PDF Download
3652597.pdf
29 December 2025
Total Citations: 56
Total Downloads:
14572

Published: 25 April 2024
Online AM: 14 March 2024
Accepted: 07 March 2024
Revised: 07 February 2024
Received: 27 March 2023

Citation in BibTeX format

TUTORIAL

Intel TDX Demystified: A Top-Down Approach

PAUCHEN CHENG, IBM Research, Yorktown Heights, NY, United States

WOJCIECH OZGA, IBM Research Europe, Ireland, Dublin, Ireland

ENRIQUILLO VALDEZ, IBM Research, Yorktown Heights, NY, United States

SALMAN AHMED, IBM Research, Yorktown Heights, NY, United States

ZHONGSHU GU, IBM Research, Yorktown Heights, NY, United States

HANI T JAMJOOM, IBM Research, Yorktown Heights, NY, United States

View all

Open Access Support provided by:

IBM Research

IBM Research Europe, Ireland

ACM Computing Surveys, Volume 56, Issue 9 (September 2024)
https://doi.org/10.1145/3652597
EISSN: 1557-7341

Intel TDX Demystified: A Top-Down Approach

PAU-CHEN CHENG, IBM Research, Yorktown Heights, NY, USA
WOJCIECH OZGA, IBM Research Europe, Zurich, Switzerland
ENRIQUILLO VALDEZ, IBM Research, Yorktown Heights, NY, USA
SALMAN AHMED, IBM Research, Yorktown Heights, NY, USA
ZHONGSHU GU, IBM Research, Yorktown Heights, NY, USA
HANI JAMJOOM, IBM Research, Yorktown Heights, NY, USA
HUBERTUS FRANKE, IBM Research, Yorktown Heights, NY, USA
JAMES BOTTOMLEY, IBM Research, Yorktown Heights, NY, USA

Intel Trust Domain Extensions (TDX) is an architectural extension in the 4th Generation Intel Xeon Scalable Processor that supports confidential computing. TDX allows the deployment of virtual machines in the Secure-Arbitration Mode (SEAM) with encrypted CPU state and memory, integrity protection, and remote attestation. TDX aims at enforcing hardware-assisted isolation for virtual machines and minimize the attack surface exposed to host platforms, which are considered to be untrustworthy or adversarial in the confidential computing's new threat model. TDX can be leveraged by regulated industries or sensitive data holders to outsource their computations and data with end-to-end protection in public cloud infrastructures.

This article aims at providing a comprehensive understanding of TDX to potential adopters, domain experts, and security researchers looking to leverage the technology for their own purposes. We adopt a top-down approach, starting with high-level security principles and moving to low-level technical details of TDX. Our analysis is based on publicly available documentation and source code, offering insights from security researchers outside of Intel.

CCS Concepts: • **Security and privacy** → **Systems security**; • **Computer systems organization** → **Processors and memory architectures;**

Additional Key Words and Phrases: Confidential computing, trusted execution environment

ACM Reference Format:
Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. 2024. Intel TDX Demystified: A Top-Down Approach. *ACM Comput. Surv.* 56, 9, Article 238 (April 2024), 33 pages. https://doi.org/10.1145/3652597

1 INTRODUCTION

Deploying computations to cloud infrastructures can reduce costs, but regulated industries have concerns about moving sensitive data to third-party cloud service providers. Confidential

Authors' addresses: P.-C. Cheng, E. Valdez, S. Ahmed, Z. Gu, H. Jamjoom, H. Franke, and J. Bottomley, IBM Research - Yorktown Heights, 1101 Kitchawan Rd, Yorktown Heights, NY, USA 10598; e-mails: pau@us.ibm.com, rvaldez@us.ibm.com, sahmed@ibm.com, zgu@us.ibm.com, jamjoom@us.ibm.com, frankeh@us.ibm.com, jejb@us.ibm.com; W. Ozga, IBM Research Europe - Zurich, Säumerstrasse 4, CH -8803 Rüschlikon, Switzerland; e-mail: woz@zurich.ibm.com.

ACM 0360-0300/2024/04-ART238
https://doi.org/10.1145/3652597

computing aims at providing end-to-end protection for outsourced computations by minimizing the root-of-trust in the processors and their vendors. All data must be protected throughout its life cycle, from leaving its owners' premises to entering certified CPU packages in the cloud. Adversaries, such as those intercepting data on the network, disk storage, or main memory, should not be able to access the data in clear form.

Cryptographic mechanisms, such as storage encryption and secure communication channels, protect the confidentiality, integrity, and authenticity of data both *at rest* and *in transit*. The emerging CPU-based Trusted Execution Environment (TEE) techniques aim at protecting data *in use*, i.e., data loaded into main memory.

Intel Trust Domain Extensions (TDX) is an architectural extension that provides TEE capabilities in the 4th Generation Intel Xeon Scalable Processors. TDX introduces the SEAM to offer cryptographic isolation and protection for Virtual Machines (VMs), which are called Trust Domains (TDs) in the TDX terminology. The threat model assumes that the privileged software, such as hypervisor or host Operating System (OS), may be untrustworthy or adversarial. TDX aims at protecting the confidentiality and integrity of CPU state and memory for designated TDs and also enables TD owners to verify the authenticity of remote platforms. TDX is built using a combination of techniques, including Virtualization Technology (VT) [64], Multi-key Total Memory Encryption (MKTME) [31], and TDX Module [40]. TDX also relies on Software Guard Extensions (SGX) [53] and Data Center Attestation Primitives (DCAP) [58] for remote attestation.

Throughout the article, we aim at giving an objective review of TDX. Our goal is to provide a thorough understanding of TDX to potential adopters, domain experts, and security researchers who want to leverage or investigate the technology for their own purposes. All the information is based on publicly available documentation [26, 32, 39, 40, 42] and source code [27–29].

The following is a roadmap of this article. We begin by outlining the security principles (Section 2) and the threat model (Section 3) of TDX. Next, we provide a comprehensive comparison of existing confidential computing technologies on the market (Section 4) and examine the existing Intel technologies that serve as the building blocks for TDX (Section 5). Once the background knowledge is established, we offer a high-level overview of TDX (Section 6) and then delve into the technical details of the TDX Module (Section 7), memory protection mechanisms (Section 8), and remote attestation (Section 9). Finally, we conclude with a summary (Section 10). To assist readers in navigating the numerous terms and abbreviations used in this article, a list of acronyms is also provided (Section A).

2 SECURITY PRINCIPLES

In cloud computing, multiple security domains, e.g., a hypervisor managed by a cloud service provider and VMs owned by different tenants, coexist on a shared physical machine. While hardware-assisted virtualization can isolate tenants' workloads, the security model still relies on a privileged hypervisor to provide trustworthy VM management. To address this issue, TDX enforces cryptographic isolation among the security domains, thereby mitigating cross-domain attacks. This eliminates hierarchical dependencies on untrusted/privileged host software and excludes hypervisors and cloud operators from the Trusted Computing Base (TCB), allowing tenants to securely provision and run their computations with confidence.

TDX guarantees confidentiality and integrity of TD's memory and virtual CPU states, ensuring that they cannot be accessed or tampered with by other security domains executing on the same machine. This is achieved through a combination of (1) memory access control, (2) runtime memory encryption, and (3) an Intel-signed TDX Module that handles security-sensitive TD management operations.

In addition, remote attestation provides tenants with proof of the authenticity of TDs executing on genuine TDX-enabled Intel processors. These guarantees are based on a specific threat model and require certain trust assumptions.

Memory Confidentiality. TD's data residing inside the processor package are stored in clear text. However, when the data is offloaded from the processor to the main memory, the processor encrypts it using a TD-specific cryptographic key known only to the processor. The encryption is performed at the cache line granularity, making it impossible for peripheral devices to read or tamper with the TD's private memory without detection. The processor can detect any tampering that may occur when loading data from the main memory.

CPU State Confidentiality. TDX protects against concurrently executing processes by managing the virtual CPU states of TDs during all context switches between security domains. The states are stored in the TD's metadata, which is protected while in main memory using the TD's key. During context switches, TDX clears or isolates the TD-specific states from internal processor registers and buffers, such as Translation Lookaside Buffer (TLB) entries or branch prediction buffers, to maintain the protection of the TD's information.

Execution Integrity. TDX protects the integrity of TD's execution from host interference, ensuring that the TD resumes its computation after an interrupt at the expected instruction within the expected states. It is capable of detecting malicious changes in the virtual CPU states, as well as injection, modification, or removal of instructions located in the private memory. However, TDX does not provide additional guarantees for the control flow integrity. It is the responsibility of the TD owner to use existing compilation-based or hardware-assisted control flow integrity enforcement techniques, such as Control Flow Enforcement Technology (CET) [25].

I/O Protection. Peripheral devices or accelerators are outside the trust boundaries of TDs and should not be allowed to access TD's private memory. To support virtualized I/O, a TD can choose to explicitly share memory for data transfer purposes. However, TDX does not provide any confidentiality and integrity protection for the data located in shared memory regions. It is the responsibility of TD owners to implement proper mechanisms, such as using secure communication channels like Transport Layer Security (TLS), to protect the data that leaves the TD's trust boundary. In the future, TDX 2.0 is planned to include TDX Connect [35, 38] to address the trusted I/O issue.

3 THREAT MODEL

TDX operates on the assumption that adversaries may have physical or remote access to a computer and may be able to gain control over the boot firmware, System Management Mode (SMM), host OS, hypervisor, and peripheral devices. The primary objective of these adversaries is to obtain confidential data or interfere with the execution of a TD. It is important to note that TDX cannot guarantee availability, as adversaries can control all the compute resources for TDs and launch Denial of Service (DoS) attacks. It is crucial for the TDX design to prevent adversaries from conducting actions that compromise the TDX security guarantees outlined in Section 2. Below, we summarize the capabilities of adversaries and identify potential attack vectors and scenarios.

Adversaries can interact with the TDX Module through its host-side interface functions, which allow them to build, initialize, measure, and tear down TDs. These interface functions can be invoked in an arbitrary order with semantically and syntax valid/invalid inputs.

Adversaries can control the compute resources assigned to TDs, including physical memory pages, processor time, and physical/virtual devices. They can interrupt TDs at any point, and try to read and write to arbitrary memory locations, as well as reconfigure the Input/Output Memory Management Unit (IOMMU).

Adversaries have the capability of manipulating the input data for TDs [30], including Advanced Configuration and Power Interface (ACPI) tables, Peripheral Component Interconnect (PCI) config, Model-Specific Register (MSR), Memory-Mapped Input/Output (MMIO), Direct Memory Access (DMA), emulated devices, hypercalls handled by the host, source of randomness, and time notion.

Adversaries can conduct physical and hardware attacks, for instance, by probing buses or accessing main memory through malicious DMA. There is no defense against physical attacks that roll back arbitrary memory regions. However, it should not be possible for adversaries to extract the secret key material baked into the processor chip's fuses. The scope of the threat model does not cover fault injections or side-channel attacks such as power glitches, time, and power analysis.

Attacking TDX attestation is within the scope as it undermines the trust model and may enable adversaries to forge a counterfeit TEE for collecting confidential information from tenants.

3.1 TCB

The TCB of TDX consists of the TDX-enabled Intel processors and the built-in technologies, such as VT, MKTME, and SGX. The TCB also includes software modules signed by Intel, including the TDX Module, the NP/P-SEAM Loaders, and architectural SGX enclaves for remote attestation. The software stacks running within TDs are owned by the tenant and are considered part of the TCB. The cryptographic primitives used in TDX are considered sound and their implementation secure, including the generation of random numbers and the absence of side-channel attacks like timing attacks.

Tenants must trust the processor manufacturer, Intel, for developing, manufacturing, building, and signing the hardware/software components used by TDX. The source code packages for the TDX Module, the NP/P-SEAM Loaders, and the DCAP for attestation are publicly available for audit purposes, allowing tenants to assess their trustworthiness. However, tenants must also trust that the version signed by Intel is equivalent to the one they have reviewed, which involves placing trust in the compilation process to protect against supply chain attacks.

Moreover, tenants are required to trust Intel's Provisioning Certification Service (PCS) for remote attestation. The PCS, which originally supported SGX attestation, has been expanded to include retrieval of Provisioning Certification Key (PCK) certificates, revocation lists, and TCB information for TDX.

4 COMPARISON OF CONFIDENTIAL COMPUTING TECHNOLOGIES

Confidential computing technologies share a common objective of protecting outsourced sensitive data and computations from unauthorized access, tampering, and disclosure on untrusted third-party infrastructures. Major processor vendors are competing to incorporate confidential computing capabilities into their chips. Despite differences in implementation and terminology, these technologies share fundamental security principles with similar system designs, such as introducing new execution modes or privilege levels, migrating VM management functions to attested firmware/software, ensuring secure or measured launch of trusted components, enforcing memory access control, and providing memory encryption protection.

In addition to Intel TDX, we provide a brief overview of the confidential computing technologies from other vendors, including AMD Secure Encrypted Virtualization (SEV), IBM Secure Execution and Protected Execution Facility (PEF), Arm Confidential Compute Architecture (CCA), and RISC-V Confidential VM Extension (CoVE), for comparison purposes. We have summarized the distinct features of these technologies in Table 1. Readers already familiar with these technologies

Table 1. Summary of Comparable Confidential Computing Technologies

Technology	Summary
AMD SEV [3, 43, 44]	- enforces cryptographic VM isolation via AMD PSP - supports memory encryption (SEV), CPU state encryption (SEV-ES), integrity protection (SEV-SNP) - provides hardware isolated layers within VMs through VMPL
IBM Secure Execution [24]	- protects SVMs on IBM Z and LinuxONE. - leverages a trusted firmware, Ultravisor, to bootstrap and run SVMs - provides end-to-end protection from the boot image to memory and throughout execution
IBM PEF [22]	- protects SVMs on Power ISA - leverages the Protected Execution Ultravisor to manage SVM execution - utilizes TPM, secure boot, and trusted boot for integrity check and bootstrap SVMs
Arm CCA [50]	- introduces Realm world for running confidential VMs - introduces Root world to enforce address space isolation through GPT - support attestation for Realm environment
RISC-V CoVE [57]	- introduces the TSM to manage TVM life cycles - uses MTT to track memory page assignment - adopts a layered attestation architecture

can skip this section and proceed directly to Section 5, where we explain the existing Intel technologies that support TDX.

4.1 AMD SEV

SEV [44] is a confidential computing feature in AMD EPYC processors. It protects sensitive data stored within VMs from privileged software or administrators in a multi-tenant cloud environment. SEV relies on AMD Secure Memory Encryption (SME) and AMD Virtualization (AMD-V) to enforce cryptographic isolation between VMs and the hypervisor. Each VM is assigned a unique ephemeral Advanced Encryption Standard (AES) key, which is used for runtime memory encryption. The AES engine in the on-die memory controller encrypts or decrypts data written to or read from the main memory. The per-VM keys are managed by the AMD Platform Security Processor (PSP), which is a 32-bit Arm Cortex-A5 micro-controller integrated within the AMD System-on-Chip (SoC). The C-bit (bit 47) in physical addresses determines memory page encryption. SEV also provides a remote attestation mechanism that allows the VM owners to verify the trustworthiness of VMs' launch measurements and the SEV platforms. The PSP generates the attestation report signed by an AMD-certified attestation key. The VM owners can verify the authenticity of the attestation report and the embedded platform/guest measurements.

AMD has released three generations of SEV. The first generation SEV [44] only protects the confidentiality of a VM's memory. The second generation SEV-ES (Encrypted State) [43] adds protection for CPU register state during hypervisor transition, and the third generation SEV-SNP (Secure Nested Paging) [3] adds integrity protection to prevent memory corrupting, replaying, and remapping attacks. Particularly, SEV-SNP provides memory integrity protection using Reverse Mapping Table (RMP). RMP tracks each page's ownership and permissions to prevent unauthorized access. SEV-SNP also introduces the Virtual Machine Privilege Level (VMPL) feature by dividing the guest address space into four levels and providing additional security isolation within a VM. The privilege levels range from zero to three, where VMPL0 is the highest level of privilege and VMPL3 is the lowest. For instance, the Linux Secure VM Service Module (SVSM) [61] makes extensive use of the RMP and VMPL features to perform sensitive services, e.g., live migration and vTPM, in a secure manner.

4.2 IBM Confidential Computing

IBM's early exploration of confidential computing can be traced back to the research on SecureBlue++ [10, 70], which included running on an emulated POWER processor on the Mambo CPU simulator [9]. Today, IBM Systems support two architectures for confidential computing: Secure Execution [24], offered on IBM Z and LinuxONE, and PEF [22], released as an open source project on OpenPOWER systems.

IBM Secure Execution. IBM Secure Execution provides support for Secure Virtual Machines (SVMs) that run inside isolated TEEs since IBM Z15 and LinuxONE III. Secure Execution protects the confidentiality, integrity, and authenticity of code and data in an SVM from any unauthorized access and snooping or tampering. Secure Execution leverages trusted firmware, called the Ultravisor, to perform security-sensitive tasks to bootstrap and run SVMs. The Ultravisor shields the SVM's memory and its state during context switches and protects the SVM from a potentially compromised or malicious hypervisor. Tenants using Secure Execution can embed their encrypted sensitive data in the VM images and rely on the Ultravisor to decrypt and expose them to the SVMs executing inside the TEEs. Specifically, tenants can encrypt their confidential data with a symmetric data key, which they embed in the IBM Secure Execution Header. They further encrypt this header with the key obtained from the verified Host Key Document and embed the header in their VM image. The header can contain multiple key slots that allow an image to run on multiple target hosts. The Host Key Document, signed by the hardware manufacturer, contains the public key linked with the private key embedded in the hardware of IBM Z or LinuxONE. Ultravisor, the only component having access to the hardware private key and the data key, enforces that only the expected tenant's SVM executing inside the TEE has access to the unencrypted data. In addition to embedding built-in secrets within the VM image, Secure Execution also supports remote attestation starting from IBM Z16 and LinuxONE Emperor 4. This allows tenants to verify the SVM's measurements before releasing their secrets.

IBM PEF. PEF provides a VM-based TEE using extensions to the IBM Power Instruction Set Architecture (ISA) that are supported in most POWER9 and POWER10 processors. PEF firmware, tooling to prepare SVMs, and OS extensions, were released as open source software [23]. To protect sensitive data and code, PEF introduces a trusted firmware called Protected Execution Ultravisor (Ultravisor) that shields the SVM execution and enforces the security guarantees with the help of the CPU architectural changes. The PEF relies on the secure and trusted boot of the system and the Ultravisor executing in a new, highest privileged CPU state called Secure State. The hypervisor starts the VM, which invokes the Ultravisor to transition to an SVM using the Enter Secure Mode (ESM) call. The Ultravisor converts the VM into an SVM by moving it to the secure memory that is inaccessible to untrusted code. Before executing the SVM, the Ultravisor performs integrity checking. It decrypts the payload attached to the SVM image to decode the integrity information and a passphrase for the encrypted file system. After ensuring the integrity of the SVM, the Ultravisor exposes the passphrase to the SVM booting system that decrypts the tenant's file system. The Ultravisor uses the Trusted Platform Module (TPM) to get access to the symmetric seed required to check integrity and decrypt the payload. The symmetric seed is guarded using the Platform Configuration Register (PCR) sealing mechanism and accessed by establishing a secure channel to the TPM. The TPM only grants access to an Ultravisor on a correctly booted system. If the Ultravisor gets access to the symmetric seed, it generates the HMAC key and symmetric key that are used to verify integrity and decrypt the passphrase.

4.3 Arm CCA

CCA [50] was introduced in the Armv9 architecture. Traditionally, Arm TrustZone allows secure execution by having two separated worlds, the Normal World and the Secure World. TrustZone prevents software in the Normal World from accessing data in the Secure World. CCA introduces the Realm Management Extension (RME) with two additional worlds, the Realm World and the Root World. The Realm World provides mutually distrusting execution environments for confidential VMs, isolating workloads from any other security domains, including host OS, hypervisor, other Realms, and TrustZone. To enforce the isolation of address spaces, CCA uses a Granule Protection Table (GPT), which is an extension to the page table that tracks the ownership of each page with different worlds. The Monitor in the Root World handles the creation and management of the GPT, preventing a hypervisor or an OS from directly changing it. The Monitor can dynamically move physical memory between different worlds by updating the GPT. CCA also supports attestation to measure and verify the CCA platform and the initial state of the Realms.

4.4 RISC-V CoVE

CoVE [57] is a reference confidential computing architecture for RISC-V. Its protected instance is called a TEE Virtual Machine (TVM). The architecture introduces the TEE Security Manager (TSM) driver, which is an M-mode (highest privilege level in RISC-V) firmware component for switching between confidential and non-confidential environments. The TSM driver tracks the assignment of memory pages to TVMs through the Memory Tracking Table (MTT). The TSM driver measures and loads the TSM, which is a trusted intermediary between the hypervisor and the TVMs. CoVE defines the Application Binary Interface (ABI) for the hypervisor to request virtual machine management services from the TSM. CoVE adopts a layered attestation architecture, which begins with the hardware and progresses through the TSM driver, TSM, and TVM. Each layer is loaded, measured, and certified by the previous layer. This approach provides a secure chain of trust that can be used to verify the integrity of the system. The TVM can obtain a certificate from the TSM that contains attestation evidence rooted back to the hardware. This certificate provides a mechanism for verifying the authenticity of the TVM and the software it runs.

5 BUILDING BLOCKS FOR TDX

TDX relies on a combination of existing Intel technologies, including VT, Total Memory Encryption (TME)/MKTME, and SGX. In this section, we provide an overview of these underpinning technologies and explain how they are used in TDX. A summary of these technologies can be found in Table 2.

5.1 Intel VT

Intel VT [64] is a set of hardware-assisted virtualization features in Intel processors. Using VT, Virtual Machine Monitors (VMMs) or hypervisors can achieve better performance, isolation, and security compared to software-based virtualization. Intel's VT portfolio includes, among others, the virtualization of CPU, memory, and I/O.

Processors with VT-x technology have a special instruction set, called Virtual Machine Extensions (VMX), which enables control of virtualization. Processors with VT-x technology can operate in two modes: VMX root mode and VMX non-root mode. The hypervisor runs in VMX root mode while the guest VMs run in the VMX non-root mode. VT-x defines two new transitions, VM entry and VM exit, to switch between the guest and the hypervisor. The

Table 2. Summary of Existing Building Blocks for TDX

Technology	**Summary**
Intel VT [64]	- provides hardware-assisted virtualization for CPU, memory, and I/O - enforces isolation among VMs via a trusted hypervisor
Intel TME	- encrypts entire main memory - uses a single and boot-time generated transient key - uses the AES-XTS algorithm with 128-bit keys or 256-bit keys
Intel MKTME [31]	- supports multiple keys for memory encryption - enables memory encryption at the page granularity
Intel SGX [53]	- encloses sensitive code and data of an application within an enclave - protects against memory bus snooping and cold boot attacks with memory encryption - supports local and remote attestation

Virtual Machine Control Structure (VMCS) is a data structure that stores VM and host state information for mode transitions. It also controls which guest operations can cause VM exits.

Intel VT-x utilizes Extended Page Table (EPT) for implementing Second Level Address Translation (SLAT). Each guest kernel maintains its page table to translate Guest Virtual Address (GVA) to Guest Physical Address (GPA). The hypervisor manages EPT to map GPA to Host Physical Address (HPA).

VMs can use different I/O models, including software-based and hardware-based models, to access I/O devices. Software-based I/O models involve emulated devices or para-virtualized devices, while hardware-based I/O models include direct device assignment, Single Root I/O virtualization (SR-IOV) devices, and Scalable I/O virtualization (S-IOV) devices.

Intel VT for Directed I/O (VT-d) enables the isolation and restriction of device accesses to entities managing the device. It includes I/O device assignment, DMA remapping, interrupt remapping, and interrupt posting. With the support of VT-d, VMs can directly access physical I/O memory through virtual-to-physical address translation with the help of the IOMMU. VT-d also provides flexibility in I/O device assignments to VMs and eliminates the need for the hypervisor to handle interrupts and DMA transfers. Overall, VT-d enhances the performance and security of virtualized environments that require direct access to I/O devices.

VT ⇒ TDX. TDX is a VM-based TEE. It relies on the VT to provide isolation among TDs. As the hypervisor is no longer trusted in the new threat model, the functionalities of managing TDs have been enclosed within the TDX Module. The TDX Module and TDs run in the new SEAM VMX root/non-root mode with additional protection. TDX still leverages EPT to manage GPA-to-HPA translation. But currently, it maintains two EPTs for each TD, a protected one for private (encrypted) memory and another one for shared (unencrypted) memory. We provide a detailed explanation of the TDX's architecture and the TDX Module in Sections 6.1 and 7.

It is worth noting that currently nested virtualization is not supported in TDX 1.0, which means that running VMs within a TD is not allowed. Attempting to use VMX instructions within a TD can result in Undefined Instruction (UD) exceptions. But the TD partitioning architecture specification draft [37] indicates that nested virtualization will be supported in TDX 1.5 in the future.

5.2 Intel TME/MKTME

TME was first introduced with the Intel 11th Generation Core vPro mobile processor. This feature is designed to protect against attackers who have physical access to a computer's memory and attempt to steal data. TME encrypts the entire computer's memory using a single transient key. The key is generated at boot-time through a combination of hardware-based random number generators and security measures integrated into the system's chipset. Memory encryption is performed by encryption engines on each memory controller. The encryption process uses the NIST standard AES-XTS algorithm with 128-bit or 256-bit keys.

MKTME [31] extends TME to support multiple keys and memory encryption at page granularity. For each memory transaction, MKTME extracts Host Key Identifier (HKID) from the physical memory address and selects a corresponding key to encrypt/decrypt memory. HKID occupies a configurable number of bits from the top of the physical address. The HKIDs range is set by the BIOS during system boot. MKTME allows for software-provided keys and introduces a new instruction, `PCONFIG`, for programming the key and encryption mode associated with a particular HKID. These $\langle \texttt{HKID}, key \rangle$ tuples are stored in the Key Encryption Table (KET), which is held by each MKTME encryption engine. The keys in the KET never leave the processor and are never exposed to software. MKTME can be used in both native and virtualized environments. In the virtualized environments, hypervisors control the memory encryption for different VMs by attaching HKIDs to VM's physical addresses in EPT.

MKTME ⇒ TDX. To use MKTME in the virtualized environments, the hypervisor must be trusted to control the memory encryption, which violates the new threat model for confidential computing. Therefore, in TDX, the TDX Module is responsible for controlling memory encryption for TDs. The HKID space has been partitioned to private HKIDs and shared HKIDs. The TDX Module ensures that a unique private HKID is assigned to each TD. Therefore, this HKID can be used to represent the identity of a specific TD. The private HKIDs can only be used for encrypting the private memory of TDs. The TDX Module still leverages MKTME to protect TD's memory. More information about how TDX uses MKTME can be found in Sections 6.2 and 8.

5.3 Intel SGX

Intel introduced SGX [53] in 2015 with the 6th Generation Core processors to protect against memory bus snooping and cold boot attacks. It enables developers to partition their applications and protect selected code and data within enclaves. The memory of an enclave can only be accessed by authorized code. SGX uses hardware-based memory encryption to protect the enclave's contents. Any unauthorized attempts to access or tamper with the enclave's memory can trigger exceptions. SGX adds 18 new instructions into Intel's ISA and enables secure offloading of computations to environments where the underlying host components (such as host application, host kernel, SMM, and peripheral devices) are untrustworthy. SGX's security ultimately depends on the security of the firmware and microcode that implement its features.

The Enclave Page Cache (EPC) is a special memory region that contains the enclave's code and data, where each page is encrypted using the Memory Encryption Engine (MEE). The Enclave Page Cache Map (EPCM) stores the page metadata, such as configuration, permissions, and type of each page. At boot time, keys are generated and used for decrypting the contents of encrypted pages inside the CPU. The keys are controlled by the MEE and never exposed to the outside. Thus, only this particular CPU can decrypt the memory. The CPU stores these keys internally and prevents access to them by any software. Additionally, privileged software out of enclaves is not allowed to read or write the EPC or EPCM pages.

SGX offers both local and remote attestation to verify the integrity and authenticity of enclaves. Local attestation is used to establish trust between two enclaves within the same platform, while remote attestation verifies the trustworthiness of an enclave to a third-party entity off the platform. In local attestation, an enclave can verify another enclave's integrity and the genuineness of the underlying hardware platform. To do so, the first enclave generates a report and uses the identity information of the second enclave to sign it. The second enclave retrieves its Report Key and verifies the report using this Report Key. A third party may want to establish trust with a remotely executed enclave before provisioning it with secrets. In this scenario, remote attestation is necessary. To perform remote attestation, SGX utilizes a special architectural enclave known as the Quoting Enclave (QE). The QE is developed and signed by Intel. The QE receives a report from another enclave, locally verifies it, and transforms it into a remotely verifiable quote by signing it with the Attestation Key. The relying party can send this quote to the Intel Attestation Service (IAS), which verifies the quote to identify and assess the trustworthiness of the SGX enclave. The QE's role is to provide a secure and trustworthy environment for the transformation of a report into a quote and to ensure the quote cannot be modified or falsified. Intel also provides DCAP [58], which is a composition of software packages, for data centers to deploy their own ECDSA attestation infrastructures for SGX enclave attestation.

Researchers have used SGX to provide secure containers (e.g., Scone [4]) and shielded execution for unmodified applications (e.g., Haven [6]). Graphene [63], an SGX-based framework, provides techniques for running unmodified applications as well as dynamic libraries inside SGX enclaves. Besides, SGX has a wide spectrum of applications ranging from the function encryption system (e.g., Iron [16]), source code partitioning to protect security-sensitive data and functions (e.g., Glamdring [51]), machine learning [20, 21, 55, 62], network security [7], fault-tolerant [8], encrypted data search (e.g., HardIDX [17]), secure databases (e.g., EnclaveDB [56]), secure coordination for distributed system (e.g., SecureKeeper [12]), and secure distributed computations (e.g., VC3 [59]).

Identifying vulnerabilities of SGX is another important line of research. Researchers also have identified a wide range of attack vectors targeting SGX, such as controlled-channel attacks [66, 67, 69, 71], cache attacks [11, 18, 54, 60], branch prediction attacks [15, 49], and speculative execution attacks [13, 48].

SGX ⇒ TDX. SGX and TDX protect memory at different granularities. But on the same platform, TDX and SGX are within the same TCB. Thus, they can locally attest to each other. TDX leverages the remote attestation mechanism provided by SGX. The attestation report of a TDX platform can be verified and signed within a QE. More details about TDX's remote attestation can be found in Sections 6.4 and 9.

It is worth noting that at the moment, running an SGX enclave within a TD is not allowed, as invoking `ENCLS` / `ENCLV` instructions within a TD can lead to UD exceptions.

6 OVERVIEW OF TDX

In this section, we give an overview of TDX, discussing its system architecture, memory protection mechanisms, I/O model, attestation, and features that have been planned for the future. Each topic also includes pointers to subsequent sections that provide more technical details.

6.1 TDX System Architecture

Figure 1 illustrates the runtime architecture of TDX. It is composed of two key components: (1) TDX-enabled processors, which offer architectural functionalities like hardware-assisted virtualization, memory encryption/integrity protection, and the ability to certify TEE platforms, (2) TDX Module, an Intel-signed and CPU-attested software module that leverages the features of TDX-enabled processors to facilitate the construction, execution, and termination of TDs while enforcing the security guarantees. The TDX Module provides two sets of interface functions, host-side interface functions for a TDX-enlightened hypervisor and guest-side interface functions for TDs. It is loaded and executed in the SEAM Range, which is a portion of system memory

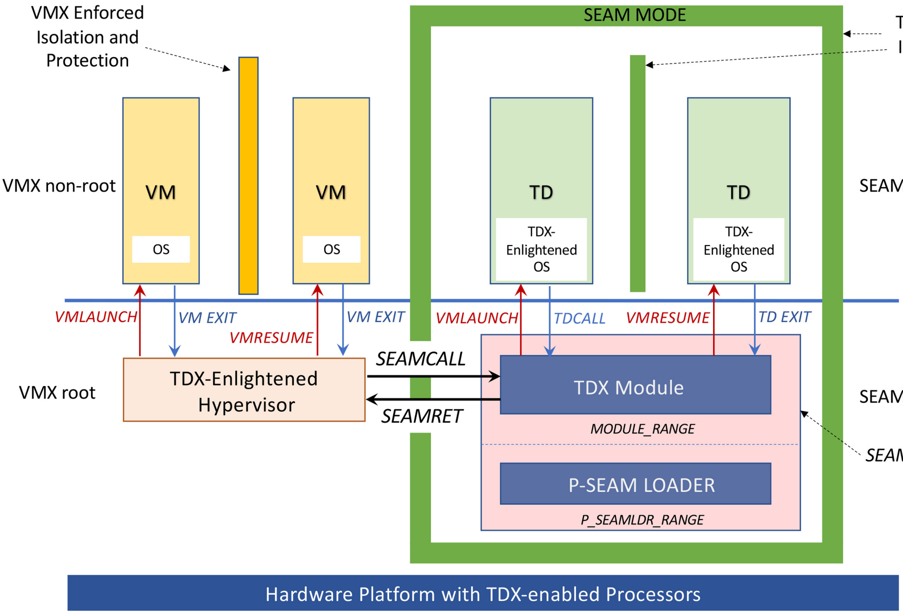

Fig. 1. TDX system architecture.

reserved via UEFI/BIOS. The P-SEAM Loader, which also resides in the SEAM Range, can install and update the TDX Module. More information on the loading process of the TDX Module can be found in Section 7.1.

Secure-Arbitration Mode (SEAM) is an extension of the VMX architecture and provides two new execution modes: SEAM VMX root mode and SEAM VMX non-root mode. A TDX-enlightened hypervisor operates in the traditional VMX root mode and utilizes the `SEAMCALL` instruction to call host-side interface functions (function names start with `TDH`) of the TDX Module. Upon execution of the `SEAMCALL` instruction, the logical processor (LP) transitions from the VMX root mode into SEAM VMX root mode and starts executing code within the TDX Module. Once the TDX Module has completed its task, it returns to the hypervisor in VMX root mode by executing the `SEAMRET` instruction.

On the other hand, TDs run in the SEAM VMX non-root mode. TDX supports the execution of unmodified user-level applications within a TD, much like in a standard VM. However, the guest OS kernel, illustrated as the TDX-enlightened OS in Figure 1, must undergo modifications to align with the underlying TDX platform, accommodating both the architectural paradigms and the security imperatives of TDX. These modifications include managing new TDX exceptions via an in-guest Virtualization Exception (VE) handler, implementing a hypercall-like mechanism for communication between a TD and the TDX Module, transitioning memory pages from private to shared for I/O operations, and integrating attestation support. The specific implementation details may vary depending on the OS type. For instance, the detailed implementation of the enlightened guest Linux kernel has been described in the kernel documentation [14]. TDs can trap into the TDX Module either through a TD exit or by invoking the `TDCALL` instruction. In both cases, the LP transitions from the SEAM VMX non-root mode into the SEAM VMX root mode and starts executing in the context of the TDX Module. The names of guest-side interface functions handling `TDCALLs` start with `TDG`. Details about the TDX context switches can be found in Section 7.5.

The confidentiality assurances offered by confidential computing render it a prime target for research on side-channel information leakage. The unveiling of a succession of micro-architectural attacks [5, 47, 52, 65, 68] exploiting the speculative execution of CPUs highlights a concerning issue: the isolation of security domains enforced in the architectural states may not be consistent as in the micro-architectural states. As TDX becomes more widely available in the market, it is expected to attract increased attention from security researchers. Our primary emphasis lies in examining the existing defenses integrated into the TDX Module to address known attack vectors. For detailed information, please refer to Section 7.8.

6.2 TDX Memory Protection

TDX leverages VMX to enforce memory isolation for TDs. Similar to legacy VMs, TDs are unable to access the memory of other security domains, such as SMM, hypervisors, the TDX Module, and other VMs/TDs. With VMX, hypervisors maintain EPTs to enforce memory isolation. However, since hypervisors are no longer trusted, TDX has moved the tasks of memory management to the TDX Module, which controls the address translation of TD's private memory.

A more intriguing aspect of TDX's security model is its protection of TD's memory from privileged software, corrupted devices, and unprincipled administrators on the host. TDX achieves this by implementing *access control* and *cryptographic isolation*. Access control prevents other security domains on the same computer from accessing a TD's data. Cryptographic isolation is utilized to prevent malicious DMA devices or adversaries with physical access to the main memory from directly reading or corrupting TD's private memory.

Memory Partitioning. With TDX enabled, the entire physical memory space is partitioned into two parts: NORMAL MEMORY and SECURE MEMORY. The sensitive data of TDs, including the private memory, virtual CPU state, and its associated metadata, should be stored in SECURE MEMORY. TDs can also specify memory regions as SHARED MEMORY for I/O, which is not protected through TDX. Thus, these memory regions belong to NORMAL MEMORY. All other software, which is not executing in the SEAM mode, belongs to NORMAL MEMORY and is not allowed to access SECURE MEMORY, regardless of its privilege level. The memory controller, an architectural component inside the processor, enforces memory access checks.

To make a physical page part of the SECURE MEMORY, the TD OWNER BIT is enabled (Section 8.2). Each TD OWNER BIT is associated with a memory segment corresponding to a cache line.[1] The TD OWNER BITS are stored in the Error Correction Code (ECC) memory associated with these segments. The TDX Module controls the conversion of physical memory pages to SECURE MEMORY by attaching PRIVATE HKIDs to their physical addresses. The HKID is encoded in the upper bits of the physical address. The set of PRIVATE HKIDs is controlled by TDX and can only be used for TDs and the TDX Module. When the memory controller writes to a physical address with a PRIVATE HKID, it sets the TD OWNER BIT to 1. When it writes to an address that does not have a PRIVATE HKID, it clears the TD OWNER BIT. Access control is enforced on each cache line read. The read request passes through the memory controller, which permits only processes executing in SEAM mode to read a cache line with a TD OWNER BIT set to 1. Any read request not in the SEAM mode receives all zeros when trying to read such a cache line.

When building a TD, the (untrusted) hypervisor selects the memory pages from the NORMAL MEMORY to become part of the SECURE MEMORY. The TDX Module gradually moves these pages to the SECURE MEMORY. It uses them for the metadata (Section 7.4) and the main memory of each TD. A TD must explicitly accept these pages before they can be used for its main memory. The TDX

[1] At the time of this writing, the size of the processor's cache line is 64 bytes; thus the address of such a memory segment is 64 B-aligned.

Module performs sanity checks of the SECURE MEMORY setup by maintaining a PHYSICAL ADDRESS METADATA TABLE (PAMT), which is described in more detail in Section 7.7.

Memory Confidentiality. TDX leverages MKTME (Section 5.2) for encrypting TD's private memory and its metadata. MKTME is responsible for transparent memory encryption and decryption of data passing through the memory controller. The TDX Module programs the keys used by the MKTME to encrypt specific cache lines when they are written to memory. The keys are associated with the HKIDs embedded in the physical addresses. MKTME decodes HKIDs and uses the referenced cryptographic keys to perform the cryptographic operations.

MKTME stores cryptographic keys in its internal memory, never exposing them to the outside. The cryptographic keys can only be referenced by their HKIDs. When building a new TD, the hypervisor selects an unused PRIVATE HKID, and the TDX Module requests the processor to generate a new cryptographic key related to this HKID. The TDX Module binds this $\langle \texttt{HKID}, key \rangle$ tuple to the TD. It guarantees that the memory of each TD is encrypted with a different cryptographic key.

MKTME encrypts memory at the cache line granularity using AES-128 XTS cryptography when the cache line is being written back to main memory. The encryption can prevent some physical attacks, like the cold boot attack. Please see Section 8.1 for more details on MKTME and HKIDs.

Memory Integrity. TDX provides two distinct mechanisms for ensuring memory integrity: LOGICAL INTEGRITY (LI) and CRYPTOGRAPHIC INTEGRITY (CI).

LI protects the integrity against unauthorized writes at the software level by using the TD OWNER BIT. Since the TDX only allows the use of PRIVATE HKIDs in the SEAM mode, any unauthorized writes to a TD's private memory from outside the SEAM mode will clear the TD OWNER BIT. When the modified private memory is read, the cleared TD OWNER BIT will trigger an exception. However, this feature cannot prevent adversaries from bit flipping (e.g., via a ROWHAMMER attack [45]) the main memory.

CI is a more advanced mechanism that addresses the limitations of LI. In addition to the TD OWNER BIT, CI also computes a Message Authentication Code (MAC) on a cache line when it is being written back to memory. The MAC is computed using a 128-bit MAC key generated during system initialization and is stored as part of the memory metadata during the write-back. When the memory is read, the MAC is recalculated. Any tampering with the memory content will be detected by CI if the TD OWNER BIT or the recalculated MAC mismatch with the stored metadata. However, neither LI nor CI can detect the memory replay attack if the adversary can roll back both the memory content and the metadata. We provide a more detailed technical discussion of the memory integrity protection in Section 8.2.

6.3 TDX I/O Model

According to the TDX threat model, hypervisors and peripheral devices are considered untrusted and are prohibited from directly accessing the private memory of TDs. It is the responsibility of TDs and their owners to secure I/O data before it leaves the trust boundary. This requires sealing the I/O data buffers and placing them in shared memory, which is identified by the SHARED bit in the GPA. Hypervisors or peripheral devices can then move the data in and out of the shared memory. This necessitates modifications to the guest kernel to support this I/O model. Furthermore, all I/O data that is transferred into the TDs from hypervisors or peripheral devices must be thoroughly examined and validated, as it is no longer considered trustworthy.

In the Linux guest support for TDX, all MMIO regions and DMA buffers have been mapped as SHARED MEMORY within the TDs. The Linux guest is enforced to use SWIOTLB to allocate and convert DMA buffers in unified locations. To protect against malicious inputs from I/O, only a limited number of hardened drivers [30] are allowed within TDs.

6.4 TDX Attestation

Remote attestation is a method for verifying the identity and trustworthiness of a TEE. The attester can provide proof to a challenger to show that computations are being executed within protected domains. The challenger validates the evidence by checking the digital signatures and comparing the measurements to reference values.

On a TDX-enabled machine, the attester operates within a TD and is responsible for handling remote attestation requests. When a request is received from a challenger, such as a tenant, the attester provides evidence of proper instantiation of the TD through the generation of a TD QUOTE. This QUOTE, which serves as the evidence, is produced by the TDX module and signed by the QUOTING ENCLAVE. It contains measurements of the TDX's TCB and the software components loaded in the TD. The QUOTE also includes a certificate chain anchored by a certificate issued by Intel. Upon receipt of the QUOTE, the challenger verifies its authenticity by checking the QUOTE and determining if the attester is running on a genuine TDX-enabled platform and if the TD has the expected software measurements. If the QUOTE is successfully validated, the challenger can proceed to establish a secure channel with the attester or release secrets to the attester. We provide a more detailed technical discussion of remote attestation in Section 9.

6.5 Future Features

Live migration and *trusted I/O* are crucial features for confidential VMs but are currently not supported in TDX 1.0. However, according to documents [35, 36, 38], Intel is planning to include the support for live migration in TDX 1.5 and trusted I/O in TDX 2.0. These plans are still in progress and may be subject to change in the future. Here we provide a brief overview of these two features and explain their design.

Live Migration. Live migration is an essential feature for cloud service providers as it enables them to transfer running VMs from one physical host to another without any service interruptions. This functionality is important for maintenance tasks such as hardware upgrades, software patches, and load balancing. However, migrating a TD is more complex than migrating a traditional VM due to the security concerns of confidential computing. Since the hypervisor is considered untrusted, it is not allowed to directly access and transfer the CPU state and private memory of the TD from the source to the destination platform. Furthermore, tenants should have the ability to define and enforce migration policies. For instance, if the destination platform does not meet the TCB requirements specified in the policy, the migration should be canceled.

Intel introduces Service TDs to expand the trust boundary of the TDX Module. Rather than making the TDX Module overly complex and bloated, it is more convenient and flexible to add customized and specialized functionalities into a Service TD. A Service TD can be bound to regular TDs via the TDX Module with access privileges to their assets.

MIGRATION TD (MIGTD) is a Service TD that is specifically designed for live migration. The entire live migration session is under the control of the TDX Module and the MIGTDs. The untrusted hypervisor, which is controlled by the cloud service provider, is only responsible for transferring the encrypted TD's assets over networks. These assets include the TD's metadata, CPU state, and private memory, and are protected by a MIGRATION SESSION KEY (MSK) that is only accessible by the MIGTDs and TDX Module.

Both the source and destination platforms have a running MIGTD. MIGTDs are respectively bound to the source TD (to be migrated) and the destination TD (initially as a TD template waiting for migration). The MIGTDs are responsible for remote attestation between source and destination platforms and evaluate their TCB levels based on security policies. Once the platforms are deemed acceptable for migration, a secure channel is established between the two MIGTDs. The source

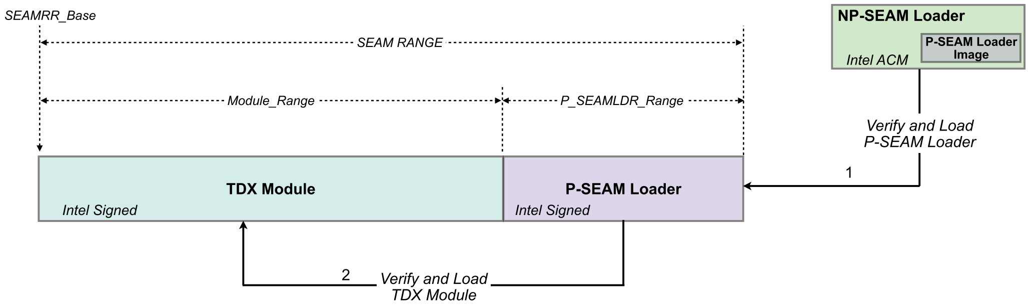

Fig. 2. Loading the TDX module.

MigTD generates an MSK, which is shared to the destination MigTD through this secure channel. Both MigTDs program the MSK into the corresponding TDX Modules. The source TDX Module exports and encrypts the TD's assets with the MSK, while the destination TDX Module decrypts the assets with the same key and imports them into the destination TD. It is worth noting that the source and destination TDs have their HKIDs assigned independently, thus protected with different TD private keys.

Trusted I/O. A computer consists of various functional components. However, confidential computing has conceptually shattered the unified trust model. As a result, each component, made by different vendors, can no longer trust each other. This creates a serious impediment to efficient I/O, as untrusted devices cannot read and write data in the private memory of TEEs. To address this issue, Intel has proposed TDX Connect in TDX 2.0, aiming to extend the trust from a TD to external devices. This requires changes to the devices and the TDX platform to use a compatible protocol to establish mutual trust and enable secure communication channels. The key principle is that a TD and a device should be able to securely exchange and verify their identities and measurements. Additionally, the data paths between a TD and a device are not trusted and may be vulnerable to interception by attackers. Therefore, an end-to-end secure channel is necessary to protect the data transmitted between a TD and a device. The detailed protocols for TDX Connect can be found in the proposals [35, 38].

7 TDX MODULE

This section provides an in-depth analysis of the TDX Module. We first discuss its loading process in Section 7.1, followed by an explanation of the physical and linear memory layout in Section 7.2. We then describe the metadata created by the TDX Module to manage TDs in Section 7.4, and the process of context switching across security domains in Section 7.5. Additionally, we provide details about the Keyhole structure (Section 7.6) and memory management (Section 7.7) of the TDX Module.

7.1 Loading TDX Module

Figure 2 illustrates the two-stage process of loading the TDX Module. The process begins with the loading of the Intel Non-Persistent SEAM Loader (NP-SEAM Loader), which is an Intel Authenticated Code Module (ACM). ACMs are Intel-signed modules that run within the internal RAM of the processor. The NP-SEAM Loader is authenticated and loaded by the Intel Trusted Execution Technology (TXT) [19] through the `GETSEC[ENTERACCS]` function. The NP-SEAM Loader contains the image of the Intel Persistent SEAM (P-SEAM) Loader, which is then verified and loaded by the NP-SEAM Loader. The P-SEAM Loader is then responsible for installing or updating the TDX Module.

It is important to note that both the P-SEAM Loader and the TDX Module are loaded in the SEAM Range, which is a portion of system memory reserved via UEFI/BIOS. The range's base address and size are specified by the `IA32_SEAMRR_PHYS_BASE` and `IA32_SEAMRR_PHYS_MASK` MSRs. This range is partitioned into Module_Range for the TDX Module and P_Seamldr_Range for the P-SEAM Loader. Both modules run in the SEAM VMX root mode and use `SEAMCALL` / `SEAMRET` to interact with external software. The NP-SEAM Loader, P-SEAM Loader, and TDX Module are all provided and signed by Intel, establishing a chain of trust to bootstrap the TDX Module.

The P-SEAM Loader provides a `SEAMCALL` interface function `seamldr_install` for loading the TDX Module. The TDX Module's image is pre-loaded into a memory buffer (not in the SEAM Range). The physical addresses of the buffer and a `seam_sigstruct` (signature of the TDX Module) are passed as the parameters to the `seamldr_install`. The `seam_sigstruct` contains the hash value and the Security Version Number (SVN) of the TDX Module, the number of per-LP stack pages, the number of per-Logical Processor (LP) data pages, and the number of global data pages. These numbers are used by `seamldr_install` to determine the physical/linear addresses and the sizes of the TDX Module's various memory regions.

The `seamldr_install` must be called on all LPs serially. When it is called on the first LP, an installation session starts. On each LP, `seamldr_install` checks that the LP is not already in an installation session (started by another LP), and clears the LP's VMCS cache. When `seamldr_install` is called on the last LP, it does the following:

(1) checking the parameters to the `seamldr_install`,
(2) verifying the signature of the TDX Module,
(3) checking the SVN of the to-be-loaded image and comparing with the resident TDX Module,
(4) determining the physical and linear addresses and sizes of the TDX Module's various memory regions in the SEAM Range: code, data, stack, page table, Sysinfo_Table, Keyhole, and Keyhole-Edit (Section 7.2),
(5) mapping the regions' physical addresses to their linear addresses (Section 7.2),
(6) loading the TDX Module's binary image into the SEAM Range, measuring the image, computing and verifying the TDX Module's hash value,
(7) setting up the TDX Module's Sysinfo_Table,
(8) setting up SEAM Transfer VMCS on each LP (Section 7.5),
(9) recording the TDX Module's hash, SVN, in the P-SEAM Loader's data region.

In addition to the `SEAMCALL` to install the TDX Module, the P-SEAM Loader also provides other interface functions to shut down itself and retrieve the loader's system information.

7.2 Memory Layout of TDX Module

Here we discuss the physical and linear memory layout for the TDX Module, respectively.

Physical Memory Layout. Figure 3 depicts the physical memory layout of the TDX Module within the Module_Range. The layout starts with a 4 KB page that holds the Sysinfo_Table of the TDX Module. The Sysinfo_Table consists of 2 KB platform information populated by Mcheck from the NP-SEAM Loader and the next 2 KB populated by the P-SEAM Loader with the TDX Module's information, such as the SEAM Range base address and size, the base linear addresses of the memory regions, number of LPs, and range of private HKIDs. After the Sysinfo_Table, there is the per-LP VMCS region. Each LP has a 4 KB SEAM Transfer VMCS (see Section 7.5). Following the per-LP VMCS region, there is the data region, which is partitioned into per-LP

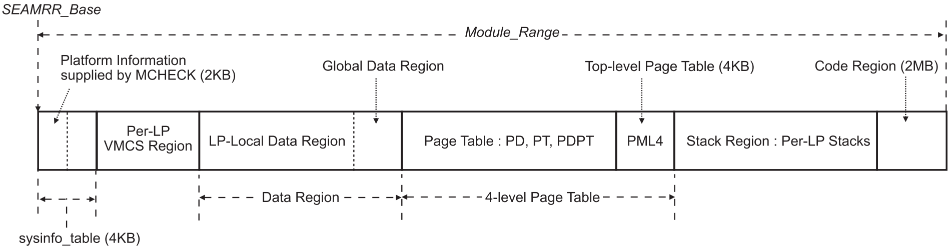

Fig. 3. TDX module physical memory layout.

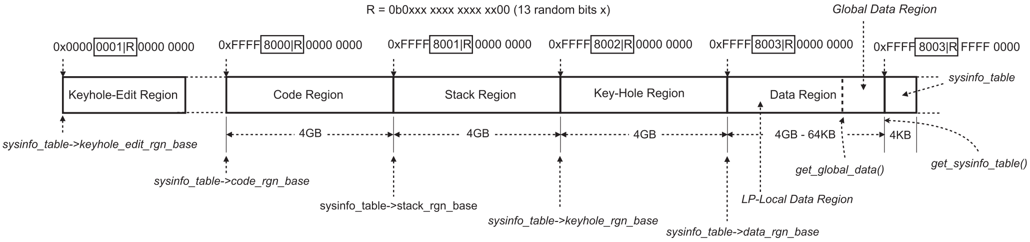

Fig. 4. TDX module linear memory layout.

data region and a global data region. Next, there is the TDX Module's 4-level page table, followed by the per-LP stack regions, and finally, the code region for the TDX Module's executable code.

Linear Memory Layout. The TDX Module has its own linear address space and maintains a page table to translate addresses. Figure 4 illustrates the layout of the TDX Module's linear address space, which is established by the P-SEAM Loader through the construction of the TDX Module's page table. To prevent memory corruption attacks, the P-SEAM Loader randomizes bits 34 to 46 of the linear addresses, which are represented by the boxes in Figure 4. The linear addresses and the sizes of all regions are recorded in the fields of the Sysinfo_Table. The Page Table Entries (PTEs) for code, stack, data, and Sysinfo_Table can be statically populated in advance and require no changes to the page table at runtime. However, the Keyhole region serves to map data passed from external software dynamically during the execution of the TDX Module. This requires the addition of the Keyhole-Edit region to allow runtime editing of the PTEs for the Keyhole's mapping. A detailed discussion of the Keyhole and Keyhole-Edit regions can be found in Section 7.6.

7.3 Initialization and Configuration of TDX Module

After the TDX Module is loaded, the host kernel is responsible for initializing and configuring the TDX Module. The host kernel makes a `SEAMCALL[TDH.SYS.INIT]` to globally initialize the TDX Module. Then, the host kernel makes a `SEAMCALL[TDH.SYS.LP.INIT]` on each LP to check and initialize per-LP parameters, such as Keyholes (Section 7.6), data regions, and stack regions (Section 7.2). Next, the host kernel allocates a global private HKID and passes it to the TDX Module through a `SEAMCALL[TDH.SYS.CONFIG]`, which also initializes the Trust Domain Memory Region (TDMR) (Section 7.7). The `SEAMCALL[TDH.SYS.KEY.CONFIG]` on each processor package generates a TDX global private key and binds the key with this HKID. This key is used to encrypt memory that holds PAMT and Trust Domain Root (TDR) (Section 7.4) of each TD. Finally, the host kernel calls `SEAMCALL[TDH.SYS.TDMR.INIT]` multiple times to gradually initialize the PAMT (Section 7.7) for each TDMR.

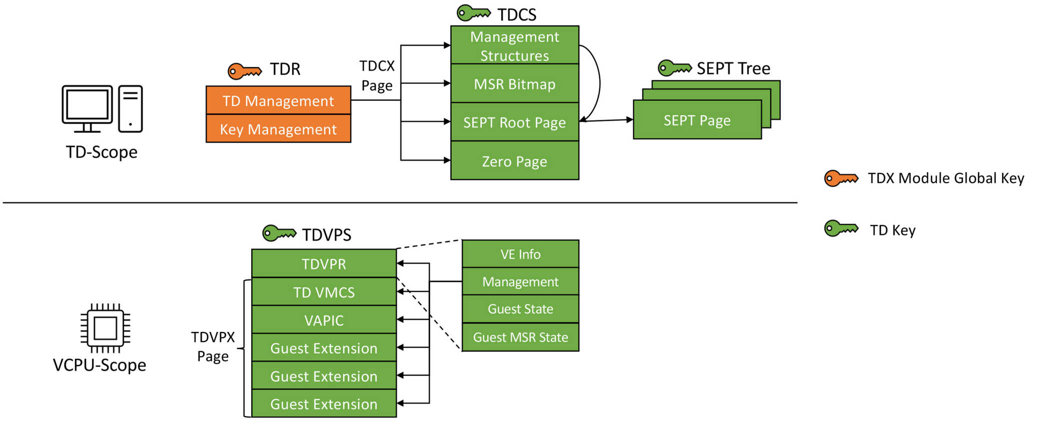

Fig. 5. Relationship of TD's metadata.

7.4 Metadata for TDs

The TDX Module is responsible for managing the entire life cycle of TDs. As such, it needs to maintain metadata for each TD instance. The TDX Module ensures that the memory encryption is applied to the metadata to prevent the hypervisor from accessing or modifying it.

Each TD's metadata consists of the following control structures: TDR, Trust Domain Control Structure (TDCS), Trust Domain Virtual Processor State (TDVPS), and Secure EPT (SEPT). Figure 5 illustrates the relationships between these control structures.

TDR. TDR is the initial structure that is created at the inception of a TD and is destroyed when the TD is terminated. During the entire life cycle of the TD, `SEAMCALLs` use the physical address of the corresponding TDR to refer to the TD. The TDR comprises the key information for memory encryption and references to the TDCX pages (physical memory pages for the TDCS). As the TDR is created before the TD's private key is generated, it is protected with the global private key of the TDX Module. The subsequent metadata (TDCS, TDVPS, and SEPT), along with the TD's memory pages, can be associated with the TDR through the owner attribute in the PAMT (Section 7.7).

TDCS. TDCS is a control structure that manages the operations and stores the state at the scope of a TD. It consists of four continuous TDCX memory pages, each allocated for a specific purpose, such as TD's management structures, MSR bitmaps, SEPT root page, and a special zero page. TDCS is encrypted with the TD's private key, which is generated when the TDR is created.

TDVPS. TDVPS is a control structure for each virtual CPU of a TD. It consists of six memory pages, starting from a TDVPR page that contains references to multiple TDVPX pages. The first TDVPR page holds the fields for VE information, virtual CPU management, guest state, and guest MSR state. The second page is for the TD Transfer VMCS (Section 7.5), which controls the TD's entry and exit. The third page is a Virtual APIC (VAPIC) page, followed by three pages for guest extension information. Like the TDCS, the TDVPS is also protected by the TD's private key.

SEPT. For legacy VMs, hypervisors manage address translations from GPA to HPA using EPT. However, in TDX, guest address translations must be protected from untrusted hypervisors. To achieve this, TDX has two types of EPT: SEPT and Shared EPT. SEPT is used to translate addresses of a TD's private memory and is protected by the TD's private key. The reference to the SEPT and the SEPT root page are stored in the TDCS. Shared EPT, on the other hand, is used to translate addresses for memory explicitly shared by the TD with a hypervisor, such as in the case of virtualized I/O. It remains under the control of the hypervisor. The guest kernel in the TD can

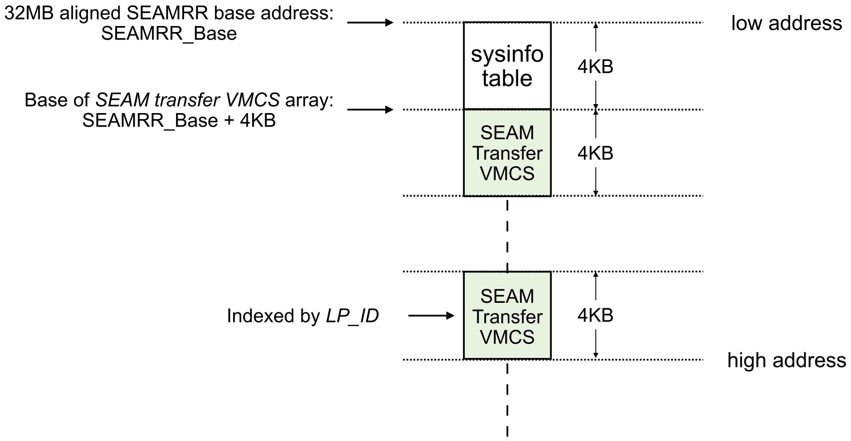

Fig. 6. Per-LP SEAM TRANSFER VMCS layout.

determine which memory pages to share by setting the SHARED bit in the GPA. SHARED MEMORY pages are not encrypted with the TD's private key.

7.5 Context Switches

There are two types of context switches for TDX: the first occurs between the hypervisor and the TDX Module, while the second occurs between TDs and the TDX Module. We delve into each of these in more detail.

Hypervisor ↔ TDX Module. In TDX, a hypervisor is prohibited from directly managing TDs. Instead, it must interact with the TDX Module through `SEAMCALL` interface functions. When a `SEAMCALL` is made, the processor transitions from the VMX ROOT MODE to the SEAM VMX ROOT MODE. The TDX Module's SEAM TRANSFER VMCS is loaded. This SEAM TRANSFER VMCS is set up by the P-SEAM Loader for each LP and is stored within the MODULE_RANGE.

The repurposing of VMCS for context switches between the hypervisor and the TDX Module may seem confusing initially, as the hypervisor is not a "guest VM" and the TDX Module is not a "host hypervisor." We can disregard the guest/host concept and only view the SEAM TRANSFER VMCS as a means of switching the execution context between the hypervisor and the TDX Module.

Figure 6 depicts the location and layout of the SEAM TRANSFER VMCS regions in the MODULE_RANGE, which begins at the `SEAMRR_Base`. The first 4 KB page in the MODULE_RANGE is the SYSINFO_TABLE. Starting from `SEAMRR_Base + 4KB`, there is an array of per-LP SEAM TRANSFER VMCS regions. Each region is a 4 KB page. This array is indexed by the identifier of the LP, denoted by `LP_ID`.

When a `SEAMCALL` instruction is executed, the processor searches for the VMCS address based on the current `LP_ID`. The address is determined by: `SEAMRR_Base + 4KB + (LP_ID × 4KB)`. The TDX Module's state is stored in the HOST STATE AREA of the VMCS. For instance, the host `RIP` is set to the `tdx_seamcall_entry_point` of the TDX Module, and the host `CR3` is set to the physical address of the TDX Module's `PML4` base. Moreover, the host `FS_BASE` is set to the linear address of the SYSINFO_TABLE and the host `GS_BASE` is set to the per-LP data region.

When the LP transitions into the TDX Module through the `SEAMCALL` instruction, the information stored in the SEAM TRANSFER VMCS is loaded onto the processor. Therefore, the `FS_BASE` and `GS_BASE` now point to the Sysinfo_Table and the local data region of the TDX Module, respectively. The `CR3` register points to the TDX Module's page table, thereby switching Memory Management Unit (MMU) to operate in the TDX Module's linear address space. The TDX Module starts to handle `SEAMCALLs` and dispatch them to corresponding interface functions.

TD ↔ TDX Module. In traditional virtualization, the hypervisor handles VM exits, which are controlled by the VM's Transfer VMCS. Each VMCS is associated with one virtual CPU and stores the virtual CPU state for recovering the guest execution in the next VM resume. However, this operation leaks the virtual CPU state as VMCS is visible to hypervisors. In TDX, synchronous `TDCALLs` or asynchronous TD exits are designed to trap into the TDX Module. This is controlled by the TD Transfer VMCS, which is set up when a virtual CPU of a TD is created and stored in the TD's TDVPS. The TDVPS is encrypted with the TD's private key (Section 7.4). Therefore, the TD Transfer VMCS is inaccessible to untrusted hypervisors. When a TD calls a `TDCALL` or triggers a TD exit, the LP loads the state of the TDX Module stored in the TD Transfer VMCS to switch context.

In TDX, certain TD exits cannot be fully handled by the TDX Module and instead require a hypervisor to emulate certain operations, such as port I/O, HLT, CPUID, and more. However, traditional hypervisors have access to the entire virtual CPU states and memory, exposing more information than necessary to handle these exits. TDX addresses this issue by introducing a new mechanism for handling TD exits. All TD exits first trap into the TDX Module, which injects a VE into the TD to handle the exit. The TD's guest kernel includes a corresponding VE handler that prepares a minimized set of parameters and invokes a `TDCALL` to re-enter the TDX Module. At this point, the TDX Module can safely ask the hypervisor to handle the requests with minimal exposure to sensitive information.

7.6 Keyholes

All memory buffers passed through `SEAMCALLs` use their physical addresses as references. The TDX Module must map these buffers into its own linear address space to access them. This mapping process is facilitated by the Keyhole and Keyhole-Edit regions, which serve as temporary "leases" of linear addresses.

The Keyhole region is a reserved linear address range specifically for address mapping. The region is comprised of an array of Keyholes. This array is further divided into 128-Keyhole segments, with each segment assigned to one LP. The TDX Module organizes free Keyholes in an LRU list when setting up per-LP data structures. Each Keyhole corresponds to a 4 KB-aligned linear address and links to a physical memory page. Since multiple memory buffers can exist within the same memory page, each Keyhole maintains a reference count to track the number of referenced buffers on the page.

When the TDX Module is installed by the P-SEAM Loader, all the linear addresses of Keyholes are mapped to an empty physical address. This is achieved by setting all the leaf-level PTEs for the Keyhole region in the TDX Module's page table to zero. Simultaneously, the physical addresses of the corresponding PTEs for the Keyholes are mapped to the Keyhole-Edit region. This enables the TDX Module to locate and modify the Keyhole's address mappings in its page table during runtime.

When processing a `SEAMCALL` that refers to an external memory buffer with a physical address, the TDX Module checks if the buffer's memory page is already mapped by a Keyhole. If so, it increments the Keyhole's reference count and returns the mapped linear address. If not, it selects a free Keyhole from the LRU list and maps the linear address of this Keyhole to the page table by updating the corresponding PTE referenced in the Keyhole-Edit region. Once the buffer is mapped, the TDX Module can access it using the Keyhole's linear address. At the end of each `SEAMCALL`, the reference counts of corresponding Keyholes decrement, and any non-referenced Keyholes return to the LRU list.

7.7 Physical Memory Management

The TDX Module manages physical memory by using a set ofTDMRs and their control structures, PAMTs. TDMRs are constructed by the hypervisor based on a list of Convertible Memory Regions (CMRs), which are the memory regions that can be used for TD's private memory or metadata. These regions are subject to MKTME encryption and TDX memory integrity protection. This list of CMRs is prepared by the UEFI/BIOS.

Each TDMR is a single range of physical memory that is 1 GB-aligned and has a size that is an integral multiple of 1 GB, but does not necessarily need to be a power of two. Two TDMRs cannot overlap. A TDMR may contain reserved areas that cannot be used by the TDX Module. A reserved area is an array of 4 KB-aligned memory pages (each page is 4 KB). Memory in a TDMR, except for the reserved areas, must be convertible. It should be noted that TDMR configuration is managed by software without using hardware range registers.

The TDX Module uses PAMT to track page attributes of each physical memory page in a TDMR. The attributes contain the information about the page owner, page type, and page size. The page attributes allow the TDX Module to ensure that a physical memory page in a TDMR has a proper type and is only assigned to at most one TD. When a page is assigned to a TD's private memory, the TDX Module can check whether the page size in the SEPT and PAMT are consistent.

A PAMT is divided into blocks, where each block tracks page addresses within the 1 GB size range. Each block has three levels to track metadata for pages with sizes 4 KB, 2 MB, and 1 GB, respectively. The first level tracks a single 1 GB page, the second level tracks 512 2 MB pages, and the third level tracks 512×512 4 KB pages. Given a physical address, the TDX Module can perform a PAMT hierarchical walk to retrieve its page attributes for a sanity check.

The TDX Module manages the data structure by updating the attributes of each page it uses during runtime. Any operation that requires accessing, removing, or adding a page causes the TDX Module to walk through PAMT to adjust the page attributes and check corresponding access rights. The memory for PAMT is allocated by the hypervisor and is encrypted with the TDX Module's global private key.

7.8 Side Channel Mitigation

Some of the known CPU vulnerabilities have been addressed in hardware fixes. During the initialization of the TDX Module, it reads the `IA32_ARCH_CAPABILITIES` MSR and verifies a set of capability bits, including `rdcl_no`, `irbs_all`, `mds_no`, `if_pschange_mc_no`, `taa_no`, `misc_package_ctls`, `skip_l1dfl_vmentry`, `energy_filtering_ctl`, and `tsx_ctrl`. Each bit corresponds to a specific vulnerability and indicates whether the processor is susceptible to that particular attack. For example, `rdcl_no` indicates that the processor has been patched against the Rogue Data Cache Load (RDCL) [2] vulnerability. The detailed list can be found in Intel's documentation [34]. Any missing capability will lead to the failure of initializing the TDX Module.

Furthermore, to counteract the Bounds Check Bypass (BCB) [1] vulnerability, a software-level mitigation strategy is deployed, employing memory barriers such as `LFENCE` to halt speculation at specific locations within the TDX Module where untrusted inputs might be encountered. These critical locations comprise the TD exit entry point, `SEAMCALL` entry point, Keyhole manager, among others. However, this approach remains manual and ad-hoc in nature. A more principled approach is needed to determine the essential placement of memory barriers.

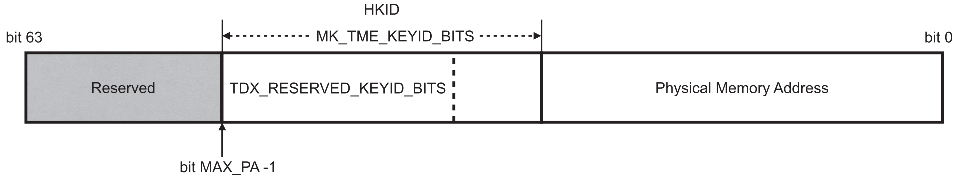

Fig. 7. HKID layout in physical memory address.

8 MEMORY PROTECTION

A TD's memory is divided into PRIVATE MEMORY and SHARED MEMORY. The PRIVATE MEMORY is only accessible by the TD and the TDX Module. The SHARED MEMORY is also accessible by the hypervisor and is used for operations that require cooperation from the hypervisor, such as networking, I/O, and DMA. TDX protects the confidentiality and integrity of a TD's private memory.

8.1 HKID Space Partitioning

The HKID space is partitioned once during the boot process into two ranges, PRIVATE HKIDs and SHARED HKIDs. Only software in the SEAM mode, namely the TDX Module and TDs, can read and write memory whose contents are encrypted by keys associated with PRIVATE HKIDs. Keys associated with SHARED HKIDs can be used to encrypt memory outside the SEAM mode, such as the memory of legacy VMs and the host kernel.

When the hypervisor requests the TDX Module to establish a TD, it allocates a PRIVATE HKID for the TD. The TDX Module, using the `PCONFIG` instruction, asks MKTME to generate a unique random key for the HKID. This key is called the TD's EPHEMERAL PRIVATE KEY. It is used to encrypt all the private memory and metadata of the TD and is never exposed outside MKTME. This $\langle \texttt{HKID}, key \rangle$ binding is valid for the lifetime of the TD.

A physical memory page associated with a HKID stores the HKID in the upper bits of the page's physical address, as shown in Figure 7. At boot time, the number of bits used for HKIDs (`MK_TME_KEYID_BITS`) and the number of bits used for PRIVATE HKIDs (`TDX_RESERVED_KEYID_BITS`) are set in the `IA32_TME_ACTIVATE` MSR. The `IA32_MKTME_KEYID_PARTITIONING` MSR can be used for reading the numbers of PRIVATE and SHARED HKIDs. Intel reserves a range of upper bits in the 64-bit physical address. The HKID uses these reserved bits. The remaining bits following the HKID correspond to the physical memory address. The upper bits of the HKID field, the `TDX_RESERVED_KEYID_BITS` are reserved for PRIVATE HKIDs. For example, if `MK_TME_KEYID_BITS` is 6 and `TDX_RESERVED_KEYID_BITS` is 4, then HKIDs from 0 to 3 are SHARED, and HKIDs from 4 to 63 are PRIVATE.

The hypervisor and the TDX Module configure the memory encryption by setting the HKID in the upper bits of the physical address of a memory page. The hypervisor can only use SHARED HKIDs, while the TDX Module can use both SHARED and PRIVATE HKIDs. An exception will be raised if any software executing outside SEAM mode tries to access memory through a physical address with a PRIVATE HKID.

8.2 TD Memory Integrity Protection

TDX always protects the integrity of the TD's private memory content. This protection is required because an entity outside the SEAM mode,e.g., a malicious hypervisor or a DMA device, can write to the TD's private memory. TDX cannot prevent such modification, but it can detect and flag it. It prevents a TD or the TDX Module from reading the tampered content. To detect such tampering, TDX supports two memory integrity modes that can be configured on a system:

(1) Logical Integrity (Li): memory integrity is protected by a TD Owner Bit.
(2) Cryptographic Integrity (Ci): memory integrity is protected by a MAC and a TD Owner Bit.

Both Li and Ci apply to a physical memory segment with the size of a cache line and whose address is cache line aligned. Ci can detect modifications made by direct physical access to the memory or bit flips, such as the Rowhammer attack [45], which Li cannot detect.

In addition to Li and Ci, if a program outside the SEAM mode reads the private memory of a TD or the TDX Module, the read will always return zeros. This is to prevent ciphertext cryptanalysis and side channels in which a program outside the SEAM mode could determine whether a program in the SEAM mode changes the memory content.

If a TD or the TDX Module writes to a memory segment belonging to a TD's private memory, the corresponding TD Owner Bit is set to 1. Due to the way a TD's memory is set up, all TD Owner Bits of a TD's private memory should be set to 1. However, if an entity outside the SEAM mode writes to a segment belonging to the private memory, the corresponding TD Owner Bit is cleared to 0. Later, when the TD or the TDX Module reads the segment, the segment is marked as poisoned. If the reader is the TD, this poisoned marking causes a TD exit for the TD. The TDX Module can capture this TD exit and put the TD into a fatal state, which prevents any further entry into the TD and leads to the tearing down of the TD. If the TDX Module reads the poisoned content, the TDX Module and the TDX's hardware extension in the processor are marked as disabled. Any further `SEAMCALLs` leads to the `VMFailInvalid` error.

If Ci is enabled, the processor generates a 128-bit MAC key during system initialization. On each write, TDX uses this key to calculate and store a 28-bit MAC in the ECC memory corresponding to the cache line. On each read, the memory controller recalculates the MAC and compares it with the value read from the ECC memory. The mismatch indicates integrity or authenticity violation and results in the cache line being marked as poisoned. The MAC is calculated over (1) the ciphertext (encrypted content of the cache line), (2) the tweak values used for AES-XTS encryption, (3) the TD Owner Bit, and (4) the 128-bit MAC key.

9 REMOTE ATTESTATION

The attestation of a TD consists of generating a local attestation report, which can be verified on the platform, and then extending this report with digital signatures and certificates to enable remote attestation of the TD off the platform. We first describe the overall process of generating and extending a local TD report in Section 9.1. Then we review the setup and the configuration of the host TDX platform to enable remote attestation in Section 9.2. Finally, we provide details on using remote attestation for establishing a secure channel and encrypted boot in Section 9.3.

9.1 Attestation Process

Several steps are involved when generating and extending a local report of a TD to enable remote attestation. The first step is to take measurements of the loaded software during the build-time and runtime of the TD. The next step is to retrieve the TD's measurements and platform TCB information, i.e., generating a TD report. The final step is to derive a quote from the TD report. A third party can use the quote to verify whether the TD runs on a genuine TDX platform with the expected TCB versions and software measurements.

Taking Measurements. TDX provides two types of measurement registers for each TD: a build-time measurement register called Measurement of Trust Domain (MRTD) and four Runtime Measurement Registers (RTMRs). These measurement registers are comparable to the TPM's

Table 3. Mapping of TDX Measurement Registers and TPM PCRs

TDX Measurement Registers	TPM PCRs	Usage
MRTD	PCR[0]	Virtual firmware
RTMR [0]	PCR[1,7]	Virtual firmware data + configuration
RTMR [1]	PCR[2-5]	OS kernel + INITRD + boot parameters
RTMR [2]	PCR[8-15]	OS application
RTMR [3]	N/A	Reserved

PCRs, see Table 3 derived from [41] showing the mapping between TDX measurement registers and TPM PCRs.

The MRTD contains a measurement of the TD build process. At the TD creation, when the hypervisor adds initial memory pages to the TD, it extends the MRTD in the TDCS with measurements of these pages. The hypervisor calls `SEAMCALL[TDH.MEM.PAGE.ADD]` to add a page to the TD's memory and to initiate the measurement of the page. It first calculates a SHA384 update over the ASCII string "MEM.PAGE.ADD" and the GPA of the page. Then it extends the MRTD with the hash value. Once the page is copied into the TD's memory, i.e., mapped and available in the SEPT, the hypervisor calls `SEAMCALL[TDH.MR.EXTEND]` multiple times to measure the content of the page. The page is measured in blocks of 256 B. For each block, the extension operation first calculates a SHA384 update over the ASCII string "MR.EXTEND" and the GPA of the block. Second, it calculates another SHA384 update over the content of this block. Both hash values extend the MRTD. These initial pages contain the TD's virtual firmware. The MRTD's measurement does not include pages containing control structures, i.e., TDR, TDCS, and TDVPS, nor the SEPT. After the initial set of pages is added, the hypervisor finalizes the MRTD measurement using the `SEAMCALL[TDH.MR.FINALIZE]`. This disables future operations to extend the MRTD. For example, when initializing a TD, KVM, as a Linux hypervisor, measures the TDVF [33] (virtual firmware of a TD) code into the MRTD.

RTMRs are general measurement registers labeled 0 through 3 for TD's runtime measurements. A TD can use these registers to provide a measured boot, i.e., measuring all software loaded after booting. These measurement registers are initialized to zero. The TD calls `TDCALL[TDG.MR.RTMR.EXTEND]` to extend the content of a RTMR. The arguments of this call consist of an index to the measurement register and a 64 B-aligned physical address of the 48 B extension buffer containing the value. This call calculates a SHA384 hash over the current value of the given index measurement register concatenated with the value in the extension buffer, i.e., `RTMR[index] = SHA384(RTMR[index] || value)`. For example, TDVF measures the static/dynamic configuration data into the `RTMR[0]` and the OS kernel, boot parameters, and INITRD into the `RTMR[1]`.

Generating TD Reports. A REPORT is generated inside a TD. The TD calls `TDCALL[TDG.MR.REPORT]`, which is the TDX Module's report function, with a newly initialized report structure and some user report data, named `REPORTDATA`. The `REPORTDATA` is 64 B and it can be used as a NONCE to verify freshness of the TD REPORT. To service the call, the TDX Module invokes the newly added `SEAMOPS[SEAMRERPORT]` instruction with the TD's measurements and `REPORTDATA`. The CPU adds TCB information related to the SEAM and returns a TD REPORT. This TD REPORT is integrity protected using an HMAC key maintained by the CPU. The HMAC key is only available to the CPU. The TDX Module returns this REPORT to the TD. Using the `EVERIFYREPORT2` instruction, an enclave can verify the REPORT on the same platform but not off the platform.

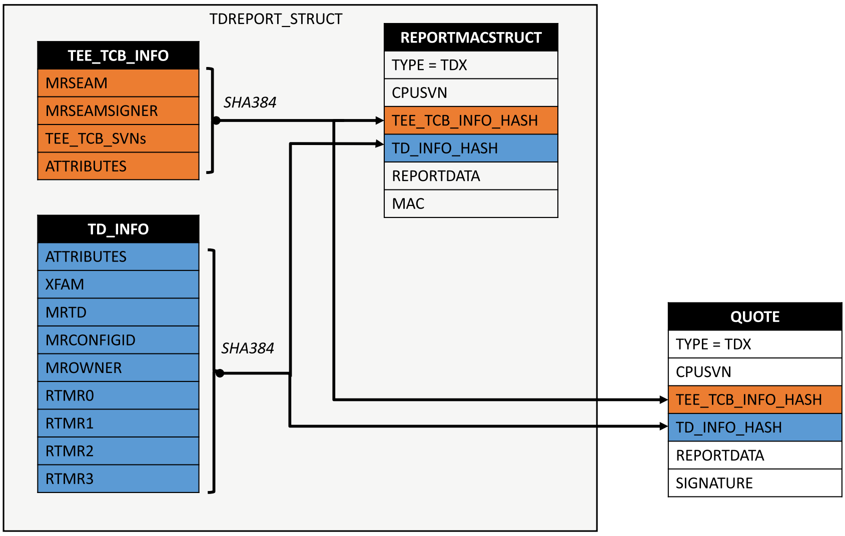

Fig. 8. TD report structure.

Figure 8 illustrates a TD REPORT consisting of three components: `REPORTMACSTRUCT`, `TEE_TCB_INFO`, and `TD_INFO`. The `REPORTMACSTRUCT` structure contains header information specifying the structure type as TDX and has fields for CPU SVN and hashes of the `TEE_TCB_INFO` and `TD_INFO` components. It also includes the `REPORTDATA` provided as the input to the `TDCALL[TDG.MR.REPORT]` function. Finally, there is an HMAC over the entire header that protects `TEE_TCB_INFO` and `TD_INFO` components. The `TEE_TCB_INFO` structure contains information about the SVN and measurements of the TDX Module. The `TD_INFO` contains TD attestable properties. Examples of these properties include the initial TD configuration and values of the measurement registers.

Deriving Quotes. To enable verification off the platform by a third party, the TD REPORT must be converted into a QUOTE. TDX tends to reuse the remote attestation mechanism of SGX. A TD makes a call to request the QE running on the host platform to sign the TD REPORT. This call can be implemented over a VSOCK or a `TDCALL[TDG.VP.VMCALL]`, depending on how the quoting service is provided on the platform. The QE calls the `EVERIFYREPORT2` instruction to verify the TD REPORT's HMAC. If this call is successful, the QE signs the TD REPORT using its certified attestation key to generate a QUOTE. This operation basically replaces the MAC integrity protection of the TD REPORT with the digital signature protection, allowing any party to verify the provenance and integrity of the QUOTE using public key certificates. Section 9.2 describes the operations for enabling the local attestation infrastructure on the platform to support remote attestation.

9.2 Platform Setup

Configuring the attestation infrastructure involves registering the platform with the Intel PCS, running architectural enclaves for generating QUOTES, and retrieving certificates required for verifying QUOTES. Intel extends the existing DCAP [58] to support remote attestation for TDX.

Registration. On multiple-package platforms, platform keys are derived at platform assembly time. These keys are shared between CPU-packages and are encrypted by the CPU's unique hardware key. PROVISIONING CERTIFICATION KEYS (PCKs) are derived from the platform keys and

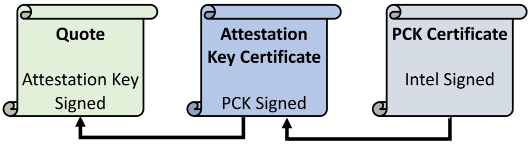

Fig. 9. Quote certificate chain.

used for certifying (signing) attestation keys. Since PCKs are not recognized by the attestation infrastructure, they must be registered with Intel PCS.

To register a platform, we need to run the PCK Cert ID Retrieval Tool to extract a manifest from the platform. This manifest contains information on CPU packages, e.g., CPU ID (128-bit), SVN, and hardware TCB information. When the Intel PCS gets the register server request, it checks whether CPUs and TCB are in good standing before issuing a PCK certificate. The manifest is signed with keys derived from the CPU package's hardware keys and the Intel PCS checks whether these signatures are valid. If registration succeeds, the Intel PCS returns an Intel-issued certificate for the PCK.

Typically in DCAP, a Provisioning Certification Caching service (PCCS) runs on the host platform to facilitate PCK certificate retrieval. This service can run anywhere. It forwards the PCK requests from the PCK Cert ID Retrieval Tool to the Intel PCS and caches the returned PCK certificates locally. The Intel PCS also provides certificates and revocation lists for PCKs in all genuine Intel platforms. PCCS maintains local caches of these artifacts as well.

Before registering, a platform must have the appropriate UEFI/BIOS settings and access to the Intel PCS. Both TDX and SGX must be enabled in the UEFI/BIOS on the host platform. An Intel account is required for retrieving API keys for registering a platform with the Intel PCS. If the PCCS is utilized, it must be configured with the API keys and Intel PCS server's address.

Architectural Enclaves. To enable quote generation on the platform. Intel provides two architectural enclaves: Provisioning Certificate Enclave (PCE) and QE. The PCE acts as a local certification authority for the QE. In its initialization process, the QE generates an attestation key pair. It sends the public part to the PCE. The PCE authenticates that this is a legitimate QE on the platform and then signs the attestation public key certificate with the PCK. This signature creates a quote certificate chain from an Intel-issued PCK certificate to the QE attestation public key. Figure 9 illustrates the quote's certificate chain. The PCK certificate is used for verifying the QE attestation public key certificate, and the QE attestation public key in turn for verifying the signature on the quote.

Remote Attestation Flow. Figure 10 shows a remote third party performing attestation with an attestation agent running on a TD. The remote party sends an attestation request providing a nonce to the attestation agent (Step 1). The nonce provides freshness to the request and prevents replay attacks. The attestation agent retrieves a TD report from the TDX Module providing the nonce as the `REPORTDATA` (Step 2) and then subsequently requests the QE to sign the TD report using its attestation key (Step 3). The QE verifies that the TD report is generated on the platform before signing with its attestation key. The attestation agent then returns the quote to the remote party (Step 4).

The remote party requires the platform's PCK certificate to verify the quote, so it may download the PCK certificate from a PCCS (Step 5) or retrieve directly from the Intel PCS (Step 6). The party then proceeds to validate the quote (Step 7). It checks for the nonce in the quote and verifies the integrity of the signature chain from the Intel-issued PCK certificate to the signed quote, walking the certificate chain to determine whether the quote has a valid signature. The party also

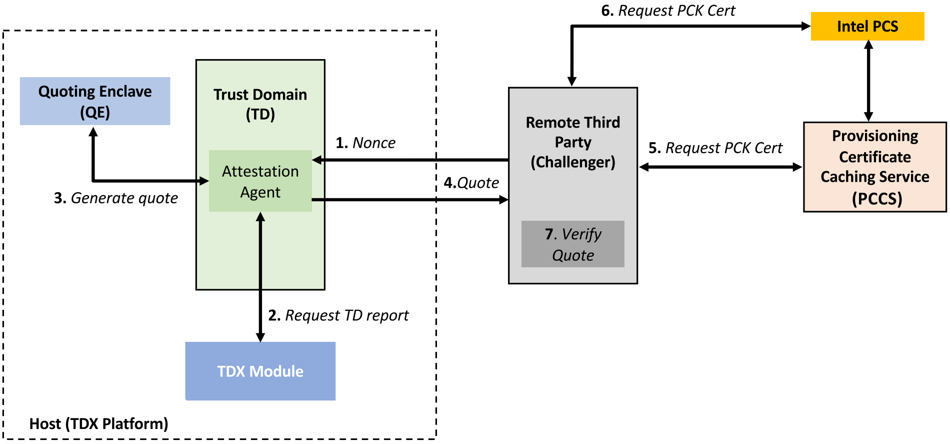

Fig. 10. A remote attestation flow.

checks that no keys in the chain have been revoked and whether the TCB is up-to-date. Finally, the party checks if the measurements, i.e., values in MRTD and RTMRs, in the QUOTE match a set of reference values. If it successfully validates the QUOTE, the remote party can trust that the TD has been properly instantiated on a TDX platform.

9.3 Use Cases

Secure Channel Establishment. Remote attestation can be integrated with establishing a secure channel [46], linking channel setup with the endpoint's TEE identity, state, and configuration. This integration prevents relay attacks since an attacker cannot forward a challenger's attestation request from a compromised system to a trusted system to service the request.

In a typical scenario when a client negotiates a secure channel with a server running in a TEE, it wants to ensure a connection with a properly instantiated server. The server, serving as an attester, generates an ephemeral public and private key pair. It computes the hash of the public key and then creates a TD REPORT providing this hash as the `REPORTDATA`. The server requests a QUOTE of the REPORT and generates a self-signed certificate with the QUOTE embedded in the certificate. It provides this self-signed certificate as the server certificate in the TLS handshake protocol. When the client, serving as the challenger, receives the server certificate, it verifies signatures on the certificate and validates the embedded QUOTE in the certificate, including the measurements. It also checks if the QUOTE includes the hash of the public key since this links the key to the TEE. When establishing a secure channel, both client and server can assume the roles of attester and verifier. This allows endpoints running in TEEs to mutually authenticate each other by validating TEEs.

Encrypted Boot. Using remote attestation, we can launch a TD with an encrypted partition image and let the tenant control the release of the partition decryption key. Figure 11 illustrates the execution flow of an encrypted boot of a TD. An ATTESTATION AGENT is placed within the TD's INITRD and starts when a TD is launched. The AGENT retrieves the TD's QUOTE, which contains the TCB measurements (from the TDX platform up to the INITRD), and is signed by Intel's QE. The QUOTE is sent over a secure channel to an ATTESTATION SERVER controlled by the tenant for verification. Once a QUOTE is verified with the expected measurements, the ATTESTATION SERVER provides the decryption key for the partition to the ATTESTATION AGENT. The AGENT then uses

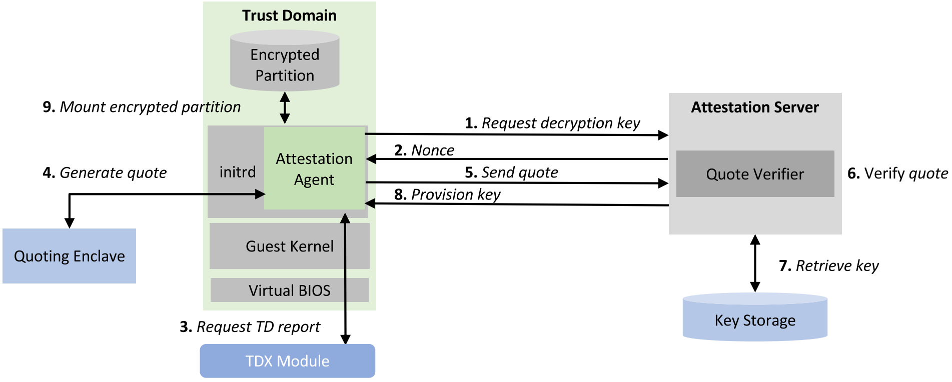

Fig. 11. Encrypted boot flow.

this key to decrypt and mount the encrypted partition that contains the secrets. This ensures that secrets can only be loaded into encrypted memory and are not visible to the host.

Tenants also have the flexibility to integrate the ATTESTATION AGENT in the virtual BIOS and wrap the entire VM workload within an encrypted image. In this case, the virtual BIOS needs to be extended to support the functionalities of retrieving the QUOTE and fetching the key for mounting the encrypted disk image.

10 CONCLUSION

In this article, we provide a top-down review of Intel TDX, covering its security principles, threat model, underpinning technologies, system architecture, and future features. We then dive deeper into the design of the TDX Module, memory protection mechanisms, and remote attestation. The review is based on publicly available documentation and source code. As confidential computing is a fast-evolving field, we highlight ongoing challenges and efforts, including the need to support live migration and trusted I/O. We will continue to conduct in-depth security analysis as the technology progresses.

APPENDIX

A LIST OF ACRONYMS

ABI	Application Binary Interface	7
ACM	Authenticated Code Module	15
ACPI	Advanced Configuration and Power Interface	4
AES	Advanced Encryption Standard	5
BCB	Bounds Check Bypass	21
CCA	Confidential Compute Architecture	4
CET	Control Flow Enforcement Technology	3
Ci	Cryptographic Integrity	13
CMR	Convertible Memory Region	21
CoVE	Confidential VM Extension	4
DCAP	Data Center Attestation Primitives	2
DMA	Direct Memory Access	4
DoS	Denial of Service	3
ECC	Error Correction Code	12
EPC	Enclave Page Cache	9
EPCM	Enclave Page Cache Map	9
EPT	Extended Page Table	8
ESM	Enter Secure Mode	6

GPA	Guest Physical Address	8
GPT	Granule Protection Table	7
GVA	Guest Virtual Address	8
HKID	Host Key Identifier	9
HPA	Host Physical Address	8
IAS	Intel Attestation Service	10
IOMMU	Input/Output Memory Management Unit	4
ISA	Instruction Set Architecture	6
KET	Key Encryption Table	9
Li	Logical Integrity	13
LP	Logical Processor	16
MAC	Message Authentication Code	13
MEE	Memory Encryption Engine	9
MigTD	Migration TD	14
MKTME	Multi-key Total Memory Encryption	2
MMIO	Memory-Mapped Input/Output	4
MMU	Memory Management Unit	20
MRTD	Measurement of Trust Domain	23
MSK	Migration Session Key	14
MSR	Model-Specific Register	4
MTT	Memory Tracking Table	7
OS	Operating System	2
PAMT	Physical Address Metadata Table	13
PCCS	Provisioning Certification Caching service	26
PCE	Provisioning Certificate Enclave	26
PCI	Peripheral Component Interconnect	4
PCK	Provisioning Certification Key	4
PCR	Platform Configuration Register	6
PCS	Provisioning Certification Service	4
PEF	Protected Execution Facility	4
PSP	Platform Security Processor	5
PTE	Page Table Entry	17
QE	Quoting Enclave	10
RDCL	Rogue Data Cache Load	21
RME	Realm Management Extension	7
RMP	Reverse Mapping Table	5
RTMR	Runtime Measurement Register	23
S-IOV	Scalable I/O virtualization	8
SEAM	Secure-Arbitration Mode	1
SEPT	Secure EPT	18
SEV	Secure Encrypted Virtualization	4
SGX	Software Guard Extensions	2
SLAT	Second Level Address Translation	8
SME	Secure Memory Encryption	5
SMM	System Management Mode	3
SoC	System-on-Chip	5
SR-IOV	Single Root I/O virtualization	8
SVM	Secure Virtual Machine	6
SVN	Security Version Number	16
SVSM	Secure VM Service Module	5
TCB	Trusted Computing Base	2
TD	Trust Domain	2
TDCS	Trust Domain Control Structure	18
TDMR	Trust Domain Memory Region	17
TDR	Trust Domain Root	17
TDVPS	Trust Domain Virtual Processor State	18
TDX	Trust Domain Extensions	1
TEE	Trusted Execution Environment	2
TLB	Translation Lookaside Buffer	3
TLS	Transport Layer Security	3
TME	Total Memory Encryption	7
TPM	Trusted Platform Module	6
TSM	TEE Security Manager	7

TVM TEE Virtual Machine 7
TXT Trusted Execution Technology 15
UD Undefined Instruction 8
VAPIC Virtual APIC 18
VE Virtualization Exception 11
VM Virtual Machine 2
VMCS Virtual Machine Control Structure 8
VMM Virtual Machine Monitor 7
VMPL Virtual Machine Privilege Level 5
VMX Virtual Machine Extensions 7
VT Virtualization Technology 2

ACKNOWLEDGMENTS

We would like to extend our sincere thanks to Guerney Hunt, Rick Boivie, Dimitrios Pendarakis, and Jonathan Bradbury for taking the time to read our draft and providing their invaluable feedback and suggestions.

REFERENCES

[1] 2018. CVE-2017-5753. Retrieved March 29, 2024 from https://nvd.nist.gov/vuln/detail/CVE-2017-5753

[2] 2018. CVE-2017-5754. Retrieved March 29, 2024 from https://nvd.nist.gov/vuln/detail/CVE-2017-5754

[3] AMD. 2020. Strengthening VM isolation with integrity protection and more. *AMD* (2020).

[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. Scone: Secure linux containers with intel sgx. In *Proceedings of the OSDI*. 689–703.

[5] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. 2022. Branch history injection: On the effectiveness of hardware mitigations against {cross-privilege} spectre-v2 attacks. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security 22)*. 971–988.

[6] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems* 33, 3 (2015), 1–26.

[7] Jethro G. Beekman, John L. Manferdelli, and David Wagner. 2016. Attestation transparency: Building secure internet services for legacy clients. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 687–698.

[8] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2017. Hybrids on steroids: SGX-based high performance BFT. In *Proceedings of the 12th European Conference on Computer Systems*. 222–237.

[9] Patrick Bohrer, James Peterson, Mootaz Elnozahy, Ram Rajamony, Ahmed Gheith, Ron Rockhold, Charles Lefurgy, Hazim Shafi, Tarun Nakra, Rick Simpson, Evan Speight, Kartik Sudeep, Eric Van Hensbergen, and Lixin Zhang. 2004. Mambo: A full system simulator for the PowerPC architecture. *ACM SIGMETRICS Performance Evaluation Review* 31, 4 (2004), 8–12.

[10] Rick Boivie and Peter Williams. 2012. SecureBlue++: CPU support for secure execution. *IBM, IBM Research Division, RC25287 (WAT1205-070)* (2012), 1–9.

[11] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure: SGX cache attacks are practical. In *Proceedings of the WOOT*. 11–11.

[12] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. Securekeeper: Confidential zookeeper using intel sgx. In *Proceedings of the 17th International Middleware Conference*. 1–13.

[13] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *Proceedings of the 2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 142–157.

[14] Linux Kernel Documentation. 2023. Retrieved March 29, 2024 from https://www.kernel.org/doc/Documentation/x86/tdx.rst

[15] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. Branchscope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices* 53, 2 (2018), 693–707.

[16] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. 2017. Iron: Functional encryption using Intel SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 765–782.

[17] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. 2017. HardIDX: Practical and secure index with SGX. In *Proceedings of the Data and Applications Security and Privacy XXXI: 31st Annual IFIP WG 11.3 Conference, DBSec 2017, Philadelphia, PA, USA, July 19-21, 2017*. Springer, 386–408.

[18] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*. 1–6.

[19] James Greene. 2012. Intel trusted execution technology: Hardware-based technology for enhancing server platform security. *Intel Corporation* (2012).

[20] Zhongshu Gu, Heqing Huang, Jialong Zhang, Dong Su, Hani Jamjoom, Ankita Lamba, Dimitrios Pendarakis, and Ian Molloy. 2020. *Confidential Inference via Ternary Model Partitioning*.

[21] Zhongshu Gu, Hani Jamjoom, Dong Su, Heqing Huang, Jialong Zhang, Tengfei Ma, Dimitrios Pendarakis, and Ian Molloy. 2019. Reaching data confidentiality and model accountability on the caltrain. In *Proceedings of the 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 336–348.

[22] Guerney D. H. Hunt, Ramachandra Pai, Michael V. Le, Hani Jamjoom, Sukadev Bhattiprolu, Rick Boivie, Laurent Dufour, Brad Frey, Mohit Kapur, Kenneth A. Goldman, Ryan Grimm, Janani Janakirman, John M. Ludden, Paul Mackerras, Cathy May, Elaine R. Palmer, Bharata Bhasker Rao, Lawrence Roy, William A. Starke, Jeff Stuecheli, Enriquillo Valdez, and Wendel Voigt. 2021. Confidential computing for OpenPOWER. In *Proceedings of the 16th European Conference on Computer Systems*. 294–310.

[23] IBM. 2020. Retrieved March 29, 2024 from https://github.com/open-power/ultravisor

[24] IBM. 2022. Introducing IBM secure execution for linux 1.3.0. Retrieved from https://www.ibm.com/docs/en/linuxonibm/pdf/l130se03.pdf

[25] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. 1–11.

[26] Intel. 2021. Intel® trust domain cpu architectural extensions specification. Retrieved March 29, 2024 from https://cdrdv2.intel.com/v1/dl/getContent/733582

[27] Intel. 2022. Retrieved from https://www.intel.com/content/www/us/en/download/738875/738876/intel-trust-domain-extension-intel-tdx-module.html

[28] Intel. 2022. Retrieved March 29, 2024 from https://www.intel.com/content/www/us/en/download/738874/intel-trust-domain-extension-intel-tdx-loader.html

[29] Intel. 2022. Retrieved March 29, 2024 from https://github.com/intel/tdx/

[30] Intel. 2022. Retrieved March 29, 2024 from https://intel.github.io/ccc-linux-guest-hardening-docs/security-spec.html

[31] Intel. 2022. Intel® architecture memory encryption technologies. *Intel Corporation* (2022).

[32] Intel. 2022. Intel® TDX loader interface specification. Retrieved March 29, 2024 from https://cdrdv2.intel.com/v1/dl/getContent/733584

[33] Intel. 2022. TDX Virtual Firmware (TDVF). Retrieved March 29, 2024 from https://github.com/tianocore/edk2-staging/tree/TDVF

[34] Intel. 2023. CPUID enumeration and architectural MSRs. Retrieved March 29, 2024 from https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/cpuid-enumeration-and-architectural-msrs.html

[35] Intel. 2023. Device attestation model in confidential computing environment. Retrieved March 29, 2024 from https://cdrdv2.intel.com/v1/dl/getContent/742533

[36] Intel. 2023. Intel TDX Module v1.5 TD Migration Architecture Specification. Retrieved March 29, 2024 from https://cdrdv2.intel.com/v1/dl/getContent/733578

[37] Intel. 2023. Intel TDX module v1.5 TD partitioning architecture specification. Retrieved March 29, 2024 from https://cdrdv2.intel.com/v1/dl/getContent/773039

[38] Intel. 2023. Intel® TDX connect architecture specification. Retrieved March 29, 2024 from https://cdrdv2.intel.com/v1/dl/getContent/773614

[39] Intel. 2023. Intel® TDX guest-hypervisor communication interface. Retrieved March 29, 2024 from https://cdrdv2.intel.com/v1/dl/getContent/726790

[40] Intel. 2023. Intel® TDX module 1.0 specification. Retrieved March 29, 2024 from https://cdrdv2.intel.com/v1/dl/getContent/733568 (2023).

[41] Intel. 2023. Intel® TDX virtual firmware design guide. Retrieved March 29, 2024 from https://cdrdv2.intel.com/v1/dl/getContent/733585

[42] Intel. 2023. Intel® trust domain extensions. Retrieved March 29, 2024 from https://cdrdv2.intel.com/v1/dl/getContent/690419

[43] David Kaplan. 2017. Protecting vm register state with sev-es. *AMD* (2017).

[44] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. *AMD* (2016).

[45] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 361–372. DOI: https://doi.org/10.1109/ISCA.2014.6853210

[46] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. 2019. *Integrating Remote Attestation with Transport Layer Security.*

[47] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2020. Spectre attacks: Exploiting speculative execution. *Communications of the ACM* 63, 7 (2020), 93–101.

[48] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre returns! Speculation attacks using the return stack buffer. In *Proceedings of the12th USENIX Workshop on Offensive Technologies.*

[49] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the USENIX Security Symposium.* 16–18.

[50] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. 2022. Design and verification of the arm confidential compute architecture. In *Proceedings of the16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22).* 465–484.

[51] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. 2017. Glamdring: Automatic application partitioning for intel SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17).* USENIX Association, Santa Clara, CA, 285–298.

[52] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. 2020. Meltdown: Reading kernel memory from user space. *Communications of the ACM* 63, 6 (2020), 46–56.

[53] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*, Tel-Aviv, Israel, 1 pages.

[54] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. Cachezoom: How SGX amplifies the power of cache attacks. In *Proceedings of the Cryptographic Hardware and Embedded Systems–CHES 2017: 19th International Conference, Taipei, Taiwan, September 25-28, 2017.* Springer, 69–90.

[55] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious multi-party machine learning on trusted processors. In *Proceedings of the USENIX Security Symposium.* 10–12.

[56] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A secure database using SGX. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP).* IEEE, 264–278.

[57] Ravi Sahita, Vedvyas Shanbhogue, Andrew Bresticker, Atul Khare, Atish Patra, Samuel Ortiz, Dylan Reid, and Rajnesh Kanwal. 2023. CoVE: Towards confidential computing on RISC-V platforms. In *Proceedings of the 20th ACM International Conference on Computing Frontiers.* 315–321.

[58] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. 2018. Supporting third party attestation for Intel SGX with Intel data center attestation primitives. Retrieved March 29, 2024 from https://cdrdv2-public.intel.com/671314/intel-sgx-support-for-third-party-attestation.pdf

[59] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy.* IEEE, 38–54.

[60] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware guard extension: Using SGX to conceal cache attacks. In *Proceedings of the Detection of Intrusions and Malware, and Vulnerability Assessment: 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017.* Springer, 3–24.

[61] Carlos Bilbao Thomas Lendacky. 2023. Linux SVSM (Secure VM Service Module). Retrieved March 29, 2024 from https://github.com/AMDESE/linux-svsm

[62] Florian Tramèr and Dan Boneh. 2019. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware.

[63] Chia-Che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A practical library os for unmodified applications on SGX. In *Proceedings of the USENIX Annual Technical Conference.* 645–658.

[64] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. 2005. Intel virtualization technology. *Computer* 38, 5 (2005), 48–56.

[65] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium.* USENIX, 991–1008.

[66] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution.* 1–6.

[67] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium*. USENIX Association, 1041–1056.

[68] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue in-flight data load. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 88–105.

[69] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2421–2434.

[70] Peter Williams and Rick Boivie. 2011. CPU support for secure executables. In *Proceedings of the Trust and Trustworthy Computing: 4th International Conference, TRUST 2011, Pittsburgh, PA, USA, June 22-24, 2011*. Springer, 172–187.

[71] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. IEEE, 640–656.

Received 27 March 2023; revised 7 February 2024; accepted 7 March 2024