A System Framework to Symbolically Explore Intel TDX Module Execution

Pansilu Pitigalaarachchi
pansilu.2020@phdcs.smu.edu.sg
Singapore Management University
Singapore

Xuhua Ding
xhding@smu.edu.sg
Singapore Management University
Singapore

Abstract

We present TDXplorer, the first dynamic symbolic analysis system for Intel's TDX Module, the software trusted computing base of TDX. Without using TDX hardware, an analyzer function on top of TDXplorer can not only apply dynamic analysis to control and instrument the TDX Module's execution, but also carry out symbolic execution for path exploration as well as security and functionality reasoning. The two types of analysis are seamlessly integrated in a way that symbolic execution is conducted directly upon the TDX Module's binary code and runtime states, which are shaped by using dynamic analysis techniques. We implement TDXplorer on Linux and measure its performance and correctness against executions on a TDX platform. Our case studies on symbolic modeling of secure EPT creation and KeyHole region management demonstrate that TDXplorer is a versatile and capable tool supporting various analysis tasks.

CCS Concepts

• **Security and privacy → Software and application security**; **Trusted computing**; • **Theory of computation → Program analysis**.

Keywords

Intel TDX Emulation; Intel TDX Module; Dynamic Program Analysis; Symbolic Execution

ACM Reference Format:
Pansilu Pitigalaarachchi and Xuhua Ding. 2025. A System Framework to Symbolically Explore Intel TDX Module Execution. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3719027.3765212

1 Introduction

The trusted computing base of Intel Trust Domain Extensions (TDX) consists of not only hardware components such as SGX and Multi-Key Total Memory Encryption (MK-TME) [43], but also a software component called the *TDX Module* (or "the Module" for short). The Module is the anchor for Trust Domain (TD) security and functionality, playing a similar role to a VMM managing regular virtual machines. It is in charge of critical tasks, such as setting up the mappings for a Trust Domain to physical page frames and managing the memory encryption keys. More notably, it is capable of reading and writing a TD's memory data in plaintext.

While the Module is shielded by a special CPU mode called the *SEAM VMX-root mode* against direct accesses from the VMM and TDs, it provides a set of APIs for them to invoke services. These interfaces constitute the attack interface to the Module. A vulnerability in its implementation or logical design could lead to a security catastrophe because of its pivotal role. Driven by the concerns of the Module's security, some cloud service providers such as Google and Microsoft have started to analyze and verify it [21, 33], based on the source code released by Intel. To our knowledge, all existing efforts thus far are either static analysis at the source code or fuzzing-aided dynamic analysis, hence are short of the rigor and the reasoning power offered by dynamic symbolic execution.

To symbolically run the Module is nontrivial. Existing popular dynamic symbolic execution engines such as KLEE [5], angr [42] and S2E [9] are not amiable to the Module's special system behavior and demands. Similar to a VMM, the Module runs in Ring 0. As a privileged software, the Module runs in the SEAM VMX root mode, providing services to TDs such as setting up their mappings to the physical page frames. By and large, the challenges are to provide all desired hardware and software resources for the Module to run properly, and to set up a foothold for another software to introspect its runtime and control its executions according to the analysis needs.

In this paper, we present TDXplorer, the first system framework for dynamic symbolic analysis upon the Module. Instead of resorting to a full-fledged hardware emulation, we retrofit a regular virtual machine into an emulation environment for Intel's TDX system software, including the Module and its loader. By supplying the desired system resources available in the host and emulating those unavailable, the environment allows the TDX software to properly execute on the native hardware (i.e., without undergoing interpretation), achieving the same functionality as in the TDX platform. Running beside the environment is TDXplorer's analysis component, which conducts both conventional dynamic analysis, e.g., single-stepping and dynamic instrumentation, and symbolic execution. The two types of analysis can be harmoniously interleaved. Like dynamic analysis, the symbolic execution in TDXplorer is also upon the Module's binary and its runtime CPU and memory states.

We have implemented a prototype of TDXplorer on a desktop PC running Linux. We rigorously measure its performance with experiments and validate the correctness of symbolic execution by running the generated test cases on a TDX server. The average per-instruction costs in our experiments range between 28 microseconds and 87 microseconds. Moreover, we present two case studies to demonstrate how TDXplorer can help users achieve various analysis goals. In the first case study, we use symbolic execution to model how the Module constructs the Secure Extended Page Table (SEPT) for a TD; in the second case, we systematically examine how the Module manages KeyHole pages, which store TD-relevant data and receive MK-TME protection. Both cases show that TDXplorer is a nimble and versatile tool to explore the TDX Module.

Organization. Section 2 explains the TDX background relevant to our design and analysis. Section 3 covers the related work. Section 4 sketches TDXplorer at the high level, with the details elaborated in Section 5 and 6. Section 7 reports our prototype implementation, its functional coverage, identified issues and inconsistencies, and evaluation results of its performance and correctness. We then present two case studies in Section 8. The last section concludes the paper.

2 Prerequisites

This section explains the TDX background techniques relevant to TDXplorer. More detailed descriptions can be found in a survey paper [8] and Intel's specification [25].

2.1 TDX Architecture

Figure 1 illustrates the TDX system architecture [23] consisting of the TDX system software (namely, the P-SEAM loader and the Module), the VMM, and two TDs. The TDX systems software and software in TDs run in a new CPU mode called the *Secure Arbitration Mode* or the *SEAM mode*. All other software, including the VMM, runs in the *non-SEAM* mode. While the Module supervises TDs, the VMM supplies computing resources needed by the Module and TDs, including the Logical Processors (LPs) and physical memory, by issuing `SEAMCALL` instructions that make a SEAM mode switch and invoke the corresponding handler of the Module. For instance, `TDH.MEM.SEPT.ADD` SEAM call invokes the Module to add a physical page to the secure EPT for the target TD. Similarly, a TD OS can also invoke the Module's service by issuing `TDCALL` instructions. Whenever the Module completes a SEAM/TD call, it passes the CPU to either the VMM or the TD kernel with the corresponding mode switch.

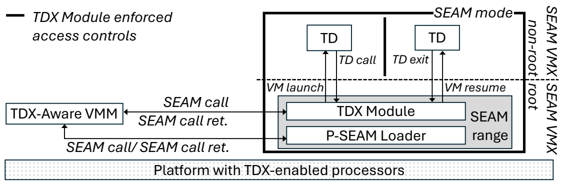

Figure 1: Illustration of TDX System Architecture.

2.2 Internals of The TDX Module

The Module's code, global/local data and stack reside on the physical pages in the *SEAM Range* which is only accessible when the CPU runs in the SEAM VMX-root mode. Hence, the VMM cannot access any page therein. The Module's VA regions are statically mapped to its SEAM range and are not changed at runtime.

The Module also uses physical pages outside of the SEAM range to store TD metadata, secure EPT, etc. Although these physical pages are provisioned by the VMM at runtime, they are accessed by the Module using predefined per-LP VA regions called the *KeyHole regions* by Intel. Since their physical addresses are decided only at runtime, the Module can update the page table pages that map the KeyHole regions from the predefined VA range called the *KeyHole edit region*. We highlight that the Module *cannot* update other parts of its paging hierarchy as the mappings are not given.

As the KeyHole region's physical pages do not receive the SEAM mode-based protection, they are protected by using Intel's Multi-Key Total Memory Encryption (MK-TME) [43]. When MK-TME is enabled, the MMU treats several leading bits of a physical address specified in the current page table entry as the so-called *KeyID* and uses the locally stored symmetric key corresponding to the KeyID to encrypt and authenticate data stored to the page at this physical address (or to decrypt and verify data loaded from it). Hence, to protect its KeyHole region, the Module configures the PTE in the KeyHole edit region with a proper *KeyID* during the mapping creation. When there are multiple TDs, the Module assigns different KeyIDs to them.

3 Related Work

Our work is akin to various efforts aimed at providing dynamic symbolic execution tools for analyzing applications [5, 6, 36, 38, 42, 49] and operating system kernels [9, 35]. These tools have demonstrated effectiveness in vulnerability discovery and exploit generation [3, 7, 26, 27, 45]. TDXplorer also applies the *re-hosting* approach which was introduced by Fasano et al. [18] and used in several firmware analysis schemes [17, 31, 50]. In general, re-hosting means that the target software is decoupled from its intended execution environment with special hardware and exported into another environment with commodity hardware to facilitate dynamic analysis.

The re-hosting approach always entails emulation, as the new home of the software does not have the needed hardware features to execute it. QEMU [4] and PIN [30] are two well-known general-purpose emulation systems, using binary rewriting and just-in-time compilation to emulate the target at the instruction level. However, TDX emulation in TDXplorer, Intel's KVM-based system [48] proposed by Isaku Yamahata at the 2022 KVM Forum, and Microsoft's Cornelius [32] do not follow this generic approach. We briefly explain Intel and Microsoft's emulation below and defer the comparison with TDXplorer to Section 4.2.4 after presenting our approach.

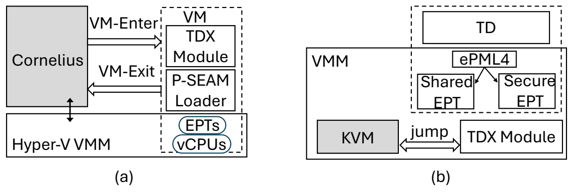

Figure 2: Illustration of TDX emulation architectures from (a) Microsoft Cornelius [32] and (b) Intel's proposal [48].

Existing TDX Emulation Projects. Figure 2 illustrates the architectures of Microsoft Cornelius [32] and Intel's TDX emulation [48]. Both aim to enable experimentation and testing of the Module without requiring actual TDX hardware. As a host application, Cornelius loads the P-SEAM loader into the VM. It then executes the P-SEAM loader in the VM to load the Module and runs the Module in VMX non-root mode, which is referred to as *VM-hosting* in this paper. In Intel's approach, dubbed in this paper as *VMM-hosting*, the Module executes in VMX root mode (i.e, inside the VMM), similar to its location in a TDX platform, except that there is no SEAM mode support. This setup allows the relocated Module to retain the capability of managing TDs, which are in the form of regular VMs. The KVM still plays the role of VMM to the Module, as in the TDX platform, and also acts as the intermediary between the TD and the Module. Specifically, the KVM invokes the Module's TDX API via a `JUMP` and the control is returned to KVM in the same manner upon completion. TD execution is supported by merging the KVM-managed shared EPT and the Module-managed secure EPT into a single EPT tree.

TDX Module and TEE Firmware Analysis. Recent work has examined the Module security. Wilke et. al. demonstrated a side channel attack targeting the Module itself [47], while another attack compromises TD confidentiality and integrity [40]. Google and Microsoft have independently assessed the TDX architecture [21, 33]; Google used static analysis tools [20, 46] and Microsoft combined manual source code review with dynamic analysis using Cornelius [32]. To the best of our knowledge, dynamic symbolic analysis has not yet been applied to the module. For AMD SEV-SNP, Paradžik et. al. [34] used Tamarin proof [41] to formally verify its software interface [1]. For Arm CCA [2], Fox et al verified the Realm Management Monitor (RMM) firmware [19] using interactive proofs and CBMC [28], a static symbolic execution tool.

4 Overview

In a nutshell, we use a two-pronged approach to enable dynamic symbolic analysis for the Module. Independent of the Cornelius project, we propose and design our own VM-hosting emulation. TDXplorer places the Module into a special bare-metal environment so that its binary runs on the CPU without undergoing interpretation and is subject to dynamic analysis such as tracing. On top of that, TDXplorer performs symbolic execution of the Module's binary using its runtime CPU and memory state in that environment, without involving any intermediary representation.

In the following, we sketch TDXplorer at the high level by explaining the primary design challenges, the system architecture, a comparison with related TDX emulation work, and a workflow example. To avoid verbosity, we use the "TDX software" to collectively refer to the Module and the P-SEAM loader.

4.1 Design Considerations

In addition to those common challenges in designing a dynamic symbolic execution engine, we face three issues stemming from the Module's unique attributes. First, the computing environment needs to be *execution-faithful* so that the Module's instructions run smoothly and function properly as if in a real TDX platform. The challenge has a twofold implication. (i) The environment must offer hardware features that the Module depends on. For instance, the Module supposes itself to be loaded in a continuous physical address range; when the Module updates its own paging hierarchy, the changes should take effect immediately through the MMU. (ii) The environment must provision both software and system states for the proper functioning of the Module. For instance, when serving certain SEAM calls, it may need a TD as the service target; it may need additional physical pages to create the secure EPT.

The second issue is that the environment is expected to be *event-faithful*. It should emit alerts whenever the Module's execution would trigger an exception on a real TDX platform. For instance, if the Module on a TDX server writes to a page using an unconfigured MK-TME Key-ID, the hardware will throw out an exception. Therefore, the environment needs to automatically catch those software errors that manifest as hardware exceptions on a real machine.

Thirdly, because of the Module's Ring 0 privilege and SEAM VMX root mode execution, it is challenging to make command-and-control over its execution to meet the runtime demand from the symbolic analysis. The requirement is not only about accessing the Module's CPU context and runtime memory, but also controlling its execution, e.g., to take a True branch as well as to save and restore its state during path exploration.

4.2 System Architecture

We tackle the challenges with a coalescence of system and software designs. TDXplorer runs on a host OS supporting CPU and MMU virtualization. From the system perspective, it consists of two components: the *SEAM emulation environment* where the TDX system software runs on the hardware; and the *Monitor*, which governs and analyzes the Module's execution dynamically and symbolically. The two components with their respective software composition are depicted in Figure 3, where the analyzer function is the user's code that invokes and controls TDXplorer for the user's task.

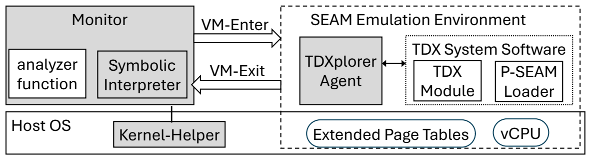

Figure 3: TDXplorer System Architecture. Shadowed boxes represent its software components.

4.2.1 *SEAM Emulation Environment.*

The environment is, in essence, a kernel-less virtual machine comprising one CPU core in the VMX non-root mode and a pool of physical pages. With CPU and MMU virtualization, it emulates those indispensable hardware properties of the genuine SEAM environment for the Module execution, such as continuous physical memory region, secure memory with MK-TME, execution on multiple logical processors and context switches between the Module and VMM/TDs. The software running inside it includes the TDX software and the TDXplorer *Agent*. We need to support the P-SEAM loader execution because it loads the Module and prepares the latter's execution. The Agent, as one of TDXplorer's software components, emulates some of the SEAM hardware features that cannot be achieved using virtualization, including special instructions in TDX, VT-x hardware virtualization and access to the TDX-relevant model-specific registers. It also serves as the Monitor's proxy to control the environment.

The vCPU core is scheduled by the Agent, which can pass control to the Module or the P-SEAM Loader and reclaim it using hardware or software breakpoints. All three run in Ring 0 and hence have the privilege to configure their respective Guest Page Tables (GPTs). Note that owing to Intel's virtualization techniques, the page tables used by the TDX software in the real SEAM environment are the same as the GPTs in our emulation environment. The Agent makes address space switches according to vCPU scheduling. The details of the emulation environment are elaborated in Section 5,

4.2.2 TDXplorer Monitor and Kernel-Helper. Running on top of the host OS, the Monitor is the host application of the emulation environment. Namely, it is the one that launches the environment as a virtual machine. TDXplorer maps the entire physical memory of the emulation environment to the Monitor for its runtime accesses. Since the Monitor only has the userspace privilege, we design the *Kernel-Helper* as a host kernel module to provide the host-level services needed by TDXplorer, e.g., to set up the desired address mappings for the Monitor to access the Module's virtual memory. The Monitor initially sets up the emulation environment with the Kernel-Helper's assistance, and starts its execution from the P-SEAM loader. Based on the runtime analysis needs, it manages the Module's execution via the Agent. It also includes the *symbolic interpreter* component which maintains runtime symbolic states, including path constraints and updates them when interpreting symbolic instructions.

4.2.3 Monitor-Environment Transition. Since the Monitor is the parent application of the environment, they share the same physical CPU core and occupy it alternately. The transitions between them are essentially VM-Enter and VM-Exit. The former starts or resumes the execution of software in the environment and the latter returns control back to the Monitor. These transitions are used for two purposes: to facilitate the Monitor's control over the environment through the Agent; and to emulate SEAM/TD calls where the Monitor plays the roles of the VMM and the TD kernel, respectively.

To transition to the environment, the Monitor configures the vCPU of the emulation environment via the Kernel-Helper to establish the initial CPU state, specifying the Agent as the entry. The Kernel-Helper then schedules the environment's vCPU for VM-Enter with `VMENTER` or `VMRESUME` instructions. As a result, the Agent is dispatched on the vCPU. Similarly, the Agent is also the exit point of the environment. It captures the ending of the Module's execution, prepares all the data, including return values, and triggers a VM-Exit, which is forwarded to the Monitor by the host kernel.

4.2.4 Comparison with related TDX emulation work. In the following, we compare TDXplorer, Intel's emulation [48] and Cornelius [32] from several angles.

Emulation Design and Goals. Intel's VMM-hosting approach eliminates the need for VMM and TD emulation by supplying live states of the KVM and TDs. Compared with TDXplorer and Cornelius's VM-hosting approach, it facilitates relatively free-formed TDX analysis instead of being prescribed by TDX APIs. Nonetheless, VMM-hosting results in less flexibility for a user-space application to mould the desired VMM/TD states. Between Cornelius and TDXplorer, the former focuses on testing SEAM and TD calls from the outside, while the latter aims for rich mid-call dynamic analysis with fine-grained control and visibility into the Module's runtime states tailored for security-centric analysis. The architectural difference also implies different ways to emulate SEAM/TD calls. With VMM-hosting, SEAM calls are emulated as jumps between the KVM and the Module, and TD calls are emulated as hypercalls. With VM-hosting, both SEAM calls and TD calls are emulated as VM entering facilitated by the KVM. Cornelius gains control at the end of SEAM/TD calls by trapping the Module's `SEAMRET`, `VMLAUNCH` and `VMRESUME` instructions.

Special Instruction Emulation. Contrary to TDXplorer's special instruction emulation by TDXplorer Agent, Cornelius handles their emulation in the host-side application, relying on hardware-triggered VM-exits due to such instructions. Root-mode instructions like `PCONFIG` naturally trap and Cornelius also forces traps on `RDMSR` and `WRMSR` by patching the Module's source code. This design, however, leads to increased VM-exit overhead in Cornelius. Intel's approach does not explicitly describe the handling of such instructions. However, the Module's root-mode execution may allow native execution of some privileged instructions, reducing the emulation and transition overheads.

Faithfulness to security critical TDX functionality. Similar to Intel's approach, Cornelius executes the Module with no real MKTME, and the secure pages are not encrypted. It simulates MKTME to pass Module checks but doesn't enforce its semantics, making it unable to detect MKTME-related violations. TDXplorer, though it also runs the module with no actual encryption of the secure pages, provides faithful MKTME emulation, enabling such detection. In Intel's approach, the merged EPTs are a divergence from the TDX's secure EPT model, limiting their value for security-critical analysis.

Access to Runtime Module states. TDXplorer's in VM agent, sharing the Module's address space, accesses both statically and dynamically mapped Module memory. Furthermore, the Module's statically mapped memory is also mapped into TDXplorer's host process, allowing access via the Module's own virtual addresses with no additional translation. Cornelius accesses Module memory only after the host regains control, either upon a SEAM/TD call completion or during an uncontrolled VM-exit, and relies on a library API that uses hypervisor services to resolve guest virtual addresses on demand. While Intel's approach provides no explicit details, the shared address space likely enables the VMM to access Module memory directly. Without requiring any additional real-time memory synchronization between TDXplorer's SEAM environment and its host-side counterpart, the setup in TDXplorer enables seamless integration with a host-side interpreter.

Support for Dynamic Analysis. Traditional dynamic analysis has not been a focus of Cornelius, nor is such support discussed in Intel's approach. TDXplorer on the other hand supports runtime introspection during SEAM/TD calls, enabling the use of standard dynamic analysis primitives such as debug breakpoints, `INT3` breakpoints and single-stepping.

4.3 An Example of Analysis Workflow

The user's analyzer function steers TDXplorer to carry out analysis steps toward his/her goal. A symbolic analysis can start anytime during the Module's native execution. The example below illustrates the main steps in a workflow (Figure 4). Consider the TD virtual processor state (TDVPS) update during TD initialization. The goal is to obtain the path constraint and derive the Module's validations applied to the input physical page address when adding a new physical page to the TDVPS.

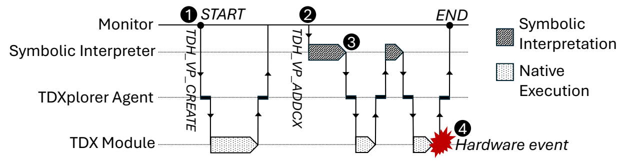

Figure 4: An exemplary analysis workflow.

1. The Monitor issues the first SEAM call, `TDH.VP.CREATE`, with all concrete arguments. The Agent receives the SEAM call and dispatches the Module to execute the corresponding handler. When the Module returns, the Agent regains control and forwards the returned value and the Module's CPU state to the Monitor. (*native execution for state initialization*)
2. The Monitor sets up the necessary contexts for the second SEAM call, `TDH.VP.ADDCX`, and invokes the interpreter to symbolize the SEAM call argument for the new TDVPS physical page address. (*setup symbolic execution*)
3. The interpreter begins single-stepping the Module. For instructions involving symbols, it interprets them and updates the Module's CPU state and memory accordingly; for those without symbols, it dispatches them to the Agent so that they are executed within the emulation environment by the hardware. (*symbolic execution*)
4. The execution stops if the native execution triggers a hardware event indicating a security policy violation or the interpreter detects an error. The corresponding path constraint and symbolic states are reported. If no hardware event occurs, the Monitor regains control once the SEAM call returns. (*termination and results output*)

The workflow shows TDXplorer empowers the user's analyzer to weave symbolic execution with dynamic binary analysis. The latter is applied to attain the desired state, whereby the former carries out symbolic exploration.

5 SEAM Emulation Environment

In this section, we elaborate on the design of the SEAM emulation environment, which provides the Module with the expected view of its physical memory, the virtual address space(s), the logical processors (LPs) and the platform configurations.

5.1 Physical Memory

As the CPU core of the emulation environment operates in the VMX non-root mode, the Module and the P-SEAM loader run on the guest physical memory whose addresses are mapped to the host physical addresses by the EPTs managed by the host kernel.
SEAM-Memory. To satisfy TDX's SEAM-memory requirement, the Kernel-Helper picks a 32-MB aligned guest physical address (GPA) region sized 2^m bytes where $m < 30$. It then sets up the EPTs for the emulation environment so that the chosen GPAs are mapped to physical page frames in the host. Furthermore, the Monitor splits the GPA region into two halves following the TDX specification, for the P-SEAM loader and the Module, respectively. The GPA range is stored as part of the emulated TDX platform state and, when needed, is returned to the P-SEAM loader or the Module through emulation as the contents in the MSRs, which store the platform's SEAM-memory base and size. As a result, the SEAM-memory emulation transparently meets the Module's and the P-SEAM loader's needs.
Non-SEAM Memory. The Kernel-Helper maps a pool of physical pages to the emulation environment as the Non-SEAM memory for the Module and the VMM to share, e.g., to exchange data in SEAM calls. It also creates the EPT mappings for a 1-GB aligned GPA region used as the secure memory for TDs and the Module's TD-related data objects. Additional EPT mappings allocate a GPA region for the Agent's memory, including the pages required for the GDT and IDT of the emulation environment. Figure 5 illustrates an example memory layout.

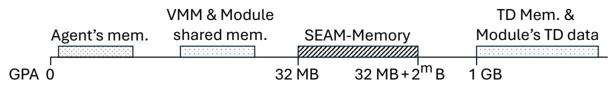

Figure 5: Memory layout in the emulation environment conforming to TDX's memory requirements (with $m < 30$).

5.2 Address Spaces

Playing the role of Intel's NP-SEAM loader, the Monitor creates the page tables for the P-SEAM loader and invokes its execution. The P-SEAM loader then loads the Module into the designated GPA region in the emulation environment and sets up the Module's page tables. The Monitor also creates the page tables for the Agent to satisfy that 1) the Agent's code and data are separated from the TDX software to avoid undesired runtime interference; and 2) the Agent can directly access the TDX software's virtual address spaces without software-based page table walking. Specifically, the Monitor initiates the Agent's PML4 page in two steps. It first clones all PML4 entries of the TDX software's PML4 pages. The cloning essentially fuses the TDX software's address space into the Agent's so that any mapping updates made by the TDX software at the lower-level page table entries[1] immediately take the same effect on the Agent. Next, it chooses an unused PML4 entry to construct the remaining hierarchy, which essentially assigns the Agent a 512 GB address space not occupied by the TDX software. The three paging hierarchies are shown in Figure 6.

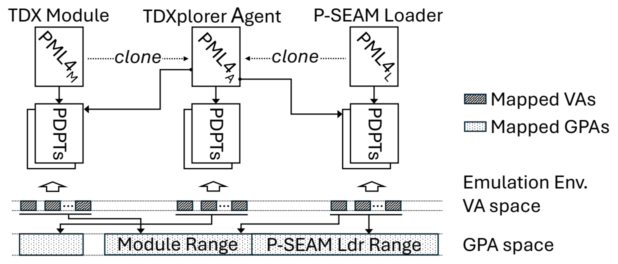

Figure 6: Three separated paging hierarchies in the emulation environment.

[1]Note that the TDX software does not modify mappings at the 512-GB level.

5.3 Special Instruction Emulation

As the hardware of the emulation environment does not support TDX, certain types of special instructions cannot be executed therein. To make the emulation transparent to the TDX software execution, TDXplorer emulates those instructions that are incompatible with the environment.

The binaries of the P-SEAM loader and the Module are scanned offline to extract special instructions. Those include new TDX instructions, MSR access instructions, `CPUID`, `PCONFIG`, instructions for VMX and MK-TME. Then, the two binaries are instrumented to install a software probe, i.e., `INT3` instruction, at the first byte of each of these instructions. To aid instruction emulations, the instrumented instructions are backed up in the data region shared between the Agent and the Monitor with their operands (if any) extracted.

When the INT3 exception is triggered during the TDX software execution, the Agent gains CPU control. If it is due to special instruction emulation, it emulates the original instruction before passing the control to the next instruction. The instruction emulation uses the information shared by the Monitor, such as the saved instruction operands, software-maintained TDX-compatible model-specific register states and CPU features as needed to do the emulation. As the majority of the special instructions are used to check or load the hardware status and configuration, their executions are not very closely related to the analysis goals.

5.4 MK-TME Emulation

As MK-TME-based memory encryption is at the core of TDX, TDXplorer is geared to emulate its impacts on the Module execution. The Module engages MK-TME in three ways: (1) It commands MK-TME to set up encryption keys for TDs and itself. (2) When it creates its mappings for non-SEAM pages, it sets up to 16 of the most significant bits in the physical address field of page table entries to specify the KeyIDs used for memory encryption. (3) Its accesses to the aforementioned pages undergo MK-TME's crypto operations. We emulate the behaviors accordingly.

Key Initialization. The Agent emulates relevant `RDMSR` instructions that check whether the underlying hardware supports MK-TME, and the `PCONFIG` instruction, which creates an MK-TME key for a given KeyID on the platform. The emulation feeds the Module with the CPU states, indicating successful instruction execution so that its execution does not abort.

KeyID Configuration. The Agent intercepts and mediates the Module's accesses to its page table pages available for dynamic mappings. This special set of pages is mapped to the Module's *KeyHole edit region* in TDX's terminology, which is a 512-GB aligned VA region. The Monitor sets the Module's PML4 entry for the KeyHole edit region as *non-present* so that the Module's access to its paging structure is then trapped to the Agent. When the Module intends to map a physical page p to a virtual page v using KeyID k, the Agent saves $\rho_v = (v, p, k)$ in its VA-Key table, and updates the PTE to create the desired mapping without any KeyID in order to be compatible with the host platform's MMU. It sets page v as *non-present* to trap any access to the newly mapped page. Note that monitoring of the KeyHole edit region updates also allows for analysis over the usage of dynamic VAs. Figure 7 depicts the mapping setup.

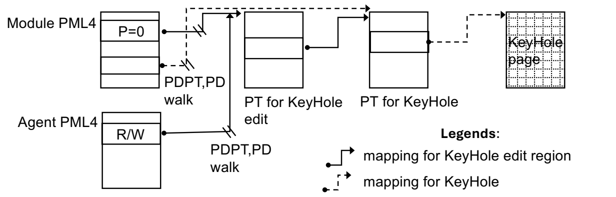

Figure 7: Mappings to KeyHole and KeyHole edit region. $P = 0$ **means page *non-present*.**

Secure Memory Access. When access to a non-present virtual page v is trapped to the Agent, it first locates the corresponding physical page number p, and operates according to the access type.

- For a write access, the Agent finds $\rho = (v, p, k)$ from its VA-Key table and inserts (or updates) $\tau = (p, k)$ in its PA-Key table which indicates that MK-TME protects p with KeyID k. It also checks whether p is mapped to other VAs. If so, it updates the Module's paging structure to mark all those VAs as *non-present* to trap accesses to them. Note that this step is necessary because MK-TME does not detect or prevent dual mappings.
- For a read access, the Agent retrieves $\rho = (v, p, k)$ from its VA-Key table and $\tau = (p, k')$ from its PA-Key table. If $k = k'$, it restores the *present* bit in the PTE for v and re-runs the instruction. Otherwise, the Agent halts the Module as the KeyID used in read does not match the KeyID used in its prior write, and reports the event to the Monitor.

Essentially, TDXplorer ensures that a VA can be used to read a page protected by MK-TME only when the KeyID in the VA's mapping is used in the immediately preceding write to the page.

5.5 Logical Processor Emulation

One of the Module's main tasks is to manage CPU resources, including logical processors (LPs) and TD vCPUs. Since the Module treats vCPUs as VMCS objects without actually running on top of them, there is no need for TDXplorer to emulate vCPUs. However, an LP is a physical resource that is not available in the emulation environment. More importantly, the Module retrieves TD-specific data from the per-LP state. It is thus necessary to emulate LPs for the Module to manage and "run" on them.

Initialization. Initially, the emulation environment makes the Module perceive that it is given n different LPs, where n is a TDXplorer configuration parameter and set to 4 by default. The Monitor prepares n CPU contexts and state objects for LP emulation and invokes the P-SEAM loader to initiate them one by one. When the P-SEAM loader queries a given LP, the Agent intercepts the `CPUID` instruction in use and emulates its return with the corresponding context and state. The execution of the P-SEAM loader eventually produces n per-LP states for the Module. A per-LP state includes a VMCS object, page table mappings (for stack and data regions), and mappings related to the dedicated KeyHole region.

Per-LP Runtime Update. To ensure the emulated per-LP state is always consistent with the Module execution, the Agent intercepts the Module's LP state modifying instructions (e.g., `VMWRITE`) and emulates them by updating the corresponding objects accordingly. The Monitor also initiates and maintains the vCPU-LP binding to remain consistent with the Module's handling of TD vCPUs. To initiate a binding, it makes a SEAM call destined to a chosen LP so that the Module creates a TD vCPU and binds it with the given LP. When making a TD call, it also ensures that it picks the LP hosting the corresponding TD vCPU as the target LP for the Module to occupy.

5.6 SEAM Call and TD Call

Supporting the issuance and handling of SEAM/TD calls is central to TDXplorer. From the functionality perspective, the Module follows the SEAM calls to manage TDs, including their creation; from the security perspective, these TDX API calls constitute the primary attack vectors the Module exposes. As described previously, the Monitor emulates the call invocation and return using the monitor-environment transition and the Agent serves as the entry and exit gates of the environment. We elaborate below on the details about the needed system resources provisioned to the Module's handlers.

LP Selection. When the Monitor emulates a SEAM/TD call invocation, it informs the Agent about the target LP for the Module's upcoming execution to occupy. Hence, the Monitor and the Module always have the same view of busy and idle LPs.

State Preparation for SEAM Call. The Monitor prepares the *SEAM transfer VMCS* object as specified by the TDX specification. This VMCS is passed to the Agent as the initial CPU context for the Module's SEAM call handling. It is *not* the VMCS for monitor-environment transition. The former is never applied to any CPU core, while the latter is applied to the emulation environment's core. The Agent also switches `CR3` to the Module's address space before performing a far-jump to the desired Module entry point.

State Preparation for TD Call. To make a TD call meaningful, the Monitor has to create and launch TDs using a sequence of SEAM calls according to the TDX specification. Note that it is unnecessary for TDXplorer to run any TD with a kernel or application, because a TD appears to the Module as a collection of data, including its CPU state and memory. It is thus sufficient for analysis if the Module's view towards a TD is genuine. For this purpose, the Agent intercepts the TD-Enter event by emulating the `VMLAUNCH` or `VMRESUME` instructions. Specifically, it emulates the CPU context switch from the Module to the TD and then exits from the emulation environment so that the Monitor gets control. When issuing a TD call, the Monitor prepares the VMCS representing the TD vCPU. Similar to the SEAM call emulation workflow, the Agent sets up the scene for the Module's TD call handler to run.

Call Return. The Agent intercepts special instructions denoting the end of an API call. For `VMLAUNCH` and `VMRESUME`, it updates the TD vCPU, emulates the CPU context switch and passes control to the Monitor as described earlier. For `SEAMRET`, the Agent updates the SEAM transfer VMCS object and passes control to the Monitor.

6 The TDXplorer Monitor

This section explains how the Monitor manages the emulation environment and how symbolic execution is carried out.

6.1 Accessing Emulation Environment

The Monitor accesses the environment's memory for two purposes. One is for the environment setup and management. Since this task requires access to all pages in the environment, TDXplorer creates for the Monitor a special VA region which has the identical layout and size of the environment's GPA space. With the assistance of the Kernel-Helper, a page in the VA region and the corresponding GPA of the environment are mapped to the same physical pages.

The other purpose is to read and write the Module's virtual memory to meet analysis needs. Leveraging the fact that the Module uses static VA-to-PA mappings for its SEAM memory, TDXplorer clones all the static mappings into the Monitor's space. As a result, the Monitor can directly reference the Module's VAs under its static mappings. To access the Module's dynamically mapped memory (i.e., the KeyHole regions), the Monitor interacts with the Agent at runtime through a shared memory buffer.

6.2 Symbolic Interpreter

As part of the Monitor, the symbolic interpreter interprets the Module's instructions involving symbolic operands. The interpretation may lead to updates in the symbolic states maintained by the Monitor and also in the Module's native runtime memory and CPU context.

6.2.1 Memory and CPU Register Model. The interpreter operates on the Module's virtual memory and CPU registers. In its local storage, it maintains a set of data structures representing the Module's current symbolic state as shown in Figure 8. The state includes, if any, the symbolic data and its associated metadata, including sizes and locations, i.e., virtual addresses and/or registers. As symbolic data is never used in executions in the emulation environment, their VA locations appear like "holes" in the Module's runtime memory.

The interpreter, as part of the Monitor, accesses the Module's virtual memory using the native VAs. The Module's CPU state is reported by the Agent after trapping the Module's execution. When interpreting an instruction, the interpreter references the Module's virtual memory or CPU context for concrete operands and references the symbolic state for symbolic ones.

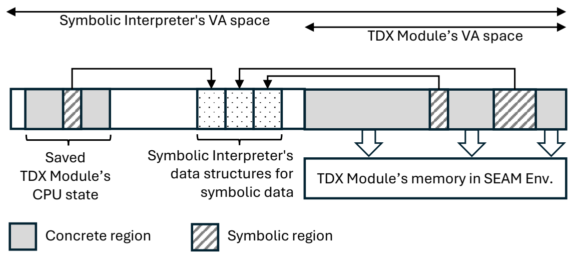
Symbolic Interpreter's VA space
TDX Module's VA space
Saved TDX Module's CPU state
Symbolic Interpreter's data structures for symbolic data
TDX Module's memory in SEAM Env.
Concrete region
Symbolic region

Figure 8: The memory and CPU model in TDXplorer

6.2.2 Single-Stepping. The interpreter starts to single-step the Module binary after being invoked by the analyzer function, which specifies the entry VA. In a nutshell, the interpreter fetches an instruction from the Module's runtime memory and analyzes all operands to determine if it is symbolic. Symbolic ones are interpreted and non-symbolic ones are dispatched to the emulation environment for the hardware to execute.

Instruction Parsing. Since the memory and CPU model used in symbolic execution is consistent with the Module's runtime, it is straightforward to determine if an instruction is symbolic. The interpreter extracts the instruction's register and memory operands to check against the symbolic states.

Instruction Interpretation. The interpreter updates the symbolic state and the Module's runtime according to the instruction's opcode and operands. One issue in instruction interpretation is to efficiently and effectively handle the flag register `RFLAGS`. We follow the approach used in angr [42] and KRover [35] to postpone emulating the flag-setting effect until a flag-dependent instruction such as `JZ` is fetched.

Single-Instruction Native Execution. To avoid the expensive costs of VM enter and exit during single-instruction native execution, TDXplorer runs the interpreter and the Agent on separate CPU cores during symbolic execution and provides shared memory for them to get synchronized. To dispatch a concrete instruction to the emulation environment, the interpreter exports to the shared memory the Module's CPU state as the latest outcome of interpretation. The Agent turns on the single-step mode in the environment's core and runs the instruction with the given CPU state. Upon the instruction completion, the Agent turns off single-step mode and returns the updated register state to the interpreter.

6.2.3 Path Selection and Exploration. When interpreting conditional branch instructions with symbolic flag bits, TDXplorer supports two ways of path selection. If the involved symbol is seeded, the interpreter evaluates the symbolic flag with the seed to determine the branching direction. Otherwise, it checks with the constraint solver about path constraint satisfiability. If both branches are satisfiable, it calls back the analyzer to decide a path with the default option of making a random decision. Exploration of a path terminates when the interpreter encounters `SEAMRET`, `VMLAUNCH` or `VMRESUME` instruction, which indicate a normal completion of a SEAM/TD call, or an error, such as `UD2` instruction.

Offline Path exploration. Similar to angr [42], QSYM [49], TDXplorer supports offline path exploration, i.e., exploring one path at a time. The exploration uses the depth-first search strategy to cover all valid paths. TDXplorer allows the user's analyzer function to determine the runtime moment to start path exploration. To prepare for path exploration, the interpreter backs up the symbolic state and turns the Module's virtual memory to read-only. Concrete writes to the Module's pages are made on a copy of the original one, following the same copy-on-write approach used in KLEE [5] and process forking. Upon completion of the path, those page copies are discarded and the same initial state is used for the next path.

6.3 Analyzer Function

Recall that the TDXplorer Monitor contains an analyzer function as shown in Figure 3. The function is programmed by the TDXplorer users for their analysis tasks. It is empowered by the architectural features of TDXplorer to combine traditional dynamic binary analysis techniques with symbolic analysis in order to make complex binary-level reasoning. Our case studies in Section 8 present two such examples.

TDX State Setup. The function can issue SEAM calls and/or TD calls to establish the desired TDX state before starting symbolic execution. During runtime, it can also set hardware breakpoints in the Module's native execution to detect fetching of a concerned instruction or access to an object. It is at the function's discretion to kickstart symbolic execution either in the midst of the Module's native execution by symbolizing runtime data or by issuing a SEAM call with symbolic arguments.

Symbol Declaration and Concretization. TDXplorer allows symbolization of the Module state at any point during the symbolic analysis. The interpreter provides APIs for defining register or memory symbols. The analyzer function can trace Module execution and dynamically define new symbols based on analysis demands. When necessary, it can query the constraint solver's APIs to evaluate a symbolic expression and concretize some symbolic data.

Accessing Module Runtime State. Running within the Monitor's address space, the analyzer function makes direct access to the Module's runtime state. It can reference the statically mapped Module memory via the Module's own VAs and uses the Module's struct definitions to access object members. Accessing dynamically mapped keyholes and the CPU state is through the interpreter's APIs. Additional APIs from the Monitor provide access to Module-specific states such as KeyHole state, LP emulation-related metadata, and the Module's page tables. The analyzer function can insert callbacks to the interpreter to monitor events such as the page faults triggered by the Module's access to the KeyHole edit region.

7 Implementation and Evaluations

We implement a prototype of TDXplorer on a Linux PC running kernel version 5.15.0. The machine is equipped with a 13th Gen Intel® Core™ i9-13900 CPU (800 MHz-4200 MHz), 64 GB RAM and 2.8 TB disk space. All TDXplorer experiments are conducted on this platform.

7.1 Prototype

The TDXplorer prototype comprises approximately 21.7K lines of C/C++ SLOC and 241 lines of inline assembly code. Component-wise, the Agent, kernel helper and the interpreter are implemented in 1.2K, 305, and 11.9K lines of code, respectively. We use Z3 [39] version 4.8.14 as our constraint solver and Dyninst [37] version 12.0.0 for binary disassembly and instruction parsing. We open-source TDXplorer and make it persistently and publicly available, along with build instructions and sample analyzer functions, at https://github.com/KRoverSystems/TDXplorer.

We build the TDX system software stack following Intel's guidelines [10, 16], compiling the P-SEAM loader (v1.5.00) and Module (v1.5.01) from source, yielding binaries of 90 MB and 397 MB, respectively. Our pre-processing identifies and instruments 534 and 37 special instructions in the Module and loader, respectively, using `INT3`. The Monitor incorporates the minimal functionality of the NP-SEAM loader (v1.5.00) to load the P-SEAM loader binary with appropriate page table mappings.

The emulation environment is created using KVM, with 64 MB of its guest physical memory reserved for the SEAM range (64-128 MB GPA) and 1 GB TD memory region (TDMR) allocated at (1-2 GB GPA range). TDXplorer emulates four LPs. The Module is installed via the `SEAMLDR.INSTALL` SEAM call, issued to the P-SEAM loader once per LP, completing in 3.69 ms. We follow Intel's instructions [10, 22] and Canonical TD suite [29] to initialize the platform (121.8 ms) and create two TDs, each taking 42.43 ms.

7.2 Functional Coverage in TDX Emulation

Currently, our TDX emulation supports a total of 54 SEAM/TD calls[2]: 38 out of 66 SEAM calls and 16 out of 21 TD calls. Figure 9 shows the supported SEAM/TD calls grouped according to Intel-defined TDX functionality categories.

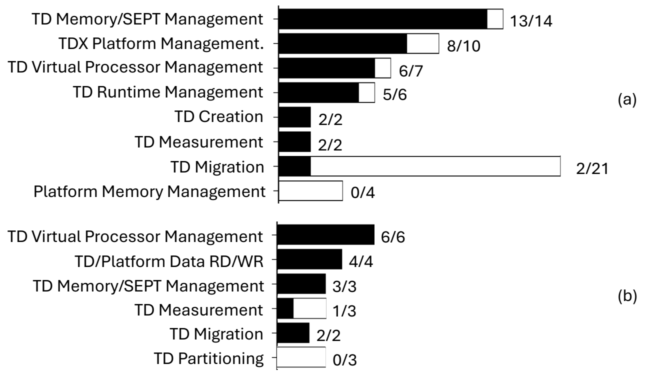

Figure 9: Number of (a) SEAM calls and (b) TD calls supported under each key TDX functionality in the current prototype.

Supported and Tested Functionality. We provide support for a wide array of SEAM and TD calls across key functionalities essential for the meaningful analysis of the Module, including, but not limited to, TDX platform management, TD creation, vCPU, memory, and SEPT management, as well as general runtime management of TDs. For instance, under TDX platform management, we support eight out of ten SEAM calls (See Figure 9 (a)) used by the VMM for the initial setup and configuration of the TDX Module, memory regions and logical processors. The two SEAM calls in this category, used for Module shutdown and update, are currently unsupported due to the high context preparation efforts they require. For a supported call, emulation support means that we can execute the SEAM/TD call and have tested its behavior under various success and failure conditions. Leveraging this support, we have tested and validated a wide range of functional use cases across the life cycle of the Module and TDs, following Intel's specifications [24, 25] and the Canonical TD suite [29]. Furthermore, TDXplorer remains compatible with future TDX Module versions, assuming the TDX architecture remains unchanged.

Emulation Support for SEAM/TD calls. Supporting a SEAM or TD call typically involves implementing the necessary emulation support, including handling special instructions, executing prerequisite calls to establish the correct Module and TD states, and preparing the appropriate input arguments and/or data structures for the target call. For example, we currently support the `TDG.SERVTD.WR` TD call, which allows a service TD to write to a metadata field of a target TD. To correctly issue this call, the analyzer function must sequentially execute several SEAM/TD calls: initialize the platform and the Module, create and build the service TD, create the target TD, bind the service TD to the target TD, launch the service TD, and finally emulate the specified TD call from the service TD with all necessary input arguments. In this example, by implementing the emulation support for all these preceding calls, we have achieved support for the `TDG.SERVTD.WR` TD call.

Unsupported Functionality. In TD migration, only the SEAM calls related to service TD binding (associating a service TD with a target TD) and the TD calls for accessing target TD metadata are supported. Our emulation software does not support SEAM calls involved in the other TD migration workflows due to their complexity. The four platform memory management calls are currently unsupported due to high data and context-preparation efforts, but can be supported in future work. TD partitioning is not supported, as our TDX emulation does not cater to the semantics of the nested virtualization of TDs. Some TD measurement-related calls are only partially supported because they involve the `SEAMOPS` instruction. Specifically, our emulation does not cover certain leaf functions of `SEAMOPS` due to the lack of documentation on their functionality and the inherent difficulty of emulation. Most other unsupported SEAM and TD calls require high context preparation efforts, but our emulation architecture is designed to support them in the future.

7.3 Identified Issues and Inconsistencies

Our dynamic testing in TDXplorer, followed by manual analysis, led to the accidental discovery of a stack canary bug in the Module [11]. It was uncovered during the TDX platform initialization sequence while testing `TDH.SYS.LP.INIT` on multiple logical processors. It causes all but one logical processor to incorrectly pick and use the same non-random value as their canary. Intel's TDX Module team confirmed the bug and acknowledged that it was already known internally.

Furthermore, our symbolic execution and subsequent analysis uncovered minor documentation issues [12–14], stemming from inconsistencies between the documentation and the Module's current behavior.

7.4 Performance Experiments

We conduct four sets of experiments to evaluate TDXplorer's performance. The first set uses seeded symbolic execution to measure the per-instruction execution overhead. Second, we assess the efficiency of path exploration during symbolic execution. Thirdly, we compare native execution performance under TDXplorer with that of a TDX-enabled server. Finally, we measure the memory footprint of TDXplorer.

7.4.1 Symbolic execution. We symbolically execute 20 SEAM/TD calls from start to return, using seeded mode with symbolic inputs that include SEAM/TD call arguments, Module's data, or both. The number of instructions executed per call ranges from 787 to 43,919,

[2]Furthermore, three of the four SEAM calls provided by the P-SEAM loader for the VMM to install the Module and query information are also supported.

resulting in symbolic execution times between 27 ms and 1239 ms per call. The average per-instruction execution cost varies between 0.028 ms and 0.087 ms. Detailed results are provided in Table 1. Each API call encounters 6-20 special instructions that are emulated by the Agent. In seeded mode, the interpreter invokes the constraint solver (Z3) to evaluate branch predicates based on the provided seed. Each solver invocation takes approximately 0.214 ms, with 0 to 41 such calls per SEAM/TD call. Additionally, there is a one-time solver initialization cost of 1.7 ms for each API call.

SEAM call/ TD call	Total ins.	IE tot.	Exec. time (ms)	
			Total	Per ins.
TDH.SYS.INIT	1314	353	45	0.034
TDH.SYS.LP.INIT	2302	649	97	0.042
TDH.SYS.CONFIG	3392	583	142	0.042
TDH.SYS.KEY.CONFIG	731	251	64	0.087
TDH.SYS.TDMR.INIT	33465	6839	944	0.028
TDH.MNG.CREATE	3518	906	137	0.039
TDH.SYS.INFO	2709	402	199	0.073
TDH.VP.INIT	4695	1043	169	0.036
TDH.MEM.SEPT.ADD	6314	1399	197	0.031
TDH.VP.ENTER	3357	770	111	0.033
TDH.SERVTD.BIND	7586	1572	270	0.036
TDH.SERVTD.PREBIND	2812	650	120	0.043
TDG.MEM.PG.ATTR.RD	2996	717	126	0.042
TDG.MEM.PG.ATTR.WR	3144	744	129	0.041
TDG.MEM.PG.ACCEPT	3767	861	119	0.032
TDG.MR.REPORT	5874	796	311	0.053
TDG.SYS.RD	2026	535	95	0.047
TDG.SYS.RD.ALL	43919	8125	1239	0.028
TDG.VM.READ	803	322	27	0.033
TDG.VP.INVEPT	787	311	42	0.053

Table 1: Results from performance evaluation of seeded execution. "IE total" refers to the number of interpreted instructions. The prefixes "TDH" and "TDG" stand for SEAM call names and TD call names, respectively.

7.4.2 Path exploration. We conduct four path exploration tests on four API calls, where input arguments or selected Module states are symbolized. Using a depth-first strategy, TDXplorer explores all feasible execution paths within each API call handler. Results are summarized in Table 2. In total, TDXplorer explores 233 unique paths across the four tests. The number of instructions per path varies widely from a few hundred to a few thousand. During each path exploration, the interpreter calls the Z3 constraint solver to determine branch satisfiability. Across all four tests, constraint solving dominates the runtime, accounting for 86.53%-91.81% of the total execution time per API call. Each path performs memory writes that are confined to 2-6 unique 4KB pages, which are backed up using TDXplorer's copy-on-write mechanism to preserve memory state across paths.

7.4.3 Native execution and special instruction emulation. We measure native execution times for 10 SEAM calls on both TDXplorer and a TDX-enabled server. On average, SEAM calls execute in 41.1 μs on the server (ranging from 21 μs to 167 μs), while the same calls in TDXplorer take 413 μs on average (ranging from 361 μs to 464 μs). While TDXplorer incurs roughly a 10x slowdown due to TDX emulation as compared to executions on a TDX server, it is acceptable from the analysis perspective because it is a tiny fraction of the total symbolic execution time per path (a few hundred microseconds vs 100+ milliseconds). We also evaluate the cost of special instruction emulation by measuring the round-trip time for handling INT3 trap; the Agent intercepts the trap, emulates the instruction and resumes Module execution via `IRET`. Each emulated special instruction costs 2.9 μs, including trap and resume.

SEAM call/TD call	# of Paths	Ins. per path	Time tot.(s)	% Z3 Cost
TDH.SYS.LP.INIT	11	359–2031	1.55	86.53%
TDH.SYS.CONFIG	25	349–3392	7.20	86.94%
TDH.MNG.CREATE	9	398–1034	1.14	88.15%
TDG.MEM.PG.AT.RD	188	850–2134	139.7	91.81%

Table 2: Path exploration statistics. The Z3 cost corresponds to the cost of constraint solving.

7.4.4 Memory consumption. We measure the memory utilization of TDXplorer during symbolic execution of `TDG.MEM.PAGE.ATTR.WR` TD call. TDXplorer explores 12 paths in 2.43 seconds, with the longest path consisting of 2281 instructions. The results are shown in Figure 10. The peak virtual memory usage is approximately 278 MB, which includes the memory allocated for the emulation environment (VM). The peak RAM footprint is around 77 MB and remains stable over time.

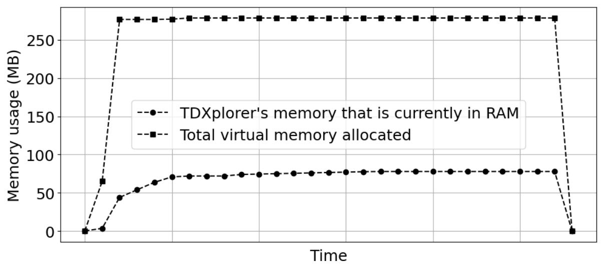

Figure 10: Memory usage of TDXplorer during symbolic path exploration. The time axis indicates the start and end of the execution.

7.5 Faithfulness Tests

The faithfulness experiments are to show that the Module's execution and the symbolic analysis outcomes from TDXplorer are faithful to the Module's execution on real hardware. Namely, the replay of test cases produces the expected outcome in real-life executions. We run the Canonical TD suite [29] to create a TD on a TDX-enabled server equipped with an Intel(R) Xeon(R) GOLD 5520+ processor (56 CPUs, clock 800 MHz - 4.0 GHz), 256 GB of RAM and 3.5 TB of disc. Both the host and the TD run Linux kernel version 6.8.0-1013-intel running Ubuntu 24.04.1 LTS. We develop a kernel module on the server to issue SEAM calls of our choice and to intercept the VMM's SEAM calls, as well as another module inside the TD to issue TD calls of our choice. Monitoring their arguments and return values allows us to maintain consistency in the TDX state between the server and the emulation environment.

Faithfulness verification is a challenging open problem. As it lacks an off-the-shelf tool to systematically verify the correctness and accuracy of an emulation, we performed manual verification on selected test cases. These represent our best effort toward faithfulness assessment. Of the dozen faithfulness tests conducted, we elaborate on two in this section.

7.5.1 Test 1: MK-TME KeyID Validation in TD Creation. In this experiment, we test whether the symbolic analysis over the Module's KeyID validation during TD creation is consistent with the execution on the server. During preparation, we clone the server's MK-TME configuration to TDXplorer so that both systems use the most significant 6 bits of the page physical address as the KeyID. In addition, we initiate the Module in both environments with a global KeyID set to 32.

Symbolic Analysis. Our analyzer function issues `TDH.MNG.CREATE`, the SEAM call that creates a TD. The function symbolizes the call's KeyID argument, which is supposedly set by the VMM. We denote the symbol as α. The function drives TDXplorer to explore all paths symbolically. During the execution, the Module reads the Module's Key Ownership Table (KOT) at its offset 8α. Located at 0xffffa00300221080, KOT stores KeyID states and ownership. To support the symbolic read, the analyzer returns an 8-byte symbolic KOT entry represented as $kote$. At the end of exploration, TDXplorer emits the path constraint:

$$(32 \leq \alpha \leq 63) \wedge (kote[0] = 0)$$

where $kote[0]$ is the least significant byte of $kote$ for all paths indicating a successful handling, i.e., no error returned to the VMM.

Replay in TDXplorer. We solve the path constraint to obtain two concrete values: $\alpha = 33$ for the success path and $\alpha = 0x8000$ for failure paths. Note that the constraint $kote[0] = 0$ requires that, in the Module's state before handling the SEAM call, the lowest byte of the KOT entry for α is 0[3]. Our dynamic analysis using TDXplorer shows that all the entries of the Module's KOT satisfy this constraint, except for $KeyID = 32$, which is the global KeyID. As expected, the replay in TDXplorer shows that: (i) for $\alpha = 33$ the SEAM call succeeds with return code 0; (ii) for $\alpha = 0x8000$ the call fails with return code 0xc000010000000000 indicating an invalid KeyID argument. To further validate the constraint's correctness, we also replay the call with $\alpha = 32$, which results in a return code 0xc000082000000000 indicating that no free KOT entry is found.

Replay on TDX Server. Using the aforementioned host kernel module, we issue `TDH.MNG.CREATE` on the server with the same three concrete KeyID values. The SEAM call return codes received by the VMM in three executions are the same as the corresponding ones in TDXplorer, confirming the accuracy of our symbolic execution and emulation in this experiment.

[3]According to the Module's code, it means the corresponding KeyID value is free.

7.5.2 Test 2: Validating SEPT Creation and GPA Mapping. We test whether the creation of the SEPT tree and the GPA mapping in TDXplorer are consistent with the SEPT states on the TDX server.

TD SEPT state creation in TDXplorer. Following the Canonical TD suite [29] and TDX ABI specification [24], we build a TD in TDXplorer. To create the SEPT tree, we first issue the `TDH.MNG.ADDCX` SEAM call to create the root SEPT page (PML5). We follow the build process and then issue a sequence of `TDH.MEM.SEPT.ADD` SEAM calls to add SEPT pages at the appropriate levels, constructing the TD SEPT tree shown in Figure 11.

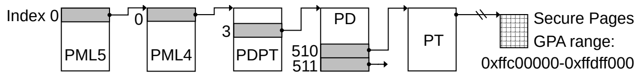

Figure 11: TD SEPT tree created to map 512 secure pages in the 2M GPA range at 0xffc00000.

We then issue 512 `TDH.MEM.PAGE.ADD` SEAM calls to map 2MB of guest physical memory at GPA 0xffc00000, attaching 512 secure 4 KB memory pages to the SEPT tree. Subsequently, we use `TDH.MEM.SEPT.RD` SEAM calls to read SEPT entries at different levels for specific GPAs. We record the return values and returned data (SEPT entry architectural contents, SEPT level and state). Specifically, we issue 3 SEAM calls to query the mapped PML5, PML4 and PDPT indexes, followed by 512 calls to read all PTEs in the PD page and another 512 calls to read entries in the PT mapped to PD index 510. All SEAM calls succeed with return code 0.

PTEs Mapping a SEPT Page. For all PTEs mapping a lower-level secure EPT page, the returned data are: (i) level matches the queried SEPT level. (ii) PTE state = 0x84 (`NL_MAPPED`), indicating the SEPT entry maps a private GPA range accessible by the guest TD. (iii) Architectural contents = 0x7, indicating the read, write and execute permissions.

Free PTEs. For the 510 free PTEs on the PD page, the returned data are: (i)level = 1, corresponding to the SEPT level 1 (PD page). (ii) PTE state = 0 (`FREE`), indicating no GPA mapping. (iii) architectural contents = 0x8000000000000000, the expected value for free PTEs.

Leaf Level PTEs. For all 512 PTEs on the PT page, the returned data are: (i) level = 0, indicating the SEPT level 0 (PT page). (ii) PTE state = 0x4 (`MAPPED`), indicating the secure EPT entry maps a private GPA page accessible by the guest TD. (iii) architectural contents include the physical address of the mapped secure page and indicate a leaf-level PTE mapping with read, write and execute permissions. For example, the PTE at index 511, which maps `0x4021f000`, returns 0x800000004021f0f7.

Replay on TDX Server. We follow the Canonical TD suite [29] to build a TD on the server. Using our kernel module, we replicate the same SEPT tree by issuing the same sequence of `TDH.MEM.SEPT.ADD` SEAM calls as in TDXplorer. We subsequently issue the same `TDH.MEM.SEPT.RD` calls to query each SEPT PTE state. The returned PTE levels, states and architectural contents match those observed in TDXplorer, confirming the correctness of SEPT tree creation and secure page mapping in TDXplorer.

8 Case Studies

8.1 Case I: Symbolic Modeling of SEPT Creation

Creation of SEPT consists of a sequence of steps to add new EPT pages to the paging hierarchy. In each step, the Module walks through the SEPT from the root to the proper parent page and modifies the PTE corresponding to the mapped GPA. Correctness of this process is crucial to security and reliability. However, it is error-prone because SEPT walking and PTE location both depend on the GPA argument used in `TDH.MEM.SEPT.ADD`.

In this case study, we symbolically analyze how the Module handles `TDH.MEM.SEPT.ADD` issuance, which adds four SEPT pages at four levels (PML4, PDPT, PD, PT), respectively, with the focus on the impact on the Module's execution from the GPA argument set by the VMM. We aim to answer the following two questions. (We do not analyze the root level (i.e., PML5) creation since its execution does not vary much with arguments.)

Q1. Given the GPA, does the Module correctly walk the SEPT to locate the parent page and its PTE, and attach the new page by updating the PTE correctly?

Q2. Does a failed SEAM call leave any undesired modifications on the SEPT after its completion? For instance, the parent page is modified; when the SEAM call is returned with an error, modifications on the PTE are not reverted.

The Module's SEPT walk with an unconstrained symbolic GPA argument leads to symbolic read operations, an open problem in the symbolic execution literature. Since a GPA in the SEAM call consists of five indexes for each SEPT level, symbolic index bits will be used as offsets to read a PTE from the page. Even though such read operations can be handled by using techniques similar to [44], it is infeasible to locate the next level page without concrete PTE content, which effectively stalls the SEPT walking.

We cope with the challenge based on the fact that the root is at a concrete location and the observation that different index bits in the GPA should be used for different SEPT levels without overlapping. Hence, we divide the whole experiment into four phases corresponding to four EPT levels. Phase 1 explores the SEAM call that adds a Level-4 page (i.e., PML4) with an unconstrained symbolic GPA. Phase 2 explores the SEAM call that adds a Level-3 page (i.e., PDPT) using a symbolic GPA with its Level-5 index being seeded with a concrete value. The symbolization-seeding cycle continues for Level-2 and Level-1 pages added in phases 3 and 4. We hence avoid the intractable problem of dereferencing a symbolic address because of the seeded values. Furthermore, the four phases of exploration, like an induction proof, collectively validate the Module's behavior on the condition that the constraints produced in symbolic exploration for each level do *not* involve irrelevant symbols in the GPA argument.

8.1.1 *Analyzer Function.*

We develop an analyzer function with 282 lines of C/C++ code following the approach above. Figure 12 depicts its workflow. Before starting to explore the SEAM call, the analyzer backs up the symbolic state and the CPU register state. At the end of the exploration, the Monitor restores the saved state and memory pages to prepare for the next level of exploration. Next, we elaborate on two implementation-relevant issues.

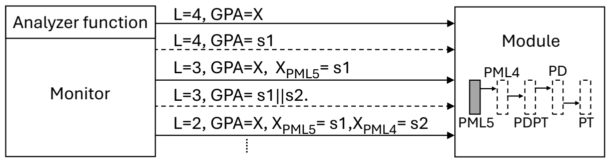

Figure 12: Workflow of the analyzer function. Solid arrows are `TDH.MEM.SEPT.ADD` calls with concrete-level numbers and symbolic GPAs. Dash arrows are concrete invocations adding four pages shown as dashed boxes, with $s1, s2$ being the seeds.

Handling Symbolic Access. When encountering an instruction accessing n bytes at a symbolic address, the analyzer function is called back by the interpreter to handle the situation. The analyzer function checks whether the symbolic address is in the form of $B + f(x)$ where B is a concrete 48-bit virtual address and $f(x)$ represents a logical and/or bit operation of symbol x followed by scalar multiplication. If not, it terminates the current path. Otherwise, the instruction is treated as accessing n bytes in an object with the base address B. The analyzer creates a shadow buffer of n bytes with metadata B, n. For a write operation, it emulates the operation upon the shadow buffer; for a read operation, it returns a new n-byte symbol representing the buffer content to the interpreter for instruction emulation.

For the subsequent access of n' bytes from a symbolic address of the form $B' + g(x)$, the analyzer function checks whether $[B' + g(x), B' + g(x) + n']$ is within $[B + f(x), B + f(x) + n]$. If so, the corresponding n' symbolic bytes from $[B + f(x), B + f(x) + n]$ are returned for a read instruction or are updated for a write access. If the two intervals are found to have no overlap, the analyzer creates a new shadow object with independent symbolization. In all other situations, including undecidable relations, it simply terminates the current path.

Address Semantics. Since the symbolic execution is in the virtual address space, it is crucial for the analyzer to acquire VA semantics to reason about the Module's behavior. For static VAs used by the Module, the needed semantics are extracted from the Module's binary. For dynamic VAs, i.e., the KeyHole regions, TDXplorer tracks the Module's mapping and un-mapping operations of KeyHoles as explained in Section 5.4. The analyzer retrieves the KeyHole state of the current LP so that it can identify the Module's virtual addresses for mapped SEPT pages. To validate the correctness of SEPT walking, the analyzer also needs the physical addresses of SEPT pages in order to compare them with the PTEs in their parent pages.

8.1.2 *Results and Discussions.*

After creating two TDs running on two LPs in the emulation environment, we run the analyzer function on top of the TDXplorer Monitor targeting one of the TDs. The exploration of adding PML4 generates 2 successful paths and 18 failure paths. For each of the other three levels' exploration, 19 paths are produced, with one successful path and 18 failed ones.

Q1. In all four phases, the modified PTE address is the sum of the concrete VA of the parent mapping the new SEPT page and 8 times a symbolic expression representing a bit segment of the symbolized GPA X. The expression occurring in each phase is in the second column of Table 3. Our manual verification confirms that the bit segments in use are correct with respect to the new page's level. The PML5 index is constrained to 3 bits and the Module attaches a new PML4 page at indexes 0 to 7 in the PML5 page, which is consistent with the TDX base specification [25]. We also confirm that the parent page's VA is mapped to the PA of the page added in the prior phase, which implies that the located PTE is on the correct parent page. In all paths where the SEAM call returns successfully, we find that the symbolic 8 bytes that are created upon reading the symbolic PTE address are all replaced with the physical address of the new page specified in `TDH.MEM.SEPT.ADD`. Hence, it confirms that the correct physical address is indeed written to the correct PTE. Additional details about the complete sequence of the Module's accesses to the parent EPT page, access sizes and path constraints are in Tables 4, 5, 6 and 7.

Added page	Modified parent SEPT index:	Constraints on parent SEPT index:	Index range
PML4	$X\_bits(48, 56)$	$X\_bits(52, 56) = 0$ $\wedge (X\_bit(51) = 0)$	{0-7}
PDPT	$X\_bits(39, 47)$	-	{0-511}
PD	$X\_bits(30, 38)$	-	{0-511}
PT	$X\_bits(21, 29)$	-	{0-511}

Table 3: Range of modified parent SEPT index in 4 phases.

Q2. Across four phases, there are 64 out of 72 SEAM call failed paths leaving a modified parent page upon call return. Table 8 reports the path constraints and symbolic expressions left on the page where *pte* represents the symbolic 8 bytes in the affected PTE. The table shows that the two constraints that appear in these paths are about the 11th bit of the PTE. It also shows that all the modifications are upon the 52nd bit of the PTE.

In fact, bits 11 and 52 of a SEPT entry represent the *entry-lock* and *host-priority* flags, which are used for synchronization among LPs according to TDX [15, 25]. The two path constraints reflect whether the Module's lock acquisition is successful or failed. For paths in which the lock acquisition is unsuccessful, the observed modification (i.e., the second `bts %rax,(%rcx)`) is to flip the *host-priority* bit from 0 to 1. It is a valid operation as it requests the priority. For paths in which the lock is successfully acquired, the observed modification (i.e., `btr %rax,(%r14)`) is to reset the *host-priority* bit from 1 to 0. The change is also valid as it indicates the priority is no longer needed. Hence, all modifications left on the PTE after the SEAM call conform to TDX functionality.

8.2 Case II: Analyzing KeyHole Mappings

8.2.1 Analysis Task. We are interested in finding out if the KeyHole mapping takes place correctly. Specifically, we aim to answer the following questions.

Q1. Is an existing matching mapping always reused? Is the returned KeyHole consistent with the matching mapped KeyHole index?

Q2. In all successful paths, is the reference count of the KeyHoles updated correctly? Specifically, is the correct KeyHole's reference count incremented by exactly one?

Since the Module's behavior depends on the KeyHole state, which is updated at runtime, we combine TDXplorer's dynamic binary analysis with symbolic execution to reason about the Module's behavior.

8.2.2 Analyzer Function. The analyzer function consists of 302 lines of C/C++ code. We specifically target the Module function `map_pa_with_memtype(KeyID_pa, mapping_type, caching_type)`. It is part of the Module's core functionality, which dynamically maps a secure page during the Module's execution. The `KeyID_pa` refers to the < KeyID | page address >, the `mapping_type` specifies if writes are allowed via the mapping and `caching_type` indicates the PTE caching type.

Start of symbolic analysis. Two TDs are created and launched in the emulation environment. Symbolic execution starts once the Module reaches the desired runtime state within the target API call handler. The Analyzer configures a hardware debug register, setting a breakpoint at the start of `map_pa_with_memtype()` and emulates the `TDG.MEM.PAGE.ATTR.RD` TD call from one TD. The TD call, requesting the GPA mapping and attributes of a TD private page, is dispatched for native execution. When the Module traps at the target function, the Agent hands control to the analyzer. The analyzer retrieves the three concrete function arguments `KeyID_pa_in`, `mapping_type_in`, `caching_type_in` from the `RDI`, `RSI` and `RDX` and saves them for result validation. To extract the mapped KeyHole VA and stop the path once the `map_pa_with_memtype()` returns, the analyzer directly reads the Module's runtime stack to obtain the function return address as follows:

```
ret_adr = *(unsigned long*)(current_rsp);
```

Initial symbolization. To achieve symbolic reasoning and enable path exploration, we symbolize selected contents in the runtime KeyHole state. The analyzer retrieves the fixed starting VA of the `tdxmod_keyhole_entry_t` array in the current KeyHole state. The array is indexed by the KeyHole index, where each KeyHole entry in this array stores metadata indicating whether the corresponding 4 KB KeyHole is currently mapped to a `KeyID_pa`. If it is, the entry also stores its `mapping_type`, `caching_type` and the reference count `ref_count`. Using its struct definition, the analyzer locates and symbolizes the `KeyID_pa` and `ref_count` elements in each of the 128 KeyHole entries in the array. The CPU and symbolic states are then backed up and the analyzer starts symbolic path exploration.

Points of analysis. When the Module creates a new mapping, the Agent intercepts the Module's KeyHole edit page write and flags the event for the use of the analyzer function. The analyzer treats the paths without a new KeyHole mapping as paths with an existing KeyHole mapping matching the target function's input arguments. Before a path is completed, the analyzer intervenes only if an error is reported by the interpreter. For errors, it records the path constraint and instructs the interpreter to begin the next path. At the end of each path, identified when the `RIP` points to the recorded `ret_adr`, the analyzer constructs a series of symbolic predicates and uses the constraint solver to gather data for symbolic reasoning as follows.

Results validation: Modified KeyHole entry. To identify the modified KeyHole entry, the analyzer compares the saved symbolic `ref_count` from the start (r_s) with that at the end (r_e) of a path. For

PML5 address	Acc. size	Read expr./val.	Write expr./val.	Path constraint at access	
$B + f'(x)$	8	$epte$	-	C_0	$C_0 : \{(\text{x\_bits}(51, 63) = 0) \wedge (\text{x\_bits}(3, 47) = 0)\}$ $C_2 :\ \text{epte\_bit}(11) = 0$ $C_3 : \{\text{0x1e00000000002c0} \wedge (\text{epte} \mid (1 << 11)) \wedge (\sim (0x1 << 0x34)) = \text{0xe0000000000000}\}$
$B + f'(x)$	8	$epte$	$epte \mid (1 << 11)$	C_0	
$B + f'(x)$	8	$epte \mid (1 << 11)$	$(epte \mid (1 << 11)) \wedge (\sim (1 << 52))$	$C_0 \wedge C_2$	
$B + f'(x)$	8	$(epte \mid (1 << 11)) \wedge (\sim (1 << 52))$	-	$C_0 \wedge C_2$	
$B + f'(x)$	8	-	0x800000004003c807	$C_0 \wedge C_2 \wedge C_3$	
$B + f'(x) + 1$	1	0xc8	-	$C_0 \wedge C_2 \wedge C_3$	
$B + f'(x)$	8	0x800000004003c807	0x800000004003c007	$C_0 \wedge C_2 \wedge C_3$	

B = PML5 base VA = 0xffffa00200105000, $f'(x) = 8 \times ((x \gg (48))\&\text{0x1ff})$, x: GPA

Table 4: Accesses to the Modified PML5 when adding a new PML4 and related path constraint at each access instance. B refers to the concrete VA of the updated PML5 page. $f'(x)$ provides the symbolic address offset. $epte$(initial symbolic value of the EPT entry). x is the SEAM call argument-GPA, symbolized at the start.

PML4 address	Acc. size	Read expr./val.	Write expr./val.	Path constraint at access	
$B + f'(x)$	8	$epte$	-	C_0	$C_0 : \{(\text{x\_bits}(51, 63) = 0) \wedge (\text{x\_bits}(3, 38) = 0)\}$ $C_1 :\ \text{x\_bits}(48, 56) = 1$ $C_2 :\ \text{epte\_bit}(11) = 0$ $C_3 : \{\text{0x1e00000000002c0} \wedge (\text{epte} \mid (1 << 11)) \wedge (\sim (0x1 << 0x34)) = \text{0xe0000000000000}\}$
$B + f'(x)$	8	$epte$	$epte \mid (1 << 11)$	C_0	
$B + f'(x)$	8	$epte \mid (1 << 11)$	$(epte \mid (1 << 11)) \wedge (\sim (1 << 52))$	$C_0 \wedge C_2$	
$B + f'(x)$	8	$(epte \mid (1 << 11)) \wedge (\sim (1 << 52))$	-	$C_0 \wedge C_2$	
$B + f'(x)$	8	-	0x800000004003d807	$C_0 \wedge C_2 \wedge C_3$	
$B + f'(x) + 1$	1	0xd8	-	$C_0 \wedge C_2 \wedge C_3$	
$B + f'(x)$	8	0x800000004003d807	0x800000004003d007	$C_0 \wedge C_2 \wedge C_3$	

B = PML5 base VA = 0xffffa00200106000, $f'(x) = 8 \times ((x \gg (39))\&\text{0x1ff})$, x: GPA

Table 5: Accesses to the Modified PML4 when adding a new PDPT and related path constraint at each access instance. B refers to the concrete VA of the updated PML4 page. $f'(x)$ provides the symbolic address offset. $epte$(initial symbolic value of the EPT entry). x is the SEAM call argument-GPA, symbolized at the start.

PDPT address	Acc. size	Read expr./val.	Write expr./val.	Path constraint at access	
$B + f'(x)$	8	$epte$	-	C_0	$C_0 : \{(\text{x\_bits}(51, 63) = 0) \wedge (\text{x\_bits}(3, 29) = 0)\}$ $C_1 :\ \text{x\_bits}(48, 56) = 1 \wedge \text{x\_bits}(39, 47) = 0$ $C_2 :\ \text{epte\_bit}(11) = 0$ $C_3 : \{\text{0x1e00000000002c0} \wedge (\text{epte} \mid (1 << 11)) \wedge (\sim (0x1 << 0x34)) = \text{0xe0000000000000}\}$
$B + f'(x)$	8	$epte$	$epte \mid (1 << 11)$	C_0	
$B + f'(x)$	8	$epte \mid (1 << 11)$	$(epte \mid (1 << 11)) \wedge (\sim (1 << 52))$	$C_0 \wedge C_2$	
$B + f'(x)$	8	$(epte \mid (1 << 11)) \wedge (\sim (1 << 52))$	-	$C_0 \wedge C_2$	
$B + f'(x)$	8	-	0x800000004003e807	$C_0 \wedge C_2 \wedge C_3$	
$B + f'(x) + 1$	1	0xe8	-	$C_0 \wedge C_2 \wedge C_3$	
$B + f'(x)$	8	0x800000004003e807	0x800000004003e007	$C_0 \wedge C_2 \wedge C_3$	

B = PML5 base VA = 0xffffa00200107000, $f'(x) = 8 \times ((x \gg (30))\&\text{0x1ff})$, x: GPA

Table 6: Accesses to the Modified PDPT when adding a new PD and related path constraint at each access instance. B refers to the concrete VA of the updated PDPT page. $f'(x)$ provides the symbolic address offset. $epte$(initial symbolic value of the EPT entry). x is the SEAM call argument-GPA, symbolized at the start.

a given KeyHole index i, it checks if the final `ref_count` is symbolic. If so, it extracts the expression r_e^i and evaluates the predicate $r_e^i = r_s^i$ using the constraint solver. If false, it checks if $r_e^i = r_s^i + 1$ with the constraint solver. If true, the KeyHole entry is identified as having the correct `ref_count` increment of 1. For a KeyHole with a symbolic `ref_count` at the start, it can only be concrete at the end if the KeyHole was previously unmapped (i.e. $r_s = 0$) and gets mapped in the current path. In such cases, the analyzer validates

PD address	Acc. size	Read expr./val.	Write expr./val.	Path constraint at access	
$B + f'(x)$	8	$epte$	-	C_0	$C_0 : \{(\text{x\_bits}(51, 63) = 0) \wedge (\text{x\_bits}(3, 20) = 0)\}$
$B + f'(x)$	8	$epte$	$epte \mid (1 << 11)$	C_0	$C_1 :\ \text{x\_bits}(48, 56) = 1 \wedge \text{x\_bits}(30, 47) = 0$
$B + f'(x)$	8	$epte \mid (1 << 11)$	$(epte \mid (1 << 11)) \wedge(\sim (1 << 52))$	$C_0 \wedge C_2$	$C_2 :\ \text{epte\_bit}(11) = 0$
$B + f'(x)$	8	$(epte \mid (1 << 11)) \wedge(\sim (1 << 52))$	-	$C_0 \wedge C_2$	$C_3 : \{\text{0x1e00000000002c0} \wedge (\text{epte} \mid (1 << 11)) \wedge (\sim (0x1 << 0x34)) = \text{0xe000000000000}\}$
$B + f'(x)$	8	-	0x800000004003f807	$C_0 \wedge C_2 \wedge C_3$	
$B + f'(x) + 1$	1	0xf8	-	$C_0 \wedge C_2 \wedge C_3$	
$B + f'(x)$	8	0x800000004003f807	0x800000004003f007	$C_0 \wedge C_2 \wedge C_3$	

B = PML5 base VA = 0xffffa00200108000, $f'(x) = 8 \times ((x \gg (21))\&\text{0x1ff})$, x: GPA

Table 7: Accesses to the Modified PD when adding a new PT and related path constraint at each access instance. B refers to the concrete VA of the updated PD. $f'(x)$ provides the symbolic address offset. $epte$(initial symbolic value of the EPT entry). x is the SEAM call argument-GPA, symbolized at the start.

Seq.	Instruction	Read expression	Write expression	Path constraints
1	`bts %rax,(%rcx)`	pte	$pte \mid (1 \ll 11)$	$pte\_bit(11) = 1$ busy lock
2	`bts %rax,(%rcx)`	$pte \mid (1 \ll 11)$	$(pte \mid (1 \ll 11)) \mid (1 \ll 52)$	
1	`bts %rax,(%rcx)`	pte	$pte \mid (1 \ll 11)$	$pte\_bit(11) = 0$ free lock
2	`btr %rax,(%rcx)`	$pte \mid (1 \ll 11)$	$(pte \mid (1 \ll 11))\&(\sim (1 \ll 52))$	
3	`btr %rax,(%r14)`	$(pte \mid (1 \ll 11))\& \sim (1 \ll 52)$	$((pte \mid (1 \ll 11))\& \sim (1 \ll 52)\& \sim (1 \ll 11)$	

Table 8: Sequence of SEPT updates in SEAM call failed paths. The memory operand of the instruction points to the PTE. pte is the initial symbolic PTE value. The ‘|’, ‘&’ and ‘≪’ represent the binary operations "OR", "AND" and "left shift", respectively.

that the `ref_count` is set to 1 and, if so, identifies the KeyHole entry as having the correctly updated `ref_count`. If the `ref_count` is incorrectly updated or the `ref_counts` of multiple KeyHole entries are updated in the same path, an error is reported.

Results validation: KeyHole, matching the input args. Once the modified KeyHole entry index (k) is determined above, the analyzer extracts its corresponding symbolic $KeyID\_pa_k$. It uses the Module’s struct definitions to read as the Module does and record the concrete `mapping_type` (M_k) and `caching_type` (C_k).

```
mapping_type = ((tdxmod_keyhole_state_t*)
    keyhole_state_va)->keyhole_array[k].
    is_writable;
caching_type = ((tdxmod_keyhole_state_t*)
    keyhole_state_va)->keyhole_array[k].
    is_wb_memtype;
```

To check if the current path has used the existing KeyHole mapping in KeyHole index k, the analyzer creates the predicate in (1):

$$KeyID\_pa\_in \neq KeyID\_pa_k \qquad (1)$$

This predicate is conjoined with the current path constraint. If evaluated to false, it indicates that the path has matched an existing mapping in KeyHole index k with `KeyID_pa_in` and the Module has updated the `ref_count` in the matching KeyHole entry. Finally, the analyzer verifies if the `mapping_type` and the `caching_type` also match by comparing the M_k and C_k with the saved input arguments and reports an error if they are not matched.

For paths with a new KeyHole mapping creation, the conjoined constraint above must be evaluated to be true. This is because there can not be a predicate in the current path constraint indicating that the `KeyID_pa` of KeyHole entry k matches the input arguments of the analyzed function. Such a predicate would only exist if the mapping for index k had already been created, which is not the case when a new mapping is created at index k.

VA assigned for the secure page. According to TDX, the base VA mapping a page at KeyHole index k is given by equation (2):

$$VA_k = B + (LP\_ID \times 128 \times 4096) + (k \times 4096) \qquad (2)$$

where B is the base VA of the Module’s KeyHole region and `LP_ID` is the current logical processor ID. The analyzer computes the expected concrete VA for the mapped KeyHole index k and compares it with the return value of `map_pa_with_memtype()` extracted from the `RAX` register when the path ends at the function return (`ret_adr`).

8.2.3 Results. The path exploration covers 180 paths, with 62 paths ending in error (via `UD2` instruction). Note that the interpreter does not dispatch the `UD2` for native execution. In these error paths, the final predicate added to the path constraint: $r_s^i + 1 = 0$, identifies an overflow scenario. Post-mortem analysis shows that the r_e^i expression from the i-th KeyHole entry equals $r_s^i + 1$. Therefore, the error occurs because the symbolic execution explored a path where the updated `ref_count` overflows.

Q1. Out of the 118 paths reaching the function return, 62 paths use existing mappings and 56 paths create new PT mappings. The analyzer’s end-of-path analysis shows that when the path constraint indicates a match with a KeyHole entry, the returned VA matches the KeyHole VA derived from equation (2), using the corresponding matching KeyHole index. In all such paths, no PT mappings are created.

Q2. All 118 paths have one KeyHole entry with updated `ref_count`. In all paths that do not create new mappings, the updated `ref_count` entry belongs to the matching KeyHole entry in the path constraint. In these paths that use an existing mapping, the final `ref_count` is symbolic and always incremented by 1. In contrast, in paths that create a new mapping, the final `ref_count` becomes concrete and is set to 1.

8.3 Lessons Learned

The two cases demonstrate that TDXplorer can be applied to generalize and reason about the Module's behavior, a basic functionality of symbolic execution. Moreover, they attest to the benefits of flexibly combining symbolic execution with commonly used dynamic analysis techniques such as hardware breakpoint, single-stepping, introspection and instrumentation. Owing to the unification of all these operations at the binary level, the analyst enjoys greater flexibility in controlling the Module execution and access to richer and more accurate runtime data than what both source-code level analysis and intermediary representation-based symbolic execution can provide.

The cases also show that symbolically exploring the TDX Module faces the roadblocks of reading or writing to symbolic addresses. Although these are well-known open problems in the symbolic execution literature, their obstruction is more evident as it is common for the Module to use the SEAM/TD call argument to locate data objects, leading to symbolic accesses. We observe that TDXplorer is likely to be more conducive to finding a solution than existing work because its architectural feature allows for nimble and easy memory modeling. It is in our future work to tackle the symbolic access challenge.

In terms of code coverage, Case 1 reaches 328 out of 687 basic blocks, while Case 2 covers 44 out of 61. It is important to emphasize that our cases are not designed with the goal of maximizing code coverage. The uncovered blocks are primarily a result of branches that depend on concrete data rather than the symbolized argument.

9 Conclusion

In summary, TDXplorer is a system framework that faithfully runs the TDX Module in an emulation environment and symbolically explores its execution. A user's analyzer function on TDXplorer can not only apply dynamic analysis techniques to shape and retrieve the Module's runtime state, but also harness symbolic execution to generalize and reason about the Module's behavior, as evidenced by our two case studies.

Acknowledgments

We are grateful to anonymous reviewers for their constructive comments and suggestions. This 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.

References

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

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

[3] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. 2014. Automatic exploit generation. *Commun. ACM* 57, 2 (2014), 74–84.

[4] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*. 41–46.

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 209–224.

[6] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 380–394.

[7] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. 2020. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In *Proceedings of the 29th USENIX Security Symposium*. 1093–1110.

[8] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. 2023. Intel tdx demystified: A top-down approach. *Comput. Surveys* (2023).

[9] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E platform: Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)* 30, 1 (2012), 1–49.

[10] Intel Corporation. 2023. *Browse Intel TDX Documentation.* Retrieved April 25th, 2024 from https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html

[11] Intel Corporation. 2024. *Issue 8: Canary validation in tdh_sys_lp_init().* Retrieved August 21st, 2025 from https://github.com/intel/tdx-module/issues/8

[12] Intel Corporation. 2025. *Issue 15: Input validation in SEAM/TD Calls for reading metadata.* Retrieved August 21st, 2025 from https://github.com/intel/tdx-module/issues/15

[13] Intel Corporation. 2025. *Issue 20: Incorrect output operand values on TD Call failure due to invalid RAX.* Retrieved August 21st, 2025 from https://github.com/intel/tdx-module/issues/20

[14] Intel Corporation. 2025. *Issue 21: Incorrect output operand values on SEAM Call failure due to invalid RAX.* Retrieved August 21st, 2025 from https://github.com/intel/tdx-module/issues/21

[15] Intel Corporation. 2025. TDX Module. https://github.com/intel/tdx-module. Accessed: 2025-04-13.

[16] Intel Corporation. 2025. TDX Module Build Instructions (tdx_1.5 branch). https://github.com/intel/tdx-module/blob/tdx_1.5/BUILD.md. Accessed: 2025-04-13.

[17] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. {FIE} on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *22nd USENIX Security Symposium (USENIX Security 13)*. 463–478.

[18] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, et al. 2021. Sok: Enabling security analyses of embedded systems via rehosting. In *Proceedings of the 2021 ACM Asia conference on computer and communications security*. 687–701.

[19] Anthony C. J. Fox, Gareth Stockwell, Shale Xiong, Hanno Becker, Dominic P. Mulligan, Gustavo Petri, and Nathan Chong. 2023. A Verification Methodology for the Arm® Confidential Computing Architecture: From a Secure Specification to Safe Implementations. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 88 (April 2023), 30 pages. doi:10.1145/3586040

[20] Frama-c. 2023. *Frama-c Software Analyzers.* Retrieved November 8th, 2023 from https://frama-c.com/

[21] Google LLC and Intel Corporation. 2023. Intel TDX Security Review Report. https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf. Accessed: 2025-04-14.

[22] Intel Corporation. 2024. Intel TDX Module: KVM Upstream Branch. https://github.com/intel/tdx/tree/kvm-upstream. Accessed: 2025-04-14.

[23] Intel Corporation. 2024. *Intel® Trust Domain CPU Architectural Extensions.* Available at: https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html.

[24] Intel Corporation. 2024. *Intel® Trust Domain Extensions Module Architecture Application Binary Interface Specification.* Available at: https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html.

[25] Intel Corporation. 2024. *Intel® Trust Domain Extensions (TDX) Base Specification.* Available at: https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html.

[26] Zheyue Jiang, Yuan Zhang, Jun Xu, Xinqian Sun, Zhuang Liu, and Min Yang. 2023. AEM: Facilitating Cross-Version Exploitability Assessment of Linux Kernel Vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 588–603.

[27] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 1–17.

[28] Daniel Kroening and Michael Tautschnig. 2014. CBMC–C Bounded Model Checker: (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*. Springer, 389–391.

[29] Canonical Ltd. 2024. Intel confidential computing - TDX. https://github.com/canonical/tdx Accessed: 2025-04-09.

[30] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices* 40, 6 (2005), 190–200.

[31] Dominik Maier, Lukas Seidel, and Shinjo Park. 2020. Basesafe: Baseband sanitized fuzzing through emulation. In *Proceedings of the 13th ACM conference on security and privacy in wireless and mobile networks*. 122–132.

[32] Microsoft. 2024. Cornelius. https://github.com/microsoft/Cornelius. Accessed: 2025-04-14.

[33] Microsoft and Intel Corporation. 2024. Microsoft and Intel joint security review of Intel TDX 1.5. https://community.intel.com/t5/Blogs/Products-and-Solutions/Security/Intel-and-Microsoft-joint-security-review-of-Intel-TDX-1-5/post/1615189. Accessed: 2025-04-14.

[34] Petar Paradžik, Ante Derek, and Marko Horvat. 2025. Formal Security Analysis of the AMD SEV-SNP Software Interface. *IEEE Transactions on Dependable and Secure Computing* (2025), 1–18. doi:10.1109/TDSC.2025.3528737

[35] Pansilu Pitigalaarachchi, Xuhua Ding, Haiqing Qiu, Haoxin Tu, Jiaqi Hong, and Lingxiao Jiang. 2023. KRover: A Symbolic Execution Engine for Dynamic Kernel Analysis. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. 2009–2023.

[36] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 1–18.

[37] Dyninst Project. 2021. *Dyninst*. Retrieved September 8th, 2023 from https://github.com/dyninst/dyninst/tree/v12.0.0

[38] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. 2012. SymDrive: Testing drivers without devices. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 279–292.

[39] Microsoft Research. 2021. *Z3*. Retrieved September 8th, 2024 from https://github.com/Z3Prover/z3/tree/z3-4.8.15

[40] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. 2024. {HECKLER}: Breaking Confidential {VMs} with Malicious Interrupts. In *33rd USENIX Security Symposium (USENIX Security 24)*. 3459–3476.

[41] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. 2012. Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties. In *2012 IEEE 25th Computer Security Foundations Symposium* (2012-06). 78–94. doi:10.1109/CSF.2012.25

[42] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 38–157.

[43] Intel® Architecture Memory Encryption Technologies. 2024. *https://cdrdv2-public.intel.com/679154/multi-key-total-memory-encryption-spec-1.4.pdf.*

[44] Haoxin Tu, Lingxiao Jiang, Jiaqi Hong, Xuhua Ding, and He Jiang. 2024. Concretely mapped symbolic memory locations for memory error detection. *IEEE Transactions on Software Engineering* (2024).

[45] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, Bingchang Liu, Kaixiang Chen, and Wei Zou. 2018. Revery: From Proof-of-Concept to Exploitable (One Step towards Automatic Exploit Generation). In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1914–1927.

[46] weggli rs. 2023. *weggli*. Retrieved November 8th, 2023 from https://github.com/weggli-rs/weggli

[47] Luca Christopher Wilke, Florian Sieck, and Thomas Eisenbarth. 2024. TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX. In *Proceedings of ACM Conference on Computer and Communications Security*.

[48] Isaku Yamahata. 2022. Allowing an Intel TDX Module to Run Without SEAM. https://www.youtube.com/watch?v=wNq6shZCYm0. KVM Forum 2022, Intel Corporation.

[49] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2020. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium*. 745–761.

[50] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. 2014. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares.. In *NDSS*, Vol. 14. 1–16.