TEEVseL4: Trusted Execution Environment for Virtualized seL4-based Systems

Borna Blazevic*, Michael Peter†, Mohammad Hamad*, Sebastian Steinhorst*
* Department of Computer Engineering, Technical University of Munich
† NIO GmbH
Email: {borna.blazevic, mohammad.hamad, sebastian.steinhorst}@tum.de, michael.peter@nio.io

***Abstract*—The growing computing power of embedded systems has led to an increase in the use of general-purpose Operating Systems (OSs) such as Linux. However, the substantial attack surface arising from their complexity makes them unsuitable for safety and security-critical use cases. Addressing this issue requires isolating the security-critical functionalities into separate execution environments and protecting them from the untrusted OS. Arm TrustZone applies this approach by providing hardware-based partitioning of the system into a secure and non-secure world, facilitating a Trusted Execution environment for the protection of security-critical functionality in the secure world. TrustZone, however, falls short when dealing with systems that virtualize multiple operating systems. Another approach to isolate functionality is employing a microkernel, such as the formally proven correct seL4 kernel, especially if it also offers virtualization functions. While current seL4-based virtualization systems offer good security and safety properties, they do not provide TrustZone-compatible security services to their virtualized guests. In this paper, we propose TEEVseL4, a TrustZone-compatible virtualization system leveraging the strengths of the seL4 microkernel, that can provide security services to the Linux guests based on the dynamic, scalable and flexible Trusted Computing Base of an seL4 system. A high-level performance benchmarking shows that TEEVseL4 can provide security services with acceptable overheads (less than 20%) when compared to a native TrustZone system, making it an attractive option for platforms with multiple, mutually-distrustful virtualized guests.**

***Index Terms*—Security, Virtualization, seL4, Arm TrustZone, TEE**

I. Introduction

Embedded systems, from smartphones and routers to smart vehicles, are increasingly running versatile general-purpose OSs like Linux [1]. Linux is a well-known open-source kernel that offers extensive functionality, mature technology, broad hardware compatibility, and a rich collection of available software applications. These attributes have made it an attractive choice for domains with high demands for safety and security, such as the automotive [1] and space industries [2]. However, like other general-purpose OSs, the complexity of Linux can render it vulnerable to attacks, especially when it is connected to the internet [3]. Moreover, Linux was not originally designed with security and safety-critical applications in mind. To enhance the security of systems relying on complex OSs such as Linux, it is necessary to isolate security-sensitive applications from the general-purpose OS and its other applications.

A Trusted Execution Environment (TEE) (§ II-A) is a separated execution environment running alongside a general-purpose OS, also designated the Rich Execution Environment (REE), and is responsible for protecting assets and executing trusted and security-critical code isolated from any threats that exist in the potentially compromised REE [3] [4]. Arm TrustZone (§ II-B) is a hardware-based security technology

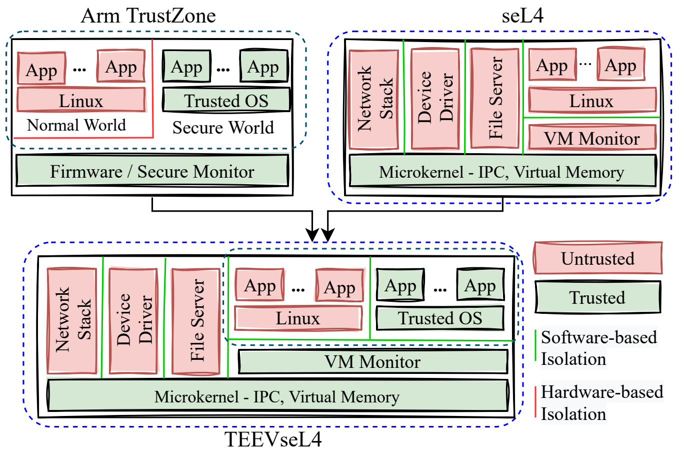


Fig. 1: The TEEVseL4 system architecture, leveraging microkernel (seL4) and Arm TrustZone-compatible software solutions, provides a trustworthy virtualization system with a TrustZone-compatible TEE for secure isolation of security-critical functions.

that allows the creation of a TEE by providing hardware-based partitioning of the system into two isolated environments: Secure World (SW) and Normal World (NW), to represent the TEE and REE respectively, as shown in Fig. 1. This functionality allowed TrustZone to become an industry standard for providing a TEE in mobile devices [5]. Despite its widespread use, TrustZone has many limitations. Due to the limit of two hardware-enforced partitions, TrustZone does not provide component isolation within its TEE but leaves this to be implemented in its TEE kernels (§ II-C), and systems relying on TrustZone have been successfully attacked through vulnerable third-party Trusted Applications (TAs) [5] [6]. In addition, TrustZone does not account for virtualization of multiple NW operating systems as virtualization in the secure world is not supported [1].

Virtualization (§ II-D) is a technology that has been widely adopted in embedded systems as a means of providing high-level isolation for safety and security-critical purposes [7] [8]. Virtualizing TEEs in the NW using a trustworthy hypervisor can address the issues of TrustZone. However, not all hypervisors are suitable for this purpose. Commodity hypervisors that include a management Virtual Machine (VM) (e.g., Xen) or host OS (e.g., KVM) have large codebase complexities that expose a large attack surface. As a result, these hypervisors are unsuitable for providing a TEE for many systems [8] [9].

[1]While we are aware that devices running Arm v8.4 have virtualization support in the secure world, they are not yet widely used and thus difficult to evaluate.

Creating a trusted hypervisor with a trusted codebase and a small-attack surface is also a widely researched topic [8]–[10].

Microkernel-based hypervisors are another way of using virtualization to provide isolation for security and safety-critical applications in embedded systems. Microkernels (§ II-E) are operating system kernels that offer functionalities like address space isolation, threads and Inter-Process Communication (IPC), within a small codebase [11] [12]. As such, they are a good basis for systems that aim to decompose the NW into components even smaller than virtual machines by extracting system components and isolating them into small execution environments, thus ensuring that the system components work independently from each other [13]. Microkernel-based hypervisors, like seL4 [14], Nova [10] or Fiasco [15] also provide hypervisor functionality to enable code reuse by virtualizing general-purpose OSs like Linux alongside the rest of the decomposed system. These microkernel-based hypervisors can be used as trusted hypervisors to provide TEEs to the virtualized general-purpose OSs, addressing the issues of Arm TrustZone [9] while providing functionality reuse to the microkernel-based OS. seL4 (§ II-G), in particular, offers a quality basis for creating a decomposed system, following the Principle of Least Authority (POLA) (§ II-F) that allows for scalable and fine-grained isolation of the security-critical from non-critical system components [16] [14]. Furthermore, seL4 can benefit from all of the functionality offered in a NW by securely virtualizing general-purpose OSs like Linux. Nevertheless, while an seL4 system allows for extracting security-critical functionality into isolated components in user-space, seL4-based systems do not provide a way for re-using Arm TrustZone compatible security solutions for securing their virtualized guests.

A. Requirements

To address the challenge of isolating security functionalities in protected components suitable for safety and security-critical systems like the automotive system, we have identified a set of critical requirements. These requirements aim to ensure that any proposed approach can meet the rigorous demands of such systems in terms of both security and safety while still offering extensive functionality. These requirements will form the basis for evaluating our system and comparing it with the state-of-the-art approaches, which we will discuss in § V.

- **Virtualization of multiple Linux guests**: The system shall be able to reuse Linux functionality without compromising the rest of the requirements, ensuring compatibility with existing software solutions and minimizing the need for additional development efforts. Furthermore, the system should be able to virtualize multiple Linux guests to accommodate dividing functionality in separate VMs.
- **Scalable system architecture**:
  - **Fine-grained decomposition**: The system shall allow to decompose critical functionality into sufficiently small components. Non-critical components should have more flexibility with their Trusted Computing Base (TCB), allowing for a good functionality-security trade-off.
  - **Support for POLA**: The system shall provide an access control mechanism that allows to approximate POLA - all components should only have access to resources or functionality necessary for their proper function, preventing unauthorized access and minimizing the potential for security breaches.
  - **Dynamic system**: The system shall support run-time reconfiguration; i.e., allow for the creation and destruction of components, including Linux VMs, based on current system needs.
- **Security Services for Linux Guests**: The system shall be able to provide an isolated execution environment that can provide security services to the virtualized Linux comparable to the Arm TrustZone security solution, ensuring that Linux guests are adequately protected against potential security threats.
- **Arm TrustZone compatibility**: The system shall be able to reuse existing Arm TrustZone software solutions in its isolated execution environment to provide security services to the Linux guest, ensuring compatibility with existing security infrastructures and minimizing the need for additional development efforts.

B. Contribution

In this paper, we propose the **TEEVseL4** system architecture that was designed to meet all the set requirements (§ I-A). We built on an seL4-based virtualization system, which we extend with the ability to provide a TrustZone-compatible security extension to its virtualized guests. In particular,

- We propose **TEEVseL4**, a TrustZone-compatible virtualization system leveraging the strengths of the seL4 microkernel that can provide security services to the Linux guests by virtualizing a mature and widely-used TEE OS, Open Portable Trusted Execution Environment (OP-TEE) [17]. Also, we report on our prototypical implementation of TEEVseL4 (§ III).
- We empirically evaluate our implementation of TEEVseL4 and compare its performance to a TEE on a non-virtualized platform. Our analysis shows that although our solution introduces a small performance overhead § IV, this degradation does not significantly impair its usability. Also, we assess how TEEVseL4 meets the set requirements.
- We demonstrate the functionality of the proposed system and its compatibility with existing security solutions by running a firmware Trusted Platform Module (fTPM) and leveraging it to execute a remote attestation of software example (§ IV).
- We compare TEEVseL4 to other approaches seeking to improve the security of Linux by system architecture modifications (§ V).

II. Background

A. Trusted Execution Environments

OSs reduce the attack surface of the system using process isolation to limit the attack-surface to individual processes. However, with growing complexity, OSs themselves become vulnerable to attacks [6]. Thus, as shown in Fig. 2, it becomes crucial to isolate the security-critical components from the untrusted OS [6]. This problem led to efforts in providing an isolated environment intended for secure execution of sensitive code [4]. The term *Trusted Execution Environment* (TEE) was first coined by GlobalPlatform in an attempt to standardize this isolated execution environment [4]. A TEE is a separate and isolated execution environment running alongside the main

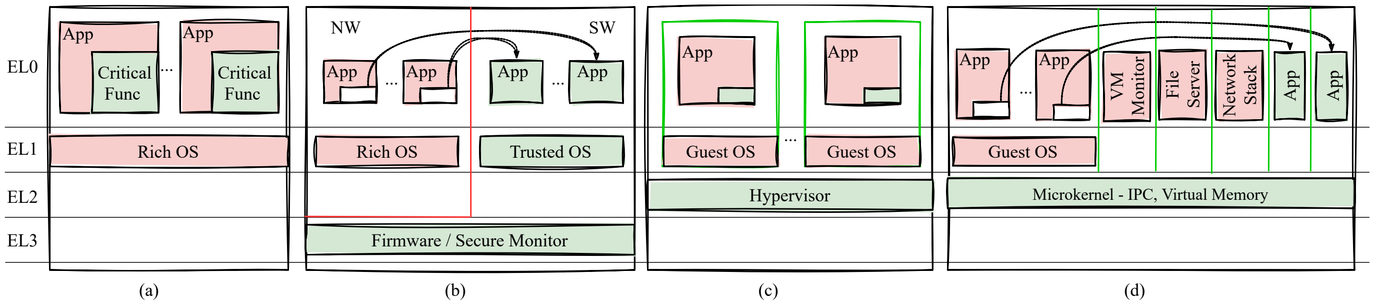


Fig. 2: General-purpose OSs have critical functionality that needs to be isolated from the rest of the untrusted OS (a) and multiple technologies can be used to address this like Arm TrustZone (b), virtualization (c) or microkernels (d) to provide systems with different security properties. Even though, these solutions have been laid out according to Arm Exception Levels (§ II-B), TrustZone is the only exclusively Arm-based solution.

operating system. A TEE is responsible for the protection of its assets against software attacks and it is possible to implement it using various technologies that provide different security properties [4]. Even though the world of TEEs is heterogeneous and the term TEE has multiple definitions, the provided definition is good enough as a basis for exploring the world of TEEs.

B. *Arm TrustZone*

Arm TrustZone was first released in 2004 and has, since then, provided security primitives for accommodating TEEs [5]. As shown in Fig. 2b, the core idea of Arm TrustZone is partitioning the system into two worlds: NW and SW, allowing for strong isolation between the untrusted and trusted components [5]. The trusted, security-sensitive components are supposed to be moved to the SW, and the untrusted components remain in the NW The interaction between the two worlds is mediated by the highest privileged system component, the Secure Monitor [5]. To achieve this, TrustZone introduces the Non-Secure (NS) bit in the Secure Configuration Register (SCR) of the processor. The value of this register is propagated throughout the system including the memory, caches and bus controllers. The NS bit marks whether the current execution is done in the secure (NS=0) or non-secure (NS=1) world [5].

The Arm architecture defines four exception levels (EL): (1) **EL0**: user/application level. (2) **EL1**: (OS) kernel level. (3) **EL2**: optional, hypervisor level. (4) **EL3**: optional, Secure Monitor level. Arm TrustZone adds two flavors to EL0 and EL1: (i) **S-EL0**: secure user/application level. (ii) **S-EL1**: secure (OS) kernel level. The Secure Monitor (running in EL3) facilitates transitions between the two worlds by updating the security status, NS-bit in SCR_EL3 [5]. A world switch is initiated by executing the Secure Monitor Call (SMC) instruction in (S-)EL1, which transfers control to the Secure Monitor in EL3 [5]. Many systems use Arm's Arm Trusted Firmware (ATF), which includes a Secure Monitor implementation, in addition to other system-level services such as Power State Coordination Interface (PSCI) [18] and Software Delegated Exception Interface (SDEI) [19]. The access to firmware services involves SMC calls, which are governed by the SMC Calling Convention (SMCCC) [20]

C. *TEE Kernels*

As shown in Fig. 2b, software running in the SW is called a trusted kernel or trusted OS and is responsible for providing secure services and allowing the execution of security-sensitive applications. However, this trusted kernel is not a part of ATF, but is left to the system provider to implement. There are many options for a trusted kernel, from closed-source proprietary solutions like Samsung KNOX or Qualcomm QSEOS, to open-source trusted kernels like OP-TEE, Open-TEE, Trusty, Andix OS, Genode etc. While systematic overviews of different properties are available [5], in this paper we will only provide the reasoning for our choice, OP-TEE, which is an open-source, GlobalPlatform compliant TEE kernel designed to run in parallel with a REE OS while leveraging Arm TrustZone hardware isolation mechanisms [17] [21]. OP-TEE has a small TCB with a suitable functionality set. While this is true for other GlobalPlatform compliant TEE kernels like Trusty and Open-TEE, OP-TEE is the only TEE kernel with integrated support in both mainline Linux and ATF. Furthermore, OP-TEE was designed with portability in mind and it does not rely on Arm TrustZone functionality directly which enables OP-TEE to use isolation providers different than Arm TrustZone. OP-TEE consists of three main components [17]:

- OP-TEE OS: the actual TEE kernel running at S-EL1 responsible for providing isolation between TAs, managing exceptions, shared memory allocation, providing secure storage and cryptography primitives.
- OP-TEE Client: the user-space framework consisting of the TEE Supplicant and the OP-TEE client library, running at EL0, that enables interaction between user-space untrusted applications and the OP-TEE Driver.
- OP-TEE driver: a crucial component, running at EL1, responsible for communication between the user-space apps and the TEE. OP-TEE driver forwards the user-space requests to the TEE and collects TEE requests and makes them available to the user-space.

D. *Virtualization*

Virtualization is a highly researched topic with different goals for various use-cases. In cloud computing infrastructure virtualization is an established way of running multiple (guest) operating systems that can be used for high availability of workloads, workload balancing, sandboxing applications that can interfere with the rest of the underlying machine [22]. Recently, in embedded devices, virtualization is researched and adopted as a security measure. Virtualization can provide spatial and temporal isolation for different processes and functions by separating (decomposing) the system into isolated VM-based execution environments, as shown in Fig. 2c. The software entity responsible for decoupling the virtualized OS from hardware, isolating it from other software components, scheduling its execution alongside other system components is called a hypervisor [22]. In general, hypervisors are divided into two types [22] [23]:

- *Standalone or Type 1 hypervisor*: hypervisor that runs directly on system hardware and completely controls all system hardware and resources (e.g., Xen, seL4).
- *Hosted or Type 2 hypervisor*: hypervisor that runs as a part of an OS that completely controls all system hardware and resources and the hypervisor (e.g., QEMU, Oracle VM VirtualBox).

E. Microkernels

Microkernels are operating system kernels that differ from more common general-purpose monolithic kernels (e.g., Linux) in that they provide only basic functionality like address-space based isolation, threads and thread scheduling and IPC [11] [12]. In comparison to monolithic kernels, microkernels cast out all other operating system features from privileged kernel-space to unprivileged user-space. With this design, microkernels drastically reduce their TCB and the attack-surface of the privileged code, while allowing the rest of the OS to be tailored for the target application [14]. Furthermore, because microkernels enforce component isolation, units like drivers, file systems and network stacks that are integral parts of monolithic kernels can be isolated into separate components that communicate through the microkernel.

Even though, microkernels offer a quality base for constructing a safety and security-critical oriented decomposed system, they offer no direct support for the features removed from the kernel-space like device drivers, file-systems and network stacks. Since they are needed for a general-purpose OS (as shown in Fig. 2d), many modern microkernels use virtualization to solve this problem as they can be used as microkernel-based hypervisors [14] [10]. This allows the system to leverage the functionality of a virtualized general-purpose OS (e.g., Linux) as just one of the isolated components in the system.

F. Principle of Least Authority (POLA)

A system that can prevent a failure of one component from influencing the behavior of other system components is desirable in industries with security and safety-critical applications like the automotive industry [11]. To achieve this property, it is necessary to ensure that the components are truly isolated and independent from each other by designing the system with the POLA in mind [24]. If a system is designed in accordance with POLA, every component has only the minimum set of privileges necessary to fulfill its function [1]. Capability-based access control models are a strong way to achieve a system that follows POLA, through its use of capabilities-forge-proof references to kernel resources that also contain a component's access right to those resources [1] [14]. This capability access control-based systems allow for a fine-grained-access control of kernel resources allowing for the creation of a system that enforces POLA [1] [14].

G. seL4

Part of the L4 family, seL4 is a third-generation microkernel, designed with security and safety in mind [14] [12]. seL4 has a number of state-of-the-art characteristics that make it particularly suitable as the base for complex systems like its minimality, typical for L4-influenced microkernels, it only provides essential features that cannot be implemented outside the kernel without security complications: threading, user-space address-space management, scheduling, IPC. Furthermore, seL4 comes with an access-control system based on object capabilities, support for policy-free kernel memory management and support for common CPU architectures, particularly ARM v8-A. In addition, seL4 offers class-leading performance of the critical IPC, both synchronous and asynchronous and support for mixed-criticality systems (MCS), allowing for better scheduling control than fixed priorities alone. seL4's most distinctive feature is a formal correctness proof that shows that the implementation conforms to a formal specification. This proof guarantees that a wide range of common implementation defects, such as null-pointer dereferencing, is impossible. Lastly, seL4 can function as a hypervisor to provide code and functionality reuse from general-purpose OSs like Linux. In hypervisor mode, seL4 runs in EL2 while the Linux guest EL1 native and Linux apps execute in EL0 as shown in Fig. 2d.

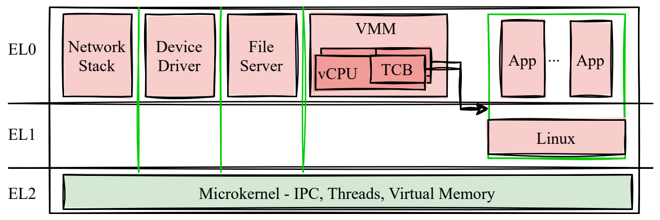


Fig. 3: seL4 Base system capable of virtualizing Linux as unprivileged isolated component alongside isolated native apps. In an seL4, the VM running is represented by a VSpace (represented with green lines), associated with a vCPU and a TCB (association represented with a directional line).

III. Approach

We leverage seL4 as a microkernel-based hypervisor, which serves as a scalable and flexible base system that can dynamically virtualize multiple Linux VMs in accordance with POLA. Before discussing the design of our contribution, we will first provide an overview of the base system architecture we contribute to.

A. Base System

seL4 can function as a microkernel-based hypervisor, virtualizing Linux guests as unprivileged, isolated component to enable code and functionality reuse. True to the microkernel principles, seL4 itself does not provide a fully-fledged VM but only primitives that allow the construction of a VM. To that end, seL4 provides two object types: virtual address space (VSpace) and a vCPU execution abstraction as shown in Fig. 3. An seL4 VSpace is used to provide the memory environment for a thread (seL4 Thread Control Object (TCB)), in that it provides a container into which memory objects can be placed. A vCPU extends the user-level context provided by a thread with the privileged execution context used by the guest kernel. When seL4 dispatches a thread with an associated vCPU, which itself is linked to a VSpace, it resumes execution in the VM context instead of a user context. On a fault, e.g., as a result of an access to non-mapped address in the VSpace or an exception generating instruction like an SMC, seL4 stops the thread, synthesizes a fault IPC, and delivers it to a fault handler. After handling the fault, the fault handler resumes the vCPU execution.

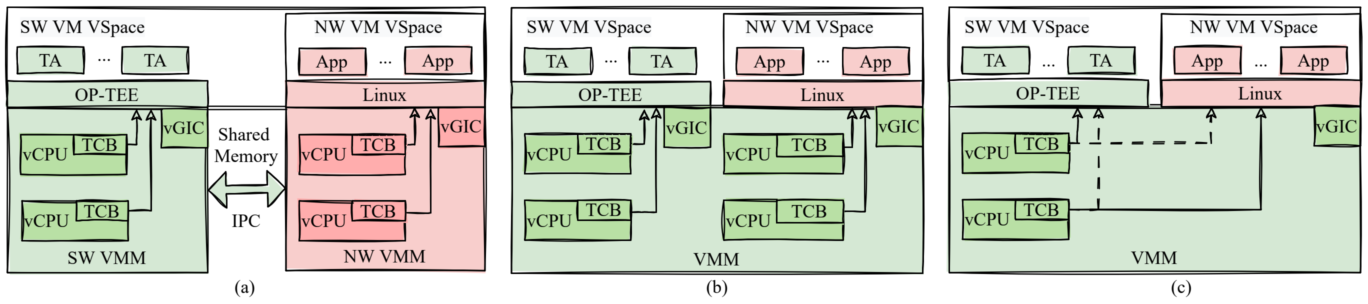


Fig. 4: Three possible design choices for providing a virtualized and isolated SW: a) using two different VM and VMMs and their seL4 infrastructure; b) Sharing one VMM and using separate VMs and interrupt controllers (vGIC); c) Sharing both the VM and VMM, providing separations through different VSpaces. The directional lines in the figure represent the VSpace association with a VM vCPU with a full line representing current use, and dashed line representing association but not current use.

A VM is managed by a so-called Virtual Machine Monitor (VMM), a regular user-level component without special privileges. The VMM activities can be broadly categorized as follows

- *setup*: allocating resources such as RAM, initializing the VM, loading the guest OS image, device initialization and mapping e.g., guest interrupt controller (vGIC [25]).
- *run-time management*: handling guest faults arising from the interaction with virtual devices, implementing virtual devices, injecting virtual interrupts.

1) Virtual Machine Setup: During the VM setup, the VMM creates a VM child process, allocates the seL4 TCB and vCPU objects based on the specified number of vCPUs for that machine. The next step is to announce the available devices to the guest and this is done through the use of Device Tree Blobs (DTBs). DTBs are essentially a list of device nodes available in the system with each node carrying information on the type of the device, its location in memory and any other device-specific information. The VMM generates the DTB matching to the set of devices it initialized and loads it into the guest VSpace. The last step to configure the VM is to configure the primary vCPU entry-point and to set the correct initial vCPU context (platform and OS specific). To start the VM, VMM starts the primary vCPU and signals to seL4 that it should switch the execution to the VM vCPU. The initial boot process is always done on the primary (v)CPU and the guest OS is responsible to start all other vCPUs as specified in the DTB.

B. Virtualizing OP-TEE

Our goal is to leverage our existing seL4 setup and extend it with OP-TEE security services. To virtualize OP-TEE, we need to add the Arm TrustZone functionality to our system. For a TZ-compatible environment, two issues need to be addressed:

1) In TrustZone, the NW (Linux OP-TEE driver in the Linux kernel) and the SW (OP-TEE OS) cannot invoke services directly. Instead, the need to pass through the EL3 Secure Monitor facilitated SMC instructions, that trap into EL3. The Secure Monitor then ensures that the execution resumes at an appropriate entry-point in the other world.
2) In the process of a TrustZone world switch, the memory access permissions have to be reconfigured. When executing in the SW, all memory is accessible. Execution in the NW cannot access the memory exclusively assigned to the SW.

The first design choice was how much of the VM components should be shared between the SW and NW. Several design paths were to be considered:

1) Complete separation of the SW and NW into separate VMs with separate VMMs. This approach could be imagined as running two VM and two VMM processes. The SW VM can map the entirety of the NW memory and both VMMs could communicate through shared memory or seL4 IPC to ensure context switching and SMC handling (Fig. 4a).
2) Semi-complete separation of the SW and NW into separate VMs with a single VMM. This approach could be imagined as running two VM in one VMM process. Running two VMs from a single VMM means that we would provide two sets of vCPUs with their matching TCBs, interrupt controllers and memory devices. The communication between these VMs would be done through the shared VMM (Fig. 4b).
3) Minimal separation of the SW and NW with one VM and one VMM. This approach is the closest to the Arm TrustZone design. The worlds share CPUs but each CPU can only be used by one of the worlds at the same time. However, the available mechanisms in this option (one VM and one VMM) do not provide isolation of these two worlds, thus this option has to include an additional solution for SW/NW isolation (Fig. 4c).

Option 1 allows the most code reuse of the existing base system at the cost of the most complex transitions between the SW and NW. Both option 1 and 2 use separate interrupt controllers, which offers easy separation of interrupt handling but raises the problem of interrupt controller synchronization and their effect on interrupt latencies. Option 3 is the most similar to Arm TrustZone and shows the most promise for integrating Arm TrustZone ecosystem solutions like OP-TEE. Option 3, however, cannot leverage existing mechanisms for world separation. The standard way of providing component isolation in seL4 is through the use of VSpaces. By adding a second SW VSpace to Option 3, as shown in (Fig. 4c), we get an Arm TrustZone-like solution.

1) World Isolation: seL4 provides a way for providing component isolation through the use of VSpaces. By providing a second VSpace to our VM and mapping all of the VM components to SW and NW VSpaces, we get memory access privileges similar to that of Arm TrustZone. The NW VSpace remains unchanged, it contains allocations for the guest OS images, DTBs, devices etc. However, the new SW VSpace is privileged and has access to the entire system. We achieve

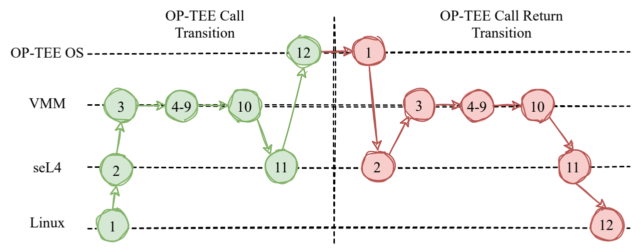


Fig. 5: Representation of the steps done to handle an SMC issued by the NW to request execution in the SW (depicted green) and the same steps when the SW requests the return of execution to the NW (depicted red).

this by allocating and mapping a new memory region for the OP-TEE OS to the SW VSpace. We consider this part of the SW VSpace secure memory. Furthermore, we expand the SW VSpace with access to the non-secure memory of the NW by mapping in all of the memory that is contained in the NW VSpace. This way, we get the asymmetric view of the memory that matches the Arm TrustZone architecture where the SW has access to the whole VM memory, while the NW has access to only the non-secure parts. Lastly, while TrustZone uses cache tagging to prevent access by the NW to SW data that is held in caches, in our system this is not needed as the hypervisor-controlled second-stage translation of guest physical addresses enforces this isolation for both cached and uncached data.

2) vCPU World Transition: We can replicate the Secure Monitor world switch by saving all of the vCPU/TCB state on each transition. Since seL4 maintains and restores the vCPU stage, we only need to leverage four seL4 syscalls for saving and restoring the vCPU context. *seL4_ARM_VCPU_ReadReg* and *seL4_ARM_VCPU_WriteReg* for manipulating the vCPU system registers and *seL4_TCB_ReadRegisters* and *seL4_TCB_WriteRegisters* for manipulating the vCPU (TCB) general-purpose registers. To reduce the number of context switches between the kernel and the VMM, we followed the example of seL4 syscalls for managing the sel4 TCB registers and extended the syscalls for managing the vCPU registers with the ability to manage multiple registers using a single call. With regards to the seL4 correctness proof, this change is negligible and could be easily incorporated in future versions of seL4. On each transition, we use these syscalls to save the current and load the previous vCPU state into the seL4 vCPU object with the updated entrypoint address and any modifications to other vCPU registers necessary.

3) SMC Handling: During run-time, the VMM handles all guest faults: traps registered during initialization of the VM (e.g., vGIC configuration accesses) or exceptions (e.g., virtual interrupt injection or trapping instructions like SMCs). Since transitions between the SW/NW are triggered by SMCs, we need to replicate the Arm TrustZone interface in the VMM SMC handler. As shown in Fig. 5, during guest VM execution, a guest can request a transition between SW/NW by executing an SMC instruction ①. The hypervisor traps SMCs into EL2 by setting the *HCR_EL2.TSC* configuration register. Then, the SMC call will be delivered to seL4 ②. seL4 forwards the fault as an IPC to the VMM together with the information about the fault ②. Upon receiving the fault, the VMM decodes this fault according to the SMCCC [20]. SMCs that come from or target the SW are delivered to the OP-TEE SMC handler ③. The handler, then, checks which VSpace was attached to the vCPU during the last execution to determine whether the source was SW or NW ④ and to determine whether OP-TEE was successfully initialized (more on this in § III-B4) ⑤. Depending on the result of previous two checks, the SMC handler has to handle four cases ⑥:

1) the SMC source was the NW and OP-TEE was initialized. In this case, the SMC handler needs to check whether the SMC is of type fast or yielding [20] and set the SW entrypoint to the matching OP-TEE exception vector. Registers X0-X7 are passed along to OP-TEE as call parameters.
2) the SMC source was the SW and OP-TEE was initialized. In this case, OP-TEE is returning from an operation and the NW entry-point does not change from its previous vCPU state. Register X0 - X4 (some cases just X0) are passed along to NW as return parameters.
3) the SMC source was the SW and OP-TEE was not initialized. In this case, OP-TEE is returning from its initialization and the NW entry-point is the Linux kernel entry-point. Register X0 contains the value of the OP-TEE exception vector table and is saved for later use.
4) Invalid SMC.

In case of invalid SMC, only the Program Counter (PC) is incremented; otherwise, the SMC handler executes the context switch (§ III-B2) by ⑦:

1) saving the current vCPU state into the appropriate (SW or NW) data structure and increments its PC (to step over the exception instruction).
2) restoring the saved target vCPU state from the appropriate (SW or NW) data structure and updating the PC and registers based on the branch of execution it took.

Lastly, the SMC handler resumes the vCPU, marking the fault handled ⑧ and returns information to the VMM which VSpace is to be run after VM is resumed ⑨. The VMM sets the proper VSpace to the TCB and signals to seL4 to schedule the VM vCPU for execution ⑩. seL4 then loads the modified vCPU state into the pCPU and executes the VM with the target VSpace ⑪. Fig. 5 shows this process when the SMC is issued in the NW ① to execute an OP-TEE call in the SW ⑫ and the same process when OP-TEE uses an SMC to signal success and request a return to the NW ① - ⑫.

4) OP-TEE Boot Procedure: Now that we designed a mechanism to transition between the worlds, we focus on virtualizing the OP-TEE OS alongside a Linux guest. While the first phase was concerned about the world transition process, in this phase we focused on ensuring the correct functionality of our SW/NW interface. For this purpose, we allocated and mapped a memory area for OP-TEE in the VM secure memory, just before the Linux image, as shown in Fig. 6, and load the OP-TEE image as part of the VM setup. The last step was to update the Linux DTB specifying that OP-TEE is available in the system. As ATF boots the SW software before NW software, we do the same, by setting our VM primary vCPU entrypoint to the OP-TEE entrypoint. To reduce the adaption effort, we selected the QEMU Armv8

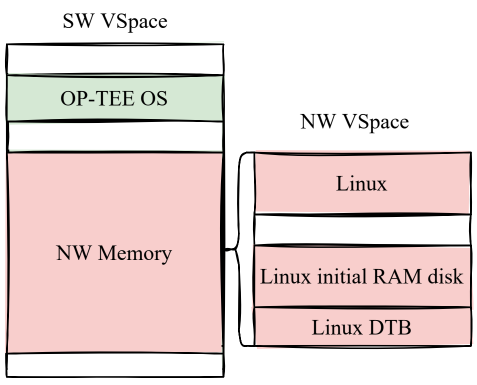


Fig. 6: The SW and NW VSpaces populated. The NW VSpace contains the Linux image, initial RAM Disk and DTB, while the SW VSpace contains everything mapped to the NW VSpace with the addition of OP-TEE OS.

platform, which due to the platform's simplicity, required minimal changes to its configuration:

- Changing the OP-TEE secure RAM location and size to match our design.
- Changing the default debug serial device.
- Changing the default address of the interrupt controller.
- Using the Arm GICv2 controller instead of Arm GICv3 (Explained in more detail in § III-B5).
- Disabling the OP-TEE feature that allows OP-TEE to announce itself in the NW DTB, as we are already doing that.

Configuration changes similar to these are necessary for porting OP-TEE to any new platform and no other changes - in the architecture-dependent part or otherwise - were necessary. After OP-TEE has successfully booted, it issues an SMC with its exception vector table address and we are ready to boot Linux. During boot process, Linux will read the OP-TEE device node from the provided DTB, load the OP-TEE driver and execute a handshake with OP-TEE to finish the initialization process. If Linux has the entire OP-TEE framework installed (OP-TEE Client, OP-TEE supplicant and optional OP-TEE TAs), the virtualized Linux has complete access to OP-TEE functionality with changes only made to OP-TEE QEMU configuration files of 29 insertions and 7 deletions. The changes to our base system were in total 2002 insertions and 491 deletions.

5) Interrupt Handling: The last part of our design is related to interrupt handling. As secure devices are an optional feature for the SW, we consider them out of scope for our paper. Since we configured OP-TEE to use the Arm GICv2 interface of the interrupt controller, OP-TEE expects secure interrupts to be delivered only as Fast Interrupt Requests (FIQs) [25] which effectively disables interrupts for the SW. We can do this because:

1) GICv3 (we are running in the system) fully supports software using it as GICv2 [25].
2) Since we are running in the NW only, no FIQs will ever be delivered to the VM, making their handling unnecessary.

This ensures, that all devices shared by the SW and NW (e.g., debug serial console) have their interrupts handled in the NW. If SW-only devices are assigned to OP-TEE it runs correctly without ever receiving an interrupt. This limitation can be solved, by making changes to the virtual interrupt controller (vGICv3) to handle secure interrupts based on the VSpace currently associated with the vCPU. However, this is out-of-scope for this paper.

IV. Evaluation

We deployed our system on the NVIDIA Jetson AGX Orin Developer Kit with 12-core Arm Cortex-A78AE v8.2 64 bit CPU and 32 GB of memory and we are comparing our system against a native Arm TrustZone OP-TEE TEE. Both systems are running on a single CPU core with a 2 GHz clock frequency with a Linaro (maintainer of OP-TEE) Linux 5.14 with accompanying OP-TEE framework (OP-TEE Linux Driver, Supplicant, Client and Test Framework). Our system is running OP-TEE OS 3.15, while the native system is running NVIDIA Tegra OP-TEE and ATF ports. The main difference between these versions of OP-TEE is that the NVIDIA OP-TEE has reworked memory management mechanisms to better suit the hardware and introduced a closed-source cryptography library with access to hardware cryptographic-accelerators. Reusing this library would eliminate any cryptography operation overhead in our system. However, since NVIDIA does not offer documentation on how such changes were made to OP-TEE and does not support running OP-TEE without these changes, using an identical OP-TEE version proved to be too challenging. Nevertheless, we believe that the results presented in the rest of the section provide a meaningful performance comparison.

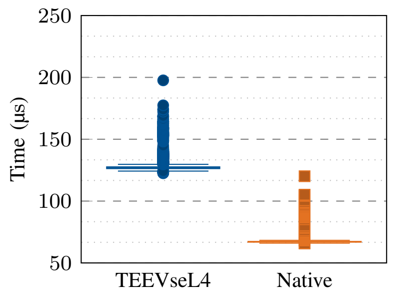
To evaluate the performance of our system, we designed five performance benchmarks: three micro-benchmarks and two high-level benchmarks. These benchmarks were used to compare our system's performance to that of the native system.

A. Micro-benchmarks

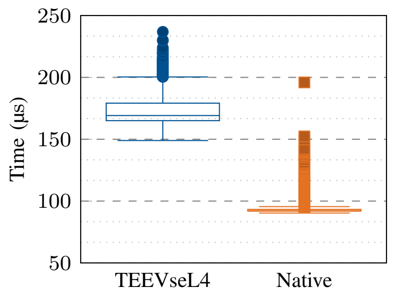
To evaluate the performance of discrete parts of our system, we designed three micro-benchmarks:

1) **SW/NW transition micro-benchmark:** we implemented a TA with an empty call. When calling this TA, the only operation performed in the SW is finding the appropriate call handler and immediately returning a success. We used this TA to measure and compare the time it takes to transition between the SW and NW on both systems.
2) **AES-CBC encryption micro-benchmark:** we leveraged an existing TA from the OP-TEE Test framework with a call that encrypts a 1024 B chunk of data using AES-CBC with 256 bit key. We used this TA to measure and compare the performance of cryptographic operations on both systems.
3) **Prime number calculation micro-benchmark:** we implemented a TA with a call that finds the prime numbers among the first 10 000 positive numbers using the Fibonacci Trial Division algorithm. We used this TA to measure and compare the performance of a CPU-only workload in the SW on both systems.

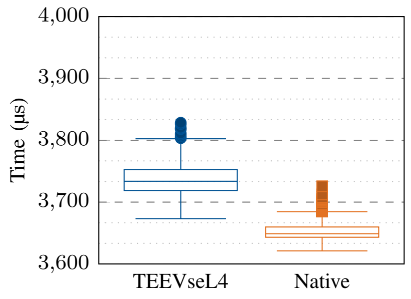
Each micro-benchmark was executed 5000 times on both systems and the results are shown in Fig. 7. The prime number arithmetic micro-benchmark (Fig. 7c) demonstrates only a 2 % performance overhead on average when running on our system compared to a native system, while both the SW/NW transition (Fig. 7a) and AES-CBC encryption (Fig. 7b) micro-benchmarks show an average overhead of 90 %. When comparing the percentage of performance overhead for the different micro-benchmarks, it is not surprising



(a) SW/NW transition

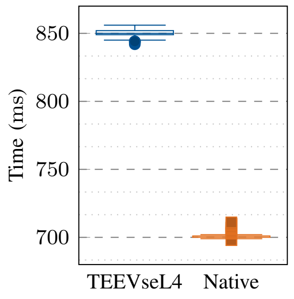


(b) AES-CBC encryption

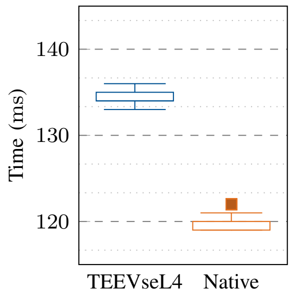


(c) Prime number arithmetic

Fig. 7: Micro-benchmark results for (a) SW/NW transition, (b) AES-CBC encryption, and (c) prim number arithmetic, comparing TEEVseL4 and native TEE system performance.



(a) Write



(b) Read

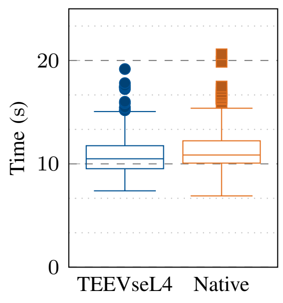
Fig. 8: Performance comparison of secure storage write (a) and read (b) of 1 MB using TEEVseL4 and native system.

that the SW/NW transitioning micro-benchmark exhibits the highest overhead. This is due to the four context switches involved in transitioning from the SW to the NW (compared to the 2 in the native system), as illustrated in Figure 5, as well as the seL4 syscalls for vCPU management and VSpace swapping. In contrast, the prime number arithmetic micro-benchmark, despite having the longest total execution time, shows the smallest percentage of performance overhead when compared to the native system. This is because the performance overhead of our system is mostly confined to the SW/NW transitions. This can be seen by subtracting the average transition micro-benchmark execution time (which is part of every micro-benchmark) from the average prime micro-benchmark time for both systems. These results show that without the transition overheads, TEEVseL4 and the native system have comparable results. The same argument can be applied to the AES-CBC micro-benchmark. However, we observed an additional 60 % overhead due to the fact that the NVIDIA OP-TEE port utilizes hardware cryptographic accelerators for its cryptographic functions. Overall, the micro-benchmarks demonstrate that the SW/NW transitions are the only significant source of overhead. This overhead could be reduced, if not eliminated, by relocating the SW SMC handler from the user-space VMM component to the seL4 hypervisor. By doing so, the number of context switches will be reduced to 2 (NW-seL4-SW), and the need for syscalls will be eliminated.

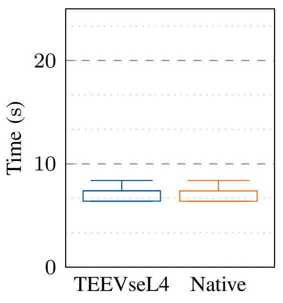
B. High-level benchmarks

To evaluate the performance of our system on real world applications, we designed two high-level benchmarks.

1) **Secure Storage benchmarks:** we leveraged an existing OP-TEE test framework TA that reads or writes predefined data of arbitrary size. We used this TA to measure and compare the performance of OP-TEE secure storage in the SW on both systems. We measured the time to execute both read and write operations of 1 MB that was written/read in chunks of 1 kB. We repeated the measurement 500 times. The results shown in Fig. 8 indicate that our system incurs an average overhead of 12 % for read operations and 20 % for write operations. Upon evaluating a well-known TEE use-case (secure storage), it becomes clear that the transition overhead visible in the micro-benchmark has a reduced impact on a real-world application as the average performance overhead drops from 90 % to 12-20 %.
This benchmark, like the cryptographic micro-benchmark, is skewed in favor of the native system because OP-TEE Secure Storage uses AES-CBC for secure storage file encryption, handled by the superior crypto libraries in the native system. AES-CBC encryption and decryption may also explain the difference in read/write overhead. The encryption (write) algorithm of AES-CBC is sequential, whereas the decryption (read) algorithm is parallelizable, so one reason for this discrepancy could be that the encryption process is more efficient than the decryption process. We cannot confirm this, however, since we do not have transparency in the cryptographic libraries the native system uses. Nonetheless, this means that our actual performance overhead is even smaller than our current numbers.
2) **Remote Attestation benchmarks:** we utilized two components: (i) TPM2Tools demo Remote Attestation (RA) project, used to showcase how to leverage TPM2Tools to provide remote attestation of software [26]. (ii) Microsoft's demo implementation of a Firmware Trusted Platform Module (fTPM) implemented as an OP-TEE TA to showcase how to use OP-TEE to provide a software only TPM [27]. TPM2Tools [28] is a collection of TPM management tools based on the Trusted Computing Group (TCG) TPM2 Library [29] that can interact with a TPM using its Linux driver, if available on the system. We provide a TPM to our system using Microsoft's demo fTPM project implemented as a TA and their fTPM driver implemented as a part of mainline Linux. The TPM2Tools RA demo executes in two steps: (i) a registration process where the OS under attestation proves its identity, and (ii) a verification process where the OS under attestation proves its data integrity. We timed the execution of both of these steps 500 times and the results



(a) Registration



(b) Verification

Fig. 9: Performance Comparison of Remote Attestation Registration (a) and Remote Attestation Registration Verification (b) using TEEVseL4 and native system.

are available in Fig. 9. When evaluating a complex TEE feature like the RA benchmark, the transition overhead becomes completely negligible as in both cases of our RA benchmark, we see no significant difference in the performance of the two systems.

V. Related Work

Arm TrustZone hardware extensions, alongside the ATF, Secure Monitor and OP-TEE OS, provide a versatile and well-integrated security solution for delivering security services to its Linux NW. However, the design of two execution environments isolated by hardware is a limitation when dealing with a multi-guest OS NW and a security concern when dealing with third-party TAs. Overcoming this limitation is a widely researched topic in both academic works [31] [32] [8] [7] [9] and industry solutions [30].

OP-TEE OS introduced a shared ”Nexus” component that is responsible for many functionalities required by all TEEs, such as starting/stopping virtualized TEEs, SMC communication, memory management, thread creation, among others. The Nexus component relies on the support of the NW hypervisor (Xen Hypervisor) to virtualize multiple TEEs in the SW to secure the multiple virtualized OSs in the NW [30]. However, the use of a commodity hypervisor like Xen, that comes with a complex TCB with a large attack surface inside its Dom0 management VM, the additional complexity of the SW with its Nexus component, and the lack of POLA compliance, makes this system unsuitable for our requirements.

TEEv [31] aims to provide reliable isolation of TAs in different paravirtualized TEEs in the SW. Instead of relying on the NW hypervisor, TEEv introduced a hypervisor in the SW for paravirtualization of TEEs using the concept of *same privilege isolation* [31]. By stripping the TEEs of any privileged instructions that could overrule the privileged trusted hypervisor and replacing them with calls to the trusted hypervisor, TEEv ensures that its trusted hypervisor becomes a privileged component in the SW [31]. Sanctuary [32], which is another solution for decomposing the SW, leverages a SW trusted kernel (OP-TEE) and the TrustZone Address Space Controller (TZASC) for temporary reservation of a single CPU core for the execution of an isolated TEE [32]. While Sanctuary does not use virtualization for isolating TEEs, it is limited in the number of TEEs that can run at the same time by the number of CPU cores that can be reserved for executing these TEEs. Even though Sanctuary and TEEv offer reliable isolation of TEEs within the SW, neither TEEv nor Sanctuary provide multiple TEEs for a NW that virtualizes multiple OSs or designs their system in accordance with POLA.

vTZ [8] is another system that relies on the NW hypervisor to virtualize multiple TEEs in the NW. Unlike OP-TEE, vTZ addresses the concerns that arise when relying on a NW hypervisor with a large attack surface by leveraging Arm TrustZone to secure components that monitor the NW hypervisor [8]. While vTZ does not address the issue of reliable isolation within a single TEE, it allows for virtualizing multiple OSs in the NW while providing each guest with access to their own isolated TEEs. A vTZ-based system can be customized to ensure that TAs use separate VMs thus confining any vulnerable TA to an isolated TEE. However, the interaction between these VMs would not be backed by a system that offers fine-grained component isolation based on POLA. Furthermore, vTZ cannot offer any component smaller than a VM which raises concerns about the scalability of its TCB.

Using purely trusted hypervisor solutions to provide TEEs to virtualized Linux guests was addressed by many works such as [7] [9]. Bao [7] is a statically partitioned embedded hypervisor that focuses on virtualizing small safety-critical embedded applications alongside virtualized general-purpose OSs. Bao provides security enclaves to the virtualized general-purpose OSs for execution of TAs. These enclaves are created by temporarily donating the virtualized guest resources (memory and vCPU) to the Bao hypervisor, which in turn, ensures secure execution of the TA. Even though Bao offers performance improvements compared to existing Arm Trust-Zone security solutions like OP-TEE, it is not compatible with them and its static architecture makes it unsuitable use-cases where dynamic systems are required. Mirzamohammadi et al. [9] proposed a system that can offer multiple TEEs to a single virtualized Linux guest allowing for compartmentalization of third-party TAs into separate TEEs. Furthermore, the authors showed that their system allows for comparable or even better performance than native Arm TrustZone, while retaining compatibility with Arm TrustZone solutions like OP-TEE. However, the proposed system is not compliant with POLA. Also, the performance benefits apply only when the system virtualizes a single general-purpose OS. Finally, their design does not support virtualization of multiple OSs in the NW.

Comparative Analysis: Table I presents a comparison between our proposal, TEEVsel4, and some of the solutions presented above. The comparison is based on the requirements introduced in § I-A. The comparison shows that TEEVseL4 has the capability to virtualize multiple Linux guests, which is not the primary focus of other solutions like [31] [32] [9]. Additionally, TEEVseL4 is supported by a scalable TCB that complies with POLA and enables fine-grained decomposition, which is not available in most systems, as observed in [30] [31] [32] [8]. Although Bao [7] and vTZ [8] are considered to be strong contenders for a comparable system, vTZ lacks fine-grained component isolation based on POLA, and Bao provides a static system that is incompatible with existing TrustZone software solutions. Finally, TEEVseL4 meets all of our requirements. Although we did not specifically emphasize the last requirement, it is important to note that our system can fulfill this requirement by separating the functionality that depends on possibly vulnerable TAs into separate VMs that come with their own TEE, similar to vTZ.

TABLE I: Comparison of TEEVseL4 with other systems.

Requirement	vOP-TEE [30]	TEEv [31]	Sanctuary [32]	vTZ [8]	Bao [7]	[9]	TEEVseL4
Virtualization of multiple Linux guests	●	○	◑	●	●	○	●
Fine-grained decomposition	○	○	○	○	●	○	●
Support for POLA	○	○	○	○	●	○	●
Dynamic System	●	●	●	●	○	●	●
Security Services for Linux Guests	●	●	●	●	●	●	●
Arm TrustZone compatibility	●	●	●	●	○	●	●
Reliable component isolation in SW	○	●	●	●	●	●	●

○ Requirement is not met ◑ Requirement is not discussed but it could be partially met ● Requirement is met

VI. Conclusion

In safety-critical systems, it is crucial to isolate security-critical functionalities from general-purpose OSs due to their complexity and large attack surface. While Arm TrustZone and its compatible security software provide a robust and versatile TEE that enables such isolation, its design limits its applicability to virtualize multiple OSs. On the other hand, microkernel-based hypervisors can virtualize multiple OSs alongside isolated native applications, but they do not offer a way for re-using Arm TrustZone ecosystem software security solutions for securing their virtualized guests. In this work, we introduce TEEVseL4, which combines the advantages of Arm TrustZone and microkernel-based hypervisors by virtualizing OP-TEE, a TrustZone-based TEE OS, using the seL4 microkernel. We leverage seL4's security properties to provide security services to the multiple virtualized Linux guests in the system, thereby addressing the limitations of both Arm TrustZone and microkernel-based hypervisors. To demonstrate the compatibility of our system with existing Arm TrustZone solutions, we used TEEVseL4 to run an unmodified complex third-party TA (Microsoft's fTPM), on top of OP-TEE OS with minimal changes to its configuration. Moreover, we conducted a performance evaluation of TEEVseL4 and demonstrated that the introduced performance overhead by our system is acceptable for real-world applications, and in some cases, negligible. Lastly, we demonstrated that our system is compatible with complex third-party solutions with no porting effort. In conclusion, TEEVseL4 presents a high-quality security solution for safety and security-critical systems that require the virtualization of multiple OSs. As a future direction, we plan to enhance TEEVseL4 with secure interrupt support and secure device sharing and explore potential performance optimizations by transitioning the mechanisms to the seL4 kernel.

References

[1] S. Jero, J. Furgala, R. Pan, P. K. Gadepalli, A. Clifford, B. Ye, R. Khazan, B. C. Ward, G. Parmer, and R. Skowyra, "Practical Principle of Least Privilege for Secure Embedded Systems," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.

[2] M. M. Madden, "Challenges Using Linux as a Real-Time Operating System," in *AIAA Scitech 2019 Forum*. San Diego, California: American Institute of Aeronautics and Astronautics, Jan. 2019.

[3] GlobalPlatform, Inc., "TEE System Architecture v1.3 — GPD_SPE_009," Tech. Rep. GPD_SPE_009, May 2022.

[4] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted Execution Environment: What It is, and What It is Not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, Aug. 2015, pp. 57–64.

[5] S. Pinto and N. Santos, "Demystifying Arm TrustZone: A Comprehensive Survey," *ACM Computing Surveys*, vol. 51, no. 6, 2019.

[6] P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Trusted Execution Environments: Properties, Applications, and Challenges," *IEEE Security & Privacy*, vol. 18, no. 2, pp. 56–60, Mar. 2020.

[7] S. Pereira, J. Sousa, S. Pinto, J. Martins, and D. Cerdeira, "Bao-Enclave: Virtualization-based Enclaves for Arm," Sep. 2022.

[8] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vTZ: Virtualizing ARM TrustZone," in *USENIX security symposium*, 2017.

[9] S. Mirzamohammadi and A. A. Sani, "The Case for a Virtualization-Based Trusted Execution Environment in Mobile Devices," in *Proceedings of the 9th Asia-Pacific Workshop on Systems*, 2018.

[10] U. Steinberg and B. Kauer, "NOVA: A microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European Conference on Computer Systems*, 2010.

[11] M. Hamad, J. Schlatow, V. Prevelakis, and R. Ernst, "A communication framework for distributed access control in microkernel-based systems," in *12th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT16)*, 2016.

[12] M. Crone, "Towards attack-tolerant trusted execution environments: Secure remote attestation in the presence of side channels," Master's thesis, AaltoUniversity, 2021.

[13] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro, "Reducing TCB size by using untrusted components: Small kernels versus virtual-machine monitors," in *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, 2004.

[14] G. Heiser, "The seL4 Microkernel – An Introduction," p. 32.

[15] "The Fiasco microkernel - Overview," https://os.inf.tu-dresden.de/fiasco/.

[16] E. de Matos and M. Ahvenjärvi, "seL4 Microkernel for Virtualization Use-Cases: Potential Directions towards a Standard VMM," *Electronics*, vol. 11, no. 24, p. 4201, Jan. 2022.

[17] "About OP-TEE — OP-TEE documentation documentation," https://optee.readthedocs.io/en/latest/general/about.html.

[18] Arm Limited, "ARM Power State Coordination Interface," Tech. Rep., Jun. 2021.

[19] ——, "Software Delegated Exception Interface (SDEI)," Tech. Rep. ARM DEN 0054C, Jan. 2023.

[20] ——, "SMC Calling Convention (SMCCC)," Tech. Rep. ARM DEN 0028E, May 2022.

[21] M. M. Quaresma, "TrustZone based Attestation in SecureRuntime Verification for Embedded Systems," 2020.

[22] Arm Limited, "Learn the architecture - AArch64 virtualization," Tech. Rep. 0100-03, Jun. 2022.

[23] K. Sandström, A. Vulgarakis, M. Lindgren, and T. Nolte, "Virtualization technologies in embedded real-time systems," in *2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*, 2013.

[24] M. S. Miller, K.-P. Yee, and J. Shapiro, "Capability Myths Demolished."

[25] Arm Limited, "Learn the architecture - Arm Generic Interrupt Controller v3 and v4," Tech. Rep. 198123_0302_00_en, Dec. 2021.

[26] "Remote Attestation With Tpm2 Tools," https://tpm2-software.github.io/2020/06/12/Remote-Attestation-With-tpm2-tools.html, Jun. 2020.

[27] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten, "fTPM: A Software-only Implementation of a TPM Chip."

[28] "Home - tpm2-tools," https://tpm2-tools.readthedocs.io/en/latest/.

[29] "TPM 2.0 Library," https://trustedcomputinggroup.org/resource/tpm-library-specification/.

[30] "Virtualization — OP-TEE documentation documentation," https://optee.readthedocs.io/en/latest/architecture/virtualization.html.

[31] W. Li, Y. Xia, L. Lu, H. Chen, and B. Zang, "TEEv: Virtualizing trusted execution environments on mobile platforms," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2019.

[32] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "SANCTUARY: ARMing TrustZone with User-space Enclaves," in *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.