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 wide adoption of virtualization technologies in embedded systems has brought up the demand for secure environments that protect against privileged adversaries. ARM TrustZone is a hardware-based TEE that provides a basis for securely executing sensitive operations. However, integrating TrustZone with virtualization platforms remains a challenge. In this work, QEMU and Xen are used to establish a robust development environment for emulating ARM TrustZone. QEMU, therefore, enhances TrustZone feature debugging, testing, and verification by offering a platform emulated from ARM for independent work from the actual physical hardware. Xen expands this virtualization capability with this hypervisor, whereby various separated operating systems can execute efficiently and at the same time not compromise on allowing supported TrustZone operations. Taken together, QEMU and Xen are a secure yet versatile combination for exploring and improving the potential of virtual systems' TrustZone. This paper is demonstrating the setup, integration, and evaluation of TrustZone-enabled virtualized environments that provide 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

TEEs combined with virtualization platforms have become increasingly popular over the past years. One such hardware-based TEE is ARM TrustZone, which provides a safe haven for sensitive operations. However, its integration into virtualized systems is still an underexplored area. This section discusses some key contributions in this domain and highlights how our solution stands out. Li et al. proposed TEEv, a virtualization framework that allows multiple isolated TEE instances to run concurrently on mobile platforms [Li et al. 2019]. Their architecture, leveraging a hypervisor (TEE-visor), allows for the isolation of TEE instances and reduces security risks related to shared TEEs. However, TEEv focuses mainly on multi-TEE environments and does not discuss the functional integration of ARM TrustZone with hypervisors for system-level virtualization.[Pinto et al. 2014] [Winter et al. 2012]

Harrison et al. proposed PARTEMU, a modular framework for emulating TrustZone software [Harrison et al. 2020a]. It is based on QEMU and PANDA and allows for the dynamic analysis of real-world TrustZone operating systems and applications. Although very powerful in finding vulnerabilities, PARTEMU does not support the integration of TrustZone with hypervisors like Xen, which plays an important role in virtualized system management.

Yang and Lee used OP-TEE with QEMU to implement TEE Client APIs between the Normal World and Secure World [Yang and Lee 2021]. In their work, they have provided a good starting point for security research. Their solution has again been designed for a single-system setup and does not extend the virtualization capability.[Mantu et al. 2023] [Yang and Lee 2021] [Cicero et al. 2018]

Our approach extends these efforts further by incorporating ARM TrustZone into a Xen-based virtualized environment. Unlike TEEv, which focuses on multi-TEE scenarios, our work focuses on taking advantage of TrustZone in order to provide secure services in a single TEE architecture. Finally, QEMU emulates the ARM platforms for ease of debugging, testing, and performance benchmarking. Using QEMU together with Xen in this context provides an extended framework for exploring and enhancing TrustZone's potential in virtualized embedded systems.

1.2 Motivation

The growing presence of embedded systems in critical infrastructure has raised the need for better security. On the other hand, hypervisors are powerful yet vulnerable to attacks that might spill over into all systems running on the hardware. This is where ARM TrustZone steps in, providing a TEE that is somewhat isolated from the rest of the system and thus protected against privileged adversaries. Hence, this project will be about studying the possibility of running Xen hypervisor with OP-TEE TrustZone Software support on an ARM QEMU-emulated environment. Through such a study, we will be able to increase the security level of embedded systems by expanding the TrustZone benefits to virtualized environments.

1.3 Objective

The key outcome expected is the development of a stable, flexible development environment to try the integration of ARM TrustZone with virtualization technologies. This translates into the following set of concrete objectives:

- Port Xen hypervisor featuring OP-TEE (TrustZone Software) running within a QEMU-emulated ARM, to enable emulation of the functionality provided by TrustZone without any dependence on physical hardware.
- It will research the performance-security consequences of combining TrustZone and Virtualization: focusing on a probably beneficial trade-off when doing virtualization. Benchmark performance-security using *xtest*, to determine enhancement and security guarantees by TrustZone provided as a result of various methods of deploying the coprocessor in both single and multi-operating system guests.

[*]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.

ACM XXXX-XXXX/2024/12-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- Basic understanding of how virtualized systems with integrated TrustZone work, which will be helpful in future research and practical applications of embedded systems.

This work not only addresses current hypervisor-TrustZone integration limitations but also provides the foundation to advance both security and functionality for virtualized embedded systems.

2 Background

2.1 ARM TrustZone Technology

ARM TrustZone is a hardware-based security technology integrated into many ARM processors, widely used in mobile devices and embedded systems. It provides a strict 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 allows a TEE for sensitive operations to be executed safely, without the possibility of any threat from the Normal World. The most important uses of the Secure World are as follows:

- 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].

The basic principle behind the TrustZone design is being an isolated world from non-trust parts, providing the means to enhance embedded and mobile system security.

2.2 Comparison of Hypervisor Architectures

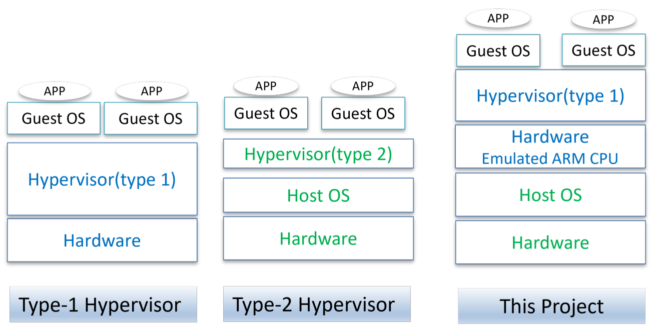

Fig. 1. Type1, Type2 and this Project used Hypervisor

Figure 1 highlights the hypervisor architectures of Type-1 and Type-2 hypervisors compared to the structure used in this project. The hypervisor set-up in this project is very different from the conventional Type-1 and Type-2 hypervisors, considering both complexity and flexibility.

Type-1 Hypervisor: A Type-1 hypervisor is also known as a bare-metal hypervisor. It runs directly on the underlying hardware. The hypervisor and its guest operating systems have exactly the same architecture as the hardware, for example, x86 or ARM. Such a design provides high performance and direct hardware access. However, it relies fully on the hardware platform.

Type-2 Hypervisor: A Type-2 hypervisor, on the other hand, executes on top of a host operating system and depends on this host OS to manage hardware resources. Similar to Type-1 hypervisors, it also requires that both the hypervisor and guest OSes use the same architecture as the base hardware. This is easier to set up but has more overhead and is tightly coupled to the hardware platform.

This Project's Hypervisor Architecture: On the other hand, this project has adopted the hypervisor architecture, which decouples the hypervisor and guest operating systems from the original hardware architecture. Using QEMU, an emulation of the ARM CPU is allowed, thereby allowing the Xen Type-1 hypervisor and guest OSs (ARM architecture) to run on an x86 host machine. This design in a hardware-independent way avoids the architectural limitation of traditional 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:** This allows the hypervisor and guest OSes to execute on any hardware, completely independent of the actual architecture, by emulating the ARM CPU.
- **Hardware Independence:** Unlike traditional hypervisors, the virtualization stack is not bound to native hardware architecture, which allows flexibility in secure experimentation, especially when physical ARM hardware is not available.
- **Enhanced Testing and Development:** This architecture is ideal to develop and test secure applications that leverage TrustZone since it provides a controlled, reproducible virtualized environment.

This project will be able to show that QEMU, Xen, and OP-TEE together can form a highly versatile and secure virtualization stack, breaking the limits of traditional hypervisor architectures.

2.3 QEMU: A System Emulator

QEMU is a multi-purpose system emulator that is widely used in research and development to emulate many architectures, including x86, ARM, MIPS, RISC-V, and more. It is one of the enabling factors for cross-platform development and testing without requiring physical hardware [Bellard 2005a; Harrison et al. 2020b; Liebergeld 2009]. [Bellard 2005b]

QEMU plays an important role in this project because it provides an ARM-emulated environment into which ARM TrustZone can be combined with the Xen hypervisor. This is because it emulates a full system: it emulates CPUs, devices, kernels, and applications - and is ideal for the evaluation process of TEEs like OP-TEE. The QEMU system would provide complete emulation of a computer system, including peripherals, thereby allowing numerous virtual machines to be run virtually on one physical host [Liebergeld 2009].

QEMU operates in various modes:

- **User-mode emulation:** This mode enables QEMU to run single Linux or Darwin/macOS programs compiled for other instruction sets. It will translate system calls for endianness and architecture compatibility, but it does so quite well and is convenient for cross-compilation and cross-debugging.

- **System emulation:** In full-system emulation mode, QEMU emulates a complete computer with its devices. This feature makes it possible to run several guest operating systems, including Linux, Windows, Solaris, and BSD, on various architectures, including ARMv7, ARMv8, x86, among others [Bellard 2005a].
- **Hypervisor support:** QEMU can be used either as a VMM or as a device emulation back-end for virtual machines running under a hypervisor. Combined with KVM or hypervisors such as Xen, near-native execution speeds are achieved while still offering broad emulation capabilities [Harrison et al. 2020b].

One of the central features of QEMU is the use of Tiny Code Generator to dynamically translate guest CPU instructions into their equivalent host assembly instructions. This flexibility and efficiency in running virtualized workloads are especially important in developing and benchmarking systems that include ARM TrustZone.

QEMU provides the base for this project, exploring secure and virtualized environments with ARM TrustZone through its powerful emulation features and support for a wide variety of architectures and virtualization setups.

3 System Design: Introduction to the Virtualization Stack

This section covers the design and implementation of a virtualized environment that can enable QEMU-based TrustZone applications. We will explain how to enable a TrustZone-based application running in a virtualized environment on QEMU on an X86 CPU. That will be done by incrementally adding virtualization technologies to an already existing base ARM CPU with TrustZone. This introduces the hypervisor, which is used to virtualize the TEE rich OS and to virtualize the ARM CPU with QEMU. Following is a high-level view 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

First, it shows the simple configuration where the Linux OS runs on an ARM-supported CPU by TrustZone. This TrustZone provides a hardware-based isolation mechanism which divides the CPU into two "worlds": the secure world - that is, TrustZone - and the non-secure world, Linux. Here, one can see that Linux is directly communicating with the ARM CPU, running in a non-secure world, whereas applications run in the secure world. This is the most simple form of running Linux with support for TrustZone.

3.2 Step 2: Virtualize the OS by introducing Hypervisor

This introduces the system to the hypervisor: Xen. A hypervisor-also called a VMM-further abstracts the hardware, thus offering the running of multiple OSs on one computer. So, Xen is going to run on top of the ARM CPU and then virtualize Linux, itself an OS, running as a guest OS. A hypervisor makes the operating systems-even guest ones-much farther away from the hardware so no guest OS acts upon it directly. This will provide the possibility to execute the secure TrustZone application along with the virtualized Linux OS.

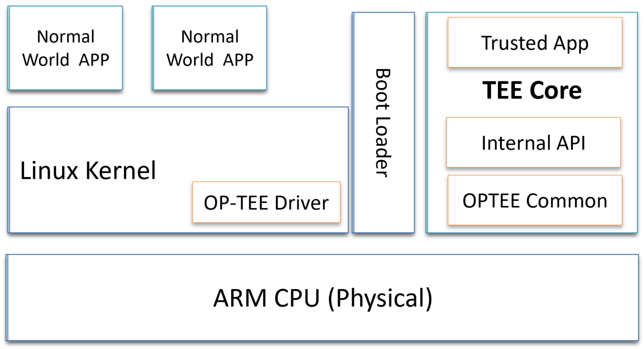

Fig. 2. ARM Trusted App, no virtualization

This is a very important step that involves introducing the hypervisor in order to enable virtualization of the system. In this case, the hypervisor used, Xen, executes directly on the ARM CPU and it sits between the hardware and the OSes to run atop of it.

In our setup, a Type 1 hypervisor is used, namely Xen. Xen runs directly on the ARM hardware; this allows it to manage most of the physical resources of the system and give them - as efficiently as possible - to VMs. The presence of Xen introduces the concept of domains:

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

Xen ensures that these domains remain separated from each other in order not to let the problems occurring in one domain compromise any other. That is because the isolation principle is key for virtualization, meaning running multiple guest OSes securely on the same hardware without disturbing the others.

3.3 Step 3: Virtualizing the ARM CPU by QEMU

The last level of the virtualization stack is using QEMU to emulate the ARM CPU on a non-ARM host machine, like an x86-based one. QEMU will offer complete system emulation, allowing the ARM architecture to be self-sufficient with no use of physical hardware. Such a configuration would mean the Xen hypervisor can run on top of the virtualized ARM CPU, which again will manage guest operating systems and also provide functionality related to TrustZone.

In this configuration:

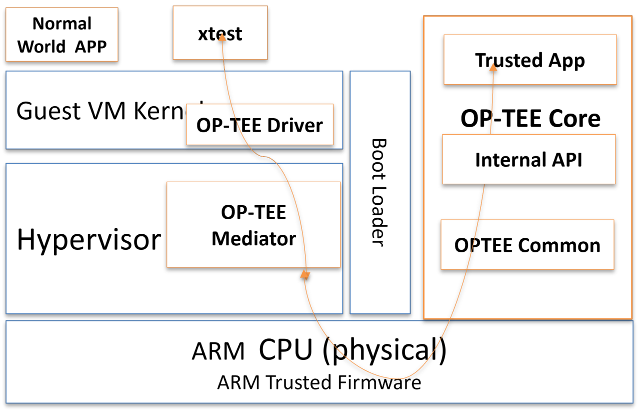

Fig. 3. Linux OS is virtualized by Xen hypervisor

- **TrustZone Application:** TrustZone application runs securely inside ARM TrustZone, separated from the non-secure world.
- **Linux Guest OS:** runs in the non-secure world on top of the virtualized ARM hardware.
- **Hypervisor (Xen):** to provide virtualization layer, maintain resource isolation and security.
- **QEMU (ARM CPU Emulator):** Provides emulation for the ARM architecture, allowing the stack to run on a non-ARM host.
- **Host OS (AnyCPU):** Provides the physical resources for Virtualization and Emulation.

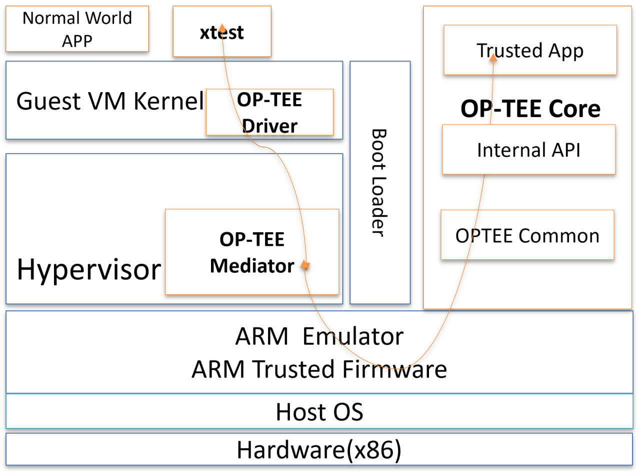

Fig. 4. Final Virtualization Stack. ARM Processor is virtualized by QEMU

3.4 Recap

The basic thought in designing the virtualization stack has been incremental towards the enabling of TrustZone applications, which are flexible but secure.

- **Step 1: Bare-Metal Setup:** It was also installed to run directly on an ARM CPU with TrustZone support, which provides the basic hardware-based isolation between Secure World and Non-Secure World.
- **Step 2: Virtualization with Xen:** The Xen hypervisor virtualized the ARM CPU, which can support multiple independent guest operating systems running simultaneously.
- **Step 3: Emulation with QEMU:** QEMU was used to emulate the ARM CPU, which allowed the entire stack to run without issue on non-ARM hardware, such as that using x86-based platforms.

This will be demonstrated in an integrated step-by-step approach with virtualization and emulation technologies to show a scalable framework for testing, debugging, and researching secure applications in virtualized embedded systems. Combining strengths from QEMU and Xen, this stack reaches hardware independence with robust isolation, thereby making it a versatile tool to advance TrustZone-based research.

4 Implementation

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

Most of the virtualization stack was implemented based on the work of Denis Obrezkov, which gave guidelines on how to set up Xen, QEMU, and Linux configurations [Obrezkov 2019]. Nevertheless, our work is using the latest versions of the software in order to be compatible and have better functionality. In terms of TEE parts, we used the official OP-TEE documentation on how to correctly implement and configure secure world operations [Contributors 2024].

4.1 System Components

Our system uses the following key components to perform the tasks assigned to it:

- **Trusted Execution Environment:** Trusted Application running in the Normal World and OP-TEE Core.
- **Hypervisor:** Xen hypervisor, experimental OP-TEE support, providing strong virtualization competency.
- **ARM Emulator:** QEMU to emulate the ARM CPU and enable 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 for system initialization and loading of the Xen hypervisor and guest operating systems.
- **Utility Tools:** BusyBox to create 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

This is to allow the hypervisor to manage safe interactions between the guest virtual machines and OP-TEE. Different challenges exist for virtualization and security on OP-TEE being directly called by the virtual machines, such as virtual address address space, requiring OP-TEE to be cognisant of VM creation/ destruction and which VM the calling is coming from. Besides, the hypervisor also provides memory isolation such that a VM cannot reference another VM's memory when operating against OP-TEE. The hypervisor solves these challenges to securely and efficiently integrate OP-TEE in the virtualized environment. OP-TEE needs to track the life cycle of VMs. So it provides two special calls:

- `OPTEE_SMC_VM_CREATED VMID`
- `OPTEE_SMC_VM_DESTROYED VMID`

Hypervisor notifies OP-TEE for VM creation or destruction by issuing the above SMCs.

4.3 OP-TEE Virtualization Support: XEN TEE Mediator

OP-TEE has experimental support for virtualization, enabling one OP-TEE instance to securely run TAs originating from several VMs. It does this by keeping the states of the different VMs isolated within OP-TEE so that operations coming from one VM cannot affect another. Such isolation is crucial for the integrity and security of TEEs in virtualized systems. In a virtualized environment, OP-TEE relies on the hypervisor to mediate access between VMs and the TEE. The hypervisor does the following:

- 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.

It achieves this through a dedicated Xen component known as "TEE Mediator." This acts to securely relay information from the VM to OPTEE and vice-versa while ensuring proper isolation and performing proper address translations in both ways.

Basic support for OP-TEE within Xen can be enabled through an interactive, menu-based configuration, or as a noninteractive, 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
```

Either way, the enabling of OP-TEE support allows Xen to securely manage interactions between VMs and the Trusted Execution Environment. It becomes crucial for testing and deploying TrustZone-based applications on virtualized environments.

4.4 OP-TEE Linux Kernel Support

The OP-TEE driver in the guest VM kernel is essential for the guest OS to interact securely with the TEE provided by OP-TEE. In a virtualized environment, the hypervisor manages hardware resources on behalf of guest VMs, but direct access to the features of TrustZone is not inherently provided to such VMs. The OP-TEE driver acts as an intermediary to the requests of guest applications for secure processing to OP-TEE. In this way, it ensures that all security-sensitive operations-such as cryptographic computations and key storage-are executed in the TEE without disclosure of sensitive data to an untrusted hypervisor or other guest VMs. Furthermore, the driver abstracts the complexity of managing TrustZone interactions, allowing for smooth use of the OP-TEE services from within the virtualized environment, thereby increasing the security features of the guest VM.

To enable the OP-TEE driver in the guest VM kernel, configuration should be made inside the Linux kernel menuconfig tool. The path to reach this driver in order to configure it 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* A benchmark suite was run in order to evaluate the performance of the OP-TEE TEE in the context of our virtualization stack. These results are important in understanding the feasibility of using OP-TEE with Xen and QEMU for enabling secure applications in virtualized embedded systems.

5.1 Results Overview

The benchmark suite ran successfully through the 24 sub-tests with no failures; in summary, the virtualized set up OP-TEE environment had shown stability. Key observations:

- **Trusted Storage Performance:** Writes performed well for smaller sizes-28.57 kB/s for 1 KB, but it tumbled to 17.09 kB/s in the case of big sizes such as 1 MB. Read operations are always better than writes with maximum of 250 KBs when 2 KB can fit inside cache and for 1MB dropped down to 134.95 kB/s.
- **Cryptographic Performance:**
  - SHA1 averaged 0.48 MiB/s, while SHA256 was much faster at 2.22 MiB/s, showing the efficiency of the TEE in performing secure hash operations.
  - AES-ENCRYPTION was quite stable; for example, ECB mode had 0.61 MiB/s, while CBC mode had 0.63 MiB/s.
- **Rewriting Data:** Rewrite also followed a similar pattern to write: for smaller sizes, it was quite efficient but bigger file sets sharply lowered the speed.

5.2 Project-Specific Insights

The results are very close to the objectives of this project, which means it is feasible to integrate OP-TEE with a Xen-based virtualized environment on QEMU. Key takeaways include:

- **Secure Storage Limitations:** While the TEE securely protects storage operations, performance degradation with respect to larger data sizes does bring out potential bottlenecks for high-throughput applications. This falls in line with testing the limitations within the current implementation, which was the focus of the project.
- **Cryptographic Workloads:** Such a high score in SHA and AES benchmarks again proved OPTEE to be suitable for performing cryptographic tasks in the secure environment, which is crucial for this project focused on studying whether the TEE would be able to support security-critical applications in virtualized setups.
- **Single-VM Context:** Benchmarks indeed confirm that in the current configuration of a single VM, OP-TEE is functionally stable, but the project requirement to support multiple VMs complicates things since performance implications for multi-VM setups remain unexplored.

5.3 Implications for Virtualized Embedded Systems

Results will highlight the strengths and challenges of using OP-TEE in virtualized environments.

- **Strengths:** Efficient handling of cryptographic operations, secure isolation was in place, meaning that OPTEE may achieve appropriate security services within a virtualized system.
- **Challenges:** Performance degradation in case of large-scale storage operations, support for multiple VMs are the key areas that need refinement before going into production-level demands.

Overall, the benchmarks show that the integration of OP-TEE with Xen and QEMU offers 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

The implementation of the virtualization stack encountered several problems. Some problems were indeed real mistakes in the original tutorial [Obrezkov 2019], while others involved differences introduced by software updates, as we used significantly newer versions of Xen, QEMU, Linux, U-Boot, and OP-TEE. That is, five years passing by means lots of changes in behavior, configuration options, compatibility requirements, which require readjustments and debugging to match the updated software ecosystem.

6.1 BusyBox Hardware Acceleration Compatibility

That is, for example, to compile BusyBox to run on QEMU with Xen, all HWA should be turned off. Most virtualized environments don't support direct hardware acceleration, like NEON on ARM platforms. Because of this, enabling HWAs at compile time may result in build breaks or runtime problems because emulated environments don't support hardware-accelerated instructions.

Due to compatibility reasons, HWA should be explicitly disabled in BusyBox configuration to make the build consistent with the constraints of the virtualized environment.

To compile BusyBox without hardware acceleration:

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
```

In the case of BusyBox, compiled binaries would disable hardware acceleration completely and rely on the software part. This guaran-tantly assures compatibility and stable performance inside virtualized environments without potential problems with unsupported hardware features.

6.2 BusyBox Compilation Failure with Traffic Control Utility

There are some problems being thrown up when trying to add on `tc` (traffic control) command under compilation of BusyBox. Reasons of security and since the feature was found largely unused, `CBQ (Class-Based Queuing)` had already been removed from the Linux Kernel in which the header definitions - namely `TCA_CBQ_MAX` utilized in BusyBox sourcecode resides.

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 such as CBQ for traffic shaping and queuing. These are deprecated and removed in newer versions of the kernel, which means that the corresponding BusyBox code using `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`

There was a problem in running QEMU with Xen to bring up Linux as `dom0`. It would hang somewhere during boot up due to some problem with the AMBA compliant controller. This would be manifested as :

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: This problem is related to the ARM `PL061` GPIO module of the AMBA controller. At boot time, Xen tries to handle the underlying hardware but, with the current configuration, does not have complete support for the `PL061` module. Due to this, Xen goes into 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. The steps are as follows:

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, enabling the system to get past this hang state.

Applying the above, QEMU boots Linux as `dom0` under Xen with no hang. This is a temporary workaround in environments where the `PL061` module is not needed or supported.

6.4 Starting QEMU with Xen and Two Linux Guests

This exercise starts QEMU to run Xen and boot two Linux guests (`dom0` and `domU1`). Since newer versions of Linux images have different sizes, the setup of memory and device tree must be adjusted accordingly.

QEMU Start Command. The following starts QEMU with Xen, two Linux kernels and their 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
```

6.4.1 *Scaling with larger Linux Images.*

The size of the compiled Linux guest OS is larger in the newer releases compared to the versions used in older tutorials. This may lead to a `FDT_ERR_NOSPACE` error at boot time since there is not enough space in the Flattened Device Tree (FDT).

U-Boot can fail with some error like this from 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 work around this, `fdt resize` needs to be executed prior to some FDT operations so that the FDT itself can support 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 takeaway here is that the modern Linux kernels and images are way larger in size compared to their ancestors. Also, using `fdt resize` prevents FDT errors, which allows smooth booting.

7 Limitation

Currently in the project, OP-TEE's feature is restricted to be enabled on only a single virtual machine (VM). Project requirement wants OP-TEE enable on two VMs but during its enablement, below was found as a limitation :

Single-VM Support in OP-TEE: OP-TEE only supports one VM. Handling more than one VM requires major changes in the existing implementation of OP-TEE. As a matter of fact, none of the tutorials/configurations in the existing OP-TEE tutorials/configurations support more than one VM for direct use.

These limitations outline the scaling issues in OP-TEE for multi-VM and provide insight into areas that could be further developed to meet the project's requirements.

8 Future Work

One promising direction for further work is the combination of the OP-TEE tutorial with the multi-VM setup from the Xen wiki tutorial to enable support for multiple virtual machines. That way, it will be able to simulate a real production environment where multiple virtual machines can securely interact with a single instance of OP-TEE.

However, reaching this involves addressing some challenges. The main point is that OP-TEE has some hard-coded configurations; for example, the memory loading address for the secure image. These settings will have to be changed if one intends to use multiple VMs, which requires more work and also a deeper understanding of both OP-TEE's architecture and the Xen hypervisor's virtualization mechanisms. Such limitations, if surpassed, will widen the advanced research on OP-TEE and Xen about the secure multi-VM environment.

9 Conclusion

This project implemented a virtualization stack using the integration of OP-TEE with a Xen hypervisor and a QEMU emulator for studying the feasibility in enabling TEE functionality on a virtualized embedded system. Results from the *xtest* benchmarks demonstrate stability and performance of OP-TEE in a single VM setting that is capable of processing cryptographic and storage operations securely.

This, in turn, demonstrated that there is significant restriction to multi-VM support and performance bottlenecks when it comes to large-scale trusted storage operations. A finding that underlines further optimizations required for scalability challenges and ease of use in the OP-TEE for production environments.

This work lays a concrete foundation for further research into secure virtualization despite these challenges. The integration of OP-TEE with Xen and QEMU not only shows the feasibility of deploying TEEs in virtualized setups but also helps identify key areas that need attention. Future work will involve support for multiple VMs and further refining the ideas of secure storage mechanisms to meet real-world application demands.

Work here contributes to the furtherance of knowledge in secure embedded systems, with insight into the integration of hardware-based security technologies with state-of-the-art 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 Țăpuș. 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