# VS-TEE: A Framework for Virtualizing TEEs in ARM Cloud Contexts


Matteo Zoia
University of Milan, Milan, Italy
Milan, Italy
matteo.zoia@unimi.it

Marco Cutecchia
Independent Researcher
Milan, Italy
marco.cutecchia@outlook.it

Davide Rusconi
University of Milan, Milan, Italy
Milan, Italy
davide.rusconi@unimi.it

Andrea Monzani
University of Milan, Milan, Italy
Milan, Italy
andrea.monzani@unimi.it

Mirco Picca
University of Milan, Milan, Italy
Milan, Italy
mirco.picca@unimi.it

Danilo Bruschi
University of Milan, Milan, Italy
Milan, Italy
danilo.bruschi@unimi.it

Andrea Lanzi
University of Milan, Milan, Italy
Milan, Italy
andrea.lanzi@unimi.it



## Abstract

Cloud computing processes and stores critical data, necessitating robust protections against unauthorized access. Confidential Computing (CC) technologies address this need by enabling secure computation in hardware-backed Trusted Execution Environments (TEEs). While solutions like AMD's Secure Encrypted Virtualization (SEV) provide strong protections, they remain vulnerable to attacks targeting applications within virtual machines (VMs). Similarly, the recent Armv9-A architecture introduces a promising Realm World for enhanced security, but its adoption is limited by hardware availability and upgrade constraints. ARM TrustZone, while widely supported, lacks native support for multiple isolated TEEs. In this paper we proposed framework eliminates the need for these components in the Trusted Computing Base (TCB), enabling secure integration of TEEs with VMs. It features a VS-TEE Driver for VM interaction and a VS-TEE Hypervisor for secure communication, ensuring compatibility with ARM TrustZone and OP-TEE libraries. We developed and evaluated an open-source prototype, demonstrating its effectiveness in addressing challenges like memory translation, resource management, and interoperability. Our framework enhances security for cloud environments, allowing multiple VMs to securely share TEE capabilities.



## CCS Concepts

• **Security and privacy** → **Trusted computing**; **Virtualization and security**; • **Computer systems organization** → *Cloud computing*.





*CODASPY '25, Pittsburgh, PA, USA*





ACM ISBN 979-8-4007-1476-4/2025/06
https://doi.org/10.1145/3714393.3726515

## Keywords

TEE; Trusted Execution Environments; Computational privacy; Virtualization; ARM; Cloud

**ACM Reference Format:**
Matteo Zoia, Marco Cutecchia, Davide Rusconi, Andrea Monzani, Mirco Picca, Danilo Bruschi, and Andrea Lanzi. 2025. VS-TEE: A Framework for Virtualizing TEEs in ARM Cloud Contexts. In *Proceedings of the Fifteenth ACM Conference on Data and Application Security and Privacy (CODASPY '25), June 4–6, 2025, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3714393.3726515


## 1 Introduction

Cloud computing has been widely adopted across diverse sectors, resulting in the extensive processing and storage of critical data within cloud infrastructures. Consequently, protecting these data from unauthorized access by both external attackers [10, 17], and internal entities, including cloud service providers, is imperative. Confidential Computing (CC) technologies have emerged to address this demand, offering secure computation within hardware-based, attested Trusted Execution Environments [14]. Although software mechanisms can ensure the confidentiality and integrity of data at rest and during transmission, securing data in use and facilitating remote attestation typically require hardware-based mechanisms. Major hardware manufacturers such as AMD, Intel, and ARM have incorporated functionalities that protect code execution from both the host and the cloud service provider, thus establishing a secure platform for sensitive computational processes and data management. For example, AMD introduced Secure Encrypted Virtualization (SEV) [32] to strengthen Virtual Machines memory. SEV enhances security by enabling users to encrypt a VM's memory, rendering the data within the VM unreadable to attackers even if a hypervisor is compromised. While this measure improves security, it remains inadequate when an attacker takes over an application inside the VM, as the application's memory is still freely accessible in plaintext to the application. Recently, it has been demonstrated that the embedded SPD chip within commercial DRAM modules

can be manipulated by researchers, enabling attackers to circumvent SEV protections, even the most recent SEV-SNP version from AMD [18]. In such instances, technologies such as application-level Trusted Execution Environments offer superior protection for sensitive data.

The Confidential Compute Architecture model, proposed with the Armv9-A architecture [27], addresses cloud contexts by introducing the *Realm World (Realm)*. This is a newly established environment that operates alongside older environments, being fully compatible with the Non-secure world (NS) to ensure that existing software stacks work seamlessly within the Realms. Each ARM world (Non-secure, Secure, Realm) manages its own Physical Address Space, and hardware mechanisms perform checks on every memory access using a table that monitors the physical address space of each memory granule and enforces access control policies to prevent unauthorized access. Although the new Armv9-A hardware feature offers a promising opportunity, it has limited availability in commercial devices. This limitation is due to both hardware constraints and the substantial effort required for software modification or full redevelopment and hardware upgrade. Updating production servers in cloud environments to incorporate the latest hardware can take several years, making solutions that utilize existing infrastructure extremely valuable.

The ARMv7 TrustZone architecture does not inherently support multiple isolated TEEs. To overcome this limitation, various methods for secure virtualization of the world have been proposed [13, 22, 26]. Notable among these are approaches such as OSP [12], PrivateZone [23], and vTZ [22], which leverage hardware virtualization extensions present in the Non-secure world (NS-EL2) to establish isolated environments. For example, vTZ achieves effective virtualization of ARM TrustZone, allowing multiple virtual machines to maintain isolated TEEs on a single physical machine without necessitating hardware alterations. Primarily, vTZ enhances the protection of existing TEEs by isolating Trusted Applications (TAs) within secure VMs in the Non-secure world, while also being potentially valuable in cloud settings by safeguarding against inter-TA breaches. The threat model presumes that an attacker is capable of breaching the confidentiality of one Trusted Application while leaving the security of other TAs intact.

In this context, our system adopts a more robust threat model by addressing such attacks directly within the TA, moving it into a TEE for improved isolation and security, and employing protocols to secure data during transmission, thus fulfilling confidentiality requirements. Our approach enables applications running within a virtual machine to utilize the functionalities of an external Trusted Application. Significantly, our system eliminates the necessity of having the host or hypervisor as trusted components, thus excluding them from the TCB. This exclusion markedly advances confidential computing, augmenting the security of applications deployed in cloud environments. The proposed framework comprises two crucial components: VS-TEE *Driver* and VS-TEE *Hypervisor*. The VS-TEE Driver operates within the guest virtual machine, designed to facilitate interaction with a specific Trusted Application, while the VS-TEE Hypervisor acts as a bridge, transmitting commands to the host Trusted Execution Environment. Our framework is fully compatible with the ARM TrustZone TEE implementation, ensuring that the invocation of TA functions remains unchanged. In addition, it utilizes standard OP-TEE libraries for easy integration. We have implemented and evaluated our system in various contexts, and the results affirm its effectiveness and ease of use, highlighting its value to cloud service providers. In conclusion, the research emphasizes the framework's potential to share TEE capabilities among multiple virtual machines within cloud-based environments. In particular, our paper makes the following contributions:

- We present a novel architecture along with communication protocol tailored to allow Virtual Machines to easily utilize the features offered by a Trusted Execution Environment.
- Our approach addresses the main obstacles in integrating VMs with TEE, including resource management and interoperability. Specifically, we encounter technical challenges related to memory address translation and buffer synchronization between the virtual machines (nonsecure world) and the TEE (secure world).
- We developed an open-source prototype of our suggested framework using the QEMU emulator. This prototype was tested in numerous scenarios that involved trusted computation using ARM TrustZone to assess its effectiveness. The source code of VS-TEE is available at [5].

## 2 Background

### 2.1 Trusted Execution Environment

A Trusted Execution Environment [33] is a tamper-resistant processing environment that runs on a separation kernel. It guarantees the authenticity of the executed code, the integrity of the runtime states (e.g., CPU registers, memory, and sensitive I/O), and the confidentiality of its code, data, and runtime states stored on a persistent memory. Commonly used in mobile and IoT devices, a TEE isolates TAs from the rest of the system, including the operating system and other applications. It combines hardware and software security measures, starting with a secure boot process that verifies system integrity.

*2.1.1 ARM TrustZone.* ARM TrustZone, introduced by ARM in 2004, offers a suite of hardware features for CPUs with Application (profile A) and Microcontroller (profile M) profiles [32]. Popular TEEs implementations using TrustZone include Samsung Knox, Qualcomm QTEE, and OPTEE. TrustZone architecture virtualizes the processor into two states: Secure World and Non-secure world. These states have access to different memory areas and are isolated by hardware protection systems. ARM's privilege modes, which range from EL-2 (highest) to EL-0 (lowest), are augmented with the secure modes SEL-0, SEL-1, SEL-2, and EL-3. The system boots at the EL3 level, where firmware from a ROM partitions memory into Secure and Non-secure regions, configures peripheral access for both modes, and sets up a handler for the Secure Monitor Call (SMC) instruction, facilitating transitions between Secure and Non-secure worlds. In ARM TrustZone, applications are categorized as Client Applications, which operate in the Non-secure world, and Trusted Applications, which reside in the Secure world. For trusted operations, a TA must be developed, signed, loaded into the Secure world, and invoked by a CA to execute secure code.

*2.1.2 Intel SGX.* Intel Software Guard Extensions (SGX) is an extension of the x86_64 architecture introduced in 2015 alongside 6th

generation Intel Core processors. SGX enables the creation of enclaves—private, encrypted memory regions of limited size. Access to data within an enclave is restricted to processes that utilize specific SGX mechanisms, such as the EENTER instruction. Unlike ARM's TrustZone, SGX does not require a separate operating system for the TEE, which facilitates its integration into cloud environments. Enclaves operate solely in protected mode and are established by the operating system. The creation of an enclave [16] is managed by the operating system, but execution authority is passed to a special Launch Enclave that coordinates with a remote Intel service for verification and to upload public certificates. An enclave must be officially signed by Intel. Designed entirely in microcode, Intel SGX allows for the mitigation of various vulnerabilities over time through software updates alone. However, as of 2021, SGX has been deprecated in 11th and 12th generation Intel Core processors [15], although its support continues for Intel Xeon processors, which are primarily used in servers and cloud environments.

*2.1.3 AMD SEV.* Secure Encrypted Virtualization (SEV) is an extension of the x86_64 architecture introduced in 2017 with Epyc processors [8]. SEV enhances security by allowing users to encrypt VMs memory, ensuring that even if a hypervisor is compromised, the data within the VM remain unreadable to attackers. Although similar to Intel's SGX, which creates protected enclaves, SEV applies its encryption mechanisms directly to entire virtual machines, making it particularly suitable for cloud infrastructures. In particular, SEV also supports secure migration of virtual machines between hosts. SEV's implementation includes an AES-128 encryption engine on the memory bus and a Secure Processor (SP), an ARM processor installed on the motherboard. The SP performs several critical functions, including generating and managing cryptographic keys for VM memory encryption and producing continuous attestation reports for each VM. These reports help establish a remote attestation system, allowing customers to verify that virtual machines on a server are derived from verified images and are operating in the intended environment. The attestation reports are signed using a private key housed within the SP, which can only be extracted through physical and destructive means.

## 3 Related work

With the introduction of Armv9-A architecture [27], the Confidential Compute Architecture model has been proposed to handle cloud contexts: it introduces the *Realm World (Realm)*, a new environment aside the older ones designed to be fully compatible with the NS allowing existing software stacks to operate seamlessly within the Realms. Each ARM world (NS, S, Realm) maintains its own Physical Address Space, and hardware mechanisms enforce checks on every memory access using a table, which tracks the physical address space of each memory granule and enforces access control policies to prevent unauthorized accesses. In this architecture, the untrusted hypervisor retains the capability to halt the scheduling of a Realm and reclaim memory assigned to it; however, it is strictly prohibited from accessing the CPU or memory state of the Realm. The ARM CCA necessitates the introduction of a new component, the Realm Management Monitor (RMM): this firmware operates within the Realm World at a higher privilege level than the Realms. The hypervisor can request the RMM to perform tasks such as delegating memory, creating Realms, executing Realms, and allocating memory to Realms. The newly introduced Armv9-A hardware feature presents a promising opportunity; however, it is not yet widely available on commercial devices. This limitation arises from both hardware constraints and the significant effort required for software adaptation or complete redevelopment. In contrast, our solution is designed to be compatible with billions of existing ARM devices, offering immediate applicability without requiring hardware upgrades. Moreover, in cloud environments, the process of updating production servers to the latest hardware is a time-consuming undertaking that often spans several years, making solutions that leverage existing infrastructure highly valuable.

Currently, ARM TrustZone does not natively support multiple isolated TEEs. To address this limitation, several approaches to Secure World virtualization have been proposed [13, 22, 26]. Among these, solutions such as OSP [12], PrivateZone [23], and vTZ [22] use hardware virtualization extensions available in the Non-secure World (NS-EL2) to create isolated environments. For instance, vTZ effectively virtualizes ARM TrustZone, enabling multiple Virtual Machines to have isolated TEEs on a single physical machine without requiring hardware modifications. While vTZ primarily enhances the security of existing TEEs by isolating Trusted Applications running inside a secure VM in the Non-secure World, it also has potential applications in cloud environments, where it can provide protection against inter-TA attacks. Their threat model assumes that an attacker can compromise a single TA, bypassing its confidentiality requirements. In contrast, our system employs a stronger threat model by mitigating such attacks within the TA itself, relocating it into a TEE for enhanced isolation and protection, and also using protocol to protect data in transit (confidential requirement). Moreover, our system offers enhanced portability and scalability, making it adaptable to a broader range of hardware and software environments.

A second line of research focuses on retaining Trusted Applications within the Secure World while enhancing isolation between them. For example, TEEv [34] and [25] introduce a minimalist hypervisor within the Secure world, enabling TAs to run on multiple isolated secure guest operating systems. These approaches perform isolation within the TEE environment and are complementary to our work, as they can be used to further improve isolation between TAs that reside within the TEE. Although TEEv's purpose is to support multiple isolated virtual TEEs in the Secure world, it is not suitable for cloud context where the setting is typically multitenant with concurrent non-secure-VMs that want to share the TEE.

From an architectural perspective, CoCoTPM [30] is quite similar to our proposed solution, as both aim to offer a hardware feature to multiple VMs in a confidential cloud computing context. Unlike traditional virtual TPM solutions that rely on the host, including the hypervisor, CoCoTPMs operate independently of the host's reliability, ensuring the secure functioning of VMs even if the host environment is compromised. Although TPMs are adept at mitigating a wide range of attacks, they do not protect against issues such as memory errors or logic programming faults. TEEs, on the other hand, can thwart these attacks by safeguarding sensitive code within the Trusted Environment. TEEs and TPMs are complementary solutions that can form a layered security approach, incorporating secure boot, cryptographic techniques, and TPM cryptographic

functions alongside Trusted Execution Environments to ensure security and privacy in cloud environments. To our knowledge, we are the first to introduce a novel framework for virtualizing the Trusted Execution Environment, specifically tailored for cloud service providers. This framework enables multiple Client Applications running in separate VMs to interact with Trusted Applications within a shared physical TEE (thereby leveraging the secure hardware features) virtualized for every VMs. This innovative approach improves the security of cloud-based applications, marking an advancement in the field of confidential computing for cloud environments.

## 4 Threat model

From a security analysis perspective, our threat model assumes the availability of a Trusted Execution Environment on the host device. Within this framework, only the Secure World and the EL-3 firmware are trusted components. In contrast, the Non-secure World, including the EL-2 hypervisor, is deemed untrusted and susceptible to compromise by malicious actors. An attacker gaining control of the Non-secure World could exploit it to access other Non-secure resources, undermining the system's integrity.

If a TEE is unavailable, the application's core logic must operate entirely in the Non-secure World, significantly increasing its vulnerability to external threats, including malware, privilege escalation attacks, and unauthorized access to sensitive data (see Figure 1, scheme 2). This creates a critical risk, as attackers could directly target the exposed application logic, leading to data breaches or system compromise. Our solution addresses this risk by migrating the application's primary logic into a TEE, effectively isolating it from the Non-secure World and potential adversaries (see Figure 1, scheme 1). This isolation ensures that even if the Non-secure World is fully compromised, the application's critical operations and sensitive data remain protected within the TEE. An untrusted VM and a TA could experience a service disruption if a malicious hypervisor selectively blocks their requests, essentially launching a DoS attack. Nevertheless, we believe that such a scenario would have limited impact in cloud contexts, since any disruption by the hypervisor, under the cloud provider's control, would amount to self-sabotage. Our main concern is protecting data confidentiality; thus, DoS attacks fall outside of our threat model.

The threat model aligns with the Trusted Execution Environment model defined by GlobalPlatform [21], which treats all resources and data outside the TEE as untrusted. Consequently, our approach emphasizes the need for robust security measures, particularly during data transfer. To safeguard data confidentiality, we advocate for end-to-end encryption for communications between the user terminal and the Client Application (CA), as well as between the CA and the Trusted Application (TA). This encryption ensures that sensitive information remains secure during transit, even in the presence of a compromised Non-secure World. By combining the TEE's isolation properties with end-to-end encryption, our solution strengthens the overall security posture, effectively mitigating risks associated with untrusted components and maintaining the confidentiality and integrity of critical application data.

## 5 Running Example

Consider, as an illustrative case as depicted in Figure 1, a health application that operates on the user's device and aggregates personal data to generate personalized health insights. The functionality of this application depends on the secure extraction of sensitive patient information from a central server. This scenario underscores the inherent limitations of conventional hypervisor security mechanisms, particularly their ineffectiveness in preventing the hijacking, theft, or modification of data processed within a Virtual Machine. The AMD SEV solution proves to be insufficient to mitigate these threats. Despite the encryption of the VM's memory, adversaries can still compromise and take over the software within the VM, thereby gaining control over the critical application. Focusing on hardware-oriented features of Trusted Execution Environments, our suggested architectural model, as shown in Figure 1, incorporates a hypervisor extension (VS-TEE hypervisor) that facilitates smooth communication between the main application and the Trusted Application. When a user terminal initiates a secure computation request, it first passes through the NS application (Figure 1, Scheme 1); VS-TEE then detects and activates the appropriate TA to execute the secure task. After the task is completed, the payload is re-encrypted and sent back to the user terminal. With this approach, numerous VMs are able to securely share the same hardware TEE.

## 6 Architectural Overview

The primary objective of our system is to provide a consolidated Trusted Execution Environment platform to multiple virtual machines within a cloud infrastructure, thereby facilitating TEE virtualization. In contrast to conventional cloud configurations that require separate TEE hardware units for individual VMs, our approach distinguishes itself by leveraging a singular TEE hardware unit supplemented by a suite of software components (Driver and Hypervisor) to accomplish virtualization. Figure 2 elucidates our comprehensive system architecture, where a single hypervisor supports three virtual machines while collectively using the same TEE. The initial component of our system is the driver VS-TEE, denoted by ❶, which is installed on the virtual machine of the user and functions as an intermediary between the virtualized TEE (within the VM) and the hypervisor, managing all requests directed to the physical TEE hardware. The primary role of the driver is to process requests from the Client Application and transform them into protocol-compliant messages for interaction with the hypervisor's virtual device.

The hypervisor ❷ receives secure computation requests and subsequently invokes the appropriate Trusted Application through the physical Trusted Execution Environment component. As stated previously, the messages are encrypted end-to-end, thereby preventing the hypervisor from accessing sensitive data. Upon arrival, the message is delivered to the TA, which decrypts and processes the payload securely. This efficient interaction between the driver and the hypervisor facilitates seamless message routing within the VS-TEE system, ensuring secure communication. VS-TEE introduces two main technical challenges essential to our approach. **(I) Address translation** (section 8.2.1): it is crucial to implement an effective memory mapping technique between the Virtual Machine

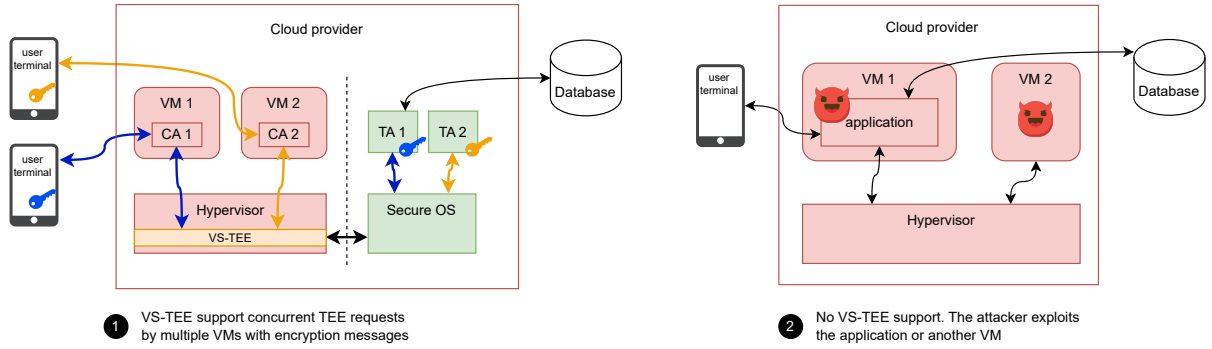


**Figure 1: Hypervisor is considered untrusted, secure world components and user terminal are considered trusted.**

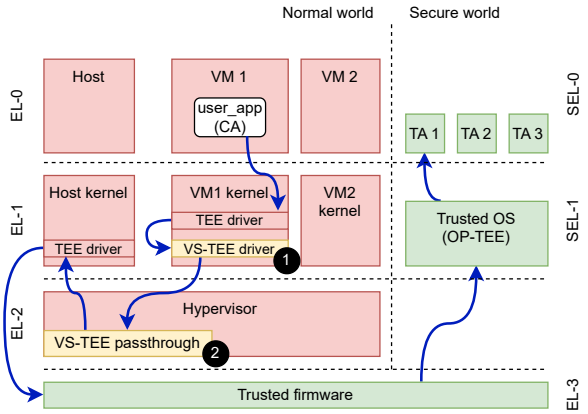


**Figure 2: Overview of the system with VS-TEE, showing the two main components: (1) VS-TEE driver installed within VMs, (2) hypervisor passthrough. Blue connections indicate encrypted communication, while black connections indicate plain communication.**

and the TEE, especially to facilitate buffer address conversion between the CA and the TA. **(II) Buffer synchronization** (Section 8.2.2): creating a method to properly synchronize memory buffers during communication. This requires a selection of sync points during every inbound /outbound request between CA and TA.

# 7 VS-TEE Communication Protocol

## 7.1 Protocol Execution Flow

The application intended for secure execution using a VS-TEE as a virtual shared TEE environment must be segmented into three parts: **(1) the user terminal**, which securely retrieves processed data, **(2) the Client Application**, which runs in a virtualized cloud setting and operates in the NS world, and **(3) the Trusted Application**, which resides in the Secure World. The sequence diagram shown in Figure 4 illustrates the operational flow of this architecture. Each invocation of secure functions (referred to as *Invokes*) necessitates both the opening and closing of a session. Given that the operational procedures for `TEE_IOC_OPEN_SESSION` and `TEE_IOC_CLOSE_SESSION` are the same as for `TEE_IOC_INVOKE`, only the Invoke flow will be detailed for simplicity. The sequence begins when a user submits a request ❶ to the CA, hosted on the server of a cloud provider. Upon receiving the request, the Client Application assesses it (II) and issues a `ioctl` command to trigger specific tasks within the TEE. This command is intercepted by the VS-TEE driver (III), which transmits an encrypted message to the hypervisor through a device file containing the secure computation request. The hypervisor processes the command, converts it into a TEE standard message, remaps the memory addresses, and finally forwards (IV) the message to the physical TEE. Upon reception, the TA decrypts the message's data and selects the appropriate secure function for execution (V). After running the secure function, the TA compiles the result and encrypts the information before storing it in a shared buffer. It then sends a message (IV) to the hypervisor with execution status; the hypervisor writes the data into dedicated registers, adjusts memory addresses if needed, and relays the status back to the appropriate VM's VS-TEE driver (III). The VS-TEE driver processes the response by placing the results in the Client Application's program memory (II), which then transmits the data to the user's terminal ❶.

## 7.2 VS-TEE Protocol and TA Selection

The comprehensive format for messages generated by VS-TEE provides guidance to the hypervisor on interpreting its contents. The `cmd` field includes one of several predefined operations (illustrated on the right side of Figure 3) which the hypervisor will execute, and this defines the `payload` contents. Because some messages may contain a varying number of arguments, the `payload_size` field specifies the size of the `payload` in bytes. The allowable values for `payload_size` depend on the nature of the transmitted message, and these constraints are detailed in Table 1. The selection of the TA is straightforward and leverages the `fd` (File Descriptor) field, which is common in all protocol messages intended to communicate with a specific TA. Similarly to Linux file descriptors, this field allows the driver to identify the active connection between the hypervisor and the TA target to receive the message. The `fd` is assigned during the OPEN operation of the virtual device, and all subsequent messages within the same connection must reference the same `fd`. The hypervisor maintains a translation mapping between the

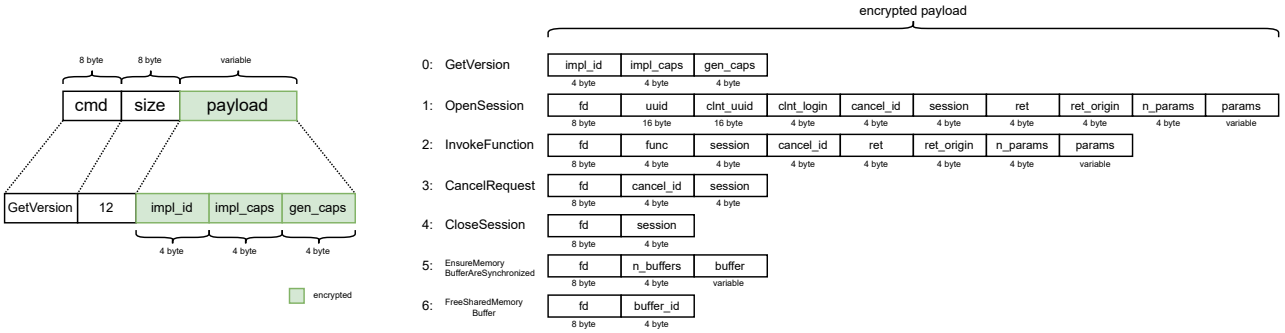


**Figure 3: Protocol messages. On the left header of all protocol messages. The first 2 value ensure that the destination is able to extract the type and the parameter of the message (inside payload). On the right the list of function that can be executed by the TEE contained in payload block.**

**Table 1: Payload size and constraint for implemented protocol messages**

| Command | Payload size | Constraint |
|---|---|---|
| `GetVersion` | 12 byte | |
| `OpenSession` | $64 \leq x \leq 192$ byte | $(x-64) \mod 32 = 0$ |
| `InvokeFunction` | $32 \leq x \leq 160$ byte | $(x-32) \mod 32 = 0$ |
| `CancelRequest` | 16 byte | |
| `CloseSession` | 12 byte | |
| `EnsureMemoryBuffers` | $12 \leq x \leq 112$ byte | $(x-12) \mod 24 = 0$ |
| `FreeSharedMemoryBuffer` | 16 byte | |

file descriptors used by the driver inside the VM and the corresponding file descriptors used by the host when interacting with the physical TEE (e.g. `VM1_CA1::FD1`↔`TA1::FD2`). This translation layer allows the hypervisor to efficiently route messages, ensuring that each CA within a VM is matched to its corresponding TA. To prevent conflicts during message transmission, the driver employs synchronization mechanisms, such as semaphores, ensuring that concurrent requests to the TEE do not clash while communicating with the virtual device.

## 8 System Design

In this section, we will outline the various design choices and explain the two main components of the VS-TEE framework. This summary aims to help the reader understand the architecture of our system and the rationale behind our design selections.

### 8.1 VS-TEE Driver

The TEE subsystem in Linux offers a dedicated kernel-space interface designed to facilitate the creation of TEE drivers. This interface is vital for the VS-TEE to handle key functions like the allocation and assignment of shared memory, message encryption, and the synchronization between the TEE and the kernel-space driver, VS-TEE. The VS-TEE driver ❶ operates within the user's VM, serving as a crucial communication link between the user-level virtual environment and the underlying physical TEE, referred to as ❷. The VS-TEE driver sets up a virtual TEE interface within the user-managed VM, primarily tasked with processing, converting, and forwarding requests from the Client Application to the hypervisor. This mechanism ensures seamless and secure communication between the CA and the TA (the trusted counterpart), as described in 4. The end-to-end encryption protocol used ensures that data exchanged between the CA and TA is safeguarded against unauthorized access. In its operation, the VS-TEE driver operates with clarity and efficiency. It starts by receiving requests from the TEE Linux subsystem, initially generated by the Client Application. These requests are then converted into standardized messages that follow a specific communication protocol. The driver then sends these messages to a virtual device using the Memory-Mapped I/O (MMIO) mechanism, ensuring reliable and secure communication between the virtual and physical TEEs.

### 8.2 VS-TEE Hypervisor

This element constitutes the central part of our framework and facilitates the primary interactions between the virtual machine and the trusted application. The VS-TEE passthrough ❷ requires a patch for the hypervisor. Specifically, the hypervisor needs to show a virtual device that the guest OS can use to manage VM requests. The passthrough monitors register states (Table 2), which get updated whenever the VM writes to the device's Memory-Mapped I/O (MMIO) region. When a write occurs at specific offsets (`0x0b000000` - `0x0b000200`), a signal is activated in the hypervisor. It takes control to manage the inbound request by making an `ioctl` call to the physical TEE and saves the results back in the VM's memory. The hypervisor allocates a portion of MMIO memory within the VM's address space, sectioning it into multiple registers for dispatching commands to the physical TEE. This mechanism is similar to user space requests, yet performed at a lower level by the VS-TEE driver, employing memory-mapped registers writable exclusively by the kernel. When a user-space application (Client Application) perform secure a operation using the TEE, the VS-TEE driver first access the `OPEN_TEE` hypervisor register. If a non-zero value is detected, a connection is established by obtaining an identifier (much like acquiring a file descriptor). Subsequently, the driver composes the command intended for the TEE in a buffer and writes the buffer's address to the MMIO `COMMAND_PTR` register. To initiate trusted execution and hand control to the TEE, the driver then writes a 4-byte value to the `SEND_COMMAND` register. The hypervisor pass-through

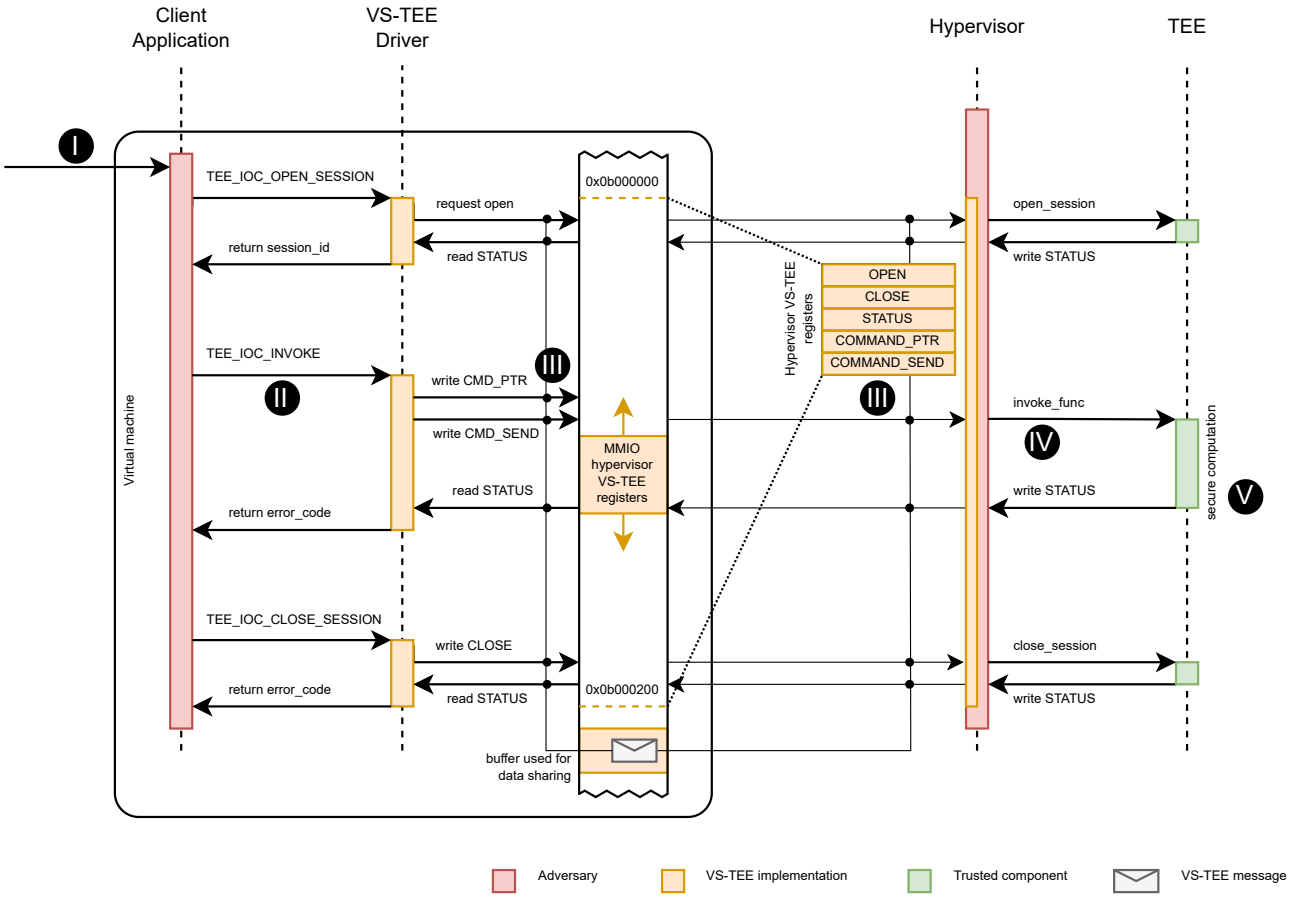


**Figure 4:** VS-TEE **communication protocol**

**Table 2: Registers mapped in MMIO zone, the offset is calculated from the starting point of the virtual device MMIO zone.**

| Register | Offset | Dimensione |
|---|---|---|
| OPEN_TEE | 0x0 | 8 byte |
| CLOSE_TEE | 0x8 | 8 byte |
| STATUS | 0x10 | 8 byte |
| COMMAND_PTR | 0x18 | 8 byte |
| SEND_COMMAND | 0x20 | 4 byte |

architecture poses two main challenges: (1) Address Translation and (2) Buffer Synchronization.

It is crucial to understand that TEE execution operates on a client-server model, requiring the Client Application to wait for a response when requesting trusted execution. The status of the request can be determined by reading the `STATUS` register, which contains the result of the TEE operation. Additionally, in the specified ARM architecture, the smallest atomic write is 4 bytes, making it unsafe to read or write to registers larger than 4 bytes during a command execution. E.g. it is possible that only the initial 4 bytes of the function entry point might be processed, leading to an incorrect execution address. To address this issue, we transfer the control for trusted execution only when the `SEND_COMMAND` (supplementary 4-byte binary register) is set to 1 (Table 2).

*8.2.1 Address translation.* Translation of addresses between a VM and the VS-TEE Hypervisor presents notable technical challenges, primarily due to the inherent separation of environments and their unique address spaces. This separation ensures that data or resources pertinent to the hypervisor's context are restricted to that environment and cannot be directly accessed or used by the VM without risking errors. Furthermore, the TEE communicates exclusively with the hypervisor, making the hypervisor the sole mediator for any requests originating from Client Applications on the VMs. Consequently, the hypervisor must remap all addresses along with the VM which have initiated such request. A key example of this issue is the difference in file descriptors utilized by the VM and the hypervisor to interact with the Trusted Execution Environment. The file descriptor used by the VM is only valid within its own address space and does not align with the file descriptor recognized by the hypervisor. Trying to use such hypervisor-specific data within the VM might result in severe errors, such as segmentation faults, caused by invalid memory access. To resolve these

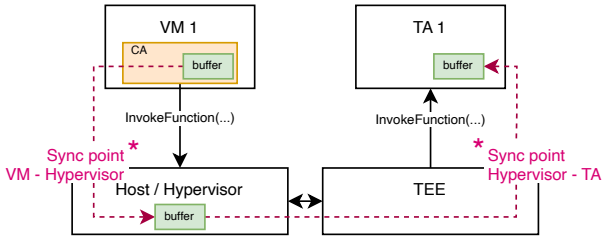


**Figure 5: Buffers shared between NS and S worlds are allocated in two distinc buffers. During an `InvokeFunction` call a `memcpy` operation is performed to synchronize the NS buffer with Hypervisor (host). Then, a second copy is performed to synchronize the secure TA buffer with the Hypervisor (host) buffer. The copy operation are executed backward when returning from the call.**

challenges, our system features a translation table that serves as a mediator for translating VM-specific addresses and identifiers to their equivalents in the hypervisor's environment. This translation table is dynamically refreshed by monitoring and capturing open and closed operations along the communication channels between the Client Applications and the Trusted Applications. This translation mechanism is vital for ensuring that resources like file descriptors and shared-memory addresses are properly translated and interpreted across the distinct address spaces. This strategy is essential to preserve the integrity and functionality of TEE between different layers of the system.

*8.2.2 Buffer synchronization.* The main major challenge in buffer synchronization is the creation of "shared" memory buffers that function effortlessly across both the Non-secure and Secure Worlds. Creating a shared buffer between two environments with differing security levels, such as the nonsecure and secure worlds, poses a significant challenge due to the inherent disparities in their trust and operational requirements. The Secure World enforces stringent security policies to protect sensitive data and resources, while the Non-secure World operates with fewer restrictions, prioritizing performance and accessibility. This mismatch makes it difficult to establish a memory-sharing mechanism that is both secure and efficient. The GlobalPlatform standard [21] acknowledges this limitation, emphasizing that directly sharing physical memory between these domains is often infeasible without resorting to data copying. To address this issue, the VS-TEE system adopts a practical solution by implementing two separate memory buffers of equal size—one allocated to the Non-secure World and the other to the Secure World. This approach avoids the complexities and risks associated with direct shared physical memory while ensuring efficient and secure synchronization between the two environments.

From a technical perspective, prior to utilizing a shared buffer, the VS-TEE driver dispatches a message to the hypervisor via the `EnsureMemoryBuffersAreSynchronized` API function. This message contains details about the buffer's unique ID within the guest, its allocated size, and its physical address. This procedure establishes the map of buffers within the hypervisor's mapping table, enabling it to select the appropriate memory address when necessary. During operation, synchronization points uphold buffer consistency by synchronizing them every time the boundary between the Secure World and the Non-secure World is traversed. This synchronization occurs under two circumstances: firstly, between the TEE and the VS-TEE hypervisor, and secondly, between the VS-TEE hypervisor and the VM. In both instances, a copy operation is performed to align the buffers. For instance, when the Client Application in the VM aims to invoke a secure function within the TEE, the procedure starts with an `InvokeFunction`, intercepted by the hypervisor, leading to the first synchronization (Figure 5). The hypervisor then forwards the request to the TEE on behalf of the VM, resulting in the second synchronization, after which the `InvokeFunction` is executed with both buffers accurately synchronized. While this technique of data transfer between buffers is less efficient compared to direct memory sharing, it is essential for ensuring data integrity and consistency between the Non-secure and Secure Worlds.

## 8.3 Provision of new TAs

Once the virtual machine setup is complete, two critical steps are required for secure communication with the Trusted Execution Environment. The initial step is to deploy the Trusted Application, followed by the creation of a key for secure communication between the user terminal and the TA. Deploying a new TA is simple, as it is loaded into the secure filesystem through a specialized Loader TA supplied by the Hypervisor. Once the TA has been uploaded, a 128-bit AES shared key is created and exchanged between the TA and the user terminal using the Diffie-Hellman algorithm. The first key is securely stored in the TEE's secure filesystem, while the second key is held in the user terminal. Since not all user terminals come with a TEE or TPM that can securely handle keys, we recommend utilizing the Signal Protocol [7]. This protocol uses the Double Ratchet algorithm, which allows two parties to swap encrypted messages based on a shared secret key. The algorithm generates new keys for each message, ensuring that previous keys cannot be inferred from later ones. This approach greatly improves the security of both prior and future encrypted messages, even if one of the party's keys is compromised.

## 8.4 TAs Migration

In cloud environments, migrating an application involves transferring it from one host server, A, to another host server, B. The migration process is divided into two sections. CA migration and TA migration. Migrating the TA is relatively simple because the Client Application can dispatch requests to the TA and wait for responses to proceed. The TA can be quickly migrated during periods when no functions are executed within the TA. If requests reach the hypervisor while the TA is offline (undergoing migration), they are queued until the TA is back online (TEE status *BUSY*). To duplicate a TA, it is necessary to transfer its memory, virtual device state, pending I/O requests, etc., to the destination host. Given that TAs are usually small in size, the migration process is expected to introduce minimal overhead. Although VM migration (including CA) has been thoroughly covered in various studies [35], our approach follows the design presented in [34]. It is detailed as follows:

after setting up a secure channel with the migration target, the hypervisor continues operating the VM, resets the dirty bit across the entire second-level page table, and copies the VM's memory to the target. The VM is then paused and the hypervisor inspects the dirty page-table bit. Any pages marked dirty are retransferred to the target platform, along with the state of virtual devices and any pending I/O requests; this process reduces VM downtime.

## 8.5 TA isolation

Within the VS-TEE prototype, TA isolation is overseen by OP-TEE. Currently, alternative solutions based on TrustZone, such as QSEE (Qualcomm) [24], Trusty (Android) [1], TrustedCore (Huawei) [11], and Kinibi (Trustonic) [4], employ various methods for the management of TA isolation. Nevertheless, these solutions are predominantly closed-source, apart from Trusty, which is tailored for Android and not optimal for cloud applications. OP-TEE maintains isolation by functioning in a higher privilege tier (SEL1) compared to TAs (SEL0). Each TA executes as an individual user mode process within the TEE, with the MMU enforcing memory isolation by assigning each TA a distinct virtual address space to avert unauthorized access. Consequently, a TA cannot directly compromise another unless a vulnerability in the secure kernel is fully exploited. Although some past vulnerabilities in OP-TEE's isolation have been documented [29], developers have addressed these issues. We claim that our TCB remains consistent with typical TEE architectures [21] and is comparable to the Android Trusty framework. While Android does not cater to cloud settings, it similarly supports the simultaneous execution of multiple TAs, safeguarding against direct attacks from a compromised TA on another, unless the secure kernel is first exploited.

# 9 Evaluation and analysis

We evaluated VS-TEE across several dimensions: functional correctness to ensure that no functionality is lost when switching from bare metal to virtualized environment, microbenchmark to assess the overhead of new components, and macrobenchmark on the overall overhead with a real use case scenario. All experiments run on QEMU, the board emulated is a Cortex-A53 (ARMv8) with TrustZone support equipped with 1GB of RAM. The system running the OP-TEE framework 3.20.0 with OPTEE-OS as the trusted OS and BusyBox as the rich OS is shipped with OP-TEE.

## 9.1 Functional Correctness Tests

To validate the transparency of VS-TEE in terms of TEE functionalities, we performed several tests using OPTEE's xtest [6], a comprehensive suite comprising 46174 across 99 categories. This suite, created by the OP-TEE project, covers a variety of common scenarios of TEE usage, including cryptographic operations, secure storage, and secure data transfer. Although it mainly uses the Linux TEE subsystem interface, the majority of tests can be applied to any TEE using a Linux driver. After testing VS-TEE with the entire suite, we did not detect any error or divergence from the expected behavior. Hence, we can declare with high confidence that VS-TEE maintains functional transparency to the host and does not compromise any OPTEE functionality.

## 9.2 System Overhead

*9.2.1 Microbenchmark overhead:* the initial batch of tests comprises seven small general-purpose user-space applications based on OP-TEE examples [2]. These programs include encryption algorithms, random generators, and secure storage. We performed our measurements using the time utility available in OP-TEE OS, specifically we used systime, that is, the time spent in kernel mode to serve the calling process, in order to assess the overhead of the VS-TEE passthrough infrastructure. The baseline we confronted our system with was collected by running the same examples on our host VM that can access directly to the TEE without using VS-TEE, even in this scenario we used the time utility and the sys time in order to ensure a fair comparison. Experiments 3 demonstrate that in this scenario the passthrough overhead is minimal, less than 2%. As additional data in Table 3, we provide the comparison between the xtest suite run with and without VS-TEE. Although this suite was used mainly to assess the functionality of our system, here the VS-TEE system exhibits an overhead 8%. We hold that the primary reason for this overhead is caused by the synchronization of shared-memory buffers that are much more prevalent in this suite than in the examples provided by OP-TEE. However, we deem this result to be acceptable considering the security advantages of sharing TEE in cloud environments.

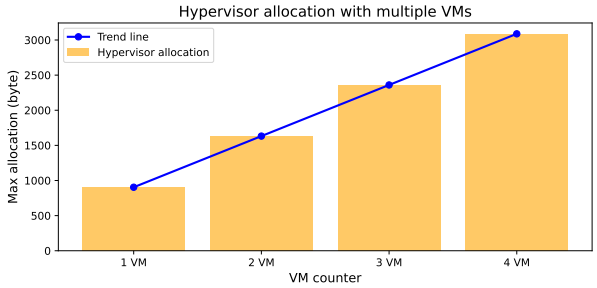


**Figure 6: VMs executing the OP-TEE AES program with VS-TEE. The graph show a linear upward trend, increasing linearly with each additional VM.**

*9.2.2 Memory overhead:* in this experiment, we focus on how the memory allocation of our hypervisor scales as the number of VMs that request secure TEE computation increases (parallel workload). The setup involved executing the same AES encryption program across 1 to 4 VMs simultaneously; in this test, we consider memory consumption by logging the allocation/free performed inside hypervisor and driver components. The results (Figure 6) showed that the memory consumption of VS-TEE grew consistently and linearly as the number of VMs increased, indicating that the hypervisor and driver's memory allocation scales efficiently with the addition of more VMs who request secure TEE calculations with VS-TEE.

*9.2.3 Overhead in Cloud context:* in this experiment, we focused on the cloud context by simulating a multi-tenant environment. Our aim was to observe how the time overhead scales when multiple virtual machines request TEE computation. Specifically, we measured the time required to encrypt/decrypt a binary data inside the

TA. To achieve this, we selected an open source available AES [3] and developed a POC program that reads a large file into memory and, block by block, iterates to perform encryption and decryption. The total execution time is measured by sending a UDP signal at the beginning and end of the operation to a time server. We have measured block encryption in two different scenarios:

- *Emulated guest with TA (w* VS-TEE*)* we record the time it took to encrypt / decrypt using the TEE from an emulated guest running inside the host with the VS-TEE driver loaded. This measure will include the overhead of the emulation and the overhead of VS-TEE.
- *Emulated guest without TA (w/o* VS-TEE*)* in order to assess the trend of the overall time taken to complete the workload, we execute encryption / decryption without TEE on the emulated guest.

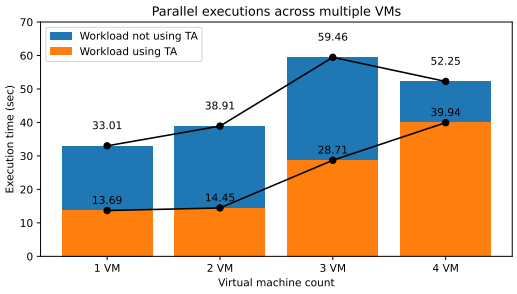


**Figure 7: Average of AES encrption workload executed multiple times on parallel VMs.** VS-TEE **have lower overhead due offloading TEE computation on the host.**

As more parallel VMs are introduced (Figure 7) into the cloud environment, the overhead of the workload with TEE (+ VS-TEE) and without TEE increases linearly and consistently in both cases. Furthermore, VMs using VS-TEE to request TEE computation experience reduced overhead compared to VMs performing identical tasks without TEE utilization. This reduction occurs because the primary encryption/decryption process is delegated to the host TEE, thereby avoiding the overhead of the emulation for the encryption process. We want to point out that the total increase in execution time as more VMs are added is not solely due to the execution itself, but is also heavily influenced by the overall consumption of resources on the machines, including CPU, RAM and disk I/O saturation. These factors contribute significantly to system performance degradation as resource contention increases.

## 9.3 End to End Performance

We conducted a comparative analysis to highlight the overall performance of the system in a real context. The experimental setup was designed to evaluate a real use case scenario: a content delivery application emulating the behavior of a popular video streaming service (e.g., Netflix). The application receives a signal from the user containing the ID of the data to be retrieved (e.g., the ID of a chunk containing a set of frames to be rendered on the user's device). This data is stored encrypted in a cloud virtual machine with a private server key. The application extracts the selected fragment, decrypts it, and reencrypts it with the user's key. This process occurs in a dedicated TA running in a trusted execution environment protected by VS-TEE to prevent data leakage and maintain integrity against hypervisor attacks. Finally, the chunk is sent back to the user. The benchmark setup involves a virtual environment with OPTEE-OS (qemu armv8), which hosts the running server application, and a client that requests data. The virtual environment consists of a VS-TEE hypervisor with physical TEE capabilities and a virtual machine running on the hypervisor with the VS-TEE driver installed. In this interactive test benchmark, we observe a 9% time increase with VS-TEE compared to the baseline program executed without VS-TEE. These data are consistent with the results seen in Table 3 of the microbenchmark. Since the emulation is nested (amd64-based system that emulates an aarch64 and the hypervisor that further emulates an aarch64 system), we have isolated the overhead of our component; in particular, we have set the same executable to use only TEE inside the VM and compared the overhead of the same executable running with TEE +VS-TEE in the same cloud context, eliminating the QEMU emulation overhead.

**Table 3: Comparison of execution time with and without pass-through across two test sets. The average overhead in time is about + 8% and drops to about +1. 7% when the parameters are passed mainly without a shared buffer.**

| Test set | W/o pass-through (ms) mean | std. dev. | Pass-through (ms) mean | std. dev. | Overhead |
|---|---|---|---|---|---|
| `xtest` | 1221,416 | 7,876 | 1319,174 | 12,790 | 8,003% |
| `OP-TEE examples` | 34,989 | 0.132 | 35.581 | 0.400 | 1,710% |

## 10 Security Analysis

The implementation of VS-TEE that leverages TEE and its interaction with the hypervisor and virtual machines is crucial to the security position of our system. In this section, we discuss the security measures and assumptions that underlie our framework to safeguard these components against potential threats.

## 10.1 Limitations and Assumptions

The proposed framework significantly improves security in cloud computing environments by addressing critical vulnerabilities in existing approaches and ensuring robust protection for sensitive data and computations. By excluding the host and hypervisor from the Trusted Computing Base (TCB), the framework minimizes the attack surface and eliminates reliance on components that are traditionally prone to compromise. This architectural decision ensures that the security of trusted applications (TA) remains independent of the host or hypervisor, effectively mitigating risks such as plaintext memory access or privilege escalation attacks, which are common in approaches such as AMD's Secure Encrypted Virtualization (SEV) [32].

A key feature of the framework is its ability to enforce strict isolation between multiple virtual machines (VMs) that share Trusted Execution Environment (TEE) capabilities. By dynamically managing address translation and resource allocation through a translation table, the framework ensures that each virtual machine interacts

exclusively with its assigned TEE resources. This prevents unauthorized access to memory or data belonging to other virtual machines, significantly reducing the risk of data leakage or interference between virtual machines [22, 23]. Furthermore, the fine-grained resource management of the framework ensures that the synchronization of memory and buffer between the VMs and the TEE is handled securely, further enhancing isolation and robustness.

The framework also provides strong protection against inter-TA attacks, a critical issue in multi-tenant environments where TAs may co-exist within the same TEE. By relocating TAs to isolated execution environments and mediating their interactions with secure communication protocols, the system ensures that the compromise of one TA does not jeopardize the integrity or confidentiality of others [12, 26]. This capability makes the framework particularly valuable for cloud deployments, where multiple clients rely on the same infrastructure while requiring strict data isolation.

Communication security is another area where the framework excels. Secures the channels between virtual machines, hypervisors, and TEEs by implementing authenticated communication protocols and end-to-end encryption. These measures ensure that sensitive data exchanges are protected from tampering, intercepting, or unauthorized modification, even in scenarios where the hypervisor or network may be compromised. This robust approach to communication security complements the overall isolation mechanisms and further strengthens the resilience of the framework to sophisticated attack vectors.

The framework is also designed to mitigate several classes of known attacks. Its strict address translation and access control policies prevent unauthorized memory access, while its isolation mechanisms reduce the risk of side channel attacks by limiting shared resource usage [18]. Additionally, the exclusion of the hypervisor from the TCB ensures that even if the hypervisor is compromised, it does not affect the confidentiality or integrity of TEE-protected data or processes.

Using widely available ARM TrustZone hardware and standard OP-TEE libraries, the framework ensures wide compatibility with existing infrastructure. Unlike solutions dependent on newer architectures such as Armv9-A, which face limited availability and require significant hardware and software upgrades [27], this approach provides immediate applicability. This makes it a practical and scalable solution for real-world cloud environments, where hardware updates often span years.

Our framework is centered on adapting the ARM TEE architecture for cloud environments without attempting to improve the security of the trusted kernel or OP-TEE. Our selection of OP-TEE was based on its user-friendliness, which simplifies prototyping and performance evaluations. In practical applications, our framework allows for an easy shift to a more secure trusted kernel when necessary, as it does not depend on OP-TEE. Rather, VS-TEE utilizes Linux's general-purpose TEE APIs, offering flexibility and allowing a smooth transition if needed.

## 10.2 Side channel attacks

Within the context of our work, side-channel attacks can be categorized into two distinct types:

(1) **Hardware-Based** – These attacks [9, 31] are not included in our study scope. Our assumption, with VS-TEE tailored for cloud settings, is that the attacker lacks physical access to the server, making hardware-based side-channel attacks implausible.
(2) **Microarchitectural** – This approach is practical within our framework [28]. Specifically, our system has the capability to incorporate microarchitectural side-channel attacks at two distinct levels: through the TA delegation mechanism and at the TA level.
   (a) *TA Delegation Mechanism* –Although the delegation mechanism in our design might theoretically act as a side channel, its potential impact on data security remains minimal because the data is protected through cryptographic means.
   (b) *TA Level* – Increasing the quantity of TA within the TEE broadens the potential attack surface for side channel threats. Although these TAs function autonomously from our framework, these risks can be managed by employing current defense strategies, such as those presented in [19][20], which cloud providers can adopt to strengthen TEE security.

## 11 Conclusion

In this work, we addressed critical security challenges in cloud computing environments by proposing a novel framework that integrates Trusted Execution Environments (TEEs) with virtual machines (VMs) while excluding the host and hypervisor from the Trusted Computing Base (TCB). Unlike existing solutions that rely on the trust of the host or hypervisor, our approach ensures robust isolation and secure communication through a VS-TEE driver for VM-TEE interactions and a VS-TEE hypervisor to mediate requests. Designed for compatibility with the ARM TrustZone and OP-TEE libraries, our framework leverages widely available hardware, avoiding the adoption delays associated with newer architectures such as ARMv9-A. The framework effectively addresses key technical challenges such as memory translation, resource management, and interoperability, ensuring that multiple virtual machines can securely share TEE capabilities. Our open source prototype, tested in various scenarios, demonstrates the practicality and scalability of the solution for real-world deployment. By enhancing security while maintaining compatibility with existing infrastructure, our framework represents a significant step forward for Confidential Computing. It provides a practical, scalable solution to protect sensitive data and computation in cloud environments, aligning with the growing demand for secure and efficient cloud infrastructure.

## Acknowledgments

This work was supported in part by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the Italian MUR. Neither the European Union nor the Italian MUR can be held responsible for them.

## References


[1] [n. d.]. Android Trusty. https://source.android.com/docs/security/features/trusty. Accessed: 2025-02-19.

[2] [n. d.]. OP-TEE Examples. https://optee.readthedocs.io/en/latest/building/gits/optee_examples/optee_examples.html.

[3] [n. d.]. Tiny-AES-c implementation. https://github.com/kokke/tiny-AES-c.

[4] [n. d.]. *Trustonic Application Protection*. Technical Report. Trustonic. Accessed: 2025-02-19.

[5] [n. d.]. VSTEE source code. https://zenodo.org/records/13902728.

[6] [n. d.]. XTest OP-TEE. https://github.com/OP-TEE/optee_test.

[7] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. 2019. The double ratchet: security notions, proofs, and modularization for the signal protocol. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 129–158.

[8] AMD. [n. d.]. White paper: AMD memory encryption. https://www.amd.com/system/files/TechDocs/memory-encryption-white-paper.pdf.

[9] Alberto Battistello, Guido Bertoni, Michele Corrias, Lorenzo Nava, Davide Rusconi, Matteo Zoia, Fabio Pierazzi, and Andrea Lanzi. 2025. Unveiling ECC Vulnerabilities: LSTM Networks for Operation Recognition in Side-Channel Attacks. arXiv:2502.17330 [cs.CR] https://arxiv.org/abs/2502.17330

[10] Luca Buccioli, Stefano Cristalli, Edoardo Vignati, Lorenzo Nava, Daniele Badagliacca, Danilo Bruschi, Long Lu, and Andrea Lanzi. 2023. JChainz: Automatic Detection of Deserialization Vulnerabilities for the Java Language. In *Security and Trust Management*, Gabriele Lenzini and Weizhi Meng (Eds.). Springer International Publishing, Cham, 136–155.

[11] Marcel Busch, Johannes Westphal, and Tilo Müller. 2020. Unearthing the {TrustedCore}: A Critical Review on {Huawei's} Trusted Execution Environment. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.

[12] Yeongpil Cho, Junbum Shin, Donghyun Kwon, MyungJoo Ham, Yuna Kim, and Yunheung Paek. 2016. {Hardware-Assisted}{On-Demand} Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 565–578.

[13] 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)*. IEEE, 1628–1633.

[14] Confidential Computing Consortium et al. 2022. A Technical Analysis of Confidential Computing. *Confidential Computing Consortium–Linux Foundation, Technical Report v1* 3 (2022).

[15] Intel Corporation. [n. d.]. 11th Generation Intel Core Processor Desktop Datasheet. https://www.intel.com/content/www/us/en/content-details/634648/11th-generation-intel-core-processor-desktop-datasheet-volume-1-of-2.html.

[16] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive* (2016).

[17] Stefano Cristalli, Long Lu, Danilo Bruschi, and Andrea Lanzi. 2019. Detecting (absent) app-to-app authentication on cross-device short-distance channels. In *Proceedings of the 35th Annual Computer Security Applications Conference* (San Juan, Puerto Rico, USA) *(ACSAC '19)*. Association for Computing Machinery, New York, NY, USA, 328–338. https://doi.org/10.1145/3359789.3359814

[18] Jesse De Meulemeester, Luca Wilke, David Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, and Jo Van Bulck. 2025. BadRAM: Practical Memory Aliasing Attacks on Trusted Execution Environments. In *46th IEEE Symposium on Security and Privacy (S&P)*.

[19] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2020. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 451–468. https://www.usenix.org/conference/usenixsecurity20/presentation/dessouky

[20] Ghada Dessouky, Alexander Gruler, Pouya Mahmoody, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2021. Chunked-cache: On-demand and scalable cache isolation for security architectures. *arXiv preprint arXiv:2110.08139* (2021).

[21] GlobalPlatform. 2022. *TEE System Architecture v1.3*. Technical Report. GlobalPlatform. Accessed: 2022-12-21.

[22] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. vTZ: Virtualizing ARM TrustZone.. In *USENIX security symposium*. 541–556.

[23] Jinsoo Jang, Changho Choi, Jaehyuk Lee, Nohyun Kwak, Seongman Lee, Yeseul Choi, and Brent Byunghoon Kang. 2016. Privatezone: Providing a private execution environment using arm trustzone. *IEEE Transactions on Dependable and Secure Computing* 15, 5 (2016), 797–810.

[24] Fatima Khalid and Ammar Masood. 2022. Vulnerability analysis of Qualcomm Secure Execution Environment (QSEE). *Computers & Security* 116 (2022), 102628. https://doi.org/10.1016/j.cose.2022.102628

[25] Donghyun Kwon, Jiwon Seo, Yeongpil Cho, Byoungyoung Lee, and Yunheung Paek. 2019. Pros: Light-weight privatized se cure oses in arm trustzone. *IEEE Transactions on Mobile Computing* 19, 6 (2019), 1434–1447.

[26] Shih-Wei Li, John S Koh, and Jason Nieh. 2019. Protecting cloud virtual machines from hypervisor and host operating system exploits. In *28th USENIX Security Symposium (USENIX Security 19)*. 1357–1374.

[27] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. 2022. Design and verification of the arm confidential compute architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 465–484.

[28] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. 2021. A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks, and Defenses in Cryptography. *ACM Comput. Surv.* 54, 6, Article 122 (July 2021), 37 pages. https://doi.org/10.1145/3456629

[29] OP-TEE. [n. d.]. *OP-TEE security advisory*. Technical Report. Accessed: 2025-03-10.

[30] Joana Pecholt and Sascha Wessel. 2022. CoCoTPM: Trusted Platform Modules for Virtual Machines in Confidential Computing Environments. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 989–998.

[31] Gábor Pék, Andrea Lanzi, Abhinav Srivastava, Davide Balzarotti, Aurélien Francillon, and Christoph Neumann. 2014. On the feasibility of software attacks on commodity virtual machine monitors via direct device assignment. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security* (Kyoto, Japan) *(ASIA CCS '14)*. Association for Computing Machinery, New York, NY, USA, 305–316. https://doi.org/10.1145/2590296.2590299

[32] Sandro Pinto and Nuno Santos. 2019. Demystifying arm trustzone: A comprehensive survey. *ACM computing surveys (CSUR)* 51, 6 (2019), 1–36.

[33] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. 2015. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/Ispa*, Vol. 1. IEEE, 57–64.

[34] Jianqiang Wang, Pouya Mahmoody, Ferdinand Brasser, Patrick Jauernig, Ahmad-Reza Sadeghi, Donghui Yu, Dahan Pan, and Yuanyuan Zhang. 2022. VirTEE: A full backward-compatible TEE with native live migration and secure I/O. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 241–246.

[35] Fei Zhang, Guangming Liu, Xiaoming Fu, and Ramin Yahyapour. 2018. A survey on virtual machine migration: Challenges, techniques, and open issues. *IEEE Communications Surveys & Tutorials* 20, 2 (2018), 1206–1243.