USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

TETD: Trusted Execution in Trust Domains

Zhanbo Wang, *Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China, and Pengcheng Laboratory, China;* Jiaxin Zhan, *Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China, and Department of Computer Science and Engineering, Southern University of Science and Technology, China;* Xuhua Ding, *Singapore Management University;* Fengwei Zhang, *Department of Computer Science and Engineering, Southern University of Science and Technology, China, and Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China;* Ning Hu, *Pengcheng Laboratory, China*

https://www.usenix.org/conference/usenixsecurity25/presentation/wang-zhanbo

This paper is included in the Proceedings of the
34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the
34th USENIX Security Symposium is sponsored by USENIX.

TETD: Trusted Execution in Trust Domains

Zhanbo Wang[1,2], Jiaxin Zhan[1,3], Xuhua Ding[4], Fengwei Zhang[3,1,†], Ning Hu[2]
[1]*Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China*
[2]*Pengcheng Laboratory, China*
[3]*Department of Computer Science and Engineering, Southern University of Science and Technology, China*
[4]*Singapore Management University*
{12131105, zhanjx}@mail.sustech.edu.cn, xhding@smu.edu.sg
zhangfw@sustech.edu.cn, hun@pcl.ac.cn

Abstract

Intel TDX empowers cloud service providers to construct confidential virtual machines called trust domains (TDs) on x86 platforms. Similar to its counterparts from AMD and Arm, TDX's hardware based protection over integrity and secrecy of virtual machine memory and vCPU states inevitably hinders legitimate virtual machine management such as introspection. At the presence of compromised high-privileged software (e.g., the guest kernel), neither the cloud service provider nor the TD owner can securely carry out a task within the TD. To tackle this problem, we propose TETD, an in-TD trusted execution technique without trusting any TD system software. Our design does not pivot on in-VM privilege layering, a popular approach used in existing VM security enhancement schemes. Instead, we leverage the virtual machine monitor's existing capability of resource management to directly separate memory and vCPU used for trusted execution from the TD system software. We implement a TETD prototype on a TDX server and run extensive experiments. The performance overhead incurred by TETD to the TD depends on the workload. In our benchmark evaluations, the highest toll is about 6.8%. Moreover, our three applications also demonstrate that TETD provides a TD owner a practical and secure foothold at the presence of a compromised kernel.

1 Introduction

Intel Trust Domain Extensions (TDX) [11, 31] allows confidential virtual machines, called Trust Domains or TDs, to be created in a cloud platform, with their private memory automatically encrypted by the hardware using Total Memory Encryption-Multiple-Key (TME-MK) [28] and their CPUs running in the Secure Arbitration Mode (SEAM). The host Virtual Machine Monitor (VMM) runs in the Non-SEAM mode and therefore is denied from accessing a TD's private memory and vCPU contexts in plaintext.

[†]Fengwei Zhang is the corresponding author.

As noted by Schwarz and Rossow [61], all confidential VM (CVM) technologies, including AMD SEV and Arm CCA, inherently conflict with existing out-of-VM introspection [15, 18, 73] as these technologies are built to thwart a VMM's guest memory access. Moreover, none of these techniques is designed to cope with internal kernel compromise. Hence, when an TD's kernel fails to function properly, its owner has no foothold inside or outside it to carry out responsive tasks such as live memory acquisition. When the kernel is compromised by an internal adversary, the owner loses the entire security control over the TD.

Schwarz and Rossow [61] tackled this introspection challenges for AMD CVMs by leveraging Virtual Machine Privilege Levels (VMPL) [4], a hardware feature providing in-VM privilege layering. The introspection agent is placed at a higher privilege level than the VM kernel's and therefore can securely access all contents of the kernel. This approach, termed *privilege layering* in this paper, has also been explored to harden a CVM without trusting its kernel, using either software techniques or hardware features. Examples include enclaves inside a CVM [68], secure log services [1], and vTPM for CVM [53].

There are several flip sides to using privilege layering. First, it trusts the *most* privileged software in a CVM. Hence, this software is a single-point of failure to CVM security. Second, to place security functions for CVM applications and/or kernel always bloats the size of the most privileged software, thus weakens CVM security. Lastly, from the practicality perspective, not all cloud users demand a higher layer of system software in their CVMs.

In this work, we propose a resource separation approach to secure TD sensitive tasks, without relying on privilege layering. Intuitively, this approach hinges on the VMM's resource management capability and turns a TD into a two- or multi-body system instead of splitting it into multiple privilege layers. Consequently, a TD owner can still retain her security foothold for her TD even if the most privileged software therein is compromised. As compared with the privilege layering approach, ours can cope with full TD

compromise without a single point of failure or invasive changes to the TD's existing system software. The approach preserves TDX's assurance of TD security as the VMM gains no advantage in attacking a TD.

Our scheme, termed *TETD* (short for *Trusted Execution in TD*), allows one or multiple subsystems to be launched and running inside a TD, however, independent of the TD system software which is referred to as the *TD kernel* for ease of presentation. A subsystem hosts and protects a pre-installed software termed *agent* with one-way memory isolation – it can be granted access to the TD kernel and other applications' memory but not vice versa. TETD supports *exclusive mode* agents to run with all TD threads suspended. Such agents are useful for TD services such as memory snapshot and forensics. TETD also supports *collaborative mode* agents which run simultaneously with TD threads, while resisting kernel-privileged attacks. We use different protection strategies based on their execution models. Our design makes *no* change to the TDX Module which is within the TDX's software TCB. As a result, most (if not all) existing virtualization based isolation techniques in the literature [10, 23, 49, 58] are infeasible to apply.

We implement a prototype of TETD and we systematically measure the performance of TETD and rigorously assess the performance and security of the agents. We also develop three applications to demonstrate its security strengths and practicality. They involve two exclusive mode agents for TD service: introspection and agent management, and two collaborative mode agents that provide hardened security for a TD application and for the TD kernel respectively. In short, we make the following contributions:

- We propose a resource separation approach to tackle security challenges originating from TD system software compromise. Compared with the privilege layering approach, ours can cope with full TD compromise and is non-intrusive to TD system software.
- Following this approach, we design TETD that builds in-TD subsystems to host agents running in exclusive/collaborative mode with security and availability assurances against a malicious TD kernel.
- We implement TETD and evaluate its performance with Apache Benchmark and LMBench. The system performance impact ranges from 0.1% to 6.8%. The prototype is available at our FigShare permalink[1].
- We also develop four agents to demonstrate its practicality in dealing with real-world security problems.

ORGANIZATION. The next section introduces TDX internals and the constraints imposed on our design. Section 3 presents the high-level view of TETD, followed by design details in Section 4 and our prototype implementation in Section 5. Section 6 evaluates TETD's security and performance with benchmark experiments and three applications. Related work is in Section 7 and discussion in Section 8. Section 9 concludes the paper. Additionally, the Appendix includes the fourth application and provides APIs for implementation reference.

[1]TETD prototype at: https://doi.org/10.6084/m9.figshare.29262146.v1

2 TDX Internals and Constraints

This section explains TDX internals from the angle of a TD's memory and vCPU management. It not only explains the technical prerequisites for understanding our design, but also highlights the constraints we face. A more comprehensive description of TDX can be found in [11].

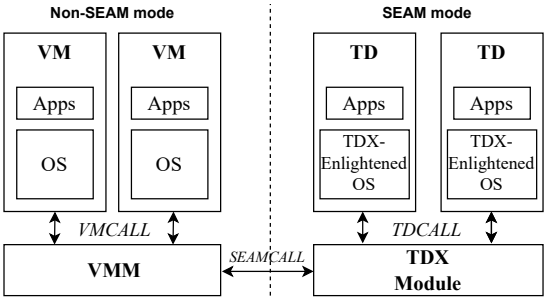


Figure 1: Architectural Component Overview for TDX.

Figure 1 illustrates a TDX-based cloud environment. A TD is a confidential virtual machine with its vCPUs running in SEAM mode and its memory encrypted by the hardware using a TD-specific AES key inaccessible to any software. The TDX Module is a trusted VMM managing TDs. It resides on the SEAM-memory—a reserved physical memory space specified by a set of registers with its vCPU also running in the SEAM mode. The VMM runs in the Non-SEAM mode. Although it cannot access the SEAM-memory, it manages the physical pages and CPUs needed by the TDX Module.

SEAMCALLs and TDCALL are two instructions for the VMM and the TD software to invoke the TDX Module's functions, respectively. Depending on the argument in use, TDCALL may cause the TDX Module to pass the invocation to the VMM, instead of handling it by itself. This type of TDCALL is noted as VMCALL.

TD Measurement. Intel TDX's TD attestation consists of two parts of measurements. The first part is the measurement of the TD building procedure which includes, among other, physical pages added by the VMM to the TD. Since a TD is a virtual machine, the second part is for the TD's bootup, similar to a TPM-based measured launch. TDX provides registers and instructions so that in-guest software can instruct the hardware to measure contents in a chosen physical memory range and extend the register with the measurement result. Since TD booting requires the VMM to import the so-called TD Virtual Firmware (TDVF) which is Intel's TD bootloader,

the TD bootup measurement starts from the TDX Module's measurement of the TDVF when it is loaded into a TD's private memory. The TDVF then measures the TD kernel to be loaded. Similarly, the kernel can further measure any software module or memory regions of concern, provided that the TD owner has the ground-truth for verification.

2.1 Memory Management for TDs

The guest physical address (GPA) space of a TD consists of *private* pages and *shared* pages. The former are those exclusively used in the TD and are encrypted by the hardware using a TD-specific key. Their GPA-to-HPA translations are managed via the *Secure EPT* (SEPT), which is set by the TDX Module and encrypted by the hardware. A TD's shared pages are for the TD to exchange data with another VM or the host, e.g., for I/O operations. Their GPA-to-HPA translations are solely controlled by the VMM without involving the TDX Module.

Each TD has exactly one SEPT. While the SEPT is set by the TDX Module, all physical pages in the TD, including the SEPT pages, are supplied by the VMM. To dynamically add a private page to a TD from scratch, the VMM provides the SEPT page to the TDX Module. Then, it adds a physical page to be mapped at the specified GPA position. Finally, an acceptance TDCALL from inside the TD finalizes the page allocation, allowing guest access.

GPA Blocking. Among all leaf functions of SEAMCALLs, the blocking-related ones are the cornerstone of TETD, as it provides the *GPA blocking* functionality which is meant to resolve the potential race condition between the TD and the VMM with respect to physical pages in transition. To block a GPA range, the VMM uses `TDH.MEM.RANGE.BLOCK` to request the TDX Module to mark the corresponding parent SEPT entry. As a result, the MMU aborts SEPT walking whenever translating any GPA within that range. Another function, `TDH.MEM.RANGE.UNBLOCK` is for unblocking.

TLB Tracking. TDX introduces the TLB tracking mechanism for the VMM to invalidate stale TLB entries. The TLB tracking begins with a GPA blocking, followed by invoking the `TDH.MEM.TRACK` in which the TDX Module increments the epoch counter of the TD. The VMM sends IPIs to all TD vCPUs. When they re-enter TD after being trapped to the TDX Module, the new epoch counter takes effect and the TLB entries with the lower epoch numbers are not used by the MMU for translation.

CONSTRAINTS. The VMM can only create a GPA-to-HPA mapping with a new physical page added to the TD, remove an existing one or relocate HPA of a page. It can neither remap an existing GPA nor set permission bits. There is no SEAMCALL to instruct the TDX Module to create two SEPTs for a TD. Since the runtime EPT pages passed by the VMM to the TDX Module are zeroed, the VMM also cannot preset a page for the TD to use.

2.2 vCPU Scheduling for TDs

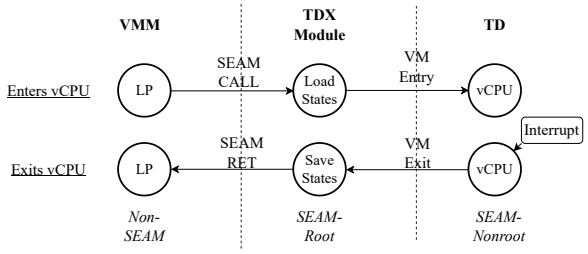


Figure 2: TD vCPU Enter/Exit.

TDX imposes more restricted vCPU management than VMX to reduce the risk of information leakage to the VMM. Each thread in VMM is called a Logical Processor (LP). By issuing a `TDH.VP` leaf function, the VMM associates a vCPU with an LP; it can later disassociate them using `TDH.VP.FLUSH`.

vCPU Enter and Exit. As shown in Figure 2, the VMM calls `TDH.VP.ENTER` to enter a TD vCPU execution. In response, the TDX Module prepares the vCPU state and then starts the vCPU on the current LP. To induce TD-Exit on a running vCPU, the VMM injects an IPI via the x2APIC interface. The interrupt causes the vCPU to trap to the TDX Module, which returns the control of the LP to the VMM. Note that the vCPU-LP association is not affected.

CONSTRAINTS. The VMM only schedules a TD vCPU's launch and execution time. It has *no* access to a vCPU context or control structures. No SEAMCALL allows the VMM to modify any vCPU register.

3 Overview

This section presents the high-level view of TETD, including the security model, challenges, and our approach. To avoid verbosity, we refer to the TDX Module as "the Module" and a TD's kernel as a "kernel". To simplify the description and understanding, the rest of the paper assumes the kernel has the *highest* privilege in the TD. The security goal of TETD is to protect a TD agent's execution integrity, data secrecy and integrity against the adversary in the TD kernel.

3.1 Security Model for TETD

TETD trusts all components within the TCB of Intel TDX, including the TDX Module and the underlying hardware. We assume the launch-time integrity of the TD kernel and the agents since it is verifiable through remote attestation. We

consider an in-TD adversary which compromises the TD kernel at runtime by exploiting its vulnerabilities. After gaining the highest privilege in the TD, it can execute arbitrary code and may attempt to attack the agent's confidentiality, integrity, or availability, e.g., by injecting malicious code or obstructing introspection tasks. Note that TETD does not address vulnerabilities or implementation flaws within the agent itself, such as memory corruption or unintended data leakage through the agent's interfaces.

The VMM is trusted to execute TETD correctly and does not collude with the in-TD adversary. This assumption is necessary as the attacks above are outside of TDX's threat model. Existing privilege-layering schemes [61] also introduce additional trust. They do so by relying on VMPL0 software (or L1VMM in TDX [32])—components that reside within the CVM and can be attested at launch time for integrity, without runtime guarantee. When the assumption on the VMM fails, an agent loses TETD's protection against the adversary even though TDX still shields the TD's entire private memory against accesses from the outside. Hence, TETD users must remain cautious and recognize that TETD security depends on whether the VMM behaves as expected.

While side-channel attacks [21, 41, 43, 52, 54, 72] are orthogonal to this study, we discuss their impacts and potential mitigation approaches when assessing TETD security in Section 6.1.2.

3.2 Challenges and Design Considerations

We face several TDX-related challenges. TDX blocks all unsolicited read or write accesses to the TD and the TDX Module's memory and vCPU context from the outside, i.e., from CPU running in Non-SEAM mode. Although the TDX Module does have the capability of accessing TD internals, we refrain from making any change to it because it is solely provisioned and maintained by Intel as the software TCB of TDX. It is thus infeasible to realize the agent's functionality (e.g., to list kernel objects) outside of the target TD. Moreover, there lacks hardware support for secure in-TD execution against TD compromise. While processors supporting TDX also support SGX, enclaves cannot be created within a TD according to the specification [31]. To uphold TD security provided by TDX, we cannot expose the agent's runtime data to the VMM.

3.3 Our Approach

3.3.1 The Idea

The idea underpinning TETD is to set up an in-TD autonomous subsystem via resource separation. With all its computing resources (i.e., vCPU and memory) securely separated from the TD system software, the subsystem securely hosts the agent's execution. The idea is embodied by using the VMM's resource management of a TD's vCPU and memory. Specifically, when the subsystem is at rest, its memory is blocked from access by using the GPA blocking mechanism (as described in Section 2.1); when it is active and its memory has to be unblocked, we use vCPU scheduling to suspend all TD threads' executions.

To reduce the performance impact and enrich functionality, we further propose to replace GPA blocking with location hiding so that TD threads and the agent can run simultaneously. We tap into the vast 52-bit GPA space and allocate a random and secret GPA location for agent execution. As the kernel adversary is forced to make online guess attacks against the secret GPA, its success probability is negligible. In short, TETD allows for a TD to be equipped with one or multiple subsystems with different agents executing in one of the two modes:

- **exclusive mode**. The agent runs with all TD threads paused. This mode is suitable for TD services, such as TD introspection. The frozen TD is conducive to maintaining consistency among fetched TD data.
- **collaborative mode**. The agent executes with other TD threads running and may interact with them through data exchanges. This mode is for security-sensitive tasks, e.g., data signing using a long-term secret key.

3.3.2 Agent Subsystem and TETD Components

Figure 3 illustrates the architectural view of a TETD-enabled cloud system. It consists of the VMM with TETD's host component (named *TETD-H*); the unmodified TDX Module; and a TD where two agent subsystems are installed and isolated from the TD kernel servicing various applications.

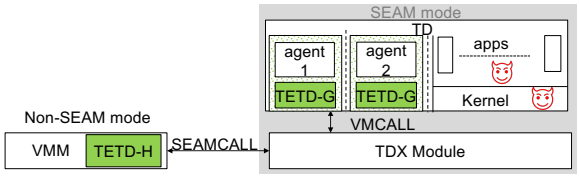


Figure 3: Illustration of two TETD agents in a TD with one-way isolation against a malicious kernel. TETD comprises TETD-G and TETD-H.

Agent. An agent is a self-contained executable a TD owner needs to execute without trusting the TD kernel. Its functionality and the privilege of accessing the TD kernel and/or applications are pre-selected and pre-determined by the TD owner before deployment. The agent is compiled and linked with *TETD-G*, i.e., TETD's guest component. All agents are built into the TD image together with the TD kernel. From the system point of view, the agent has its exclusive subsystem consisting of a dedicated vCPU referred to as the *agent vCPU* and a pool of private physical pages mapped to a GPA range.

Its vCPU and GPA pages are not shared with other agents unless they are explicitly designed so. During TD bootup, it is initialized and put into the *sleep* state in which the vCPU is dissociated with the LP. Upon receiving a job request, it is activated to enter the *running* state and completes the stipulated workload before hibernation. While the agent's workload may involve data accesses to the kernel or an application, the agent's control flow is self-contained. Namely, the agent vCPU does not run instructions outside of its assigned GPA region.

CAVEAT. We stress that while TETD protects an agent, it neither promotes nor demotes the agent's privilege which is chosen at the user's discretion.

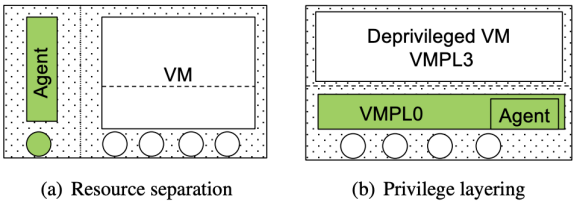
TETD Architecture. TETD comprises only two components: TETD-H and TETD-G. TETD-H manages vCPU and memory resources upon requests from TETD-G. It makes use of the following VMM capabilities described in Section 2:

- GPA blocking/unblocking. It closes or opens access the GPA regions allocated to the agent subsystem.
- vCPU management. It schedules the TD's and the agent's vCPUs onto or off from logical processors.

Running in Ring 0, TETD-G provides two services to the agent. One is to request TETD-H operation on behalf of the agent, such as to put the subsystem into sleep; and the other is to provide mission-critical system services such as preparing the agent's paging hierarchy and ensuring graceful abort upon exceptions. Although with Ring-0 privilege, TETD-G is far from a full-fledged kernel. For instance, it offers no system call services except TETD related invocations. Note that TETD-G provides no services to TD applications, hence not replacing the TD kernel. Also note that each agent subsystem has its own TETD-G instance with independent code and data isolated from all others in the TD.

3.4 Resource Separation versus Privilege Layering

Figure 4 illustrates two approaches to trusted execution in CVM: resource separation used in TETD and privilege layering used in the literature (for AMD SEV). Their architectural differences manifest in security strength, deployment, functionality, and performance.



(a) Resource separation (b) Privilege layering

Figure 4: Two approaches to trusted execution within a TD or a confidential VM in general.

Security. Table 1 below compares security of two approaches. Resource separation's reliance on the VMM to realize protection is a limitation, and also a strength. In the event of malicious VMM, this approach fails to safeguard the agent against in-TD threats. However, note that TD security against a malicious VMM is still upheld by TDX, which potentially dampens the cloud provider's incentive not to fulfill its duties.

VMM-based resource separation requires no control flow or CPU context switches from the untrusted execution to protected agents, thus making it easier to ensure availability and security of protected agents against the most-privileged adversary inside the TD. In contrast, security offered by privilege-layering hinges on access controls enforced by the most privileged software upon less-privileged ones. In the event of a full-stack TD compromise (e.g., the adversary in the TD kernel in regular TDs or in L1VMM with TDs using partitioning) the TD owner loses all security in her TD, including those supposedly protected agents.

Moreover, resource separation strictly complies to the lest-privilege-principle, as it neither promotes the protected agent to a higher privilege nor modifies existing privileged software. However, privilege-layering does not, as it fundamentally taxes a high-privilege layer to resolve a low privilege layer's *internal* security problems. Hence, schemes following this approach [2, 61, 68] refrain from adding complex application logic to VMPL0, in order to restrain VMPL0 code size bloating. A side-effect is that they are *unscalable* in terms of supporting multiple or complex agents.

Table 1: Security Model Comparison.

Compromised	Priv. Layering	Res. Separation
VMM	□ ◊	□ ♦
VMPL0 / L1VMM	■ ♦	■ ◊
User-mode Agent	■ ♦	□ ♦
Kernel-mode Agent	■ ♦	■ ♦

□/■: secured/broken TD; ◊/♦: secured/broken agent

Deployment. Our approach is relatively easier to deploy as it makes no changes to a TD's system software. The privilege layering approach implies that the (normal) VM kernel is dislodged from the highest privilege level. The relocation becomes onerous if the VM privilege level is not transparent to the kernel. However, our approach requires the cloud support, which impedes its practicality.

Functionality. Privilege layering empowers the agent to control the target software by accessing its CPU context [68]. Unfortunately, resource separation restricts an agent to use the pre-assigned vCPU. Hence, it is difficult for an agent protected by resource separation to control others' executions or access their CPU contexts.

Performance. While the two approaches have similar speed to read the TD memory, they have different types of performance tolls. The primary cost in resource separation is due

to resource provisioning changes. Privilege layering incurs persistent overhead due to the additional tasks in the VMPL0 layer. The cost is much more considerable if the rich OS kernel is relocated. Moreover, schemes with complex logic suffer from the costs due to decoupling bulky functions from the essential ones.

In short, it is imprudent to claim that one approach is generally superior to the other. Resource separation offers stronger security, easier deployment and better scalability with minimal system changes. The privilege layering approach, while more complex and invasive, is suitable for security tasks demanding close interactions with untrusted executions. In fact, these two approaches do not conflict with each other and a hybrid approach can better harden in-TD executions.

4 Design Details

We elaborate TETD's design details for exclusive mode and collaborative mode agents.

4.1 Exclusive Mode Execution

An exclusive mode agent appears as a parallel subsystem to the TD kernel. At any point of time, either the agent subsystem or the TD is active.

4.1.1 Agent Bootup

Agent bootup proceeds in two phases. During the kernel loading stage of TD creation, the agent subsystem image (including its TETD-G and the agent binary) is loaded into the TD private memory as a kernel module. Next, the loaded TETD-G initializes the runtime of the subsystem with the VMM's assistance.

Attestation. Agent bootup does not disrupt existing TD attestation. Instead, it can be easily included into attestation, thanks to TDX's flexible support for measured TD launch similar to TPM-based authenticated boot. As explained in Section 2, TDX allows loaded software during TD creation to build the chain of measurements starting from the TD virtual firmware to the TD kernel. Since the measurements are invoked by software, the chain can be further extended to kernel modules including the agent subsystem image by having the kernel make further measurement. The measurement results are stored in the so-called Runtime Measurement Registers and used for remote attestation. Note that the measurement only reports the integrity of the loaded agent image.

System Environment Initialization. To enable the agent to execute independently without relying on services from the TD kernel, TETD-G sets up the needed environment using resources not shared with the kernel. Besides the memory pages used by the agent and TETD-G, the environment also comprises a paging hierarchy and a suite of system data structures e.g., IDT and GDT.

Mappings. The VMM chooses the GPA region within the TD's GPA range for the agent subsystem. Specifics of the region which is dubbed as the agent's *worksite* is passed to TETD-G so that it prepares a four-level guest paging hierarchy to map the agent and TETD-G's code (including interrupt and exception handlers), data and stack to the assigned GPAs.

Depending on the agent's functionality, the hierarchy may optionally include VA-to-GPA mappings used by the TD kernel. However, the kernel virtual memory pages are mapped as non-executable for the sake of security. Since TETD-G runs in Ring 0, it can update the guest page tables to satisfy the agent's runtime needs for reading/writing targets in the TD. Once the new paging hierarchy is constructed, TETD-G switches `CR3` to effect it immediately.

As explained in Section 2.1, the TD kernel and applications share the same SEPT as the agent subsystem since the TDX Module assigns one SEPT per TD only, as illustrated by Figure 5. Thus, GPA blocking makes the same impact to both the TD and the agent subsystem at the same time.

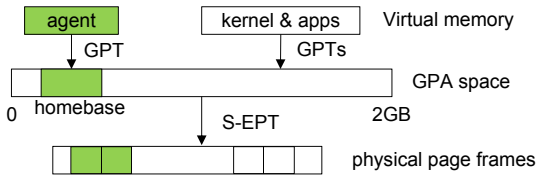


Figure 5: One Secure EPT serves for the agent and other TD software which have separated GPTs.

Interrupt and Exception Handling. TETD-G masks external interrupts. It also has dedicated handlers to handle interrupt and exceptions, without relying on the TD kernel. TETD-G prepares the handlers by configuring dedicated IDT and GDT pages. For convenience, these structures use the same virtual addresses as their counterparts in the kernel.

4.1.2 Agent Sleep & Activation

After initializing the system environment, TETD-G puts the agent to sleep. Specifically, it issues a VMCALL to the VMM. In response, TETD-H activates the GPA blocking based on the received arguments and runs TLB tracking to shoot down TLB entries holding previously used GPA mappings. (Note that the register context of the paused agent vCPU is saved by the TDX Module, not by the VMM.) As a result, the agent enters sleep with its memory regions and vCPU context securely isolated from any other TD thread's access.

When an activation request is delivered to the VMM, TETD-H wakes up the agent in sleep so that it runs with all other TD threads being paused, as illustrated in Figure 6. The agent state transition is controlled by TETD-H with the following steps.

Step 1. Pause TD. TETD-H schedules off all running vCPUs

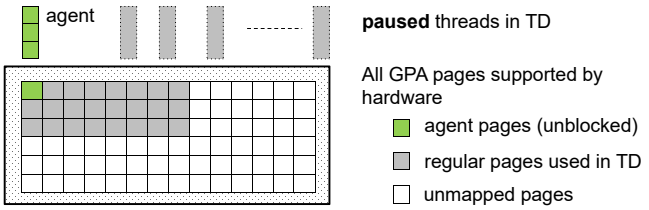


Figure 6: Illustration of exclusive mode execution.

assigned to the TD from their hosting LPs by sending out IPIs (as described in Section 2.2). Following the resulting TD-Exit, the TDX Module passes the control of these LPs to TETD-H.

Step 2. Unblock Agent. TETD-H notifies the TDX Module to unblock the agent worksite. As a result, the agent GPAs are allowed to be translated to physical addresses.

Step 3. Resume Agent vCPU. TETD-H issues a SEAMCALL for TD-Enter so that the TDX Module reschedules the agent vCPU to the target LP. Once the vCPU is scheduled on, it resumes TETD-G's execution following the VMCALL that puts the agent to sleep, i.e., to fetch the instruction next to the VMCALL invocation. TETD-G activates the agent execution environment and passes the control to the agent.

Step 4. Re-entering Sleep. After completing its workload, the agent requests for sleep. TETD-G uses another VMCALL to re-enter into sleep as described in Section 4.1.2. TETD-H schedules all TD vCPUs back to their LPs. Due to the TLB tracking in Step 2, the resumed vCPUs do not use the stale TLBs which may contain unblocked mapping to the agent worksite.

To summarize, the full-cycle of an exclusive execution session comprises TETD's one IPI broadcast to the TD, three SEAMCALLs (GPA unblocking, TLB tracking, and TD-enter) to activate the agent, followed by three SEAMCALLs (GPA blocking, TLB tracking, and TD-enter) to put it back to sleep and resume other vCPUs. Note that an exclusive mode agent runs and sleeps at a fixed worksite assigned during bootup. Next, we show that a collaborative mode agent behaves differently.

4.2 Collaborative Mode Execution

Recall that the agent in the collaborative mode runs simultaneously with untrusted TD threads. Since GPA blocking takes effect on all vCPU cores in the TD including the agent's, it cannot be used to protect the running agent. Our approach is to have the agent run in a secret worksite, shown in Figure 7. The TD adversary is forced to correctly guess GPAs before making any access. Given the vast GPA space, the success probability is practically negligible. More importantly, any incorrect access immediately triggers a TD-Exit and is caught by the VMM. In other words, the adversary can only make online guess attacks. To proactively protect the agent, the GPA region is re-randomized after detecting an EPT violation outside the GPA range assigned to the TD or on a periodical basis (See Section 6 for detailed analysis). Hence, a collaborative mode agent does not follow the fixed cycle of an exclusive agent. Instead, it behaves like a normal daemon thread.

4.2.1 Agent Bootup, Sleep & Activation

The bootup procedure of a collaborative agent is the same as an exclusive agent, except the following aspects. During bootup, TETD-H randomly selects a GPA from the 51-bit private GPA space and outside of the range (e.g., 0 to 2GB) assigned to the TD. This GPA is set at the starting address as the agent's worksite. Similar to loading an exclusive agent, TETD-H augments the page to the Module to update the TD's SEPT. Then the newly added pages are explicitly accepted by TETD-G, so that the GPA mapped take effect. Note that corresponding SEPT pages are added whenever needed. Another difference from the exclusive agent bootup is about interrupt/exception handling. Since the agent co-executes with TD threads, it is desirable to safeguard the agent vCPU against threads on the TD vCPUs. Hence, TETD-G masks all external interrupts and its interrupt handlers ensure that non-maskable IPIs from TD cores are discarded without further processing.

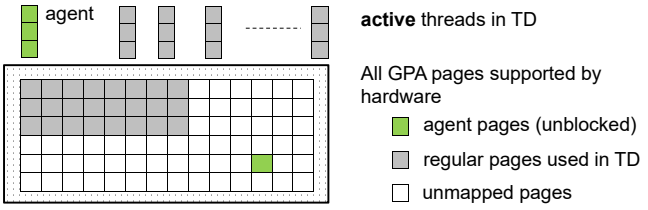


Figure 7: Illustration of collaborative mode execution.

After bootup, TETD-G issues a VMCALL to enter sleep. Similarly to the exclusive agent scenario, its vCPU is scheduled off and the context is saved by the TDX Module. Note that TETD-H does not apply GPA blocking in this scenario.

The activation request for a collaborative agent comes from within the TD. The requesting TD thread on an LP issues a VMCALL which notifies TETD-H to wake up the sleeping agent. TETD-H, on the calling LP, returns to the requesting thread immediately and wakes up the agent on another LP.

4.2.2 Worksite Relocation

A worksite is relocated to a new GPA region by the VMM either periodically as a precaution or after detecting an attack. To detect attacks on the agent, TETD-H hooks the VMM's EPT violation handler. If the offensive GPA is not within the TD's assigned GPA range, TETD-H treats the exception as an attempted attack upon the agent and immediately kicks start the incident response steps as follows.

Step 1. New Worksite Creation. TETD-H pauses the TD vCPUs as in Section 4.1.2. Similar to the bootup, TETD-H

chooses a fresh random GPA region as the new worksite and requests the Module to update the SEPT.

Step 2. Agent Notification. We consider two scenarios according to the agent state. For an agent in sleep, TETD-H wakes it up as in Section 6.5 and passes the new worksite location as the return value of the prior VMCALL. For an agent paused in Step 1, TETD-H issues an IPI to the agent vCPU. TETD-G's IPI handler recognizes the IPI as an alert for relocation and issues a VMCALL to retrieve the worksite location from TETD-H.

Step 3. Agent Copying. TETD-G accepts the new worksite with TDCALL so that the new GPA pages take effect. It then copies all agent code and data pages from its current worksite to the new one. Next, it constructs a new paging hierarchy to clone its virtual address space to the worksite. Finally, it switches to the new hierarchy by loading `CR3` with the new root and issues a VMCALL to inform TETD-H that relocation is completed. Note that the new worksite agent uses the same virtual address layout as the current worksite, only changing the GPA underlying. This design easily supports use of pointers on virtual addresses.

Step 4. Expired Worksite Disposal. TETD-H issues SEAMCALLs to remove GPA mappings of the previous worksite and perform TLB tracking.

5 Implementation

We implement a prototype of TETD and evaluate it on a server with Intel Xeon Silver 4510 processor (24 physical cores) and 512 GB of 16-channel RAM. The host OS is Ubuntu 24.04 LTS. The guest TD is configured with 16 vCPUs and 2 GB RAM and installed with Ubuntu 24.04 LTS. The host OS uses 46-bit physical addresses and 57-bit virtual addresses, while the guest uses 52-bit GPAs and 48-bit VAs. TETD-H and TETD-G consist of 823 lines and 1,637 lines of C code, respectively. TETD-H resides in the host KVM codespace and manages an expandable 16 MB memory pool for agents, while TETD-G with a minimal stub agent occupies 0.2 MB.

TETD-H and TDX TCB jointly form the TCB of TETD. However, as explained in Section 3.1, its compromise does not affect TD security assurances provided by TDX. In other words, TDX TCB is not modified. Similar to code inserted to an SGX enclave by Intel's SDK, TETD-G provides the necessary functionalities an agent needs to benefit from TETD protection. Strictly speaking, it is not part of the TCB for the agent or TETD, because it is subject to the TD owner's vetting and modification. A malicious TETD-G in one agent subsystem does not affect others.

5.1 TETD-H

TETD-H is a suite of 8 VMCALL handlers providing TETD related services (e.g., to activate an exclusive agent in sleep). These handlers reside in the KVM code space. A full list of APIs are in Appendix A. These VMCALLs from a TD to the VMM are issued via TDCALL with register `RAX` bit 15:0 specifying the type as VMCALL. Register `R10` is used to specify VMCALL number, `R11`–`R14` are used as arguments. We use the VMCALL number range $[13-20]$ for the new calls.

Physical Pages Handling. Pages used for TD agents must adhere to the standard TD private page requirements, which means they should be located in CMR regions (just like SGX enclaves). TETD-H manages and keeps track of all pages allocated. One minor issue is that, these pages are not mapped to the TDP and not accounted for the QEMU process. Thus, the host OS cannot automatically reclaim it when the TD is destroyed. To circumvent this problem, we use a garbage collector that runs periodically, checks whether the TD is still active and updates the agent metadata table (Table 6). Once a TD no longer exists in the system, its agent pages are reclaimed.

VE Suppression. When the guest TD accesses a GPA that does not exist in either EPT, two mechanisms may handle this: a Virtualization Exception (#VE) or an EPT-violation. A #VE is handled by the guest OS, and the VMM is unaware of it; while a EPT-violation directly exits the VM and handle it by the VMM. When using secret worksite, to make sure that each malicious activity triggers a relocation, we enforce #VE suppression for the whole TD. To control this behavior, we can use Secure EPT entries to manage it on a page-by-page basis, or configure it globally in the VMCS. Although the VMCS, as part of the TDVPS, is managed by the TDX module. Although managed by the TDX Module, the VMCS is accessible to the VMM using `VMPTRLD` and `VMPTRST` instructions. In our prototype, we manually configured the VMCS to suppress #VE across the entire TD, ensuring that any attempt to access blocked memory or guessing non-existed GPA would be detected.

5.2 TETD-G

TETD-G handles system-level tasks such as managing the paging hierarchy and also works as the middle-man between the agent and TETD-H. In the following, we describe four key components in TETD-G: ① Agent SDK, ② Agent Environment, ③ TDCALL Wrapper and, ④ Agent Encapsulation.

Agent SDK. TETD-G enables the agent with capabilities through Agent SDK, a static library that is compiled together with the agent. By calling the library functions, the agent can partition its workflow into phases.

Each agent begins its workflow using *init*. This call registers a function pointer that handles agent initialization, enabling TETD-G to invoke it during Bootup. At this phase, agents also *register* call IDs and entry points. The *ret* interface is used to manage the graceful and secure exit of agents. If not invoked, TETD-G returns a failure status after a user-space

agent reaches end-of-execution. To safeguard against eavesdropping in the VMM, TETD-G encrypts the arguments and return values using `AES-NI` with a pre-distributed key. In collaborative mode, agents process return values through shared buffers. However, this requires additional security measures from the developer, as TETD-G does not enforce encryption on the buffer here.

Agent Environment. TETD-G functions similarly to a mini-OS. It deals with system environment such as paging hierarchy, interrupt and stores related metadata. Its workflow is described as followed.

Upon bootup, a separate stack, guest page table, IDT, and GDT are generated for the agent, locating in its private memory. Necessary page table hierarchies are constructed, so that the agent accesses its own private pages without trusting the guest kernel. In TETD-G paging, page table entries belonging to the guest kernel GPAs are linked to the original hierarchies. These entries are set to non-executable to prevent accidental calls to the kernel. A dedicated IDT is also initialized, while empty handlers are used to mitigate attacks from other cores. Additionally, a separate GDT is created into the agent's private space to protect against segment overwrite. For agents running in userspace, TETD-G also controls the privilege switch of the agent vCPU, by setting return address to TETD-G in the `IA32_LSTAR` MSR.

TDCALL Wrapper. The wrapper library handles all the assembly operations used by other components. It consists of an assembly file containing the core instructions and a shim written in C. The shim packs and parses parameters passed through registers. This allows TETD-G to seamlessly transit between high-level functions and assembly instructions.

Agent Encapsulation. TETD-G and the agent are a kernel module loaded during system boot. The compiled binaries are placed in the extra modules directory. To load the module automatically during system boot, we leverage the systemd [56] to this ends. A custom systemd service unit is created for each agent, ensuring they are initiated at the correct stage in the boot sequence (After SYSINIT.TARGET).

6 Evaluation

6.1 Security Analysis and Evaluation

TETD security addresses the risk of an in-TD adversary accessing an agent's memory or interfering with its execution. We assess it from both system and software perspectives. As TETD defends exclusive agents using GPA-blocking and collaborative ones using secret locations, our system security analysis addresses them separately.

6.1.1 Effectiveness of GPA Blocking

We experiment with two attacks targeting a given GPA-blocked memory page. One is to read the blocked memory using the agent's previous mapping, which is to validate whether TLB entries used by the agent remain valid. The other is to build a new mapping to read the GPA. In both experiments, the read, write, or execute attempts on the blocked GPA trigger an EPT violation and are caught by the host VMM, which GPA-blocking effectively prevents any TD thread from accessing the protected page.

Since current TDX versions do not allow a peripheral device to access a TD's private pages [31], probably due to incompatibility to MK-TME, DMA attacks cannot threaten TETD agents. In the event that the future Trusted Execution Environment I/O (TEE-IO) [33] allows DMA accesses to private pages, it is also promising to extend TETD to counter the attacks, because the host VMM retains its role of resource management.

6.1.2 Secrecy of Worksite Relocation

The runtime security of a collaborative agent depends on the secrecy of its worksite location. Since the secure EPT is shared by the agent and the TD kernel, the GPA-to-HPA mappings for the agent are also open for the TD kernel to use. However, the adversary must guess the GPA page used by the running agent in order to set up its VA-to-GPA mappings before making any access. Note that this GPA guessing is an *online* attack in TETD, in the sense that any direct access using an incorrect guess leads to an EPT-violation exception and the core is trapped to the host VMM to handle. In other words, a failed guess can be immediately caught by TETD-H.

Probability of Successful Guess. We further show that the success probability of the adversary's one guess is negligible. Suppose that a worksite of size n bytes is in use. As the platform uses 52-bit GPA addresses, the entire GPA space is 2^{52} bytes. Note that TETD-H randomly picks the GPA site from the *private* GPA space (51-bit) except those used by the TD. Hence, the success probability ρ of one guess upon any GPA in the worksite is $\frac{n}{(2^{51}-M)}$, where M represents the guest physical memory allocated to the TD. Considering a typical CVM with memory between 8 GB and 512 GB, i.e., $2^{33} \leq M \leq 2^{39}$ and an agent with 2 MB size ($n = 2^{21}$), we have $\rho \approx 1/2^{30}$, which implies that the attacker is expected to make approximately $1/2^{29}$ guesses before success. Note that after the worksite is relocated, the attacker's efforts targeting the previous location become invalid.

Side-Channel Attack Mitigation. The relocation scheme is geared to reduce the risk of side-channel exposure of secret GPA, as all TD threads are stalled and cannot launch side-channel attacks during relocation. It is generally more difficult to attack GPAs secrecy than VA or PA secrecy, since GPAs neither affects the agent's control flow or data flow, nor

appear in the memory or general registers. Note that speculative execution attacks [35, 42] could potentially be used to stealthily guess a GPA without triggering an EPT violation. The approach is that speculative execution is applied to leave information in the cache reflecting whether the guessed GPA is valid and then the attacker uses cache side-channel [70] to deduce it. While such attacks are more silent than direct GPA accesses, they do not reduce the prior deduction of the expected amount of guesses. There are two countermeasures to mitigate the risk. One is to increase the frequency of worksite relocation so that the adversary cannot complete the needed trials within the shortened time window. The other is that the agent can disturb the adversary's cache-based side-channel analysis. The rationale is that the traces left by speculative execution using a wrong GPA only appear in L3 cache, not in L1 or L2 cache (otherwise it implies the adversary has already accessed the data before.) Hence, the agent can introduce noises to L3 cache, e.g., to randomly load data or flush it, so that the adversary cannot have a reliable side-channel to infer traces left by speculative execution. Some studies achieve side-channel resistance through constant-time hardware design, resource reuse, or threshold implementation with provable security [22, 34].

6.1.3 Software Attack Analysis

As a security system, TETD is not meant to cope with software attacks exploiting vulnerabilities in protected agents. Nonetheless, it is worth analyzing how TETD's architectural features can help protect agents, especially in view of TD kernel exploits being the primary motivation of TETD. Vulnerabilities in commodity kernels, such as CVE-2024-1086 [24] and CVE-2024-53141 [13], are due to both its widely exposed interface (more than 300 system calls in Linux) to support a multitude of services and also the inclusion of various third-party device drivers. TETD ensures no control flow transfer or CPU sharing between an agent and any TD thread. Hence, an agent's interactions with a TD thread are restricted to data passing only. As an agent's functionality is much simpler than the kernel, it is easier to enforce input sanitization. Note that TETD-G in an agent subsystem exposes no interface to the TD kernel or other agents. In short, to develop an agent with less or zero vulnerabilities is more achievable than secure kernel development.

6.2 TETD Performance Evaluation

In the following, we measure TETD's own operation overhead by experimenting with an empty-load agent. Performance in realistic settings is studied through two representative TETD applications in Section 6.3 and 6.4 respectively.

6.2.1 CPU Time of TETD Primitives

We first measure the round-trip time of six SEAMCALLs and two TDCALLs involved in TETD operations, i.e., the interval between the caller's instruction issuance and its receiving of the return value(s). Except that GPA-blocking/unblocking use 2 MB as the workload and VMCALL has an empty handler in the host VMM, the rest have a fixed workload determined by TDX specification. Table 2 reports the experiment results.

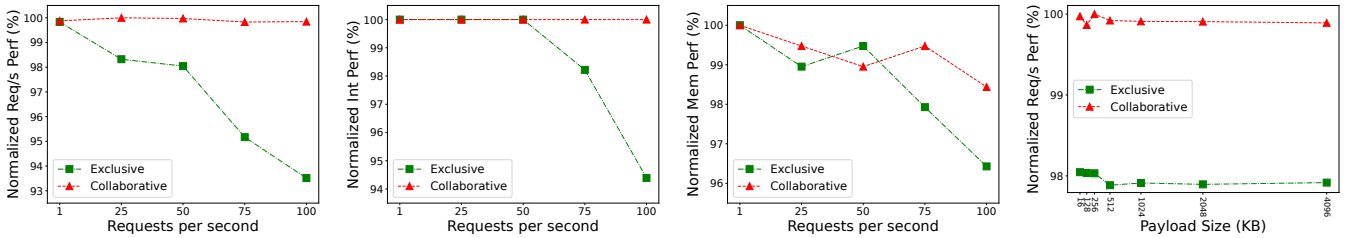
Table 2: Roundtrip Time of TETD Relevant SEAMCALLs and TDCALLs (in μs).

SEAMCALL/TDCALL	Time	Workload	Adjustable
`TDH.MEM.RANGE.BLOCK`	10.2	2 MB	Yes
`TDH.MEM.RANGE.UNBLOCK`	4.7	2 MB	Yes
`TDH.MEM.TRACK`	2.1	N/A	N/A
`TDH.MEM.SEPT.ADD`	6.9	1 Page	No
`TDH.MEM.PAGE.AUG`	8.0	1 Page	No
`TDH.MEM.PAGE.REMOVE`	3.4	1 Page	No
`TDG.MEM.PAGE.ACCEPT`	4.2	1 Page	No
`TDG.VP.VMCALL`	10.6	N/A	N/A

To measure CPU time of TETD's primitive operations, we develop a zero-function agent with a 128 KB memory dummy payload. Table 3 shows their average costs for the agent to run in exclusive mode and collaborative mode. The bootup time, i.e., the interval between TETD-G starts and the agent init function returns, dominates the cost of agent initialization (about 1.2 milliseconds) which barely impacts the whole TD kernel launch (about 4.5 seconds). The table also shows that the runtime operation overheads for exclusive execution are much higher than for collaborative execution, which is due to the former's need for blocking and unblocking the agent's GPA region. However, collaborative agents incur relocation costs. Since the relocation cost varies with the amount of memory to copy, we run experiments with various agent sizes, as reported in Table 4. Note that the cost is close to the initialization cost since both are dominated by memory copy and Secure EPT configuration. To put the data in perspective, it takes the kernel about 680 µs to copy 2 MB of data on our experiment platform.

Table 3: CPU Time of TETD Operations (in μs).

	Operation	Exclusive	Collaborative
Init	Bootup	1103.5	1197.2
	Sleep	172.2	23.1
	Total	1275.5	1220.3
Runtime	Activate	203.2	32.1
	Sleep after exec	109.4	10.3
	Total (one round exe.)	312.6	42.4



(a) System behavior impact on Apache throughput

(b) System behavior impact on Lm-Bench (CPU performance)

(c) System behavior impact on Lm-Bench (memory performance)

(d) Agent size impact on Apache throughput

Figure 8: Normalized Benchmark Data Reporting TETD's Impact on TD Performance.

Table 4: CPU Time for Relocation (in μs).

Payload Size (MB)	1/8	1/4	1/2	1	2	4
Relocation Time	221	249	353	549	916	1402

6.2.2 TD Performance Impact From TETD

We study TETD's impact on TD performance from three angles: TETD's system behaviors, the size of the memory to isolate, and relocation. In order to focus on the impacts owning to TETD as a system, instead of an agent's own execution, we conduct experiments with an empty-load agent. All experiments are in a TD allocated with 16 vCPUs with one assigned to the agent.

Common System Behavior Impact. TETD's system operations harming TD performance include cache flushing and TD exit/resume as they either slowdown or temporarily suspend thread execution in the TD. We consider them as the primarily performance impact from TETD, as they are introduced by it and common to all kinds of agents.

We use Apache Benchmark [17] to assess impacts from TD disruptions such as exit and pausing and use LMbench [50] to assess cache miss effects upon memory and CPU performance. In the experiments, we tune the frequency of agent activation/sleep where these operations are taken. Albeit incurring negligible execution time due to no workload, our agent occupies 16 KB memory requiring protection. We run the agent in both exclusive and collaborative mode. The results are reported in Figure 8.

Both Figure 8(a) and 8(b) show the clear trend that whole-TD pausing takes a noticeable toll on TD performance, approximately 6.8% drop due to 100 times of pausing per second. The performance hit is primarily due to the time of TD pausing. The conclusion is drawn by comparing the curve for collaborative execution with the one for exclusive execution. As both agents are requested with the same frequency, the callers face the same number of TD exit/resume. However, each request of the exclusive mode incurs a whole-TD pausing, while the request for collaborative mode does not. It implies that exclusive mode agents are not ideal for tasks demanding frequent invocations. Figure 8(c) shows TETD's impact on TD memory performance, with both types of executions exhibiting a similar pattern.

Protected Memory Size. The second group of experiments shows how the performance impact varies with the agent size. We also use an empty agent, with different amounts of static data to expand its memory footprint without increasing its execution time. The agent is activated with a constant frequency of 50 Reqs/s.

Figure 8(d) shows the benchmark results with agent payload ranging from 16 KB to 4 MB. As the size increases, there is a visible performance drop in exclusive mode, suggesting that increasing memory footprint introduces some overhead due to guest page table operations. On the other hand, the collaborative mode performance degradation is less noticeable as the agent size increases, demonstrates better scalability.

6.3 TETD Application I: TD Introspection

To assess exclusive mode execution performance in a realistic setup, we build an agent called *TDReader* for TD introspection services and compare its performance against 00SEVen [61].

6.3.1 TDReader

The TD owner's remote server holds a pair of certified public and private keys and communicates with TDReader through an untrusted proxy in a non-confidential VM residing in the same platform as the TD. TDReader establishes a TLS-like end-to-end channel with the server through a shared key k to ensure communication integrity and secrecy. Figure 9 illustrates how TDReader receives commands and returns results. The proxy and TDReader exchange data and synchronize their operations through a TD-VM shared memory region using QEMU's Inter-VM Shared Memory (IVSHMEM) [57] which maps a set of physical pages to the TD via the TD's shared EPT and to the VM via its EPT. Note all data written to the buffer is encrypted under k by either TDReader or the owner's

remote server.

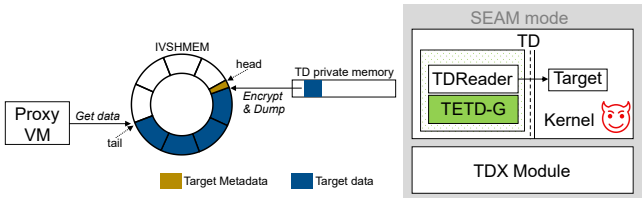


Figure 9: Communication Channel between TDReader and the Proxy VM.

To support introspection, we grant TDReader Ring-0 privilege so that it can adaptively create mappings and has the sufficient privilege to read the TD kernel. For physical memory acquisition, TDReader builds its own VA-to-GPA mappings to dump the contents out. For virtual memory introspection, TDReader accesses the target virtual memory by using the target's VA-to-GPA mappings, which are copied into its own page tables with execution permissions (if any) stripped off. As explained in Section 4.1.1, TDReader's paging hierarchy already has kernel mappings after its bootup. For additional target mappings, TDReader can copy the needed entries provided by the remote server into its own paging table.

6.3.2 Implementation and Experiments

We implement a TDReader agent prototype with 1.8 KLoC of C code (excluding dependencies such as cryptographic libraries and symbol table handling). The binary of TDReader, compiled along with TETD-G, occupies 2.9 MB. TDReader uses ECDSA with secp256r1 (P-256) curve for request authentication and AES-NI to encrypt the outcome in AES-GCM mode. The TD-VM shared memory comprises of 64 pages and is organized as a ring buffer.

We run two experiments to measure TDReader's performance and dependability, i.e., guaranteed launched without kernel support. In both experiments, we manually introduce the TD kernel crash by using `echo c > /proc/sysrq-trigger`. We then issue the request from the proxy VM to the VMM to activate TDReader.

Experiment 1: Full-TD Memory Acquisition. We create a TD configured with 2 GB system RAM in our platform and run experiments to dump all contents in the RAM. As a virtual machine, the TD's RAM is set at seven GPA regions whose base addresses and sizes are chosen by default. Different from non-confidential VMs, all assigned GPA pages are mapped with host physical pages by the VMM during TD bootup. TDReader reads those GPA regions and writes all contents to the shared buffer after encryption. In total, 1.994 GB are retrieved and it takes about 3.1 seconds to complete.

Experiment 2: Kernel Object Retrieval. In our second experiment, TDReader retrieves data objects from the crashed kernel. We select eight compatible introspection targets from previous work([44], [61]). TDReader retrieves kernel objects by using virtual addresses obtained from the symbol table and accesses them through the agent's own guest page table.

Table 5: Inspection Time Comparison (in milliseconds).

Task	TETD	00SEVen*	Speedup
Task list traversal	18.5	120	6.5
Privilege analysis	17.9	125	7.0
VFS hook detection	16.7	115	6.9
TTY keylogger checks	21.8	120	5.5
System call checks	4.4	60	13.6
Process memory map	21.6	140	6.5
Module list traversal	19.3	130	6.7
Open file list	28.9	160	5.5

* The numbers for 00SEVen are estimated floor values (i.e., fastest speed) of the corresponding time in Figure 6 of [61] which does not provide raw data.

Table 5 compares TDReader's performance with 00SEVen's introspection using a VMPL0 agent with a local-TCP analysis client. The data shows that TDReader is multiple times faster than 00SEVen across all introspection tasks. Note that the results should be interpreted with caution, as the two tools run on different hardware platforms whose architectural differences, CPU speed, and DRAM chip performance, etc. need to be factored into consideration. However, we observe that TDReader reaps its performance benefits from its architectural advantage over 00SEVen's.

00SEVen consists of a lightweight front-end in VMPL0 for memory reading only and a backend in another VM for kernel object parsing. Note that kernel introspection typically follows the read-parse-read paradigm due to the need for semantics extraction. Hence, the 00SEVen front-end and back-end depend on each other's results to proceed, which impairs its overall performance due to the incurred communication overheads and inability to run in parallel. In contrast, TDReader has a unified design with all functionalities in one space. We remark that the reason behind the design difference is 00SEVen's privilege-layering approach and TETD's resource-separation approach. As explained in Section 3.4, 00SEVen must refrain from placing the complex backend into VMPL0 in order to keep the CVM TCB small. For TETD, the agent is *not* a TCB to the TD and importing bulky code is only an engineering issue.

6.4 TETD Application II: TDSigner

The second application, called TDSigner, demonstrates how TETD can be used to harden a critical TD application by creating an enclave-like environment to run functions with secrets. We apply it to a TLS server and measure its performance using benchmarks.

6.4.1 TDSigner

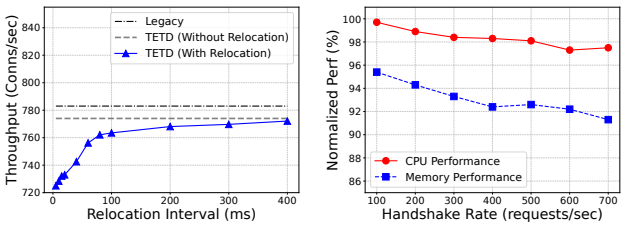
Consider a network service TD (e.g., a cloud-based DNS server) that needs to sign outgoing messages. To safeguard the owner's signing credentials against runtime kernel compromise, we partition the signing service into (1) a front-end residing in the TD for application-related functionalities, and (2) TDSigner as a user-space TETD agent in collaborative mode. In a similar vein to an SGX enclave, TDSigner runs within the address space of the service application and holds the long-term signing key protected by TETD-imposed one-way isolation. It can directly read and write virtual addresses used by the front-end.

Signing Key Installation. TDX provides the hardware-based attestation mechanism that allows the TD owner to authenticate her TD together with TD-provided data such as a public key generated within the TD. However, the attestation cannot differentiate between a TETD agent and the TD controlled by its kernel. Hence, we propose a double-lock scheme to securely import the owner's signing key to TDSigner without exposing it to the TD kernel or the VMM. Let S be the signing key. During TD image preparation, the owner generates a random AES key k, and embeds the ciphertext $Enc(S,k)$ to TDSigner. After her TD is up and running, the owner authenticates the TD and returns k to the TD through the encrypted channel established following TD attestation. The TD then passes k to TDSigner which can decrypt S. Note that only the TD has access to k, however, not the ciphertext of S. Hence, S is secure when the TD and the VMM do not collude, which is consistent with our adversary model. We apply TDSigner to harden a TLS server's use of its private key in TLS handshakes where the key is used to sign `ServerHello` messages in TLS v1.3 [59]. As our TETD prototype currently does not support multi-threading, the environment includes one TLS server and one client thread.

6.4.2 Implementation and Experiments

TDSigner is implemented with 477 LoCs, excluding dependencies, and occupies 1.2 MB. We use RSA-3072 and SHA-256 [25] as the crypto algorithms. We instrument the TLS process of the `s_server` application in the OpenSSL 3.5 suite. It issues a VMCALL to invoke TDSigner, who receives the signing request and returns the signature via designated buffers. The signature is computed over a structured message consisting of: octet 32 (0x20) repeated 64 times, a context string, a single 0x00 byte separator, and the transcript hash. Our experiments show that signing 130 bytes data takes about 0.89 milliseconds, modestly higher than the original 0.73 milliseconds. The main overhead is for waking up the agent from sleep and switching into and back from user-space.

Experiment 1: TLS Handshake Performance. We evaluate the impact of agent relocation frequency on TLS handshake throughput. In this experiment, we use three configurations.



(a) TLS handshake throughput under varying relocation interval δ.

(b) System behavior impact under varying workload, $\delta = 5\ ms$.

Figure 10: System performance for TLS workload.

The first measures legacy OpenSSL performance, the second uses TDSigner with relocations at fixed intervals ranging from 5 to 400 ms, and the third disables relocation, serving as a comparison baseline. As presented in Figure10(a), smaller relocation intervals (e.g., 5–100 ms) offer stronger security by reducing the window for potential side-channel attacks. However, this comes with performance overhead. As the interval increases beyond 200 ms, throughput approaches the configuration without relocation (774 conn/s), eventually nearing the legacy performance (783 conn/s).

Experiment 2: TD Performance Impact. In Figure 10(b), we measure system CPU and memory performance using LMbench [50] with a server actively executing with TDSinger at various handshake rate (with the relocation interval set at 5 ms to evaluate the highest-pressure overhead). As the handshake rate increases from 100 to 700 requests per second (near the highest throughput), CPU performance remains relatively stable. In contrast, memory performance shows a more noticeable decline, reaching 91% at peak load.

The application demonstrates TETD's capability to protect a segment of critical code and data in an application against a corrupted kernel, and also shows how a secret can be securely imported from the owner to an agent. It is in our future work to dynamically launch authenticated and authorized agents within a running TD.

6.5 TETD Application III: SuperAgent

6.5.1 Multi-Agent Support

With a slight extension, TETD can support multiple agents in a TD. To manage them, TETD-H sets up an agent table for each TD as shown in Table 6, which is updated according to runtime states of the agents.

TETD permits more than one collaborative agents to sleep and wait for execution at the same time, which offers more flexible ways of trusted execution. It is the owner's responsibility to ensure the installed collaborative agents do not have overlapping interfaces exposed to the TD threads. Concurrent executions of exclusive agents are disallowed, as exclusive agents are meant TD level services, such as introspection.

However, installing multiple exclusive agents boosts system reliability by avoiding a single-point-of failure. Noted that TETD introduces two control bits to manage asynchronous state discrepancies. One for signaling the agent to pause, and the other reflects if the pause has taken effect.

Table 6: An agent table managed by SuperAgent. Mode 0 stands for Exclusive and Mode 1 for Collaborative.

ID	Mode	Worksite	Size	State
0	0	0xFF..20 ...	2 MB	Sleep
1	1	0xFF..10 ...	64 KB	Active

6.5.2 SuperAgent Implementation

To show TETD's support for multi-agent, we develop SuperAgent that governs others in the TD and is able to access their memory. The owner constructs SuperAgent image with a unique flag indicating its special role. After SuperAgent is loaded, it is given the agent ID-0. To enable SuperAgent, we utilize the following admin API: TOGGLE_DISABLE and ADMIN_ACCESS. The former instructs TETD-H to stop the target agent from waiting for execution and to reject future activation request. The latter instructs TETD-H to unblock an exclusive agent's worksite and expose it to SuperAgent. For a collaborative agent under checking, TETD-H returns its secret worksite. Note that TETD-G underneath SuperAgent configures the guest page table to map the target GPA.

Experiment. We implement SuperAgent based on TDReader by adding 42 lines of C code, and test it overseeing TDSigner, the collaborative agent in our previous application 6.4. Suspecting that a TD adversary attacks TDSigner by feeding it with poisonous data, the TD owner first remotely activates SuperAgent to suspend and disable TDSigner as a precautionary measure. She then instructs SuperAgent to dump all memory of TDSigner. Our experiment successfully enables the owner to acquire TDSigner's memory. The operation of temporarily pausing the TDReader agent takes 22.9 μs. The total procedure costs 322.9 μs.

In short, our study with TDReader and SuperAgent shows that a TD owner benefits from TETD to gain a reliable and secure foothold to manage and service a TD, despite the barrier induced by TDX. On the one hand, an exclusive agent like TDReader has the flexibility of reading target virtual memory, similar to in-VM introspection agents [9, 63]. On the other hand, it has the same security and dependability like out-of-VM introspection techniques [18, 73].

7 Related Work

Confidential VM Security. Our work builds upon Intel TDX, which is part of a broader area of research focused on the security of confidential virtual machines across various platforms. Heckler [60] tampers with TDX registers using interrupts. Google Project Zero has published a security report on TDX [3]. Intel also conducts research on security practices and side-channels mitigation on TME-MK [26, 27, 29]. AMD SEV-SNP is used by Cabin [51], Veil [2], and NestedSGX [68] to achieve in-CVM isolation, although SEV virtual machines are also vulnerable to side channels [37, 38]. Li *et al.* built Realms for CCA to protect VM security [40]. Shelter complements CCA's primary Realm architecture to provide a user-level TEE [71]. CAGE provides confidential computing for GPU in Realms [67].

VM Introspection. While TETD is a general framework for securing in-TD agents, introspection remains a key application. A substantial body of work has been devoted to this area [12, 14, 18, 20, 36, 46, 55, 64, 66, 69, 73], including studies on applicability [20, 36, 69], semantics [14, 18, 73], modeling [55], and efficiency [46, 73]. More recent research focuses on introspection challenges within TEEs, with SMILE being the first to explore live memory introspection of enclaves [74].

8 Discussions

Detectability. TETD does not claim to be undetectable by the TD kernel. Guest kernel-level attackers could detect its presence by comparing the NUM_VCPUS value obtained from the TDG.VP.INFO TDCALL and the number of CPUs actually usable by the kernel. A mismatch between these values would indicate that additional components are present. Upon detecting such discrepancies, an attacker may become aware of TETD's existence and choose to abstain from malicious actions to avoid triggering defensive mechanisms.

Compatibility. Although TETD is designed primarily for TDX, its architecture is adaptable to other CVMs, such as Arm CCA [5]. The fundamental protection mechanisms of TETD, including memory blocking and vCPU scheduling, can be ported onto Realm VMs by revising the RMM [6].

TDX Features. Several TDX features could support security tasks similar to TETD. The first is TD Partition, which supports Nested Virtualization [32], similar to the VMPL. The second is the Service TD [31], allowing Service TDs to be bound to a target TD to access its metadata. This capability is primarily intended for TD migration [30]. We plan to explore these features as more details become available.

VMM Security. TETD 's security assumes that the VMM behaves correctly. Recent fuzzing efforts—such as ViDeZZo [45], HyperPill [8], and Truman [48]—have shown promise in uncovering hypervisor bugs. Other directions include formal verification of VMMs (e.g., SeKVM [39], Differential Testing([47]), VRM [65]), and memory-safe implementations like Firecracker and Cloud-Hypervisor [7, 16]. Detecting runtime VMM misbehavior, however, remains an open

challenge. One direction is to leverage the TDX Module's trusted role. For instance, the module could log SEAMCALLs related to EPT updates, enabling post-hoc validation of the VMM's behavior.

9 Conclusion

TETD provides a framework for ensuring trusted execution for an agent inside a TD. By leveraging vCPU scheduling, memory blocking and the secret GPA mechanism, TETD achieves isolation from the untrusted VMM and malicious kernel. Our work offers a secure yet flexible solution for hosting sensitive agents without breaking any TDX guarantees. We conduct extensive evaluation on overhead and provide three detailed case studies to show TETD's practicality and potential in real-world use.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. We also appreciate the helpful discussions with members of the COMPASS group and acknowledge partial support from Ant Group. This work is partly supported by the National Natural Science Foundation of China under Grant No.62372218, No.U24A6009, and Peng Cheng Laboratory Grant PCL2024A05-1. Xuhua Ding's research is supported by the National Research Foundation, Singapore, and the Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Proposal ID: NCR25-DeSSMU-0001). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore, and the Cyber Security Agency of Singapore.

Ethics Considerations

This research introduces a software solution that maintains service security even if Intel TDX's TD is compromised and guarantees that providers cannot covertly eavesdrop on tenant data. It introduces no attack vectors and thus poses no risk to Intel's interests or reputation. All experiments run exclusively on local machines using synthetic datasets, with no use of cloud servers, ensuring no impact on personal privacy or public services; it involves no human subjects and imposes no physical or psychological burden on the research team.

While this research has rigorously avoided ethical concerns and introduces no new vulnerabilities, we commit to following coordinated disclosure practices in future research. Specifically, any discovered vulnerabilities will be responsibly reported to affected vendors for remediation prior to public disclosure. We pledge to proactively address potential issues arising from this work and develop contingency plans to mitigate unintended consequences.

Open Science

We confirm that the submitted paper complies with the open science policy of USENIX Security'25. We will make the TETD source code (including the guest and host module, along with example agent and deployment guidance) publicly available.

References

[1] Adil Ahmad, Sangho Lee, and Marcus Peinado. HARDLOG: Practical Tamper-Proof System Auditing Using a Novel Audit Device. In *Proceedings of IEEE Symposium on Security and Privacy*, 2022.

[2] Adil Ahmad, Botong Ou, Congyu Liu, Xiaokuan Zhang, and Pedro Fonseca. Veil: A Protected Services Framework for Confidential Virtual Machines. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2023.

[3] Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, and Felix Wilhelm. Intel Trust Domain Extensions (TDX) Security Review. Technical report, Google, 2023.

[4] AMD. SEV Secure Nested Paging Firmware ABI Specification. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56860.pdf, 2024. Accessed: 2024-05-22.

[5] Arm. Arm CCA Security Model 1.0. https://developer.arm.com/documentation/DEN0096/A_a, 2021.

[6] Arm. Realm Management Monitor Specification. https://developer.arm.com/documentation/den0137/1-0rel0, 2022. Accessed: 2024-10-09.

[7] AWS. Firecracker. https://firecracker-microvm.github.io.

[8] Alexander Bulekov, Qiang Liu, Manuel Egele, and Mathias Payer. HYPERPILL: Fuzzing for Hypervisor-bugs by Leveraging the Hardware Virtualization Interface. In *Proceedings of USENIX Security Symposium*, 2024.

[9] Martim Carbone, Matthew Conover, Bruce Montague, and Wenke Lee. Secure and Robust Monitoring of Virtual Machines through Guest-Assisted Introspection. In *Proceedings of International Symposium on Research in Attacks, Intrusions and Defenses*, 2012.

[10] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *ACM SIGOPS Operating Systems Review*, 2008.

[11] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel TDX Demystified: A Top-Down Approach. *ACM Computing Surveys*, 2024.

[12] Thomas Dangl, Stewart Sentanoe, and Hans P Reiser. VMIFresh: Efficient and Fresh Caches for Virtual Machine Introspection. In *Proceedings of International Conference on Availability, Reliability and Security*, 2022.

[13] Divya. PoC Released for Linux Kernel Vulnerability Allowing Privilege Escalation. https://gbhackers.com/poc-released-for-linux-kernel-vulnerability/, 2025. Accessed: May 22, 2025.

[14] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Proceedings of IEEE Symposium on Security and Privacy*, 2011.

[15] Bryan Payne et al. LibVMI: Simplified Virtual Machine Introspection. https://libvmi.com.

[16] Linux Foundation. Cloud-Hypervisor. https://www.cloudhypervisor.org.

[17] The Apache Software Foundation. Apache HTTP Server Benchmark. https://httpd.apache.org, 2024.

[18] Yangchun Fu and Zhiqiang Lin. Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *Proceedings of IEEE Symposium on Security and Privacy*, 2012.

[19] Varun Gandhi, Sarbartha Banerjee, Aniket Agrawal, Adil Ahmad, Sangho Lee, and Marcus Peinado. Rethinking System Audit Architectures for High Event Coverage and Synchronous Log Availability. In *Proceedings of USENIX Security Symposium*, 2023.

[20] Tal Garfinkel, Mendel Rosenblum, et al. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of Network and Distributed System Security Symposium*, 2003.

[21] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *Proceedings of USENIX Security Symposium*, 2018.

[22] Shiyang He, Hui Li, Qingwen Li, and Fenghua Li. A Time-Area-Efficient and Compact ECSM Processor over GF(p). *Chinese Journal of Electronics*, 32(6):1355–1366, 2023.

[23] Owen S Hofmann, Sangman Kim, Alan M Dunn, Michael Z Lee, and Emmett Witchel. Inktag: Secure applications on an untrusted operating system. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[24] Google Inc. CVE-2024-1086: Linux Kernel nf_tables Use-After-Free Vulnerability. https://nvd.nist.gov/vuln/detail/CVE-2024-1086, 2024. Accessed: May 22, 2025.

[25] Intel. Intel SHA Extensions. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sha-extensions.html, 2013.

[26] Intel. Intel Trust Domain Extension Guest Kernel Hardening Documentation. https://intel.github.io/ccc-linux-guest-hardening-docs/index.html, 2023.

[27] Intel. MKTME Side Channel Impact on Intel TDX. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/mktme-side-channel-impact-on-intel-tdx.html, 2023.

[28] Intel. Multi-Key Total Memory Encryption Specification. https://cdrdv2-public.intel.com/679154/multi-key-total-memory-encryption-spec-1.4.pdf, 2023. Accessed: 2024-04-25.

[29] Intel. Trust Domain Security Guidance for Developers. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/trusted-domain-security-guidance-for-developers.html, 2023.

[30] Intel. Intel TDX Module Architecture Specification: TD Migration. https://cdrdv2.intel.com/v1/dl/getContent/733580, 2024. Accessed: 2024-12-01.

[31] Intel. Intel Trust Domain Extensions Module Base Architecture Specification. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, 2024. Accessed: 2024-07-01.

[32] Intel. Overview and architecture specification for TD partitioning of the Intel TDX Module. https://cdrdv2.intel.com/v1/dl/getContent/773039, 2024.

[33] Intel Corporation. Intel TDX Connect TEE-IODevice Guide. https://cdrdv2-public.intel.com/772642/whitepaper-tee-io-device-guide-v0-6-5.pdf, 2023.

[34] Zhipeng Jiao, Hua Chen, Jingyi Feng, Xiaoyun Kuang, Yiwei Yang, Haoyuan Li, and Limin Fan. A Combined Countermeasure Against Side-Channel and Fault Attack with Threshold Implementation Technique. *Chinese Journal of Electronics*, 32(2):199–208, 2023.

[35] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of IEEE Symposium on Security and Privacy*, 2019.

[36] Tamas K Lengyel, Justin Neumann, Steve Maresca, Bryan D Payne, and Aggelos Kiayias. Virtual Machine Introspection in a Hybrid Honeypot Architecture. In *Proceedings of USENIX Workshop on Cyber Security Experimentation and Test*, 2012.

[37] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In *Proceedings of IEEE Symposium on Security and Privacy*, 2022.

[38] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *Proceedings of USENIX Security Symposium*, 2021.

[39] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. Formally verified memory protection for a commodity multiprocessor hypervisor. In *Proceedings of USENIX Security Symposium*, 2021.

[40] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and Verification of the Arm Confidential Compute Architecture. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, 2022.

[41] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *Proceedings of IEEE Symposium on Security and Privacy*, 2021.

[42] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of USENIX Security Symposium*, 2018.

[43] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. Frequency Throttling Side-Channel Attack. In *Proceedings of ACM Conference on Computer and Communications Security*, 2022.

[44] Hongyi Liu, Jiarong Xing, Yibo Huang, Danyang Zhuo, Srinivas Devadas, and Ang Chen. Remote Direct Memory Introspection. In *Proceedings of USENIX Security Symposium*, 2023.

[45] Qiang Liu, Flavio Toffalini, Yajin Zhou, and Mathias Payer. Videzzo: Dependency-aware virtual device fuzzing. In *Proceedings of IEEE Symposium on Security and Privacy*, 2023.

[46] Yutao Liu, Yubin Xia, Haibing Guan, Binyu Zang, and Haibo Chen. Concurrent and Consistent Virtual Machine Introspection with Hardware Transactional Memory. In *Proceedings of International Symposium on High-Performance Computer Architecture*, 2014.

[47] Hongyi Lu, Zhibo Liu, Shuai Wang, and Fengwei Zhang. DTD: Comprehensive and Scalable Testing for Debuggers. *Proceedings of the ACM on Software Engineering*, 2024.

[48] Zheyu Ma, Qiang Liu, Zheming Li, Tingting Yin, Wende Tan, Chao Zhang, and Mathias Payer. Truman: Constructing Device Behavior Models from OS Drivers to Fuzz Virtual Devices. In *Proceedings of Network and Distributed System Security Symposium*, 2025.

[49] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of IEEE Symposium on Security and Privacy*, 2010.

[50] Larry McVoy and Carl Staelin. LMbench: A Benchmarking Tool for Memory and Network Performance. https://lmbench.sourceforge.net, 1996.

[51] Benshan Me, Saisai Xia, Wenhao Wang, and Dongdai Lin. Cabin: Confining Untrusted Programs within Confidential VMs. In *Proceedings of International Conference on Information and Communications Security*, 2024.

[52] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered Insecure: GPU Side Channel Attacks are Practical. In *Proceedings of ACM Conference on Computer and Communications Security*, 2018.

[53] Vikram Narayanan, Claudio Carvalho, Angelo Ruocco, Gheorghe Almási, James Bottomley, Mengmei Ye, Tobin Feldman-Fitzthum, Daniele Buono, Hubertus Franke, and Anton Burtsev. Remote attestation of confidential VMs using ephemeral vTPMs. In *Proceedings of Annual Computer Security Applications Conference*, 2023.

[54] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *Proceedings of USENIX Security Symposium*, 2021.

[55] Jonas Pfoh, Christian Schneider, and Claudia Eckert. A formal model for virtual machine introspection. In *Proceedings of ACM Workshop on Virtual Machine Security*, 2009.

[56] Lennart Poettering et al. systemd: System and service manager. https://systemd.io.

[57] QEMU. Inter-VM Shared Memory device. https://www.qemu.org/docs/master/system/devices/ivshmem.html.

[58] Jianbao Ren, Yong Qi, Yuehua Dai, Xiaoguang Wang, and Yi Shi. AppSec: A Safe Execution Environment for Security Sensitive Applications. In *Proceedings of ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2015.

[59] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018. Proposed Standard.

[60] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. Heckler: Breaking Confidential VMs with Malicious Interrupts. In *Proceedings of USENIX Security Symposium*, 2024.

[61] Fabian Schwarz and Christian Rossow. 00SEVen–Re-enabling Virtual Machine Forensics: Introspecting Confidential VMs Using Privileged in-VM Agents. In *Proceedings of USENIX Security Symposium*, 2024.

[62] R Sekar, Hanke Kimm, and Rohit Aich. eAudit: A Fast, Scalable and Deployable Audit Data Collection System. In *Proceedings of IEEE Symposium on Security and Privacy*, 2024.

[63] Monirul I Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure In-VM Monitoring Using Hardware Virtualization. In *Proceedings of ACM conference on Computer and communications security*, 2009.

[64] Monirul I Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-VM monitoring using hardware virtualization. In *Proceedings of ACM Conference on Computer and Communications Security*, 2009.

[65] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. Formal verification of a multiprocessor hypervisor on arm relaxed memory hardware. In *Proceedings of ACM Symposium on Operating Systems Principles*, 2021.

[66] Donghai Tian, Qiang Zeng, Dinghao Wu, Peng Liu, and Changzhen Hu. Kruiser: Semi-synchronized non-blocking concurrent kernel heap buffer overflow monitoring. In *Proceedings of Network and Distributed System Security Symposium*, 2012.

[67] Chenxu Wang, Fengwei Zhang, Yunjie Deng, Kevin Leach, Jiannong Cao, Zhenyu Ning, Shoumeng Yan, and Zhengyu He. CAGE: Complementing Arm CCA with GPU Extensions. In *Proceedings of Network and Distributed System Security Symposium*, 2024.

[68] Wenhao Wang, Linke Song, Benshan Mei, Shuang Liu, Shijun Zhao, Shoumeng Yan, XiaoFeng Wang, Dan Meng, and Rui Hou. The Road to Trust: Building Enclaves within Confidential VMs. In *Proceedings of Network and Distributed System Security Symposium*, 2025.

[69] Fangzhou Yao, Read Sprabery, and Roy H Campbell. CryptVMI: A flexible and encrypted virtual machine introspection system in the cloud. In *Proceedings of International Workshop on Security in Cloud Computing*, 2014.

[70] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of USENIX security symposium*, 2014.

[71] Yiming Zhang, Yuxin Hu, Zhenyu Ning, Fengwei Zhang, Xiapu Luo, Haoyang Huang, Shoumeng Yan, and Zhengyu He. SHELTER: Extending Arm CCA with Isolation in User Space. In *Proceedings of USENIX Security Symposium*, 2023.

[72] Mark Zhao and G Edward Suh. FPGA-Based Remote Power Side-Channel Attacks. In *Proceedings of IEEE Symposium on Security and Privacy*, 2018.

[73] Siqi Zhao, Xuhua Ding, Wen Xu, and Dawu Gu. Seeing Through The Same Lens: Introspecting Guest Address Space At Native Speed. In *Proceedings of USENIX Security Symposium*, 2017.

[74] Lei Zhou, Xuhua Ding, and Fengwei Zhang. Smile: Secure Memory Introspection for Live Enclave. In *Proceedings of IEEE Symposium on Security and Privacy*, 2022.

A Interface Function Definition for TETD

The interface functions provided in TETD implementation is shown in Table 7.

Table 7: TETD Interface Function Definition

Category	API Name	Arguments	Return Value
TETD-G and H Communication	tetdg_exclu_init	agent_id, agent_type, homebase	agent_call_id, agent_call_args
	tetdg_collab_init	agent_id, agent_type, homebase	secretzone
	tetdg_collab_relo_done	-	agent_call_id, agent_call_args
	tetdg_exclu_done	agent_id, results	agent_call_id, agent_call_args
	tetdg_collab_done	agent_id, results	-
Agent Support	tetdg_agent_init	*agent_init_func()	-
	tetdg_agent_call_register	agent_call_id, *agent_call_func()	-
	tetdg_agent_ret	results	-
	tetdg_admin_access	agent_id	homebase
	tetdg_admin_toggle_disable	agent_id	-
VMM-side Request	tetdh_vmm_agent_call	agent_id, agent_call_id, agent_call_args	agent_call_results
TD-side Request	tetdg_td_agent_call	agent_id, agent_call_id, agent_call_args	agent_call_results

B TETD Application IV: TDLogger

Our fourth application, TDLogger, is a kernel-space agent in the collaborative mode. It hardens the kernel's log entries for critical system events by safeguarding them from modification, i.e., to attain forward security of committed entries. Note that TDLogger guarantees neither the genuineness of all entries (i.e., whether they are not factual or fabricated) nor their analysis effectiveness. However, it offers the opportunity to capture traces of kernel compromise with forward security, depending on whether a log action is triggered. Several schemes have been proposed in conventional system [1, 19, 62] to enhance kernel log security. None of them is applicable to the TD.

TDLogger works with a TETD-enlightened TD kernel which is instrumented to invoke the former to save critical event data via VMCALL. TDLogger also periodically uploads encrypted log entries to the owner via a network proxy VM, in the same fashion as TDReader. As a proof of concept, we handpick two security critical events deserving an immutable log. One is the dynamic kernel module loading and the other is the root login, which are two popular actions taken by malware to escalate its privilege and to gain the ability to conceal its existence. For kernel module loading, the instrumented code records the module name, `uid`, the parent process name and ID, and the timestamp. For the root login, it records the `pid`, `uid`, the TTY number, the session ID and the timestamp.

Implementation and Experiments. TDLogger is implemented with 60 lines of assembly code and 106 lines of C code, occupying 34 memory pages. The instrumented kernel places a log messages (up to 128-bytes) in a buffer page and passes its GVA to TDLogger. TDLogger stores the message in its own memory and periodically uploads them (with encryption) to the owner. Generating and secretly storing a 128 bytes message with TDLogger takes 190.3 microseconds. We experiment TDLogger with a remote attacker played by ourselves. The attacker remotely logs in to the TD's root account with stolen passwords, and loads a malicious kernel module using the `insmod` command. Before logging out, the attacker erases the traces in the system logs, including user login and kernel module load logs, to conceal its activity. We report the tampered log in Figure 12. It illustrates that no traces of the attacker are found in the recent legacy user login logs or kernel module load logs. Figure 11 shows TDLogger securely stores those entries, whose timestamps do not appear in the system logs.

```
PID: 1309, UID: 0, TTY Number: 0, Session ID: 1, Current time:
 2024-11-12T10:21:51.317135+00:00
```

(a) Login message.

```
NAME: Malware, UID: 0, PPID: 1357, Parent Process Name: bash,
 Current time: 2024-11-12T10:23:59.433219+00:00
```

(b) Module loading message.

Figure 11: TDLogger reveals evidence of the attacker's malicious activities.

```
2024-11-12T10:21:23.672679+00:00 tdx-guest login[1227]:
 pam_unix(login:session): session opened for user root(
uid=0) by root(uid=0)
2024-11-12T10:24:32.927227+00:00 tdx-guest login[1378]:
 pam_unix(login:session): session opened for user root(
uid=0) by root(uid=0)
```

(a) Legacy login messages.

```
2024-11-12T10:21:40.873842+00:00 tdx-guest kernel:
 Normal module have been loaded.
2024-11-12T10:25:10.184851+00:00 tdx-guest kernel:
 Normal module have been loaded.
```

(b) Legacy module loading messages.

Figure 12: Legacy logs show that the attacker's malicious activities have been removed.