Emulating Virtualized ARM TrustZone on a Xen Hypervisor in an x86 Environment

YONG LI, LINGLIN WU, RUI LI*, University of Massachusetts, Amherst, USA

Abstract–The increasing adoption of virtualization technologies in embedded systems highlights the importance of secure environments to protect against privileged adversaries. ARM TrustZone, a hardware-based Trusted Execution Environment (TEE), provides a foundation for secure execution of sensitive operations. However, the integration of TrustZone with virtualization platforms remains a challenge. This work leverages QEMU and Xen to establish a robust development environment for emulating ARM TrustZone. QEMU facilitates debugging, testing, and verification of TrustZone features by providing an emulated ARM platform, enabling developers to work independently of physical hardware. Xen, as a hypervisor, extends virtualization capabilities, allowing the execution of multiple isolated operating systems while maintaining support for secure TrustZone operations. Together, QEMU and Xen provide a flexible and secure framework for exploring and enhancing TrustZone's potential in virtualized systems. This study demonstrates the setup, integration, and evaluation of TrustZone-enabled virtualized environments, offering insights into their performance and practical applications.

ACM Reference Format:
Yong Li, Linglin Wu, Rui Li. 2024. Emulating Virtualized ARM TrustZone on a Xen Hypervisor in an x86 Environment. 1, 1 (December 2024), 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

1.1 Literature Review

The integration of Trusted Execution Environments (TEEs) with virtualization platforms has gained significant attention in recent years. ARM TrustZone, a hardware-based TEE, provides a secure environment for sensitive operations, yet its potential integration into virtualized systems remains underexplored. This section reviews key contributions in this domain and highlights how our solution stands out.

Li et al. proposed TEEv, a virtualization framework that supports multiple isolated TEE instances (vTEEs) running concurrently on mobile platforms [Li et al. 2019]. By leveraging a hypervisor (TEEvisor), their architecture enables the isolation of TEE instances, reducing security risks associated with shared TEEs. However, TEEv primarily focuses on multi-TEE environments and does not address the functional integration of ARM TrustZone with hypervisors for system-level virtualization.[Pinto et al. 2014] [Winter et al. 2012]

Harrison et al. introduced PARTEMU, a modular framework for emulating TrustZone software [Harrison et al. 2020a]. Built on QEMU and PANDA, PARTEMU enables dynamic analysis of real-world TrustZone operating systems and applications. While effective for vulnerability identification, PARTEMU lacks support for the integration of TrustZone with hypervisors like Xen, which is critical for managing virtualized systems.

Yang and Lee demonstrated the use of OP-TEE with QEMU to implement TEE Client APIs for communication between the Normal World and Secure World [Yang and Lee 2021]. Their work offers a foundation for security research but remains limited to single-system setups and does not explore virtualization capabilities.[Mantu et al. 2023][Yang and Lee 2021][Cicero et al. 2018]

Our approach builds on these efforts by integrating ARM TrustZone into a Xen-based virtualized environment. Unlike TEEv, which emphasizes multi-TEE scenarios, our work focuses on leveraging TrustZone for secure services in a single TEE architecture. Additionally, we utilize QEMU to emulate ARM platforms, facilitating debugging, testing, and performance benchmarking. This combination of QEMU and Xen provides a comprehensive framework for exploring and enhancing TrustZone's potential in virtualized embedded systems.

1.2 Motivation

The rise of embedded systems and their integration into critical infrastructure has necessitated enhanced security measures. Hypervisors, while powerful, are vulnerable to attacks, which can compromise all systems running on the hardware. ARM TrustZone provides a solution by creating a Trusted Execution Environment (TEE) that is isolated from the rest of the system, offering protection against privileged adversaries. This project aims to explore the potential of running Xen hypervisor with support for OP-TEE (TrustZone Software) on a QEMU-emulated ARM environment. By doing so, we can extend the benefits of TrustZone to virtualized environments, enhancing the security of embedded systems.

1.3 Objective

This project aims to establish a robust and flexible development environment for exploring the integration of ARM TrustZone with virtualization technologies. Specifically, the objectives are as follows:

- Implement a Xen hypervisor with support for OP-TEE (TrustZone Software) within a QEMU-emulated ARM environment, enabling the emulation of TrustZone functionality without the need for physical hardware.
- Facilitate research on the performance and security implications of combining TrustZone with virtualization, focusing on the potential benefits and trade-offs in a virtualized system.
- Conduct benchmarking using tools like *xtest* to evaluate the performance enhancements and security guarantees provided

*Both authors contributed equally to this research.

Author's Contact Information: Yong Li, Linglin Wu, Rui Li, {yonli,linglinwu,ruili}@umass.edu, University of Massachusetts, Amherst, Amherst, MA, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM XXXX-XXXX/2024/12-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

by TrustZone in single and multi-guest operating system scenarios.
- Develop a deeper understanding of the operational mechanisms of virtualized systems with TrustZone integration, providing insights for future research and practical applications in embedded systems.

This work not only addresses the limitations of current hypervisor-TrustZone integration but also lays the foundation for advancing the security and functionality of virtualized embedded systems.

2 Background

2.1 ARM TrustZone Technology

ARM TrustZone is a hardware-based security technology integrated into many ARM processors, commonly used in mobile devices and embedded systems. It establishes a separation between two execution environments: the Secure World and the Normal World [Amacher and Schiavoni 2019; Harrison et al. 2020b; Yang and Lee 2021].

The Secure World provides a trusted execution environment (TEE) for performing sensitive operations, ensuring protection from potential threats in the Normal World. Key applications of the Secure World include:

- Storing cryptographic keys securely.
- Processing financial transactions and payments.
- Running secure boot mechanisms to validate system integrity.

[Amacher and Schiavoni 2019; Harrison et al. 2020b; Li et al. 2019].

This dual-world architecture enables TrustZone to enhance the security of embedded and mobile systems by isolating critical operations from untrusted components.

2.2 Comparison of Hypervisor Architectures

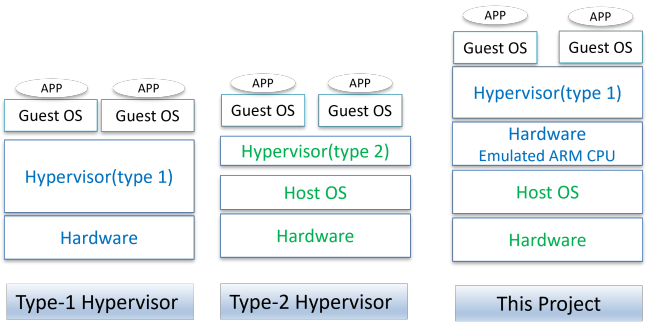

Fig. 1. Type1, Type2 and this Project used Hypervisor

Figure 1 illustrates the hypervisor architectures of Type-1 and Type-2 hypervisors compared to the structure used in this project. The hypervisor setup in this project differs significantly from conventional Type-1 and Type-2 hypervisors in both complexity and flexibility.

Type-1 Hypervisor: A Type-1 hypervisor, also known as a bare-metal hypervisor, runs directly on the underlying hardware. Both the hypervisor and its guest operating systems share the same architecture as the hardware (e.g., x86 or ARM). While this design offers high performance and direct hardware access, it depends entirely on the hardware platform.

Type-2 Hypervisor: A Type-2 hypervisor operates on top of a host operating system, relying on the host OS to manage hardware resources. Similar to Type-1 hypervisors, both the hypervisor and guest OSes must share the same architecture as the underlying hardware. This structure is easier to set up but adds overhead and remains tightly coupled to the hardware platform.

This Project's Hypervisor Architecture: In contrast, the hypervisor structure implemented in this project provides greater flexibility by decoupling the hypervisor and guest operating systems from the underlying hardware architecture. Using QEMU, the ARM CPU is emulated, enabling the Xen Type-1 hypervisor and guest OSes (ARM architecture) to run on an x86-based host machine. This hardware-independent design eliminates the architectural constraints imposed by conventional Type-1 and Type-2 hypervisors, allowing cross-platform testing and development.

Advantages of This Project's Architecture: The added complexity of this setup is justified by the increased flexibility it provides:

- **Cross-Platform Compatibility:** By emulating the ARM CPU, this architecture allows the hypervisor and guest OSes to run on any hardware, regardless of the underlying architecture.
- **Hardware Independence:** Unlike conventional hypervisors, the virtualization stack does not depend on the hardware's native architecture, enabling secure and flexible experimentation in environments where physical ARM hardware is unavailable.
- **Enhanced Testing and Development:** This architecture is ideal for developing and testing secure applications that leverage TrustZone, as it provides a controlled and reproducible virtualized environment.

This project demonstrates that by combining QEMU, Xen, and OP-TEE, it is possible to create a highly versatile and secure virtualization stack that surpasses the constraints of traditional hypervisor architectures.

2.3 QEMU: A System Emulator

QEMU is a versatile system emulator widely used in research and development to emulate various architectures, including x86, ARM, MIPS, RISC-V, and more. It serves as a critical tool for enabling cross-platform development and testing without requiring physical hardware [Bellard 2005a; Harrison et al. 2020b; Liebergeld 2009]. [Bellard 2005b]

In the context of this project, QEMU plays a pivotal role by providing an ARM-emulated environment that supports the integration of ARM TrustZone with the Xen hypervisor. Its ability to emulate full systems, including CPUs, devices, kernels, and applications, makes it ideal for evaluating Trusted Execution Environments (TEEs) like OP-TEE. QEMU's system emulation mode allows the emulation of a complete computer system, including peripherals, enabling the

virtual hosting of several virtual machines on a single physical host [Liebergeld 2009].

QEMU operates in multiple modes:

- **User-mode emulation:** This mode allows QEMU to execute individual Linux or Darwin/macOS programs compiled for different instruction sets. It translates system calls for endianness and architecture compatibility, making it a useful tool for cross-compilation and debugging.
- **System emulation:** In this mode, QEMU emulates an entire computer system, including its peripherals. This capability supports the execution of various guest operating systems, such as Linux, Windows, Solaris, and BSD, across architectures like ARMv7, ARMv8, x86, and more [Bellard 2005a].
- **Hypervisor support:** QEMU can act as a Virtual Machine Manager (VMM) or as a device emulation back-end for virtual machines running under a hypervisor. When paired with a Kernel-based Virtual Machine (KVM) or hypervisors like Xen, it achieves near-native execution speeds while maintaining extensive emulation capabilities [Harrison et al. 2020b].

One of QEMU's core features is its use of Tiny Code Generator (TCG) to translate guest CPU instructions into equivalent host assembly instructions. This dynamic translation capability enables flexible and efficient execution of virtualized workloads, making QEMU an indispensable tool for developing and benchmarking systems that incorporate ARM TrustZone.

Through its robust emulation features and support for a variety of architectures and virtualization setups, QEMU provides the foundation for this project's exploration of secure and virtualized environments with ARM TrustZone.

3 System Design: Introduction to the Virtualization Stack

This section outlines the design and implementation of a virtualized environment to enable TrustZone-based applications on QEMU. we will explain how to enable a TrustZone-based application to run in a virtualized environment on QEMU in an X86 CPU. The process involves incrementally layering virtualization technologies, starting with the basic ARM CPU with TrustZone, then introducing the hypervisor to virtulize the TEE rich OS, and finally virtualizing the ARM CPU with QEMU. Below is a high-level overview of the stack:

- TrustZone Application (via OP-TEE)
- Linux Guest OS
- Hypervisor (Xen)
- QEMU (emulates ARM CPU)
- AnyCPU (host OS)

We will explore each step in detail, showing how the system is built up layer by layer.

3.1 Step 1: Bare Metal Machine. OS directly running on ARM CPU

The first step is to show the basic setup where Linux OS runs directly on an ARM CPU that supports TrustZone. TrustZone provides a hardware-based isolation mechanism, dividing the CPU into two "worlds": the secure world (TrustZone) and the non-secure world (Linux). Here, Linux directly interacts with the ARM CPU, running in the non-secure world while TrustZone applications run in the secure world. This setup represents the simplest form of running Linux with TrustZone support.

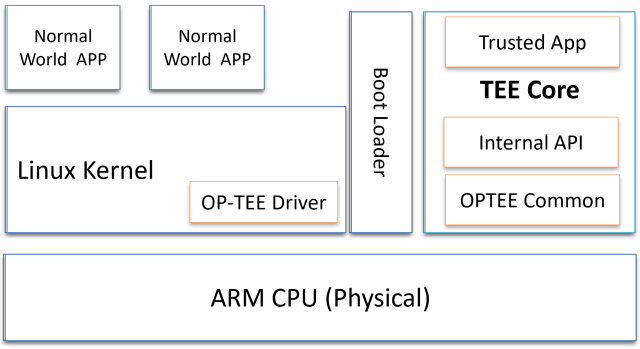

Fig. 2. ARM Trusted App, no virtualization

3.2 Step 2: Virtualize the OS by introducing Hypervisor

The next step involves introducing a hypervisor (in this case, Xen) to the system. The hypervisor provides a layer of abstraction, allowing multiple operating systems to run concurrently on the same hardware. In this scenario, Xen is installed on the ARM CPU, and Linux is virtualized into a guest operating system. The hypervisor ensures that the guest OSes are isolated from one another and the underlying hardware. This setup enables the secure TrustZone application to run alongside the virtualized Linux OS.

In this step, we introduce the concept of a hypervisor, which is critical to enabling the virtualization of the system. The hypervisor, in this case, Xen, runs directly on the ARM CPU and acts as a layer between the hardware and the operating systems (OSes) that need to run on top of it.

In our configuration, we utilize Xen as a Type 1 hypervisor. It runs directly on the ARM hardware, which allows it to manage the system's resources and allocate them efficiently to virtual machines (VMs). The Xen hypervisor introduces the concept of domains:

- **Domain 0 (Dom0):** This is the privileged domain that has direct access to hardware resources and manages other domains. It typically runs a Linux-based OS, which is responsible for managing virtual devices and the overall virtualization environment.
- **Domain U (DomU):** These are the unprivileged domains or virtual machines that run on top of Dom0. Each DomU runs its own guest operating system, which in our case is a Linux OS. DomU does not have direct access to hardware, and its interactions with the hardware must go through Dom0 or the hypervisor.

Xen ensures that these domains are isolated from one another, preventing one compromised domain from affecting the others. This isolation is a critical feature of virtualization as it allows multiple guest OSes to run safely on the same hardware, without interference.

3.3 Step 3: Virtualizing the ARM CPU by QEMU

The final step in the virtualization stack leverages QEMU to emulate the ARM CPU on a non-ARM machine, such as an x86-based host.

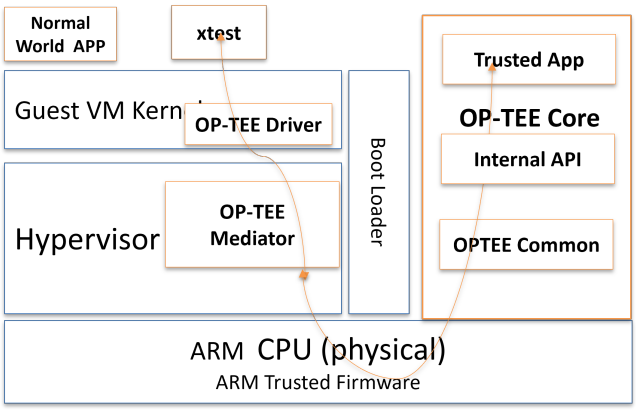

Fig. 3. Linux OS is virtualized by Xen hypervisor

QEMU provides full system emulation, enabling the ARM architecture to function independently of physical hardware. This setup allows the Xen hypervisor to run on a virtualized ARM CPU, which, in turn, manages guest operating systems and facilitates TrustZone functionality.

In this configuration:

- **TrustZone Application:** Securely runs in the ARM TrustZone, isolated from the non-secure world.
- **Linux Guest OS:** Operates in the non-secure world on the virtualized ARM hardware.
- **Hypervisor (Xen):** Provides the virtualization layer, ensuring resource isolation and security.
- **QEMU (ARM CPU Emulator):** Emulates the ARM architecture, enabling the stack to run on a non-ARM host.
- **Host OS (AnyCPU):** Supplies the physical resources for virtualization and emulation.

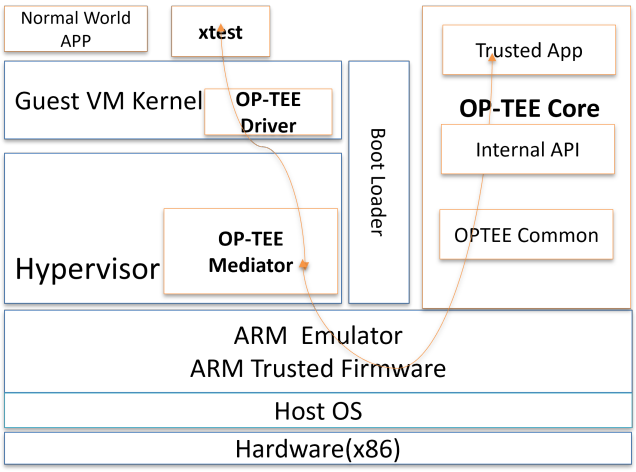

Fig. 4. Final Virtualization Stack. ARM Processor is virtualized by QEMU

3.4 Recap

In summary, the virtualization stack was built incrementally to enable TrustZone applications in a flexible and secure environment:

- **Step 1: Bare-Metal Setup:** Linux was configured to run directly on an ARM CPU with TrustZone support, establishing the foundational hardware-based isolation between the Secure World and Non-Secure World.
- **Step 2: Virtualization with Xen:** The Xen hypervisor was introduced to virtualize the ARM CPU, enabling the concurrent execution of multiple isolated guest operating systems.
- **Step 3: Emulation with QEMU:** QEMU was employed to emulate the ARM CPU, allowing the entire stack to operate seamlessly on non-ARM hardware, such as x86-based platforms.

This step-by-step approach integrates TrustZone with virtualization and emulation technologies, showcasing a scalable framework for testing, debugging, and researching secure applications in virtualized embedded systems. By combining the strengths of QEMU and Xen, the stack achieves both hardware independence and robust isolation, making it a versatile tool for advancing TrustZone-based research.

4 Implementation

This section describes the implementation of the virtualization stack to enable TrustZone-based applications.

The implementation of the virtualization stack primarily followed the guidance provided by Denis Obrezkov for setting up Xen, QEMU, and Linux configurations [Obrezkov 2019]. However, our work leverages the most up-to-date software versions to ensure compatibility and improved functionality. For the Trusted Execution Environment (TEE) components, we relied on the official OP-TEE documentation to ensure proper integration and configuration of secure world operations [Contributors 2024].

4.1 System Components

Our implementation relies on the following primary components:

- **Trusted Execution Environment:** Trusted Application Benchmark (xtest) in Normal World, along with OP-TEE Core within Secure World to manage and execute trusted applications.
- **Hypervisor:** Xen hypervisor with experimental support for OP-TEE, providing robust virtualization capabilities.
- **ARM Emulator:** QEMU configured with ARM virtualization extensions to emulate the ARM CPU and facilitate the virtualization stack on x86-based hardware.
- **Guest Operating Systems:** Linux kernel, compiled for both secure and non-secure environments, running as guest OSes.
- **Bootloader:** U-Boot to initialize the system and load the Xen hypervisor and guest operating systems.
- **Utility Tools:** BusyBox for creating a lightweight root filesystem for the guest Linux environments.

The key software components used in our implementation include:

- **OP-TEE:** Version 4.3.0
- **Xen:** Version 4.18.3

- **QEMU:** Version 9.1.0
- **BusyBox:** Version 1.37.0
- **Linux Kernel:** Version 6.1.18
- **U-Boot:** Version 2024.07

4.2 Role of the Hypervisor in OP-TEE Integration

The hypervisor plays a critical role in facilitating secure interaction between virtual machines (VMs) and the Trusted Execution Environment (TEE) provided by OP-TEE. VMs cannot directly call OP-TEE due to several virtualization and security challenges, including the lack of physical address awareness, the need for OP-TEE to track VM creation and destruction, and the requirement to identify which VM is making a request. Additionally, the hypervisor ensures memory isolation, preventing one VM from referencing another VM's memory when interacting with OP-TEE. By addressing these challenges, the hypervisor enables secure and efficient integration of OP-TEE into the virtualized environment. OP-TEE need to track life cycle of VMs. So it provides two special calls:

- `OPTEE_SMC_VM_CREATED VMID`
- `OPTEE_SMC_VM_DESTROYED VMID`

Hypervisor informs OP-TEE about VM creation or destruction by issuing above SMCs.

4.3 OP-TEE Virtualization Support: XEN TEE Mediator

OP-TEE provides experimental support for virtualization, allowing a single OP-TEE instance to securely execute Trusted Applications (TAs) from multiple virtual machines (VMs). This is achieved by isolating VM-specific states within OP-TEE, ensuring that operations from one VM cannot affect another. Such isolation is critical for maintaining the integrity and security of Trusted Execution Environments (TEEs) in virtualized systems.

In a virtualized setup, OP-TEE relies heavily on the hypervisor to mediate interactions between VMs and the TEE. The hypervisor is responsible for:

- Identifying which VM is making a request to OP-TEE.
- Informing OP-TEE about the creation and destruction of VMs to manage secure resources effectively.
- Translating Intermediate Physical Addresses (IPAs) used by VMs into Physical Addresses (PAs) required by OP-TEE, as OP-TEE cannot process IPAs directly.

To support these functions, Xen includes a specialized component called the "TEE Mediator." This mediator ensures secure communication between VMs and OP-TEE while maintaining isolation and proper address translations.

To enable OP-TEE support in Xen, two configuration methods can be used: interactive menu-based configuration and non-interactive command-line configuration.

Interactive Menu Configuration:

(1) Navigate to the Xen source directory.
(2) Run the `make menuconfig` command to access the Xen configuration menu.
(3) Enable the `OPTEE` option under the "Architecture Features" section.
(4) Ensure that the "TEE mediators" option is also enabled.
(5) Save the configuration and exit the menuconfig tool.
(6) Compile and install Xen with the updated configuration.

The specific path in the Xen interactive menu for enabling OP-TEE support is: `Xen/arm 4.18.3 configuration -> Architecture Features -> TEE mediators -> Enable OP-TEE mediator (UNSUPPORTED)`.

Non-Interactive Command-Line Configuration: Alternatively, the configuration can be performed directly by modifying the Xen '.config' file. The following changes enable OP-TEE support: edit the `xen/.config` file and update the following options:

Listing 1. Enable OP-TEE Support in Xen Configuration

```
# Xen configuration file: xen-4.18.3/xenxen/.config

CONFIG_TEE=y     # Enable Trusted Execution Environment
    (TEE) support
CONFIG_OPTEE=y   # Enable OP-TEE mediator support
```

By enabling OP-TEE support through either method, Xen becomes capable of securely managing interactions between VMs and the Trusted Execution Environment. This setup is critical for testing and deploying TrustZone-based applications in virtualized environments.

4.4 OP-TEE Linux Kernel Support

The OP-TEE driver in the guest VM kernel is essential for enabling secure communication between the guest operating system and the Trusted Execution Environment (TEE) provided by OP-TEE. In a virtualized environment, guest VMs rely on the hypervisor to manage hardware resources, but direct access to TrustZone features is not inherently available to these VMs. The OP-TEE driver acts as an intermediary, allowing guest applications to issue secure requests to OP-TEE. This integration ensures that security-critical operations, such as cryptographic computations and secure storage, can be performed in the TEE without exposing sensitive data to the untrusted hypervisor or other guest VMs. Furthermore, the driver abstracts the complexity of managing TrustZone interactions, enabling seamless use of OP-TEE services within the virtualized ecosystem, thus enhancing the security capabilities of the guest VM.

To enable the OP-TEE driver in the guest VM kernel, the configuration must be set within the Linux kernel menuconfig tool. The specific path to access and configure this driver is: `.config - Linux/x86 6.1.18 Kernel Configuration -> Device Drivers -> Trusted Execution Environment Support`

The following commands and configurations are applied via the command line:

Listing 2. Enabling OP-TEE in the Linux Kernel Configuration

```
# Linux configuration file: linux-6.1.18/.config
    CONFIG_XEN=y        # Enable Linux Kernel support for
        XEN
    CONFIG_XEN_DOM0=y   # Support Linux running as a Xen
        Dom0 guest
```

```
CONFIG_TEE=y        # Enable Linux Kernel support for
    Trusted Execution Environment (TEE)
CONFIG_OPTEE=y      # Enable Linux Kernel support for
    OP-TEE
```

5 Benchmark Performance Analysis

The *xtest* benchmark suite was executed to evaluate the performance of the OP-TEE Trusted Execution Environment (TEE) in the context of our virtualization stack. The results provide critical insights into the feasibility of integrating OP-TEE with Xen and QEMU to enable secure applications in virtualized embedded systems.

5.1 Results Overview

The benchmark suite completed 24 subtests successfully, with no failures, indicating the stability of the OP-TEE environment within the virtualized setup. Key observations include:

- **Trusted Storage Performance:** Write operations were efficient for smaller data sizes (up to 28.57 kB/s for 1 KB), but performance decreased significantly for larger files (17.09 kB/s for 1 MB). Read operations consistently outperformed writes, peaking at 250 kB/s for 2 KB, but dropped to 134.95 kB/s for 1 MB.
- **Cryptographic Performance:**
  - SHA1 achieved a mean speed of 0.48 MiB/s, while SHA256 performed significantly better at 2.22 MiB/s, demonstrating the TEE's capability to handle secure hash operations efficiently.
  - AES encryption showed stable performance, with ECB mode achieving 0.61 MiB/s and CBC mode slightly higher at 0.63 MiB/s.
- **Rewriting Data:** Rewrites exhibited a similar trend to writes, with small data sizes performing efficiently but a sharp decline in speed for larger files.

5.2 Project-Specific Insights

The benchmark results align closely with the objectives of this project, demonstrating the viability of integrating OP-TEE with a Xen-based virtualized environment on QEMU. Key takeaways include:

- **Secure Storage Limitations:** While the TEE effectively secures storage operations, the performance degradation for larger data sizes highlights potential bottlenecks in scenarios involving high-throughput applications. This aligns with the project's focus on testing and identifying limitations in the current implementation.
- **Cryptographic Workloads:** The strong performance in SHA and AES benchmarks validates OP-TEE's suitability for handling cryptographic tasks within a secure environment. This result is significant for the project's goal of evaluating the TEE's ability to support security-critical applications in a virtualized setup.
- **Single-VM Context:** The benchmarks confirm the functional stability of OP-TEE in the current single-VM configuration. However, the project's requirement to support multiple VMs presents a challenge, as performance implications for multi-VM setups remain unexplored.

5.3 Implications for Virtualized Embedded Systems

The results underscore the strengths and challenges of leveraging OP-TEE in virtualized environments:

- **Strengths:** Cryptographic operations and secure isolation were efficiently managed, indicating that OP-TEE can provide robust security services within a virtualized system.
- **Challenges:** The performance decline for large-scale storage operations and the absence of multi-VM support reveal critical areas for improvement to meet production-level demands.

Overall, the benchmarks demonstrate that the integration of OP-TEE with Xen and QEMU provides a secure and functional foundation for virtualized embedded systems. Future work will focus on addressing identified limitations to scale this solution for multi-VM environments, aligning with the project's long-term objectives.

6 Challenges

During the implementation of the virtualization stack, several challenges were encountered. While some issues stemmed from real mistakes in the original tutorial [Obrezkov 2019], others were due to differences introduced by software updates, as we used significantly newer versions of Xen, QEMU, Linux, U-Boot, and OP-TEE. These updates, spanning over five years, led to changes in software behavior, configuration options, and compatibility requirements, necessitating adjustments and debugging to align with the updated software ecosystem.

6.1 BusyBox Hardware Acceleration Compatibility

When compiling BusyBox for use in a virtualized environment like QEMU with Xen, it is essential to disable hardware acceleration (HWA) features. This is because virtualized environments typically do not provide direct access to hardware acceleration, such as NEON on ARM platforms. Enabling HWA during compilation can result in build failures or runtime issues due to the lack of support for hardware-accelerated instructions in emulated environments.

To ensure compatibility, HWA must be explicitly disabled in the BusyBox configuration, aligning the build with the constraints of the virtualized environment.

To compile BusyBox without hardware acceleration, follow these steps:

Listing 3. Solution to HWA Fault

```
# Disable Hardware Acceleration in configuration
sed -i 's/CONFIG_SHA1_HWACCEL=y/CONFIG_SHA1_HWACCEL=n/'
    .config
sed -i 's/CONFIG_SHA256_HWACCEL=y/CONFIG_SHA256_HWACCEL=n/'
    .config
```

By disabling hardware acceleration, the compiled BusyBox binaries rely solely on software implementations. This ensures compatibility and stable performance in virtualized environments, avoiding potential issues with unsupported hardware features.

6.2 BusyBox Compilation Failure with Traffic Control Utility

During the compilation of BusyBox, a issue was encountered when enabling support for the `tc` (traffic control) utility. The problem arises due to the removal of the `CBQ (Class-Based Queuing)` feature from the Linux kernel. Specifically, the kernel headers no longer define the constant `TCA_CBQ_MAX`, which is referenced in the BusyBox source code.

Listing 4. Failure on compile BusyBox

```
networking/tc.c: In function cbq_print_opt:
networking/tc.c:236:27: error: TCA_CBQ_MAX undeclared
 (first use in this function); did you mean TCA_CBS_MAX?
  236 |        struct rtattr *tb_TCA_CBQ_MAX+1_;
      |                       ^~~~~~~~~~
      |                       TCA_CBS_MAX
```

Root Cause: The `tc` utility relies on kernel features like CBQ for traffic shaping and queuing. With these features deprecated and subsequently removed from newer kernel versions, the corresponding BusyBox code referencing `TCA_CBQ_MAX` cannot compile.

To resolve this, `tc` support needs to be disabled in BusyBox's configuration file (`.config`). This can be done using the following command:

Listing 5. Solution to Disable Traffic Control

```
sed -i 's/CONFIG_TC=y/CONFIG_TC=n/' .config
```

Disabling `tc` is a practical workaround if traffic control functionality is not required, this change prevents the compilation of the `tc` utility, avoiding the error.

6.3 QEMU stuck on loading unsupported GPIO module

QEMU with Xen Stuck on Loading Linux as `dom0`

A challenge was encountered when running QEMU with Xen to start Linux as `dom0`. The system would get stuck during the boot process due to issues related to the Advanced Microcontroller Bus Architecture (AMBA) compliant controller. This problem manifested in the output as follows:

Listing 6. QEMU stuck on unsupported GPIO module

```
# Start the QEMU virtual emulation
$ qemu-system-aarch64 -machine virt,gic_version=3 -machine
     virtualization=true \
  -cpu cortex-a57 -machine type=virt -m 4096 -smp 4 -bios
       u-boot.bin \
  -device loader,file=xen,force-raw=on,addr=0x49000000 \
  -device loader,file=Image.gz,addr=0x47000000 \
  -device loader,file=virt-gicv3.dtb,addr=0x44000000 \
  -device loader,file=rootfs.img.gz,addr=0x42000000 \
  -nographic -no-reboot \
  -chardev socket,id=qemu-monitor,\
   host=localhost,port=7777,server,nowait,telnet \
  -mon qemu-monitor,mode=readline

... [Truncated output] ...

### XEN booting up, but stuck at loading DOM0 ###
(XEN) d0v1: vGICR: SGI: unhandled word write
      0x000000ffffffff to ICACTIVER0
```

The process would hang indefinitely, requiring manual intervention to terminate.

Root Cause: The problem is linked to the ARM `PL061` GPIO module, which is part of the AMBA controller. Xen attempts to manage this hardware during boot, but it lacks full support for the `PL061` module in the current configuration. As a result, Xen gets stuck in an unhandled state.

The solution involves disabling the `PL061` controller in the device tree. This prevents Xen from trying to initialize unsupported hardware during the boot process. This can be achieved with the following steps:

Listing 7. Solution to QEMU stuck

```
dtc -I dtb -O dts virt-gicv3.dtb > virt-gicv3.dts
sed 's/compatible = "arm,pl061.*/status = "disabled";/g'
     virt-gicv3.dts > virt-gicv3-edited.dts
dtc -I dts -O dtb virt-gicv3-edited.dts > virt-gicv3.dtb
```

This procedure modifies the device tree to disable the incompatible `PL061` controller, allowing the system to proceed past the stuck state.

After applying these changes, QEMU successfully boots Linux as `dom0` under Xen without hanging. This solution provides a temporary workaround for environments where the `PL061` module is unnecessary or unsupported.

6.4 Launching QEMU with Xen and Two Linux Instances

This challenge involves configuring QEMU to run Xen and boot two Linux instances (`dom0` and `domU1`). The procedure requires careful memory and device tree setup due to changes in the size of Linux images in newer releases.

QEMU Launch Command. The following command initializes QEMU with Xen, two Linux kernels, and their corresponding root file systems:

Listing 8. Designed Device Layout

```
qemu-system-aarch64 \
    -machine virt,gic_version=3 \
    -machine virtualization=true \
    -cpu cortex-a57 \
    -machine type=virt \
    -m 4096 \
    -smp 4 \
    -bios u-boot.bin \
    -device loader,file=xen,force-raw=on,addr=0x50000000 \
    -device loader,file=Image,addr=0x47000000 \
    -device loader,file=Image,addr=0x53000000 \
    -device loader,file=virt-gicv3.dtb,addr=0x44000000 \
    -device loader,file=rootfs.img.gz,addr=0x42000000 \
    -device loader,file=rootfs.img.gz,addr=0x58000000 \
    -mon qemu-monitor,mode=readline
```

The size of the compiled Linux guest OS has increased in newer releases compared to the versions used in older tutorials. This increase can result in a FDT_ERR_NOSPACE error during boot due to insufficient space in the Flattened Device Tree (FDT).

U-Boot may failed with some error like this due to the larger image size.

Listing 9. U-Boot error due to different image size

```
libfdt fdt_setprop(): FDT_ERR_NOSPACE
chosen {
riscv,kernel-start = <0x00000000 0x80200000>;
};
```

To resolve this, it's necessary to include the `fdt resize` command before certain FDT modifications. This step ensures that the FDT can accommodate larger configurations.

Below is the updated U-Boot configuration sequence:

Listing 10. Solution to U-Boot error

```
setenv xen_bootargs 'dom0_mem=512M'
fdt addr 0x44000000
<truncated>

# resize fdt to accommodate the real image size
fdt resize
fdt mknod /chosen/domU1 module@1
<truncated>
```

The key takeaway is that modern Linux kernels and images are significantly larger than their predecessors. This necessitates the use of `fdt resize` to prevent FDT errors and ensure a smooth boot process.

7 Limitation

At the current stage of the project, the OP-TEE feature is limited to supporting a single virtual machine (VM). While the project requirements specify the need to enable OP-TEE across two VMs, the following limitations were encountered:

Single-VM Support in OP-TEE: The current implementation of OP-TEE is designed to handle Trusted Applications (TAs) for a single VM. Extending this functionality to multiple VMs requires significant modifications and is not directly supported by the existing tutorials or configurations.

These limitations underscore the challenges of scaling OP-TEE for multi-VM setups and highlight areas for further development to meet the project's requirements.

8 Future Work

One promising direction for future work is to combine the OP-TEE tutorial with the multi-VM setup outlined in the Xen wiki tutorial to enable support for multiple virtual machines (VMs). This enhancement would simulate a more realistic production environment, where multiple VMs can securely interact with a single OP-TEE instance.

However, achieving this goal requires addressing several challenges. Notably, OP-TEE has certain hardcoded configurations, such as the memory loading address for the secure image. Modifying these settings to accommodate multiple VMs demands additional effort and a deeper understanding of both OP-TEE's architecture and the Xen hypervisor's virtualization mechanisms. Overcoming these limitations would pave the way for advanced research into secure multi-VM environments using OP-TEE and Xen.

9 Conclusion

This project successfully implemented a virtualization stack integrating OP-TEE with the Xen hypervisor and QEMU emulator to evaluate the feasibility of enabling Trusted Execution Environment (TEE) functionality in a virtualized embedded system. The results from the *xtest* benchmarks demonstrated the stability and performance of OP-TEE within a single-VM setup, validating its capability to securely handle cryptographic and storage operations.

However, the project also highlighted key limitations, including the lack of multi-VM support and performance bottlenecks for large-scale trusted storage operations. These findings underscore the need for further optimization and development to address scalability challenges and enhance the usability of OP-TEE in production environments.

Despite these challenges, this work provides a solid foundation for future research into secure virtualization. The integration of OP-TEE with Xen and QEMU not only demonstrates the viability of deploying TEEs in virtualized setups but also identifies critical areas for improvement. Future efforts will focus on extending support to multiple VMs and refining secure storage mechanisms to better meet the demands of real-world applications.

This project contributes to advancing the field of secure embedded systems, offering insights into the integration of hardware-based security technologies with modern virtualization platforms.

References

Julien Amacher and Valerio Schiavoni. 2019. On the Performance of ARM TrustZone. In *Distributed Applications and Interoperable Systems*, José Pereira and Laura Ricci (Eds.). Springer International Publishing, Cham, 133–151.

Fabrice Bellard. 2005a. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. USENIX Association. https://www.usenix.org/legacy/event/usenix05/tech/general/bellard.html

Fabrice Bellard. 2005b. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 41.

Giorgiomaria Cicero, Alessandro Biondi, Giorgio Buttazzo, and Anup Patel. 2018. Reconciling security with virtualization: A dual-hypervisor design for ARM TrustZone. In *2018 IEEE International Conference on Industrial Technology (ICIT)*. 1628–1633. https://doi.org/10.1109/ICIT.2018.8352425

OP-TEE Project Contributors. 2024. *OP-TEE Documentation*. https://optee.readthedocs.io/en/latest/ Accessed: December 4, 2024.

Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. 2020a. PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 789–806. https://www.usenix.org/conference/usenixsecurity20/presentation/harrison

Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. 2020b. PartEmu: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation. In *29th USENIX Security Symposium*. USENIX, 789–805. https://www.usenix.org/conference/usenixsecurity20/presentation/harrison

Wenhao Li, Yubin Xia, Long Lu, Haibo Chen, and Binyu Zang. 2019. TEEv: virtualizing trusted execution environments on mobile platforms. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Providence, RI, USA) *(VEE 2019)*. Association for Computing Machinery, New York, NY, USA, 2–16. https://doi.org/10.1145/3313808.3313810

Steffen Liebergeld. 2009. Efficient Virtualization on ARM Platforms. In *Proceedings of Technische Universität Dresden, Institut für Systemarchitektur*. Technische Universität

Dresden, Dresden, Germany.

Radu Mantu, Florin Stancu, Mihai Chiroiu, and Nicolae Ţăpuş. 2023. Approaches to teaching embedded development. In *2023 22nd RoEduNet Conference: Networking in Education and Research (RoEduNet)*. 1–5. https://doi.org/10.1109/RoEduNet60162.2023.10274941

Denis Obrezkov. 2019. *Xen on ARM and QEMU.* https://medium.com/@denisobrezkov/xen-on-arm-and-qemu-1654f24dea75

S. Pinto, D. Oliveira, J. Pereira, N. Cardoso, M. Ekpanyapong, J. Cabral, and A. Tavares. 2014. Towards a lightweight embedded virtualization architecture exploiting ARM TrustZone. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*. 1–4. https://doi.org/10.1109/ETFA.2014.7005255

Johannes Winter, Paul Wiegele, Martin Pirker, and Ronald Tögl. 2012. A Flexible Software Development and Emulation Framework for ARM TrustZone. In *Trusted Systems*, Liqun Chen, Moti Yung, and Liehuang Zhu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–15.

Heedong Yang and Manhee Lee. 2021. Demystifying ARM TrustZone TEE Client API using OP-TEE. In *The 9th International Conference on Smart Media and Applications* (Jeju, Republic of Korea) *(SMA 2020)*. Association for Computing Machinery, New York, NY, USA, 325–328. https://doi.org/10.1145/3426020.3426113