially aimed at mobile devices, TrustZone has since become integral to embedded systems, IoT devices, and other platforms requiring TEEs. In real-world applications, TrustZone plays a crucial role in Digital Rights Management (DRM) [16], ensuring secure playback of copyrighted content on platforms like Netflix and Disney+ by performing video decryption within the Secure World to prevent piracy. Moreover, TrustZone is integral to mobile payment systems [17], as it isolates sensitive processes such as PIN entry and encryption on mobile devices, ensuring the secure processing of sensitive data.

TrustZone divides the system into two distinct worlds: the secure world, which handles security-sensitive operations such as encryption and key management, and the normal world, where general, non-critical software runs. This separation is enforced by hardware mechanisms, allowing sensitive operations and data to remain protected even if the normal world is compromised. Figure 2.1 illustrates the TrustZone architecture, which shows how the system operates across different exception levels (ELs). In the normal world, user applications run at EL0 while the operating system runs at EL1. In the secure world, sensi-

doi:doi:10.6342/NTU202404660

tive applications run in a similar hierarchy, with trusted applications at EL0 and the secure operating system at EL1.

To manage the CPU state between the secure world and the normal world, TrustZone introduces an additional privilege layer known as EL3. EL3, acting as a higher privilege level, is responsible for handling the Secure Monitor Call (SMC) [13] and thus managing transitions between the secure world and the normal world. EL3 relies on a set of system registers to manage these world transitions and maintain security policies. These registers include the Secure Configuration Register (SCR_EL3), which controls the security state of the CPU, and the Saved Program Status Registers (SPSR_EL3), which are used to store and restore the CPU's state during transitions between exception levels. Together, these registers help the CPU maintain the state during SMC handling and isolation between the secure world and the normal world.

In addition to CPU features, TrustZone extends memory and I/O devices to support security. It divides memory into secure and non-secure areas, ensuring that sensitive data is protected from the normal world. I/O devices are also divided into two worlds. In addition, TrustZone ensures that the devices assigned to the secure world can't be accessed by the normal world.

2.2 OP-TEE

OP-TEE [1] is a widely used, open-source Trusted Execution Environment (TEE) built on TrustZone architecture. One of the key advantages of OP-TEE is that it allows developers to implement TAs within the secure world. TAs are usually used to handle sensitive operations and are protected from the normal world. The main part of OP-TEE

doi:doi:10.6342/NTU202404660

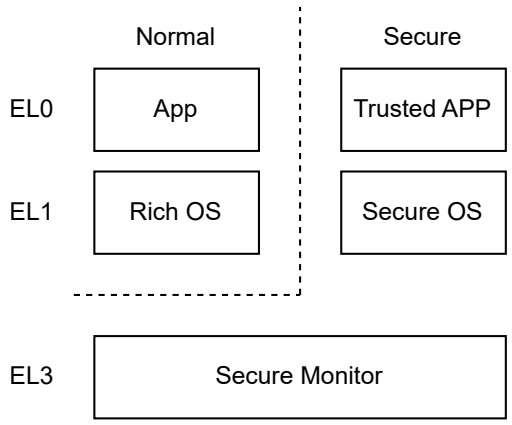


Figure 2.1: Arm TrustZone Architecture

is the OP-TEE kernel, which operates in the EL1 of the secure world and is responsible for managing the execution of TAs. OP-TEE kernel provides multiple APIs that follow the GlobalPlatform standards to TAs and communicate with normal world OS to perform various operations. Since the OP-TEE kernel is designed to be lightweight, most of its operations rely on the normal world OS support, including most system calls, process scheduling, and file access.

2.2.1 Arm Trusted Firmware

In OP-TEE architecture, Arm Trusted Firmware [21] is responsible for executing at the highest privilege level, EL3, during the boot process and managing the transitions between worlds using the Secure Monitor. It initializes the system, sets up the security configuration, and loads OP-TEE OS into the secure EL1. The firmware ensures that the hardware is properly configured to enforce the isolation between the secure and normal worlds before handing over control to the operating systems running in both environments.

After Arm Trusted Firmware loads OP-TEE OS into the secure EL1, it continues

doi:doi:10.6342/NTU202404660

to reside at EL3 as a secure monitor of OP-TEE, managing future transitions between the secure and the normal world by handling SMC. As shown in Figure 2.2, when an OS in the normal EL1, typically Linux under OP-TEE architecture, needs to execute a TA, it generates an SMC to trap to EL3. Arm Trusted Firmware then handles the SMC and switches the CPU context to the secure world. Once the switch is complete, Arm Trusted Firmware executes an Exception Return (ERET) [7] to return to secure EL1 and hand over the control to OP-TEE OS. The OP-TEE OS then invokes the appropriate TA. Once the operation is complete, the result is passed back to the normal world through the same SMC and ERET mechanism, ensuring that the separation between the secure and the normal world is maintained.

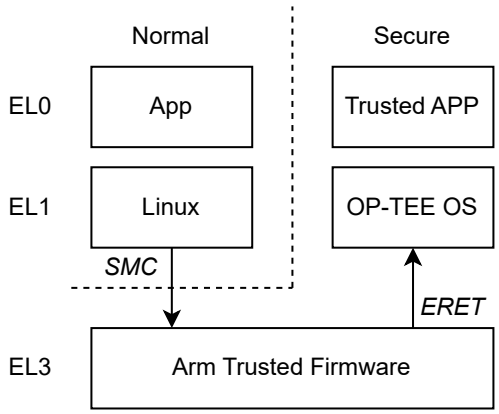


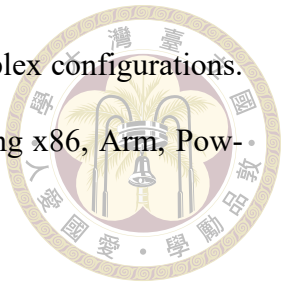
Figure 2.2: OP-TEE World Switch Procedure

2.3 QEMU

QEMU[3] is a generic and open-source machine emulator and virtualizer used to run software and operating systems across different hardware platforms. Its ability to emulate various CPU architectures and full system environments makes it a valuable tool in development, testing, and virtualization scenarios. Moreover, QEMU can emulate peripheral

doi:doi:10.6342/NTU202404660

devices, allowing developers to emulate entire systems and test complex configurations. QEMU's versatility supports various hardware architectures, including x86, Arm, PowerPC, and RISC-V.



2.3.1 TCG

At the core of QEMU's emulation capabilities is the Tiny Code Generator (TCG) [4]. TCG can dynamically translate instructions from one CPU architecture (the guest) into instructions that can be executed by the host CPU. TCG's flexibility lies in its ability to support a wide range of hardware architectures without relying on specific hardware features, making it ideal for cross-platform development. Notably, TCG supports the emulation of TrustZone. This makes TCG useful for research and development involving security features. However, despite its flexibility, TCG has a significant drawback: its execution speed. Since the binary translation is done in userspace and happens dynamically during runtime, TCG is substantially slower than hardware-assisted virtualization. This performance limitation makes TCG impractical for scenarios where high-speed execution is required.

2.3.2 KVM

KVM [2] is a Linux kernel module that enables hardware-assisted virtualization. In situations requiring high-performance execution, TCG's slow speed is inadequate, and KVM becomes the preferred solution. Compared to TCG, KVM does not dynamically translate instructions but directly executes the guest instruction on the host CPU. In QEMU/KVM architecture, QEMU manages device emulation and virtual machine configu-

doi:doi:10.6342/NTU202404660

ration, while KVM handles the execution of guest code natively, achieving near-native performance. Moreover, KVM allows guest systems to execute in kernel space by leveraging CPU virtualization extensions like Intel VT-x [26] and AMD-V [6], providing direct access to the CPU and memory management functions. This enables the guest VMs to perform tasks much faster and with more efficient resource use compared to TCG's userspace emulation.

On Arm platforms, KVM leverages Arm Virtualization Extensions (VE) to achieve native performance of VMs. Arm VE introduces an additional privilege level, EL2, specifically for hypervisors. KVM operates at EL2, while the host OS and guest OS operate at EL1, allowing KVM to manage the isolation between the host and guest VMs efficiently. However, a limitation arises when it comes to supporting the virtualization of TrustZone on KVM. TrustZone relies on EL3 to perform transitions between the normal world and the secure world. Since KVM operates with guests confined to EL1, TrustZone's EL3 is inaccessible to guest VMs, which is a critical challenge in virtualizing TrustZone on KVM.

doi:doi:10.6342/NTU202404660

Chapter 3 Design

3.1 Overview

The increasing demands of isolated environments on embedded and mobile devices have been highlighted in recent years. Arm TrustZone offers a solution by dividing the CPU into the secure world and the normal world, thus allowing users to create an isolated TEE. However, TrustZone's hardware limitations, such as its lack of full virtualization support, restrict its scalability and flexibility on testing and developing TAs in virtualized environments that utilize KVM's performance benefits. Thus, this work focuses on extending KVM to support virtualized TrustZone in VMs. We introduce a new design approach that achieves the set of goals below.

Enable full TrustZone software stack Our KVM extensions aim to allow VMs to run the entire TrustZone software stack, including the secure monitor, TEE kernel, TAs, and the REE kernel and its applications. Previous work [22; 24] focuses on virtualizing TrustZone environments to run the TEE and REE but does not support executing the secure monitor.

This limitation prevents VMs from utilizing essential secure monitor features like secure boot. Virtualizing the secure monitor introduces unique challenges, particularly

doi:doi:10.6342/NTU202404660

since Arm VE does not virtualize EL3 mode. We address the challenges and execute the secure monitor in the VM on KVM.

Retain TrustZone's security guarantees Our design must isolate the execution context of (1) the secure world from the normal world and (2) the virtualized EL3 from EL0/EL1. In addition, we must ensure that the resources of the secure world are inaccessible to the normal world while permitting the secure world to access the normal world's resources.

Achieve portability and compatibility with hypervisor features We aim to enhance deployment, allowing users of increasingly prevalent Arm devices to create VMs with virtualized TrustZone capabilities. Further, our design aims to expose the virtualized TrustZone to standard (unprotected) VMs and confidential VMs (protected by pKVM) on KVM, given confidential computing has become increasingly essential to secure VM deployments. This enables users of these VMs to leverage TrustZone to enhance the safety of their systems. Finally, our design aims to reuse existing virtualization support from the existing KVM software stack (e.g., QEMU) to simplify maintenance and development efforts.

3.2 CPU virtualization

This section introduced how we virtualize TrustZone's CPU features, including how we create a virtual EL3 environment for guest CPU and detect and handle those sensitive instruction.

doi:doi:10.6342/NTU202404660

3.2.1 Virtual System Registers

The first challenge of CPU virtualization is that guest VMs can only execute in EL1 under KVM architecture, which means there is no EL3 environment for the secure monitor of TrustZone. To address this issue, we need to create a virtual EL3 environment for the secure monitor. This includes creating a set of virtual EL3 system registers to emulate the whole EL3 environment. Here, it is introduced how we virtualize EL3 system registers and how they work. An important observation is that most of the EL3 system registers, including SPSR [14; 15], VBAR [18; 19], SCTLR [11; 12], and so on, are also banked at EL1. According to the observation, we propose a novel technique, *EL multiplexing*, to multiplex virtual EL3 and virtual EL1 system registers onto physical EL1 system registers. By doing that, we can create a virtual EL3 and a virtual EL1 environment based on the physical EL1 environment. Figure 3.1 shows the key concept of EL multiplexing. What we actually do is to replace all accesses of EL3 registers with their EL1 equivalent registers in guest VM. For example, if the guest VM attempts to execute the instruction msr x0, spsr_el3 to read the spsr_el3 to x0, we will change it to msr x0, spsr_el1. It is important to note that even when the CPU in EL3 try to modify the EL1 registers, it won't cause any impact on the actual hardware state, since hardware state of a CPU in EL3 can only be controlled by EL3 registers. This ensures that multiplexing EL1 and EL3 registers does not cause any conflicts, as the CPU either operates in a virtual EL3 environment or within a virtual EL1 environment, but not both at the same time.

Another challenge is how we maintain the virtual system registers' value when the exception level changes. In order to address this issue, we let KVM intercept all the SMC and ERET instructions which will change the exception level. Figure 3.2 shows how we

doi:doi:10.6342/NTU202404660

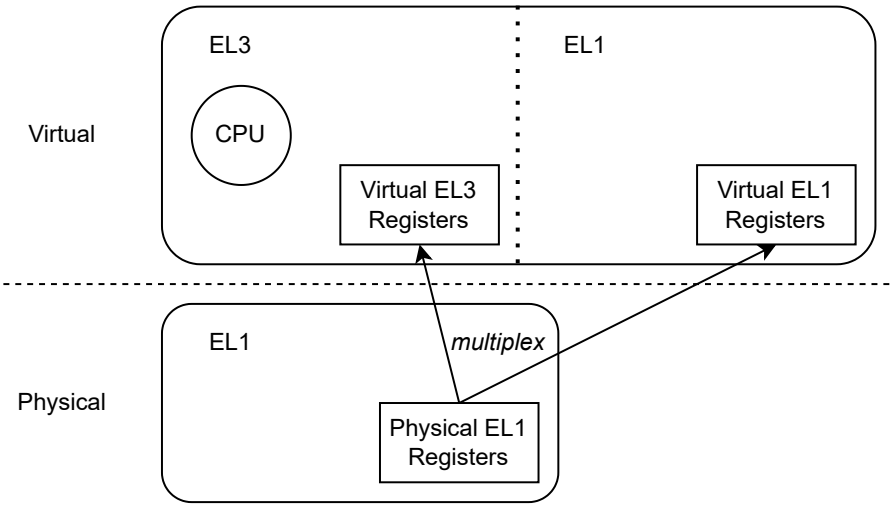


Figure 3.1: Design of Virtual System Register by multiplexing EL1 of Arm64

maintain the virtual registers' state at KVM when exception level changes. For example, if the CPU makes an ERET to return to EL1 from EL3, KVM will intercept the instruction, save the virtual EL3 registers, and restore the virtual EL1 registers onto the physical EL1. This is done to switch the CPU to virtual EL1 from virtual EL3 and ensure the system state is correctly saved and restored. The following section introduces how we make KVM to intercept SMC and ERET.

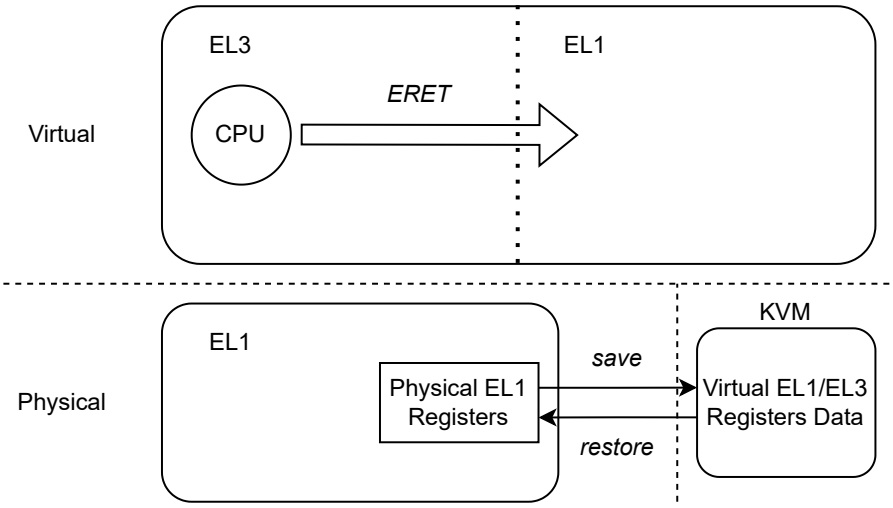


Figure 3.2: Maintain Virtual System Register during ERET

In addition to the registers that exist at both EL1 and EL3, there are a few registers

doi:doi:10.6342/NTU202404660

that only exist at EL3. These registers, such as SCR_EL3 (Secure Configuration Register) [10] and MDCR_EL3 (Monitor Debug Configuration Register) [9], play a crucial role in managing the security state and debug configurations of the system. Currently, because these registers have no direct equivalent registers in EL1, our design cannot multiplex it on the hardware. Instead, accesses to these EL3-exclusive registers are replaced with Hypervisor Call (HVC) [8], and KVM can keep tracking the state of these registers. This means the guest VM can also update and check the state of these registers in KVM. However, the state of the registers will not affect the actual hardware.

3.2.2 Sensitive Instructions Handling

Although we have emulated virtual EL3 system registers, there are still some EL3-specific instructions we need to handle. We consider those instructions as sensitive instructions since they can't be directly executed in EL1 and should be handled carefully. For example, ERET is considered a sensitive instruction in our approach, since it should be executed in EL3 and may cause undefined behavior if we execute it in EL1. Although the SMC can be executed in EL1, it needs EL3 system registers to save and restore the CPU context, we still consider it as a sensitive instruction. Halt instruction (HLT), often used for semihosting calls in Arm64, is also considered as a sensitive instruction since it may cause the CPU to stop executing.

To manage those sensitive instructions, the most common strategy is to trap-and-emulate them at KVM. However, not all sensitive instructions would be trapped to KVM; for example, ERET and HLT may not be trapped. If we directly execute them on the CPU, ERET may cause undefined behaviors, and HLT will just stop the CPU. So, we adopted a para-virtualization method to replace those sensitive instructions in the guest

doi:doi:10.6342/NTU202404660

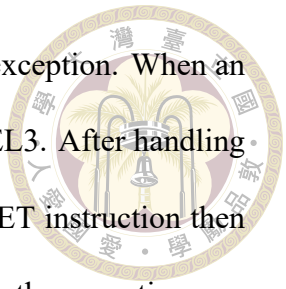
VM with HVCs. By doing so, we can force those sensitive instructions to be trapped to EL1, allowing KVM to handle them.

Next, we introduce how we emulate SMC, ERET, and HLT in KVM.

SMC emulation SMC is an instruction used in ARM architecture to trigger a switch between the normal world and the secure world. When a secure function needs to be executed, the system issues an SMC instruction, which traps the CPU to EL3 and redirects execution to the secure monitor. The secure monitor then handles the switch between the normal world and the secure world.

To emulate SMC, we must first replace all SMC instructions in the guest VM with HVC instructions. Next, we can simulate the behavior of SMC in KVM with the help of virtual EL3 system registers using the trap-and-emulate approach. When the HVC instruction is trapped to KVM, we first save the current program counter to the virtual ELR_EL3. This ensures that we can return to the correct address when the SMC handling is complete. Next, we save the current process state to the virtual SPSR_EL3. This includes saving important information like the condition flags and processor mode to correctly restore the system state after the exception handling. After that, we set the PC to the virtual VBAR_EL3 with the corresponding offset. Which will redirect execution to the EL3 exception vector table, allowing the system to handle the SMC as though it were operating in EL3. And finally, as mentioned in the previous section, we need to perform a context switch to save and restore virtual system registers. In SMC, KVM saves the current virtual EL1 registers and then restores the virtual EL3 registers onto the hardware EL1.

doi:doi:10.6342/NTU202404660



ERET emulation The ERET instruction is used to return from an exception. When an SMC is executed, the processor saves the current state and jumps to EL3. After handling the exception, the CPU makes an ERET to go back to EL1. The ERET instruction then restores the saved state and returns the execution to the address where the exception was triggered.

The handling of ERET in our implementation follows a similar approach to that of SMC, which just simply restores the state saved by SMC. First, we restore the program counter from the virtual ELR_EL3, then restore the process state from the virtual SPSR_EL3. They were both saved by KVM during the SMC emulation. Finally, we perform another context switch, but this time we save the current virtual EL3 registers and restore the virtual EL1 registers onto the hardware EL1.

HLT emulation HLT instruction is used to halt the CPU. When executed, it causes the processor to stop, waiting for an external event, such as an interrupt or a reset, to resume execution. In arm64 architecture, the HLT instruction with a specific immediate value (0xF000) is used to trigger a semihosting call.

However, HLT will not be trapped under QEMU/KVM environment. To emulate the HLT of a semihosting call in our virtual TrustZone environment, the same as SMC and ERET, we need to replace HLT with HVC first. Futhermore, we can emulate the behavior of a semihosting call in KVM.

doi:doi:10.6342/NTU202404660

3.2.3 Memory Virtualization

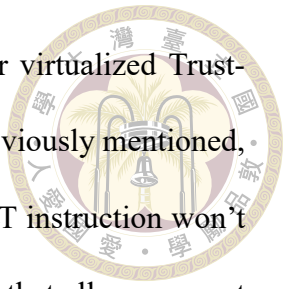
In TrustZone, secure memory is a critical component that ensures sensitive data is isolated and protected from potentially malicious access by software running in the normal world. Secure memory is a designated region of physical memory in a specific address that is only accessible to applications and the operating system of the secure world. This separation is enforced at the hardware level by TrustZone, which enables a system to establish secure and non-secure memory regions, each restricted to its respective domain.

Since certain TrustZone software components, such as the secure monitor, can only execute in secure memory, we need to virtualize secure memory to ensure these components function correctly in our virtualized TrustZone. In the guest VM of QEMU/KVM, QEMU is responsible for managing the virtual memory region and mapping guest address space to host memory. So, we can create a memory region at QEMU and map it to the address of the secure memory to virtualize TrustZone's secure memory. In this way, TrustZone's memory is successfully virtualized and can be utilized by the software in the secure world.

3.3 I/O Virtualization

In the context of virtualized environments, I/O virtualization is a critical component that enables multiple VMs to share physical I/O devices efficiently. In QEMU/KVM architecture, I/O virtualization is performed by QEMU through virtual devices that emulate physical hardware, allowing guest VMs to request I/O operations without direct interaction with the underlying hardware.

doi:doi:10.6342/NTU202404660



However, I/O virtualization of QEMU/KVM is insufficient for virtualized TrustZone. For example, semihosting call is not supported by KVM, as previously mentioned, because semihosting call is triggered by HLT instruction, and the HLT instruction won't be trapped to KVM. Semihosting call is an important I/O mechanism that allows a guest VM to perform I/O operations on the host's devices. This mechanism is especially useful for embedded systems development, where direct access to I/O may not be available. Semihoting call is also critical in the virtualization of TrustZone, since OP-TEE uses semihosting calls to load bootloader images and secure kernel to memory. Due to the design of sensitive instruction handling in section 3.2.2, we can trap HLT instruction to KVM and thus emulate semihosting call in KVM. Although KVM doesn't have a semihosting call handler, fortunately, QEMU's TCG has support for semihosting calls, which means we can simply leverage it to handle the semihosting call. Once the semihosting call is trapped to KVM, we return the control from KVM back to QEMU to allow QEMU to take over and handle the semihosting request. In QEMU, we leveraged the existing ARM TCG's semihosting handler. This means we didn't modify the semihosting handler; we directly utilized the TCG's semihosting mechanism to handle semihosting calls. When semihosting call is complete in QEMU's TCG, QEMU will execute another `kvm_vcpu_ioctl` to hand over the control to KVM, and KVM then resumes the execution of the guest VM. By virtualizing semihosting calls, guest VMs in our virtualized TrustZone environment can utilize semihosting calls to request I/O operations to the host's devices.

doi:doi:10.6342/NTU202404660

Chapter 4 Implementation

In this chapter, we introduce the implementation of our approach, which includes the technical detail of how we adapted KVM and QEMU to support our virtualization model. In addition, we will introduce the para-virtualization part, detailing the modifications we made to OP-TEE to integrate it into our virtualized TrustZone environment.

4.1 KVM

In KVM, we implement three functions to handle the emulation of sensitive instructions and virtual system registers. When HVC is trapped to KVM, KVM executes the corresponding handler function based on the immediate value of HVC, specifying which sensitive instruction needs to be emulated.

4.1.1 Virtual System Registers

Next, it is introduced how we implement virtual system registers and how we maintain them at KVM. As shown in Listing 4.1, we define all virtual EL3 system register at `vcpu_sysreg`. This will give each of them a `sysreg` number, and each `sysreg` number is mapped to a descriptor of system register, which maintains register data, including value, name, access function, and so on. In addition, we can access those system registers

doi:doi:10.6342/NTU202404660

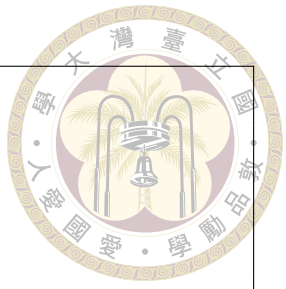
through `vcpu_write_sys_reg` function and `vcpu_read_sys_reg` function with the corresponding `sysreg` number. We don't need to define virtual EL1 registers since physical EL1 registers have already been defined in `vcpu_sysreg`. Thus we can simply leverage it to support virtual EL1 registers.

```
enum vcpu_sysreg {
    // Pre-defined physical EL1 system register
    SCTLR_EL1,
    VBAR_EL1,
    ELR_EL1,
    ...
    ...
    // Our virtual EL3 system register
    SCTLR_EL3,
    VBAR_EL3,
    ELR_EL3,
    ...
    ...
}
```

Listing 4.1: Virtual EL3 Register Declaration

Listing 4.2 shows how we context switch system registers between virtual EL1 and virtual EL3 environment. When the exception level changes between EL1 and EL3, for example, a SMC is triggered, KVM will call `kvm_el13_ctx_switch` to switch the value between physical EL1 system registers and virtual EL3 system registers. In `kvm_el13_ctx_switch`, KVM traverses all the virtual EL3 system registers that have EL1 equivalent registers, and swaps the value between the EL3 registers and the EL1 registers.

doi:doi:10.6342/NTU202404660



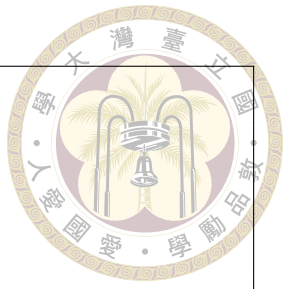
```
void kvm_el13_ctx_switch(struct kvm_vcpu *vcpu)
{
    int i;
    for(i = 0; i < ARRAY_SIZE(el13_regs); i++) {
        switch_virtual_sysreg(vcpu, el13_regs[i]);
    }
}
```

Listing 4.2: kvm_el13_ctx_switch

4.1.2 kvm_el1_smc

This section introduces the implementation of `kvm_el1_smc`, which is the handler function of SMC emulation. Listing 4.3 shows the code of this function. In kvm_el1_smc, we first read the process state and the current program counter from the CPU. Then, we read virtual VBAR_EL3, which points to the exception vector table of EL3, from the virtual system registers' data saved at the CPU. After reading out those values, we can set virtual SPSR_EL3 to process state and set virtual ELR_EL3 to program counter, which will help KVM to restore the execution when the exception has been handled and needs to go back to EL1. Following this, we need to configure the process state and program counter for EL3. We set the process state to EL3_INITIAL_PSTATE according to the documentation of ARM64 SMC. We set the program counter to VBAR_EL3 with the offset of the synchronous exception handler to redirect the execution to the synchronous exception handler of EL3. Finally, we can perform a context switch to put virtual EL3 system registers on hardware and save virtual EL1 system registers by invoking the kvm_el13_ctx_switch function.

doi:doi:10.6342/NTU202404660



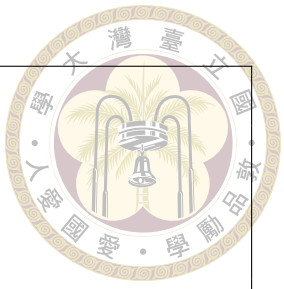
```
void kvm_el1_smc(struct kvm_vcpu *vcpu)
{
    u64 el1_pstate = vcpu_gp_regs(vcpu)->pstate;
    u64 el1_exit_pc = vcpu_gp_regs(vcpu)->pc;
    u64 vbar_el3 = vcpu_read_sys_reg(vcpu, VBAR_EL3);

    vcpu_write_sys_reg(vcpu, el1_pstate, SPSR_EL3);
    vcpu_write_sys_reg(vcpu, el1_exit_pc, ELR_EL3);

    vcpu_gp_regs(vcpu)->pstate = EL3_INITIAL_PSTATE;
    *vcpu_pc(vcpu) = vbar_el3 + sync_exception_handler_offset;
    kvm_el13_ctx_switch(vcpu);
}
```

Listing 4.3: kvm_el1_smc

4.1.3 kvm_el3_eret

This section introduces another handler function, `kvm_el3_eret`, which is the handler function of ERET emulation. Listing 4.4 shows the code of this function. In kvm_el3_eret, we just reverse what we have done in kvm_el1_smc. We first do a context switch to update the virtual EL3 system registers from physical EL1 system registers and restore virtual EL1 system registers on physical EL1 system registers. Then, we can simply restore the process state and the program counter from virtual SPSR_EL3 and virtual ELR_EL3. By doing so, the CPU can resume the execution of what it was doing before the SMC.

doi:doi:10.6342/NTU202404660



```
void kvm_el3_eret(struct kvm_vcpu *vcpu)
{
    kvm_el13_ctx_switch(vcpu);
    u64 spsr_el3 = vcpu_read_sys_reg(vcpu, SPSR_EL3);
    u64 elr_el3 = vcpu_read_sys_reg(vcpu, ELR_EL3);

    vcpu_gp_regs(vcpu)->pstate = spsr_el3
    *vcpu_pc(vcpu) = elr_el3;
}
```

Listing 4.4: kvm_el3_eret

4.2 QEMU

To support our virtualized TrustZone environment, we also made several modifications to QEMU, particularly for emulating the secure memory of TrustZone and handling IO operation such as semihosting call.

4.2.1 Virtual Secure Memory

We defined a virtual secure RAM region in QEMU to emulate the secure memory used by TrustZone. We allocated and initialized a secure RAM region with a specified size and base address according to TrustZone's spec, then integrated it into the system's main memory. Furthermore, we added a device tree node to represent this secure RAM, specifying properties such as its address, size, device type, and secure status. We also created a tagged memory region associated with the virtual secure RAM. Within the virtual secure RAM region, we mapped secure peripherals, such as secure Universal Asynchronous Receiver/Transmitter (UART) and secure General-Purpose Input/Output (GPIO). By con-

doi:doi:10.6342/NTU202404660

figuring secure UART, we are able to set up the console of the secure world of OP-TEE.

4.2.2 Semihosting Call Handling

As mentioned in Section 3.3.3, we handled semihosting call by leveraging QEMU's TCG mechanism. The implementation in QEMU is quite simple, when KVM return to QEMU with an exit reason telling it's a semihosting call, we will directly invoke TCG's semihosting handler to process it. After that, QEMU returns the control to KVM and KVM resumes the execution of guest VM.

4.3 OP-TEE

This section introduces the details of para-virtualization in our approach. Based on our design of sensitive instruction handling in section 3.3, we need to modify the source code of OP-TEE and guest kernel in the implementation of our approach, to support our virtual TrustZone environment. All we have to do is (1) replace MSR/MRS with EL3 registers with it's EL1 equivalents, (2) replace sensitive instructions with HVC.

4.3.1 Arm Trusted Firmware

Most modification we made to OP-TEE is about the ARM Trusted Firmware, which is the firmware of OP-TEE. Since it is the only software component of OP-TEE that will be executed in EL3, we need to replace all it's instructions which will access EL3 registers with its EL1 equivalents. Besides that, we also need to replace sensitive instructions, including SMC, ERET and HLT, with HVC.

doi:doi:10.6342/NTU202404660

4.3.2 OP-TEE OS

Since OP-TEE OS is executed in EL1, it won't access EL3 registers or execute a ERET, the only sensitive instruction it can execute is SMC. Meaning that we don't need to do many modifications to OP-TEE OS, other than replacing SMC with HVC.

After all the modifications made to OP-TEE, we successfully set up an OP-TEE VM on our virtualized TrustZone with QEMU/KVM.

doi:doi:10.6342/NTU202404660

Chapter 5 Evaluation

The evaluations were performed on the AVA Developer Platform [25] with Arm Neoverse N1 CPU, 32 cores and 32 GB memory. The Linux version of our modified KVM is 5.15. The OP-TEE version is 4.0.0. To integrate OP-TEE with our virtualized TrustZone, we added 483 lines of code (LoC) to OP-TEE and 457 LoC to KVM.

Since the main objective of this paper is about performance optimization, we evaluate the implementation by measuring the execution time of several common trusted applications. These applications represent typical use cases in a TEE, making them suitable for assessing the effectiveness of the performance enhancements. By comparing their execution times with an OP-TEE VM hosted by QEMU/TCG, we can quantify the speedup achieved by our KVM approach. Table 5.1 lists the applications we run. Figure 5.1 shows the execution time speedup of our KVM approach compared to QEMU/TCG on different applications. Our virtualized TrustZone on KVM is about ten times faster than QEMU/TCG in all applications.

The significant performance improvements of KVM over QEMU/TCG are primarily due to the following two factors: (1) hardware-assisted virtualization enabled by Arm VE, which allows guest code to execute natively on the host CPU, eliminating the overhead of binary translation; (2) the efficient handling of sensitive TrustZone instructions, such

doi:doi:10.6342/NTU202404660

as SMC and ERET, through a trap-and-emulate mechanism that minimizes latency while preserving security guarantees. These combined optimizations result in high-performance virtualized TrustZone environments.

Table 5.1: Trusted Applications

Application	**Description**
acipher	Generates an RSA key pair and encrypts a supplied string
aes	Runs an AES encryption and decryption
hello_world	A simple Trusted Application to answer a hello command
hotp	Generates a HMAC based One Time Password
random	Generates a random UUID
secure_storage	Reads/writes raw data into the OP-TEE secure storage
plugins	Interacts with Linux syslog service as a plugin

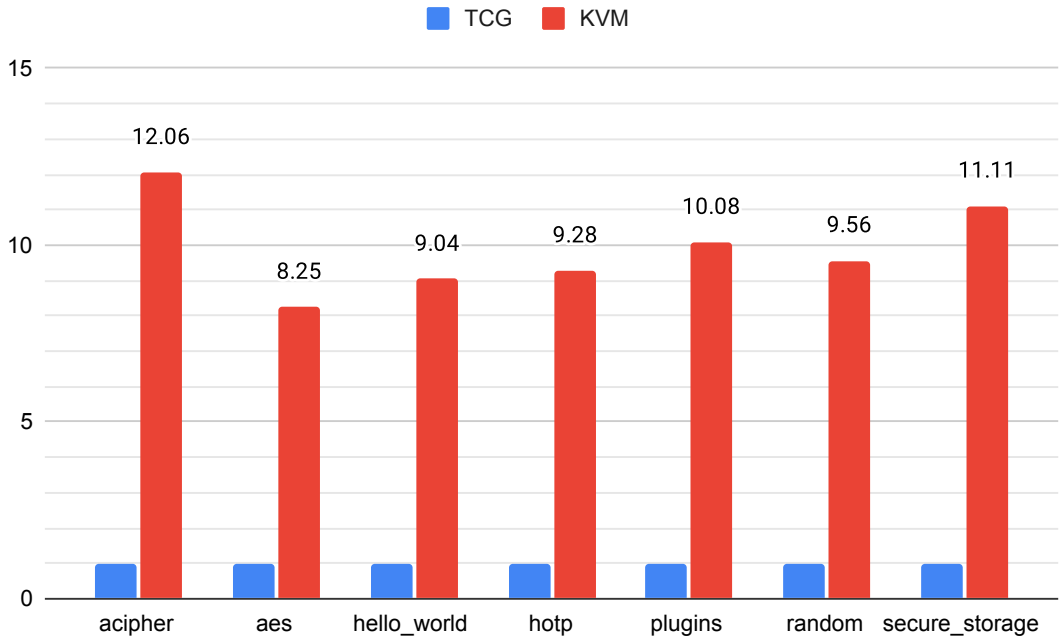


Figure 5.1: Results of KVM and TCG

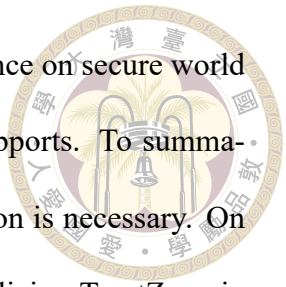
doi:doi:10.6342/NTU202404660

Chapter 6 Related Work

In recent years, the virtualization of ARM TrustZone has been an active area of research, with multiple approaches aiming to address the inherent limitations of TrustZone's original design.

vTZ [24], as another framework for virtualizing ARM TrustZone, allowing multiple VMs to each have their own secure world, or TEE. In vTZ, each guest VM is assigned a guest TEE, leveraging a small hypervisor in EL2 to interpose memory mappings and manage world-switching operations across guest TEEs. This architecture achieves isolation between TEEs through a set of components also running in EL2 called Constrained Isolated Execution Environment (CIEE), which are trusted, self-contained code modules used to emulate the functionalities of TrustZone. Additionally, vTZ leverages physical TrustZone to create several secure modules in the secure world to ensure memory safety and control flow integrity for guest TEEs. To perform a secure world switch, vTZ uses the same approach as we do, adopting the trap-and-emulate method to handle SMC calls and ensure isolation between the two worlds during the switch. Moreover, vTZ requires users to install custom software, including a modified secure monitor, on the physical TrustZone environment. This requirement presents significant practical challenges. Arm device vendors typically implement secure monitors within proprietary firmware that users cannot modify, creating a vendor lock-in that prevents the deployment of vTZ-aware firmware. In

doi:doi:10.6342/NTU202404660



contrast to vTZ, our approach requires no firmware updates and reliance on secure world software. It can be easily deployed to Arm platforms that KVM supports. To summarize, vTZ excels in scenarios where strong hardware-enforced isolation is necessary. On the other hand, our work provides a more flexible solution for virtualizing TrustZone in software, mainly when TrustZone hardware is not available.

TEEv [27] aims to address the challenge of simultaneously hosting different TrustZone-based TEEs by virtualization. TEEv introduces a hypervisor, TEE-visor, to manage various secure kernels from different vendors that coexist on the same device, allowing users to choose a secure kernel that is most suitable for the TA they want to execute. Furthermore, TEEv enforces strict isolation between different vTEEs, as well as between vTEEs and the normal world OS, by carefully configuring memory mapping. This means that a compromised vTEE won't affect the security of other vTEEs or the normal world, preventing inter-TEE vulnerabilities and further enhancing overall system security. Although the goal of TEEv seems similar to our work's, the difference is that TEEv virtualizes those TrustZone-based TEEs (software level); more specifically, TEEv virtualizes secure EL1 to host multiple secure kernels. In contrast, our work virtualizes the entire TrustZone hardware stack (hardware level), including the secure world, EL3 environment, and the world switch mechanism.

TEEVseL4 [22] introduces a microkernel-based approach to virtualized TrustZone-based TEE on seL4 architecture, pursuing minimal Trusted Computing Base (TCB) and leveraging formal verification properties of seL4. By integrating with seL4's virtualization capabilities, TEEVseL4 reuses seL4's security solutions across multiple virtualized Linux guests, offering TrustZone-compatible virtualization that balances scalability and isolation. Both TEEVseL4 and our work don't depend on physical TrustZone to provide

doi:doi:10.6342/NTU202404660

virtualization of TrustZone, and the only difference is that TEEVseL4 is built on seL4 architecture while our work is built on KVM. However, TEEVseL4 has less flexibility in developing firmware since it doesn't virtualize the EL3 environment as our work.

Another significant approach is MyTEE [23], which aims to build a TEE with basic TrustZone CPU features. MyTEE was explicitly designed for machines where key TrustZone hardware, such as the TrustZone Address Space Controller (TZASC) and TrustZone Protection Controller (TZPC), is not available. To achieve as strong memory isolation as TZASC provides, MyTEE manages stage-2 page tables to isolate address space between software components and prevents malicious Direct Memory Access (DMA) attacks through a DMA filter that verifies and emulates MMIO transactions. To establish secure I/O without TZPC, MyTEE introduces a novel technique, temporal privilege escalation, to temporarily escalate the privilege and enable access to specific secure memory regions, such as I/O buffers and memory-mapped registers, as required by trusted applications. This approach prevents directly exposing device drivers to the TEE and dynamically controls I/O channels by filtering DMA traffic and protecting I/O buffer integrity. To summarize, MyTEE offers a robust solution for machines that only have basic TrustZone CPU features to create a TEE. In contrast, our work provides a more fundamental solution, virtualizing the TrustZone execution environment by leveraging KVM for machines without TrustZone-enabled CPU to utilize TrustZone CPU features. However, our work currently is not able to virtualize the secure I/O of TrustZone like MyTEE.

In summary, those works provide valuable approaches for enhancing TEE flexibility and isolation by virtualization. However, they focus on virtualizing the TEE layer, which means that they cannot emulate the full TrustZone CPU environment, particularly the EL3, which is required for running secure firmware. In contrast, our approach virtualizes the

doi:doi:10.6342/NTU202404660

entire TrustZone environment, which enables developers to deploy and manage different secure firmware. This virtualization capability allows greater flexibility in developing and testing firmware, making our solution more adaptable.

doi:doi:10.6342/NTU202404660

Chapter 7 Limitation and Future Work

7.1 Limitation

The biggest limitation of our approach is the lack of robust security guarantees for guest VMs. Our current implementation, while providing a functional virtualized TrustZone, does not offer strong security protections. More specifically, our approach does not guarantee memory isolation between guest VMs. This means sensitive data or operation in guest VMs may be attacked by either the host or other VMs on the same machine if the machine does not have advanced memory protection mechanisms. As the virtualized environment grows to support confidential computing, the limitations of our approach become more pronounced, leaving guest VMs at risk.

7.2 Future Work

To address these limitations, one of our primary goals for future work is to extend our approach to KVM-based confidential VM. This means we want to leverage existing confidential VMs like pKVM or Arm Confidential Compute Architecture (Arm CCA) to

doi:doi:10.6342/NTU202404660

provide strong memory isolation and protection to our approach. These technologies can ensure sensitive data and operation in guest VMs is protected from other system components, including other guest VMs and the hypervisor itself.

In particular, pKVM introduces a software-based mechanism to partition memory and enforce strict memory isolation between the guest VM and the host through page table management. On the other hand, Arm CCA introduces an additional isolated environment reinforced by hardware, the realm world. Unlike the traditional TrustZone architecture with secure and normal worlds, the realm world isolates workloads from both system software and other applications running on the same hardware, reducing the attack surface. By integrating these technologies, we aim to enhance our system's security model, providing robust protections for guest VMs without sacrificing performance.

Additionally, current QEMU hardware abstraction only supports a few TrustZone based devices, so we plan to extend QEMU to virtualize more TrustZone extensions. For example, we currently aim to virtualize the TZASC. The TZASC can define memory regions that can be accessed by the secure world while preventing unauthorized access from the normal world. By virtualizing the TZASC in QEMU, we can better emulate secure memory access and improve the isolation between the secure world and the normal world.

doi:doi:10.6342/NTU202404660

Chapter 8 Conclusions

In this thesis, we presented a novel approach to virtualizing Arm TrustZone on a KVM-based hypervisor, enabling guest VMs to utilize TrustZone's security features without relying on TrustZone-enabled hardware. By virtualizing TrustZone's EL3 and secure world components, we demonstrated the capability of exposing a virtual TrustZone environment to guest VMs on QEMU/KVM, significantly enhancing performance compared to traditional QEMU/TCG solutions.

One of the key technical contributions of our work is the design of virtual system registers by EL multiplexing to support the virtualized TrustZone environment. By multiplexing the virtual EL3 and the virtual EL1 on the physical EL1, we successfully emulated the critical CPU features necessary for the TrustZone functionality. Our approach ensures that sensitive instructions, such as SMC and ERET, are correctly handled through a trap-and-emulate mechanism, providing a fully virtualized TrustZone environment to guest VMs without relying on physical TrustZone hardware.

Through performance evaluation on OP-TEE VMs, we demonstrated that our approach achieves up to 10 times the performance improvement compared to QEMU/TCG. This performance gain highlights the potential of virtualizing TrustZone's CPU features with the efficiency of KVM-based virtualization.

doi:doi:10.6342/NTU202404660

In conclusion, our system enables VMs to run a full TrustZone software stack, including the Secure Monitor, TEE kernel, and TAs. This provides a foundation for the future of virtualized secure environments, offering a scalable and efficient solution to leverage TrustZone in virtualized infrastructures, with significant potential for further security enhancements.

doi:doi:10.6342/NTU202404660

References

[1] About op-tee. URL: https://optee.readthedocs.io/en/latest/general/about.html.

[2] KVM. URL: https://www.linux-kvm.org/page/Main_Page.

[3] QEMU. URL: https://www.qemu.org/docs/master/.

[4] Tiny code generator. URL: https://www.qemu.org/docs/master/devel/index-tcg.html.

[5] Introducing Amazon EC2 A1 Instances Powered By New Arm-based AWS Graviton Processors, Nov. 2018. https://aws.amazon.com/about-aws/whats-new/2018/11/introducing-amazon-ec2-a1-instances.

[6] I. Advanced Micro Devices. AMD virtualization (amd-v) technology. URL: https://www.amd.com/en/technologies/virtualization.

[7] Arm. ERET (exception return). URL: https://developer.arm.com/documentation/ddi0602/2024-09/Base-Instructions/ERET--Exception-return-.

[8] Arm. HVC (hypervisor call). URL: https://developer.

doi:doi:10.6342/NTU202404660

arm.com/documentation/ddi0602/2024-09/Base-Instructions/HVC--Hypervisor-call-.

[9] Arm. MDCR_EL3, monitor debug configuration register (el3). URL: https://developer.arm.com/documentation/ddi0595/2021-03/AArch64-Registers/MDCR-EL3--Monitor-Debug-Configuration-Register--EL3-.

[10] Arm. SCR_EL3, secure configuration register. URL: https://developer.arm.com/documentation/ddi0595/2021-03/AArch64-Registers/SCR-EL3--Secure-Configuration-Register.

[11] Arm. SCTLR_EL1, system control register (el1). URL: https://developer.arm.com/documentation/ddi0595/2021-03/AArch64-Registers/SCTLR-EL1--System-Control-Register--EL1-.

[12] Arm. SCTLR_EL3, system control register (el3). URL: https://developer.arm.com/documentation/ddi0595/2021-03/AArch64-Registers/SCTLR-EL3--System-Control-Register--EL3-.

[13] Arm. SMC (secure monitor call). URL: https://developer.arm.com/documentation/ddi0602/2024-09/Base-Instructions/SMC--Secure-monitor-call-.

[14] Arm. SPSR_EL1, saved program status register (el1). URL: https://developer.arm.com/documentation/ddi0595/2021-03/AArch64-Registers/SPSR-EL1--Saved-Program-Status-Register--EL1-.

[15] Arm. SPSR_EL3, saved program status register (el3). URL: https://developer.

doi:doi:10.6342/NTU202404660

arm.com/documentation/ddi0595/2021-03/AArch64-Registers/SPSR-EL3--Saved-Program-Status-Register--EL3-.

[16] Arm. Trustzone's example use cases, content management. URL: https://developer.arm.com/documentation/PRD29-GENC-009492/c/TrustZone-System-Design/Example-use-cases/Content-management.

[17] Arm. Trustzone's example use cases, mobile payment. URL: https://developer.arm.com/documentation/PRD29-GENC-009492/c/TrustZone-System-Design/Example-use-cases/Mobile-Payment.

[18] Arm. VBAR_EL1, vector base address register (el1). URL: https://developer.arm.com/documentation/ddi0595/2021-03/AArch64-Registers/VBAR-EL1--Vector-Base-Address-Register--EL1-.

[19] Arm. VBAR_EL3, vector base address register (el3). URL: https://developer.arm.com/documentation/ddi0595/2021-03/AArch64-Registers/VBAR-EL3--Vector-Base-Address-Register--EL3-.

[20] Arm. Trustzone for armv8-a, 2020. URL: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Learn%20the%20Architecture/TrustZone%20for%20Armv8-A.pdf?revision=c3134c8e-f1d0-42ff-869e-0e6a6bab824f.

[21] Arm. Arm trusted firmware, 2022. URL: https://www.trustedfirmware.org/projects/tf-a.

[22] B. Blazevic, M. Peter, M. Hamad, and S. Steinhorst. Teevsel4: Trusted execution environment for virtualized sel4-based systems. In 2023 IEEE 29th International

doi:doi:10.6342/NTU202404660

Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pages 67–76, 2023. doi:10.1109/RTCSA58653.2023.00017.

[23] S.-K. Han and J. Jang. Mytee: Own the trusted execution environment on embedded devices. In NDSS, 2023.

[24] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vTZ: Virtualizing ARM TrustZone. In 26th USENIX Security Symposium (USENIX Security 17), pages 541–556, Vancouver, BC, Aug. 2017. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/hua.

[25] A. T. Inc. Ava developer platform, 2021. URL: https://www.ipi.wiki/pages/com-hpc-altra.

[26] Intel. Intel virtualization technology (vt-x) architecture. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/virtualization/virtualization-technology-vt-x.html.

[27] W. Li, Y. Xia, L. Lu, H. Chen, and B. Zang. Teev: virtualizing trusted execution environments on mobile platforms. VEE 2019, page 2 – 16, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3313808.3313810.

[28] C. Williams. Microsoft: Can't wait for ARM to power MOST of our cloud data centers! Take that, Intel! Ha! Ha! The Register, Mar. 2017. https://www.theregister.co.uk/2017/03/09/microsoft_arm_server_followup.

doi:doi:10.6342/NTU202404660