Intel Trust Domain Extensions (TDX) Security Review

April, 2023

Erdem Aktas[1], Cfir Cohen[1], Josh Eads[1], James Forshaw[2], Felix Wilhelm[2]

[Executive Summary](#)

[Background](#)

[Terminology](#)

[Intel TDX Threat Model](#)
- [Design Goals](#)
- [Adversarial Goals](#)
  - [Leaking TD Secrets](#)
  - [Manipulating TD Behavior](#)
  - [Host Denial-of-Service](#)
- [Attack Vectors](#)
  - [Malicious Hardware](#)
  - [Malicious BIOS](#)
  - [Malicious SMM](#)
  - [Malicious VMM](#)
  - [Malicious TD/VM](#)

[MCHECK](#)
- [System Validation](#)
- [Security Concerns](#)

[Non Persistent SEAM Loader](#)
- [Threat Model](#)
- [Attestation and Rollback Prevention](#)
- [Security Concerns](#)
- [Security Vulnerabilities](#)
  - [Unsafe Performance Monitoring VMCS Configuration](#)
    - [Variant Analysis](#)
    - [Remediation](#)
  - [Exit Path Interrupt Hijacking](#)

---

[1] Google Cloud Security
[2] Google Project Zero

        Exploitation
        Variant Analysis
        Remediation
    Mitigating controls

Persistent SEAM Loader
    Overview
    Install initiation
    TDX module authentication
    Module installation
    Page tables “keyhole” mechanism
    Misconfiguration bugs
    Mitigating controls

TDX Module
    Attack Surface
        Malicious TDs
        Malicious Host
    Security Review
        weggli
        Frama-C
    Discovered Issues
        Incorrect loop boundary in tdh_sys_tdmr_init
        Incorrect error handling in tdh_mng_rd_wr
        Off-by-one in shared_hpa_check
    TLB tracking
        Address translation
        Secure EPT
        TLB Tracking Algorithm
        revert_tlb_tracking_state() vulnerability

Uncore Attack Vector
        ECC Disablement Vulnerability

MSRs
    Review Methodology
    Attack Vectors
        TD-to-TDX Module Attacks
        VMM-to-TDX Module Attacks
        Address-Based Attacks
    Security Concerns
        VMM-to-TDX Privilege Inversion

Side Channel Attacks and Mitigations

- Speculation based side channel attacks
  - Transient execution attacks
  - Speculation variants
    - Prediction based
    - Fault / Assist based
  - Value injection variants
  - Secret output variants
  - Applications to TDX
  - Mitigations
  - On hyperthreading
- Traditional side channel attacks
- Access oracles
  - Blocked private pages
  - Poisoned cache lines
  - MONITOR and MWAIT
  - Boosting cache based side channel attacks
- Zero step / Single step mitigations
- Baseboard Management Controller
- Conclusions

TDX Logical Integrity and Memory Corruption Attacks
- Corruption Targets
  - VMM allocated control structures
  - SEAM Range
- Mitigations

Attestation
- Measurements

Debug Security
- TD Debugging
- TDX System Debugging

Security Review Results

Future Research Areas

Acknowledgments
- Google
- Intel

Appendix A - MSRs of Interest

Executive Summary

This report contains the results of Google's security review into Intel's Trust Domain Extensions (TDX). The Intel TDX feature was added to limited SKUs of the 4th generation Intel Xeon Scalable CPUs[3]. TDX provides hardware isolated virtual machines referred to as Trust Domains (TD), which isolates sensitive resources, such as virtualized physical memory from the hosting environment. This is a valuable addition to the Google Cloud platform as it provides assurances to customers that Google can not access their virtual machine's private information even with full control over the host control mechanisms such as the kernel services and hypervisor.

The primary goal of the security review was to provide assurances that the Intel TDX feature is secure, has no obvious defects and works as expected so that it can be confidently used by both cloud customers and providers. Any defects or weaknesses discovered during the review were fed back to Intel for remediation. A secondary goal was to have a better understanding of the expected threat model for TDX and identify limitations in the design and implementation that would better inform Google's deployment decisions.

The review encompassed source code inspection of the core Intel TDX software components and a review of the design and documentation provided by Intel. Each major area of TDX was reviewed for defects and weaknesses which would impact the security and availability of a deployed virtual machine. Some of the issues inspected were:

- Arbitrary code execution in a privileged security context.
- Cryptographic weaknesses and oracles.
- Temporary and permanent denial of service.
- Weaknesses in debug or deployment facilities.

During the review there was close collaboration between Google and Intel engineers. Questions and issues were handled through a shared issue tracker and regular technical meetings. This allowed Intel to provide deep technical information about the function of the TDX components as well as enabling the reviewers to resolve potential ambiguities in documentation and source code. The review resulted in 81 potential attack vectors and resulted in 10 confirmed security issues and 5 defense in depth changes over a period of 9 months.

This report begins by detailing the Intel TDX threat model as described through a review of the available documentation and discussions with Intel. It then follows with separate sections for each major component of TDX. Each component is described in detail along with what review process was undertaken by the engineers to verify its security properties. Any issues that were discovered during the review are also detailed as well as the status of their mitigations. Intel mitigated the issues discovered before the production release of the 4th gen Intel Xeon Scalable processors.

[3] TDX is expected to be in general availability in a future Intel Xeon Scalable CPU.

The most serious implementation issue discovered during the review was a bug in the Authenticated Code Module (ACM) responsible for initializing the TDX feature. When the ACM transitions from its privileged execution context back to an untrusted context it incorrectly handles interrupts. The bug allowed untrusted code to execute within the privileged execution mode and compromise the integrity of the TDX feature and the security of any deployed virtual machines.

It was also realized that all ACMs, of which TDX is only one type, can be a weakness in the design as they all run within the same privileged execution context. This means that issues with one ACM could be used to compromise any other. Any ACM that is provided for a platform should be thoroughly reviewed before being used on a production system.

The team identified additional design-level issues during the review process. These issues included the significant number of Machine Specific Registers (MSR) that could interact with the functionality of the TDX. For example, some MSRs could interact with physical addresses assigned to the TDX feature allowing corruption or leaking of sensitive information. Rowhammer was also considered a significant risk to the security of TDX.

All implementation issues were identified in pre-release code. They will all be remediated before the TDX feature officially ships, both for the targeted release on 4th gen Intel Xeon Scalable CPUs, as well as the future general release; however, this does mean they will not be assigned a public identifier such as a CVE. Design level issues have been fed back to the Intel engineering team for review internally.

Intel has also opened the source code to the components the team reviewed so that further research can be performed in public. The source includes the TDX Module and Seam Loader SW; however, it omits low-level code such as the microcode used for some of the system's configuration.

Overall, the review has been considered to have met its initial goals of finding and remediating security issues in the implementation and design. There were limits to what was available to review, such as the lack of access to microcode or low-level hardware documentation; therefore, some aspects of the review are based on trust. However, it has also given Google a far greater appreciation of the expected threat model which will be beneficial for subsequent deployment stages.

Background

In existing virtualization deployments, such as cloud hosting, a virtual machine needs to fully trust the hosting environment and system administrators for the security of any data stored by the machine. While a virtual machine could encrypt data at rest there's little it can do to completely protect the runtime state. The hosting environment has the privilege to inspect and modify the memory and the CPU context to extract secret information while the virtual machine is operational.

Intel Trust Domain Extensions (TDX) is a new architectural component first being introduced in the 4th Gen Intel Xeon Scalable CPUs (formerly code-named “Sapphire Rapids”) to provide hardware and cryptographic isolation between virtual machines so that less trust needs to be placed in the hosting environment. This includes protecting against a compromised Virtual Machine Monitor (VMM) and boot environment.

Intel TDX functions by providing a new type of virtual machine guest called a Trust Domain (TD). This guest can only be controlled by a signed Intel TDX module running within a special privilege level, Secure Arbitration Mode (SEAM), which can't be directly accessed by normal code running on the CPU. The memory and context switch state of the virtual CPU is protected using the Total Memory Encryption - Multi-Key (TME-MK) feature built into the memory controller to encrypt data written to physical memory to protect the confidentiality of the TD's state. The integrity of these encrypted contents is additionally protected with either a SHA256-HMAC or a software-inaccessible access control bit.

Intel TDX is of interest to Google Cloud to increase isolation for customer virtual machines and provide a higher level of assurance that Google engineers or systems can't access their data. To be suitable for deployment in customer facing products Google desired additional assurances that the TDX design and implementation met the expected security requirements. For that reason, a team of Google engineers from Cloud and Project Zero were assembled to perform a review of the general design and the software implementation of TDX version 1.0 on Sapphire Rapids (SPR) with full cooperation from Intel.

Security reviews are limited in nature based on the amount of time available and how much access there is to the platform. Based on an initial assessment of publicly available documents it was apparent that TDX is a very complex and dense system, so the team decided to focus only on the areas of most importance.

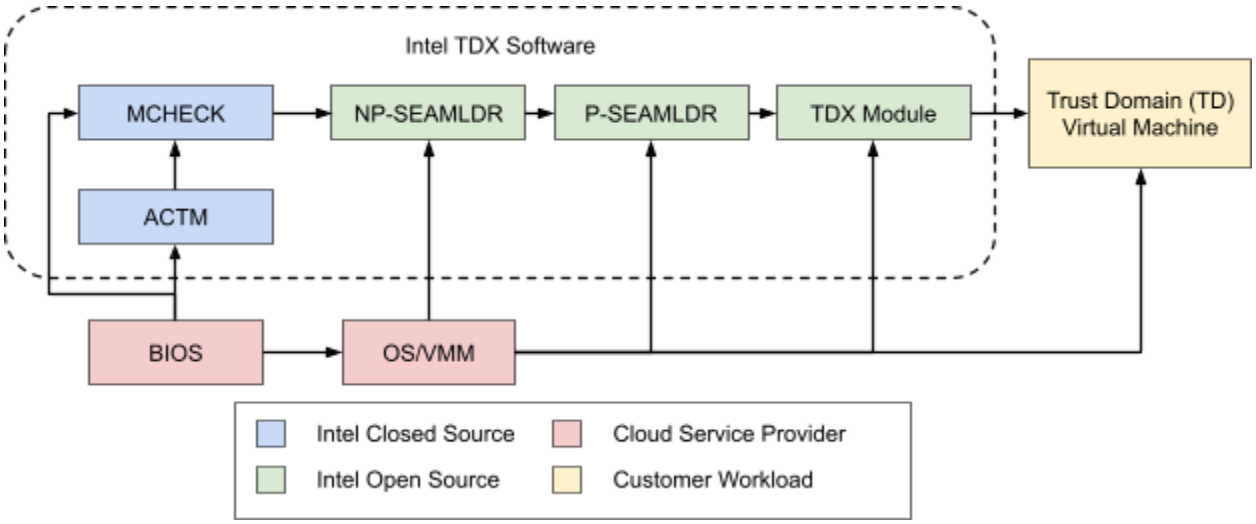

Figure 1: Diagram of Intel TDX Platform Initialization

The diagram above provides an overview of the initialization process for Intel TDX. For the security review we focused on the areas implemented by Intel. Specifically, the following components:

- MCHECK - Used by the BIOS to initialize the platform memory configuration
- Non-Persistent SEAM Loader (NP-SEAMLDR)
- Persistent SEAM Loader (P-SEAMLDR)
- TDX Module

These components are all interrelated and a failure of one would have a significant impact on the rest. For example, a flaw in MCHECK has the potential to compromise the entire initialization chain and therefore the security of the encrypted VM.

Anything outside of the four areas, such as the BIOS, Virtual Machine Manager (VMM) and any VM platform support such as Linux kernel changes were not reviewed. The team also did not review the system attestation provided by SGX. Therefore, the scope of the security review encompassed:

- Review of the public and private documentation
- Analysis and review of the source code for the following TDX software components:
    - The TDX module
    - The Secure Arbitration Mode (SEAM) loaders to bootstrap the TDX module
    - Host VMM to TDX and Guest to TDX APIs
- Provide feedback to Intel of issues discovered and general security improvements

Intel provided the review team with the design documentation and two source code repositories:

- Seam-loader - Contains the implementation of NP-SEAMLDR and P-SEAMLDR.
- TDX-module - Contains the implementation of the TDX module.

A critical building block in the TCB for TDX is the MCHECK module used to verify a number of system configuration parameters typically set by untrusted elements like the BIOS. At this time the source code for MCHECK is not available for review (and is delivered from Intel encrypted),

which limits the assurances providers and customers can derive from TDX's security properties. The team was also not provided with test systems to perform black-box testing. Additionally, the hardware implementation of the various TDX features was not provided for review. Therefore, analysis of these areas focused around documentation and design review.

This document is a summary of the security review performed by Google including technical details of some issues that were discovered during the process. The source code and specifications for the TDX components have been made open source and are available for download from Intel's TDX web page. This document should provide a useful introduction for further research into the security of the Intel TDX implementation.

Terminology

Secure Arbitration Mode (SEAM): A new x86 execution mode designed to isolate the TDX module and SEAMLDRs from entities outside the TDX TCB.
Non Persistent SEAM Loader (NP-SEAMLDR): The root-of-trust for Intel TDX. This signed module bootstraps the P-SEAMLDR.
Persistent SEAM Loader (P-SEAMLDR): Authenticates and installs (or uninstalls) the TDX module.
TDX Module: The software component which manages TDs. The VMM directs it and exposes both guest-facing and VMM-facing APIs.
Trust Domain (TD): A VM running under control of the TDX module. Its memory and CPU state are encrypted and integrity protected.
MCHECK: Validates the platform has been configured securely before SGX and TDX are initialized. Embedded within the signed and encrypted microcode update blob and run during BIOS.
Authenticated Code Module (ACM): Intel's format for signed blobs which are authenticated against a fused key hash and provide a dynamic root of trust.
Uncore: All components other than the x86 cores that are still within the Intel SoC. The L3 cache, memory controller, and power control unit are some examples.
Security Version Number (SVN): A monotonically increasing number attached to microcode updates, ACMs, and the TDX module. This number is independent from the functional version number and only incremented when security properties change (e.g., vulnerability patch).
Virtual Machine Monitor (VMM): The host system used for managing virtual machines, virtual devices, and interacting with external services. In this document and some Intel documents, VMM is used to indicate all host code outside of the TDX TCB once the host OS is running.
Intel Total Memory Encryption - Multi-Key (TME- MK): A feature on recent Intel memory controllers which encrypts DRAM using a key selector stored in the upper bits of the physical address. (Note: TME-MK is sometimes referred in code as "MKTME").
Host Key ID (HKID): The TME-MK key selector associated with a given TD. There is only one HKID associated with each TD (and one for the TDX module itself), and they are immutable.
Poison: A tracking mechanism within Intel CPUs used to propagate memory errors through the busses until CPU consumption. TDX utilizes memory poisoning to detect and respond to TD memory corruption.

Intel TDX Threat Model

This section presents our threat model for the Intel TDX technology based on the original design and claims by Intel and our understanding of the various goals an adversary may have when attacking this system. This model informed where we focused our efforts during the review as well as which attacks were out of scope. Intel also published a paper at IEEE SEED in 2021 which describes their threat model and security analysis of TDX.

Design Goals

Intel has published a white paper for Intel TDX which provides an overview of the technology, the threats it is designed to mitigate, and the methods for implementation and attestation. TDX asserts a very conservative trusted-computing base (TCB) which only includes the following components:

- Intel TDX module (including P-SEAMLDR)
- Intel Authenticated Code Modules (ACM)
- TD Quoting Enclave (SGX)
- Intel CPU hardware

All other system components are outside of the TCB, including the BIOS, SMM, host OS, and VMM. Additionally, some forms of physical attacks, such as cold-boot and DRAM traffic modification (except for replay), are also outside the TCB and should be protected against. At a high level, only the Intel hardware and signed core firmware should need to be trusted – all other software and design implemented by the cloud service provider can be considered untrusted.

The TDX module handles the bulk of the complex system interactions and is essentially a peer hypervisor interposing between the TD and the host HV/VMM. Writing a bug-free hypervisor is challenging[4] due to the complexities of handling arbitrary guest states and correctly handling complex hardware interactions. Given that bugs in the system are inevitable, a robust attestation system has been incorporated into TDX so that customers can trust their TD is running the latest versions of microcode, firmware, and TDX software.

Overall, this model assumes a sophisticated attacker who potentially has machine ownership/administrative privileges and some physical access – a model many existing technologies were not designed to protect against. Intel TDX is built on a combination of both legacy Intel technologies as well as new hardware and firmware additions.t is important to verify that each of these independently and in conjunction with each other can withstand an attacker who controls so much of the system.

[4] See previous vulnerabilities for KVM, Xen, VMware, Hyper-V

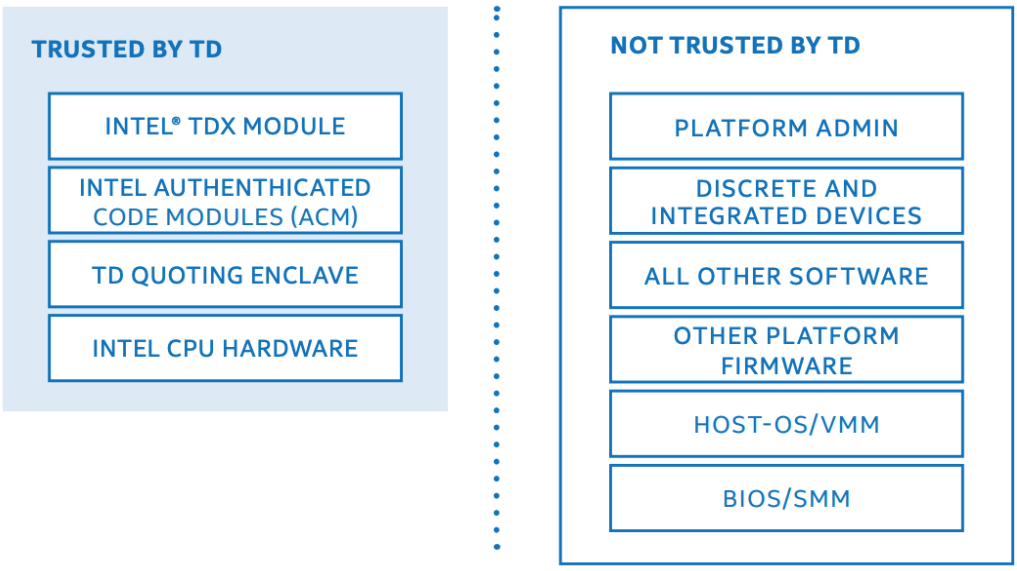

Figure 2: Trust Boundaries for TDX[5]

Adversarial Goals

An attacker targeting Intel TDX may focus their efforts on different components depending on what their goals are. In general, an adversary is interested in leaking sensitive information from the TDs, manipulating the behavior of TDs, or using a malicious TD to deny service to the host machine.

Leaking TD Secrets

After the attestation report is generated and verified, the third-party TD owner is expected to provision secret material to the TD. This information could be in the form of cryptographic keys, intellectual property, private user information, or similar. The goal of TDX's isolation design is to prevent this information from being indirectly or directly leaked to an adversary party.

For example, if there are side channels which exist (e.g., Spectre gadgets in the TDX module, shared resource side channels) then a neighboring TD or the VMM itself may be able to extract partial or complete information from the victim TD. Furthermore, if the TDX module is compromised then an attacker can directly read memory from the victim TD.

Manipulating TD Behavior

Similarly, an adversary may be interested in modifying the behavior of a victim TD. This could come in the form of direct memory or register corruption which leads to unexpected behavior or execution control. It could also take a more subtle form such as a VMM altering the scheduling pattern of a victim TD or tampering with I/O traffic between the TD and external world.

[5] Taken from the Intel TDX White Paper

Additionally, the VMM can modify the memory mapping of the TD (within limits) which may lead to differences in guest behavior.

Host Denial-of-Service

Finally, an adversary may simply desire to reduce the availability of a cloud provider by preventing other workloads (TDs, VMs) from being scheduled on a machine. Some of these attacks are more severe than others. For example, there could be a bug where a TD can cause itself to be shut down while another bug in the TDX module may require all TDs on the machine to be immediately halted. Furthermore, some forms of memory corruption and unexpected behavior under TDX can cause an unrecoverable machine check to occur which requires a full power cycle to recover from. Care must be taken in the TDX module to ensure that the risk of these machine checks occurring is minimized in order to prevent widespread availability attacks.

Attack Vectors

Due to so few elements on the system being within the TCB for Intel TDX, there are many different attack vectors an adversary can utilize when attempting to degrade the system. Additionally, by combining various attack vectors (e.g., a malicious VMM and TD working together) attackers can reach complex edge cases that the system designers may not have planned for. The sections below list the main pathways an attacker has to interact with the TDX components.

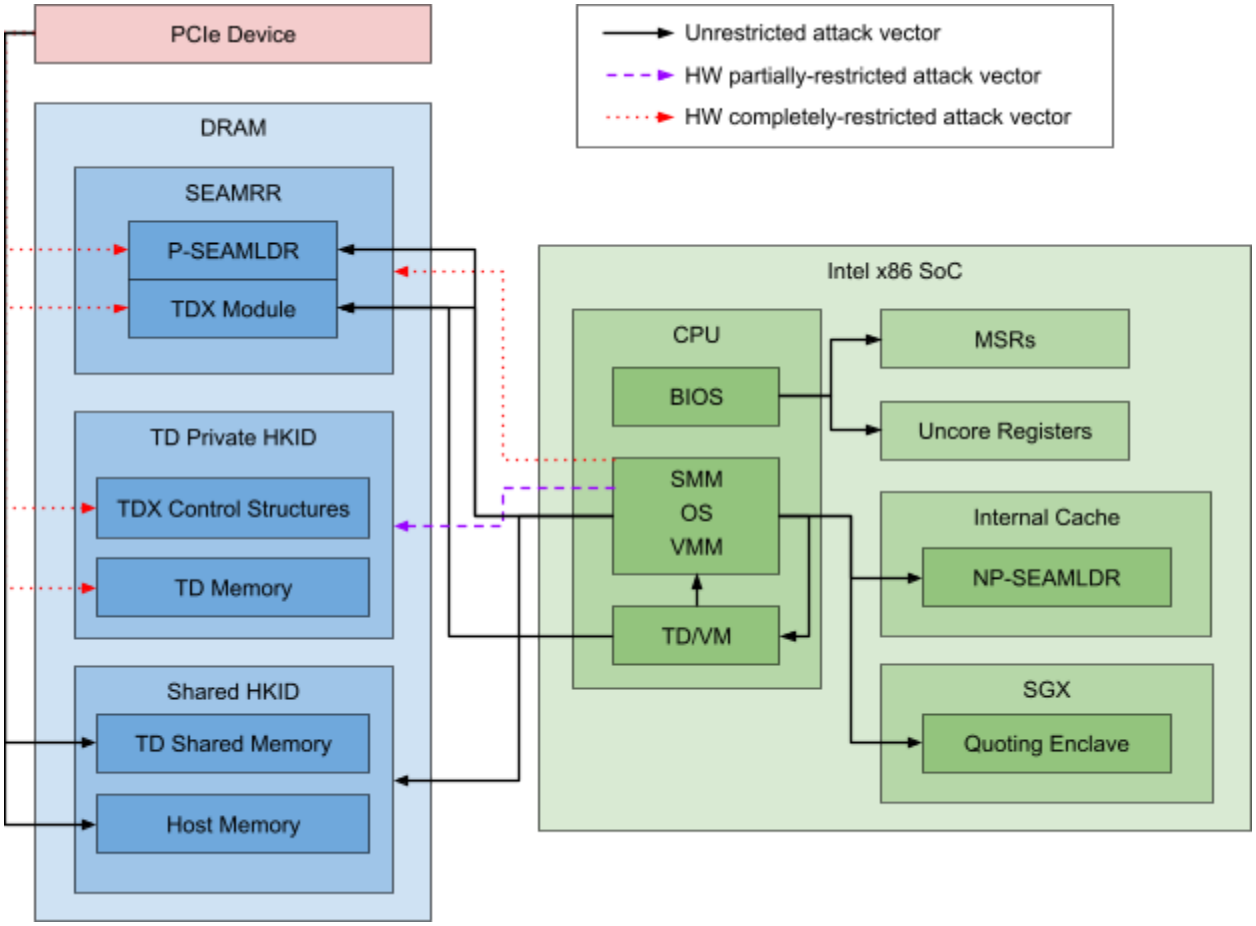

Figure 3: Diagram illustrating the main attack vectors available on Intel TDX. The system is decomposed into external devices (red); external memory (blue); and SoC code, registers, and internal memory (green).

Malicious Hardware

Intel states in the TDX whitepaper that TDX 1.0 is designed to withstand some forms of physical attacks such as DRAM capturing and modification; however, there are no protections against physical replay attacks.

PCIe devices such as GPUs and TPUs are sometimes attached directly to VMs in cloud environments. These devices use DMA to directly access the x86 host machine's DRAM and traditionally the host OS is responsible for programming an IOMMU to restrict access and enforce VM isolation. However, when the OS and VMM are outside the TCB, the correct programming of the IOMMU can no longer be relied on. To protect against this attack, PCIe memory TLPs targeting TD private memory (physical addresses with a private HKID set) will be dropped by the IOMMU regardless of host programming. External devices can only interface with TD shared memory which the TD operating system explicitly marks as shared in its guest page tables.

Since physical machines supporting TDX were not available during this review, we left malicious hardware attacks outside the scope of the review.

Malicious BIOS

In the Intel TDX threat model, the BIOS (i.e., UEFI) and all code launched by it (OS, VMM) are considered untrusted. This is a significant difference compared to traditional virtualization offerings where no guarantees are made about the trustworthiness of the underlying software. The BIOS is responsible for bringing up all system components and applying the appropriate configuration. Additionally, the BIOS has additional privileges explicitly recognized by the hardware which are significantly restricted when transitioning (by setting the BIOS_DONE MSR bit) to the operating system. Given this extensive level of system access, ensuring that the integrity of TDX is not compromised by a malicious BIOS is challenging. Intel provides technologies to protect the integrity of BIOS to different degrees (e.g., Bootguard, TXT, PFR). At Google, we use the Titan security chip to ensure that the initial BIOS image that the CPU boots is authentic and produced by Google.

The BIOS has similar access to system configuration as an operating system, but there are additional registers it can access – either due to hardware access controls or lockable registers which will normally be locked by the time the OS gains execution. The BIOS is also responsible for launching MCHECK (responsible for verifying the system has been securely configured) and SGX initialization, both of which are critical for TDX integrity. We enumerated all MSR and Uncore registers accessible by the BIOS, SMM, and OS and attempted to identify any TDX security-relevant registers and bits; however, given the scope of the search space this was not comprehensive[6]. Many of the sensitive registers identified were confirmed with Intel engineers to be checked by MCHECK or had other restrictions which prevented BIOS access from being exploitable.

Malicious SMM

The system management mode (SMM) code is responsible for handling a variety of system management tasks and interrupts (SMIs), is more highly privileged than the OS, and is launched by the BIOS. From a threat analysis point of view, if the BIOS is compromised then the SMM module should also be considered compromised. An additional attack vector that can lead to SMM compromise is the SMI handler which services a limited number of requests from the OS. From the hardware point of view, SMM privilege is a mixture somewhere between the BIOS and OS with a few extra bits included (e.g., exclusive access to SMRAM). The main difference for TDX is that SMM persists past the BIOS and is able to attack the TDX module and TDs that are running and potentially attested.

For this review, we did not focus in-depth on SMM but did confirm that many of the access controls that explicitly prevent BIOS and OS access also prevent SMM access. Additionally, we

[6] This is an area where future research may be useful to verify there are no gaps.

reviewed the SMM-only MSRs and uncore registers which appear to have effects on TDX security.

Malicious VMM

The host OS, hypervisor and VMM are responsible for kicking off TDX initialization, creating new TDs, and managing the lifecycle of these TDs. The host OS launches NP-SEAMLDR which triggers the measured installation of P-SEAMLDR, later used to trigger the measured installation of the TDX module. Once the system is initialized, the VMM manages the TD lifecycles through the TD Host (TDH) APIs; loading the initial guest image, configuring guest memory, assisting with attestation, scheduling vCPUs, and so on. These responsibilities lead the VMM to have the largest attackable interface into the TDX system. Additionally, the VMM is uniquely responsible for passing information between TDX and the SGX quoting enclave – compromising this chain of trust would remove the legitimacy of attestation.

Malicious TD/VM

Finally, the TDs running on the system are guaranteed to be under attacker control and have a unique interface into the TDX system. Legacy VMs do not run under the TDX module, and we did not identify any security concerns regarding their interactions with TDX, but there may be undiscovered issues given their sharing of resources and similar connections to cloud services. The TDs can interact with the TDX module directly through the `TDG.*` API calls and VM exit handlers triggered during sensitive operations such as MSR and control register access. Additionally, the TD is a full-fledged virtual machine and can configure the vCPU and memory in any number of unexpected combinations – the TDX module must be able to handle all of these safely.

MCHECK

For technologies where the BIOS is outside the TCB, such as SGX and TDX, the system requires a mechanism to ensure that the BIOS has configured all security sensitive settings to be within an acceptable range. Intel has developed the MCHECK firmware to provide this assurance and deliver the results to TDX in a way that is trusted. MCHECK is implemented as a non-persistent XuCode module and embedded within the CPU microcode update file. It is executed as part of the BIOS boot sequence. The entire microcode update, including the MCHECK XuCode program, is encrypted and signed by Intel. While this provides a trusted way for only Intel to execute microcode updates and MCHECK programs, it also results in a completely opaque security validation which TDX relies upon. Reviewing the MCHECK source code was outside the scope of this security review.

System Validation

While much of what MCHECK does is not made publicly available by Intel, for TDX there are a few known areas that it must validate before TDX will consider the system in a trusted state. First, MCHECK verifies that the physical DRAM memory has been configured correctly. This includes the absence of address aliasing and checking that security-relevant settings such as refresh timings and ECC are in expected ranges.

Additionally, MCHECK is responsible for verifying that DMA protections are enabled, DDR5 ECC is enabled, memory encryption is configured correctly, and MSR and uncore values are consistently programmed. For TDX, MCHECK derives the HMAC key using `RDRAND` and later used by the `SEAMREPORT` for attestation report generation for SGX verification during quote generation.

Security Concerns

MCHECK is an interesting area for security research since TDX (and SGX) rely on it to ensure the BIOS hasn't degraded system security in ways which may open up new exploit pathways. For example, if the BIOS were able to alias two physical memory addresses this would enable bypassing memory access controls such as the SEAMRR region where the TDX module and control structures are located.

Due to the lack of source code and that MCHECK's logic is encrypted, the main security concern we have for MCHECK is its reliance on security through obscurity. There are many edge cases that Intel has confirmed MCHECK validates, but a complete list of what it checks or how exactly these checks are performed is not available. The attack surface of MCHECK is somewhat limited and additionally it can only be executed while in the BIOS execution mode (i.e., before BIOS_DONE MSR is set). The BIOS populates a somewhat complex data structure that contains a feature bitmap, SGX information, TDX convertible memory ranges (CMRs), and memory topology information. This structure is passed to MCHECK and is the main attack

surface in which one might discover edge cases that are not covered or memory corruption bugs.

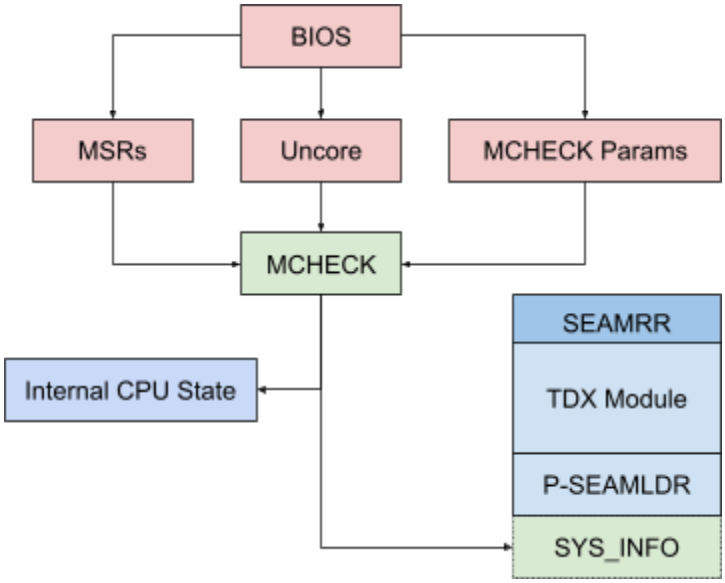

Figure 4: Diagram of MCHECK's inputs from the BIOS and output into SEAMRR

Outside of direct attack vectors, we additionally investigated the possibility of the BIOS using software-based fault injection attacks to cause MCHECK to misbehave[7]. There are a couple of design choices that make this attack challenging (MCHECK prevents any other CPU threads from running in parallel, and the OS_MAILBOX_INTERFACE from Plundervolt is disabled on Sapphire Rapids), but at least one window remains open. There is an alternative BIOS_MAILBOX_INTERFACE for sending commands to the power control unit (PUnit), this includes the ability to send raw SVID commands which result in voltage adjustments. Since there is a delay between when these commands are sent to the voltage regulator and when the voltage actually changes, an attacker might be able to drop the voltage right before executing MCHECK and have the voltage drop hit mid-execution. Evaluating this attack was beyond the scope of this review since we didn't have access to hardware.

MCHECK is Intel-trusted code which runs on the x86 cores and is foundational for ensuring the system has been securely configured before launching SGX or TDX. Outside of limited publications, Intel has provided no public comprehensive details on MCHECK's design and its implementation is a black box (neither plaintext binary nor source available). Based on our discussions with Intel engineers, MCHECK appears to validate and prevent many attacks. Intel also confirmed significant efforts on their part to review and validate this module; however, we could not verify the robustness of these checks ourselves. We strongly encourage Intel to publish the MCHECK source code to enable third party review.

[7] See https://plundervolt.com/ for similar attacks on SGX

Non Persistent SEAM Loader

Given that the startup BIOS code is outside the TCB for Intel TDX, there needs to be a method for dynamically establishing a root of trust on which the rest of the TDX infrastructure can be loaded. Intel has solved this problem by leveraging the existing Intel TXT and Authenticated Code Module (ACM) technologies to create a new ACM named Non Persistent SEAM Loader (NP-SEAMLDR). In this design, the OS loads NP-SEAMLDR which validates the system configuration, installs the Persistent SEAM Loader (P-SEAMLDR) into the SEAMRR memory region, and returns control to the OS. The OS can then interact with the trusted P-SEAMLDR to install a signed TDX module into the SEAMRR memory region. Finally, the OS interacts with the trusted TDX module to initialize TDX and manage the TD lifecycle.

More details about NP-SEAMLDR and P-SEAMLDR can be found in the SEAMLDR Interface Specification and furthermore Intel has open sourced the code.

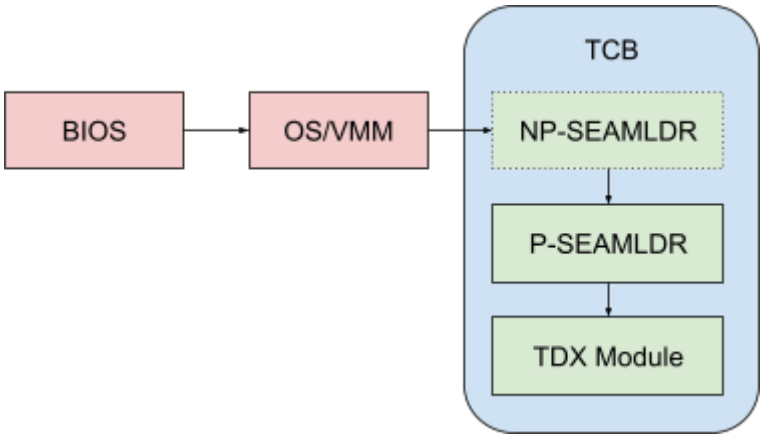

Figure 5: Establishment of trust during TDX initialization

An ACM consists of a binary blob of x86 executable code and a header which includes an RSA signature that covers the entire binary except for a scratch section. The Intel SMX `GETSEC[ENTERACCS]` instruction is used to load and execute an ACM and `GETSEC[EXITAC]` is used within the ACM to return to the caller. By design, NP-SEAMLDR executes entirely within the internal CPU cache on the primary core (BSP) and requires all other cores on the socket to be idle (i.e., in the wait-for-SIPI state). This reduces the attack surface to preclude system modification while the ACM is running by forcing the CPU into a single-threaded state.

After executing `GETSEC[ENTERACCS]` and validating the RSA signature, the x86 core switches into 32-bit mode and jumps to the entry point specified in the header. Additionally, the core sets an internal flag which indicates that the processor is now executing in AC mode. One privilege unlocked by entering AC mode is the ability for the CPU to directly access the SEAMRR memory region which is otherwise blocked for BIOS, OS, and SMM execution modes. NP-SEAMLDR requires this access in order to install the P-SEAMLDR binary (which is encapsulated within the NP-SEAMLDR binary) into SEAMRR.

Given that NP-SEAMLDR establishes the trust which the remainder of TDX relies upon, any vulnerabilities here can lead to a cascading compromise of the entire system.

Threat Model

The overall TDX threat model states that all code outside of the TDX chain of trust is outside the TCB and thus this is the code which can attack NP-SEAMLDR. This includes all ring 0 code on the system; however, NP-SEAMLDR prevents calls from the BIOS progressing beyond the entry code. Additionally, guest VMs and TDs unconditionally trigger a VM exit for all `GETSEC` instructions and therefore can't attack this interface. This leaves the BIOS (very limited), OS/VMM, and SMM as attack vectors into NP-SEAMLDR.

The interface into NP-SEAMLDR is relatively small, but the attacker has uniquely broad control over the environment in which it runs. There is an explicit ABI which includes 6 general purpose registers to pass arguments which are used either by uCode during load or x86 code within the payload:

- **R9**: GDT base to be established when returning to the OS
- **R10**: RIP where control is transferred when returning to the OS
- **R11**: CR3 value to be established when returning to the OS
- **R12**: IDTR base value to be established when returning to the OS
- **EBX**: NP-SEAMLDR ACM physical address base
- **ECX**: NP-SEAMLDR ACM size

There is also the implicit interface which includes the machine configuration and system register contents. The BIOS has broad privileges to configure all parts of the SoC, including the uncore registers and CPU MSRs. Some of these controls may have effects on the ACM mode transitions and others are parsed by the ACM code itself. Both the implicit and explicit attack surfaces were covered as part of this review.

The NP-SEAMLDR binary protects itself from exploitation in a few different ways. First, the uCode for GETSEC[ENTERACCS] masks all external interrupts such as NMIs and SMIs while also disabling hardware breakpoints. Next, software exceptions are inhibited by setting the IDTR limit to zero which leads to any exception causing a triple fault and system shutdown. Additionally, stack canaries are used but Intel CET control flow integrity and shadow stacks are not used (but are in P-SEAMLDR and the TDX module). Lastly, the binary is loaded at a known virtual address and no ASLR is applied, unlike P-SEAMLDR and the TDX module which have ASLR.

Attestation and Rollback Prevention

Because NP-SEAMLDR is the root-of-trust for all Intel TDX code, it is critical that customers can verify the system booted their TD using only the latest version of NP-SEAMLDR. If an older, vulnerable version ever loaded it may be possible to corrupt the system's integrity given an

ACM's elevated privileges. Through a combination of hardware registers, SGX, microcode, and NP-SEAMLDR code, the system enforces these properties:

Attestation: The end user must be able to cryptographically verify the **lowest**-SVN NP-SEAMLDR that has executed during the boot cycle. This is accomplished through the next two properties.

Anti-rollback: For every NP-SEAMLDR executed during a boot cycle, only NP-SEAMLDRs of equal or greater SVN may execute later within the same boot cycle.

Anti-spoofing: The recorded SVN for NP-SEAMLDR must never increment.

In combination, these properties provide confidence that the SVN recorded (which is used to generate the attestation report and signed quote) indicates the lowest versioned NP-SEAMLDR executed during the current boot cycle. The anti-rollback protection is implemented in the `GETSEC[ENTERACCS]` microcode which checks the ACM header's SVN against the value stored in `BIOS_SE_SVN.SEAMLDR_SE_SVN` – if this value is lower than the previously recorded version a TXT shutdown occurs. On first execution, the `SEAMLDR_SE_SVN` value has not yet been written and execution is allowed.

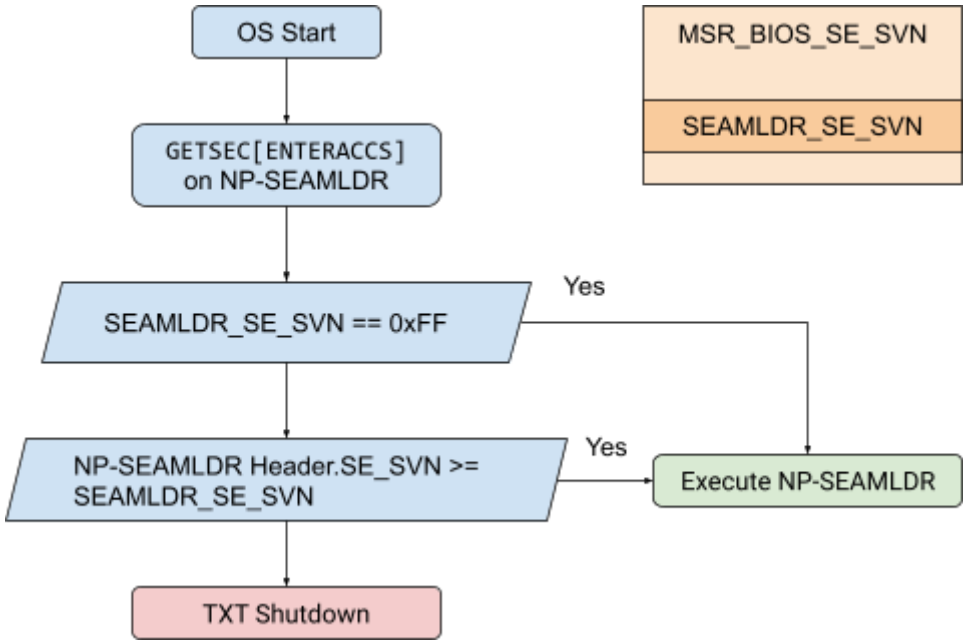

Figure 6: Anti-rollback protection for NP-SEAMLDR, implemented in ENTERACCS microcode

The anti-spoofing protection prevents a compromised low-SVN NP-SEAMLDR from stating that it was in fact a higher-SVN module which ran. This is implemented in the microcode for WRMSR when writing to the `BIOS_SE_SVN` register and depends on internal SGX state. As soon as the first non-faulting SGX instruction executes on the system, all SVNs in `BIOS_SE_SVN` are locked – this enables SGX to report an accurate minimum security state in attestation. If no SGX

instruction has been executed, then the SVN variables are only allowed to be decremented (and default to 0xFF on reset). SVN's are part of the integrity/authentication controls over Intel issued blobs.

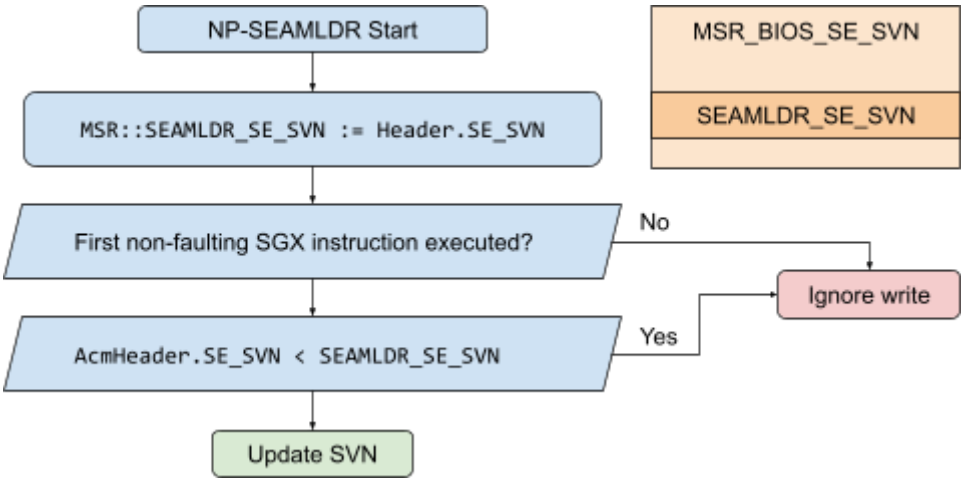

Figure 7: Anti-spoofing protection for NP-SEAMLDR SVN, implemented in WRMSR microcode

Security Concerns

So far, the focus of this section has been on the specific NP-SEAMLDR ACM; however, due to the overall system design, we must also pay attention to all other ACMs which are loadable on the system. Previously, it was mentioned that when the x86 core transitions into AC mode it now has elevated privileges, including the ability to access SEAMRR protected memory. This privilege is not unique to NP-SEAMLDR and in fact applies to all ACMs on the system. This universal application of privileges significantly widens the attack surface for compromising the TDX root of trust. In addition to NP-SEAMLDR, the following ACMs are also known to be loadable on Sapphire Rapids CPUs:

- **Alias Checking Trusted Module (ACTM)**: New ACM for DRAM configuration validation
- **BIOS ACM**
- **BIOS Guard**
- **SINIT**

A vulnerability in any of these ACMs could lead to the same kind of TDX system compromise as previously discussed above. Previous research has discovered multiple vulnerabilities in some of these ACMs. Reviewing each of these modules was outside the scope of this review, but remains an interesting area for future research.

Within NP-SEAMLDR, there are several areas where generic defense-in-depth strategies could add extra hurdles for exploitation. As discussed above, the memory layout of the module is known to the attacker and ASLR of this space (at least for the 64-bit portion) would make attacks more challenging. Similarly, Intel CET features are enabled for P-SEAMLDR and the TDX module but not NP-SEAMLDR. Lastly, the `GETSEC[ENTERACCS]` instruction disables CR0.WP which enables a memory write primitive to overwrite read-only memory such as the

code itself. After raising this concern with Intel, the NP-SEAMLDR code was updated to enable CR0.WP during the 32-to-64-bit transition.

Security Vulnerabilities

While reviewing NP-SEAMLDR, we discovered a variety of vulnerabilities in the pre-release code, some of which are described below. Each of these vulnerabilities have been fixed and verified in the current release.

Unsafe Performance Monitoring VMCS Configuration

By design, the core performance monitors should be disabled while executing in P-SEAMLDR or the TDX module. TD guests should also have these features disabled unless the guest is created with the attestable `ATTRIBUTES.PERFMON = 1` or `ATTRIBUTES.DEBUG = 1` values set. The reason for this is twofold: first, to prevent information leakage of TDX and TD secrets to the host; and second, to prevent host control over performance features which write records during TDX execution.

These performance monitoring controls are configured by writing to CPU MSRs such as `IA32_PERF_GLOBAL_CTRL`. In order to prevent these configurations from persisting from host execution into TDX or TD execution, there are VMCS entry and exit controls which direct the CPU to context switch on transition.

For the TDs, the TDX module is in charge of configuring the VMCS and will set the entry/exit controls based on the `ATTRIBUTE` bits described above. For the TDX module, a similar method is used where the transfer VMCS (taken from the STM design) is configured such that `IA32_PERF_GLOBAL_CTRL` is always context switched. There is one transfer VMCS per physical CPU, each of which are located within the TDX module and configured by P-SEAMLDR during the install command. P-SEAMLDR follows a similar design to the TDX module, but instead has a single transfer VMCS which is installed by NP-SEAMLDR.

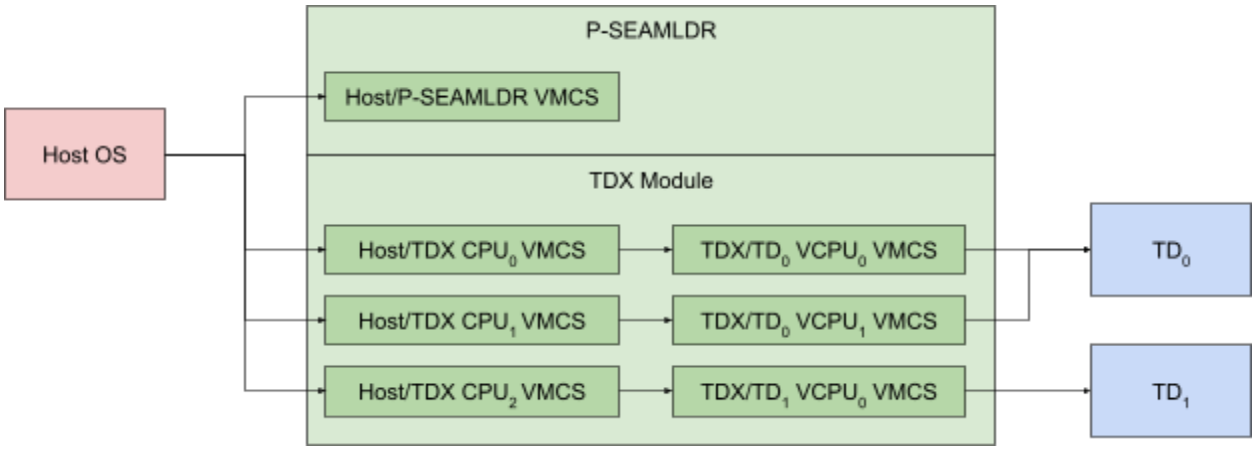

Figure 8: Example VMCS association with 3 physical CPUs and 2 TDs (2 vCPU, 1vCPU)

While reviewing how performance monitors are context switched as described above, a discrepancy was discovered between the P-SEAMLDR transfer VMCS and TDX module transfer VMCS. In P-SEAMLDR, the VM-exit control[8] for saving `IA32_PERF_GLOBAL_CTRL` was set but the similar control for *loading* the MSR was cleared. This results in the host MSR value persisting into P-SEAMLDR's execution. The resulting effect is that core performance counters continue incrementing while P-SEAMLDR is executing and if any of them rollover then a performance monitoring interrupt (PMI) will occur. P-SEAMLDR doesn't contain any secret material, so using the performance monitors to leak information was not relevant. However, the system can be configured such that a PMI triggers the CPU to write information about the event into memory at a user-specified address. For example, the Processor Event-Based Sampling (PEBS) feature can be programmed via the IA32_DS_AREA MSR to write such data on a PMI. The result is that a malicious host OS can configure the PMUs and PEBS such that a PMI occurs during P-SEAMLDR execution which then writes semi-controlled data into an arbitrary address. Given that P-SEAMLDR is responsible for authenticating and loading the TDX module, this vulnerability could lead to a full TDX compromise.

Variant Analysis

To proactively discover similar issues where the P-SEAMLDR and TDX transfer VMCS differ, Intel dumped both of these structures in a test environment and searched for any unexpected differences. Additionally, the VMCS configuration code was reviewed with a focus on finding similar edge cases where performance monitors could be enabled.

Remediation

We did not attempt to exploit this vulnerability due to the fact that an ASLR defeat would also be required to determine the base of P-SEAMLDR. Since NP-SEAMLDR is responsible for initializing the transfer VMCS for P-SEAMLDR, the bug is in NP-SEAMLDR but only affects P-SEAMLDR. This issue was fixed by configuring the VMCS to load the IA32_PERF_GLOBAL_CTRL MSR on VM exit (VMM to P-SEAMLDR transition).

Exit Path Interrupt Hijacking

Previous research has highlighted the importance of ensuring that trusted-to-untrusted domain transitions are complete and well understood by both sides. For TDX, the transitions into NP-SEAMLDR, P-SEAMLDR, and the TDX module must be complete and not implicitly trust any attacker-controlled data that may be present in system registers. Similarly, the transition back to the VMM or TD must ensure that the context switch is complete and no sensitive TDX state remains in system registers. This section describes a vulnerability that was discovered in the NP-SEAMLDR where attacker-controlled content was implicitly trusted during a short window during the exit transition.

[8] For the P-SEAMLDR and TDX module transfer VMCS, the transition from host OS to P-SEAMLDR/TDX is considered a VM-exit while the transition back is considered a VM-entry.

As mentioned in the overview of NP-SEAMLDR above, the ACM code masks all external interrupts and translates software exceptions into a system shutdown. External interrupts are masked throughout ACM execution due to configuration registers which are set by the `GETSEC[ENTERACCS]` microcode on entry. For software exceptions however, the ACM entrypoint x86 code disables interrupts by quickly reconfiguring the interrupt descriptor table (IDT) to point to a null descriptor. This effectively makes any software fault (e.g., page fault, general protection, …) cause the hardware to fault again when performing the interrupt lookup, leading to a triple fault which finally leads to system shutdown. Conversely, on the ACM exit path the original IDT descriptor is restored before returning to the host via the `GETSEC[EXITAC]` instruction.

```
AcmEntryPoint PROC NEAR
  nop
  nop
  nop

  ; Right after ENTERACCS, SEAMLDR 32-bit assembly code will do the following:
  ; SIDT saved_OS_IDTR
  ; LIDT null_IDTR // a 48-bit variable that contains 0's

  sidt    fword ptr ds:[ebp + stackStart + 4*6]

  ; Make sure that Null IDTR is actually zero
  mov     dword ptr ds:[ebp + stackStart + 4], 0
  mov     dword ptr ds:[ebp + stackStart + 4 + 4], 0
  lidt    fword ptr ds:[ebp + stackStart + 4]          ; Load NULL IDTR
```

32-bit entry code for NP-SEAMLDR

```
  lidt    FWORD PTR [rcx].SEAMLDR_COM64_DATA.NewIDTR   ; Load attacker IDTR
  lgdt    FWORD PTR [rcx].SEAMLDR_COM64_DATA.OriginalGdtr

  ;; <truncated for report>

DoExitAC:
  ;; Parameters for EXITAC

  ; uCode restores the RIP from RBX during EXITAC
  mov     rbx, QWORD PTR [rcx].SEAMLDR_COM64_DATA.ResumeRip
  ; uCode restores the CR3 from R8 during EXITAC
  mov     r8, QWORD PTR [rcx].SEAMLDR_COM64_DATA.OriginalCR3
  ; SEAMLDR Error code is reported in R9
  mov     r9, QWORD PTR [rcx].SEAMLDR_COM64_DATA.RetVal
  ; Clear all flags
```

```
mov     rdx, 0
; Do ExitAC
mov     rax, EXITAC
push    2
popfq

;; Clear other registers as described in spec - not xor to avoid changing flags
mov     rcx, 0
;; <truncated for report>

GETSEC[EXITAC]  ; drop privileges and return to host x86 code
```

64-bit exit code for NP-SEAMLDR

From an attacker's perspective, this presents an interesting window: there is a point of time shortly after ACM entry and shortly before ACM exit where the host's IDT is still configured. **If an exception can be forced to occur within these windows the attacker can gain control over RIP while in privileged AC mode.**

On the entry path, the attacker has no influence over the instruction operand values or the EBP base register (this points to the base of the ACM) which is dereferenced while the attacker's IDT is still active. However, on the exit path, the code restores the host state before executing `GETSEC[EXITAC]` which completes the transition back to regular x86 mode. In this path, we noticed that if a non-canonical GDT base address is passed to the ACM then the context switch which executes LGDT will raise a #GP exception. At this point, we wrote a quick proof of concept to demonstrate control of RIP while still in AC mode.

Exploitation

While the attacker does have control of IDTR, there are some environmental restrictions that must be overcome. Critically, the ACM only has its own module image mapped into memory after transitioning to 64-bit mode which means an attacker can't place their malicious IDT outside of the ACM image or it will #PF and triple fault. Since the ACM image is signed and verified, this means that an attacker also can't modify the signed image to inject an IDT.

Luckily for the attacker, there is a portion of the ACM image which is unsigned – the scratch space. For NP-SEAMLDR, this space is 832 bytes long which is plenty of room for the required IDTR and IDT descriptor. In Simics[9], this scratch space is still accessible in ACRAM and still contains the modified values; however, on real hardware a subset of this space is used for RSA calculations.

[9] Due to hardware access limitations during this early technology review, we relied on the Simics simulation environment for dynamic testing.

A proof of concept was created as a UEFI application that runs from the UEFI shell in Simics with the SPR module installed.

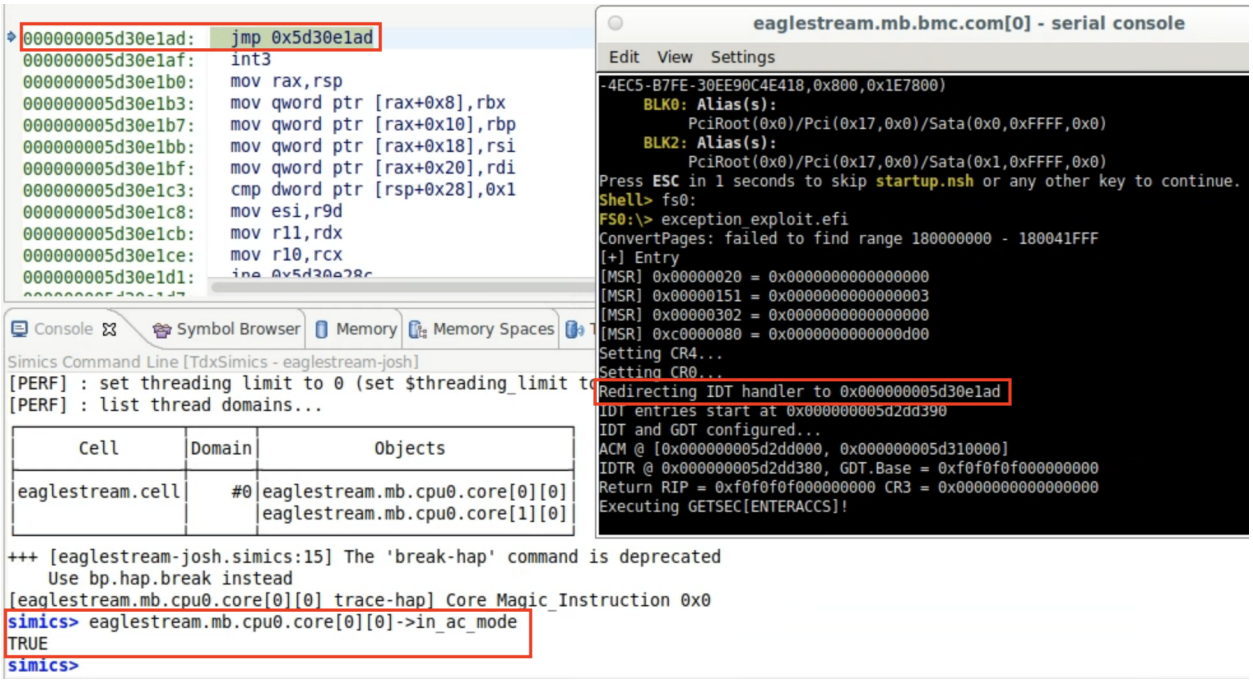

Figure 9: Demonstration in simulation of RIP control while in privileged AC mode

Variant Analysis

After this vulnerability was discovered, we continued to look for any other method for causing an exception during these execution windows. One additional variant was discovered in the final instruction which executes, GETSEC[EXITAC]. This is a complex, microcoded instruction that implements the context switch from AC mode back to regular x86 mode. Looking through the Intel specification for the operations performed by this instruction show several opportunities for causing an exception, although most are not possible given the constraints.

However, one condition is possible to reach given the original code. The NP-SEAMLDR ACM needs to know where to return execution to on exit and in this design, the host OS originally passes the return address during `GETSEC[ENTERACCS]` via the R10 register. Similar to the original issue, an attacker can specify a non-canonical value for R10 which will cause a #GP exception to occur on execution of `GETSEC[EXITAC]`. Critically, this exception occurs **before** the transition out of AC mode, so the attacker retains this privilege.

```
ELSIF (
    (in VMX operation) or ( (in 64-bit mode) and ( RBX is non-canonical ) )
    (CR0.PE=0) or (CPL>0) or (EFLAGS.VM=1) or
    (ACMODEFLAG=0) or (IN_SMM=1)) or (EDX ≠ 0))
THEN
```

```
  #GP(0);
... // segment cut for brevity
ACMODEFLAG := 0;
```

Portion of GETSEC[EXITAC] operation listing

Other ACMs have a similar return path where caller state is restored before executing `GETSEC[EXITAC]`. As far as we know, all these ACMs run in 32-bit mode so the specific issue concerning non-canonical addresses is not applicable; however, there may be alternative ways to trigger an exception after IDTR has been loaded[10]. We tested many theories locally using Simics and worked with Intel engineers to test some of these experiments on SPR hardware. Compromise of any ACM leads to execution in a highly privileged mode which can impact TDX and platform security.

Remediation

Intel fixed both variants of this vulnerability in the 1.0 release. The original exception path during `LGDT` was fixed by moving the LIDT instruction directly before the register clearing and `GETSEC[EXITAC]` instruction. The non-canonical return address exception path was mitigated during entry by verifying the requested return address is actually canonical.

While the vulnerability is fixed in the latest signed version of NP-SEAMLDR, there must also be mechanisms in place to prevent (or detect) the older vulnerable versions from being loaded. There is no persistent revocation list for TDX, so older versions of NP-SEAMLDR can always be loaded on supported hardware; however, the version loaded is securely stored and later used during attestation[11]. To get around this attestation artifact, an attacker might try first loading a low-version vulnerable NP-SEAMLDR, then compromising the TDX boot chain, and finally loading a high-version NP-SEAMLDR before attestation occurs. This technique is thwarted by the anti-spoofing mechanism described in the previous section. Therefore, a TDX customer can trust that only a specific range of NP-SEAMLDR modules were loaded before provisioning secrets to their TD.

Mitigating controls

The NP-SEAMLDR has a set of defensive measures that make exploitation harder. This includes the following:

- **Constrained execution model**: The `GETSEC[ENTERACCS]` instruction used to launch NP-SEAMLDR requires all logical processors in the socket be in the WAIT-FOR-SIPI

[10] For example, 32-bit ACMs can #GP on GETSEC[EXITAC] if the return address is above the CS limit. However, in these ACMs the return address is not arbitrary and instead is set to the address of the instruction following GETSEC[ENTERACCS].
[11] Included as CPUSVN by the SEAMOPS[SEAMREPORT] instruction executed from the TDX module.

state (not able to execute instructions). Additionally, external interrupts and debug features like hardware breakpoints are disabled during NP-SEAMLDR execution.

- **Fail-closed**: Exceptions cause a triple fault and a machine shutdown.
- **Heap**: No dynamic heap allocations prevent the risks commonly associated with bad object management, such as use-after-frees.
- **Extremely small attack surface:** There is almost no user input to NP-SEAMLDR, mostly just CPU state to restore on the exit path. Additionally, this input is only given on entry with no post-launch command handling.
- **Secrets**: NP-SEAMLDR, by design, does not hold or process any secret key material.

Persistent SEAM Loader

Overview

The persistent SEAM loader, or P-SEAMLDR, is responsible for **authenticating**, **loading** and **measuring** the TDX module. It is embedded in the NP-SEAMLDR binary and is loaded dynamically to the top of SEAM range by the NP-SEAMLDR ACM.

NP-SEAMLDR sets up the environment for the P-SEAMLDR: copies code and data pages, initializes stack pages, configures page tables that translate linear addresses to physical addresses, and configures the VMCS required to enter SEAM root-mode (figure 10).

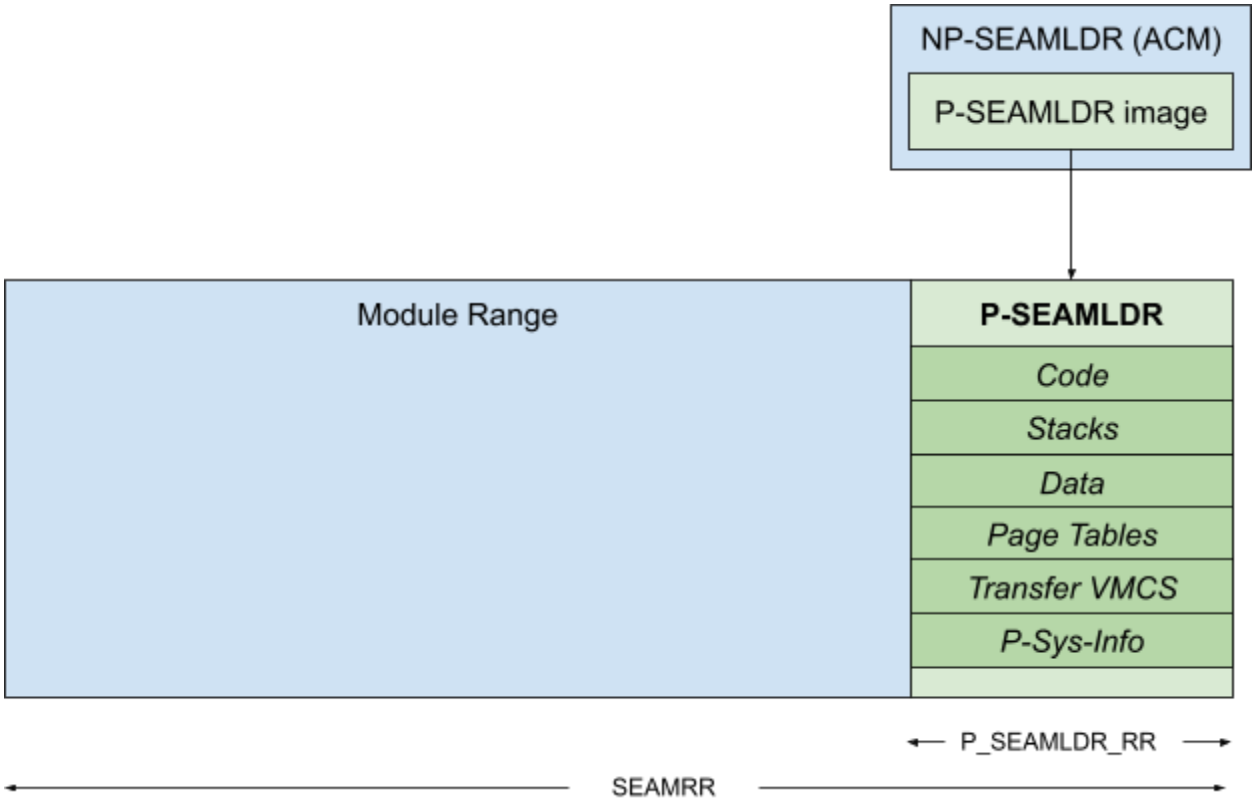

Figure 10: SEAM setup after NP-SEAMLDR

SEAM state is captured in a special platform-scope register (only accessible by uCode or SEAM code) called CR_SEAMEXTEND. It includes the following fields:

P-SEAMLDR Ready	Flag indicating P-SEAMLDR is loaded successfully in SEAM range. CPU uCode checks this flag before entering SEAM root-mode on a SEAMCALL instruction.
P-SEAMLDR Mutex	Lock for entering P-SEAMLDR. Acquired on SEAMCALL, released on SEAMRET. CPU enters P-SEAMLDR only when this lock is clear.
SEAM SVN, Late SE SVN	“Security Version Numbers” captured during load. These are included in the attestation report.
SEAM Ready	Flag indicating TDX module is loaded successfully in SEAM range. CPU uCode checks this flag before entering SEAM root-mode on a SEAMCALL instruction.
SEAM Under Debug	Flag indicating system is under debug. A system under debug does NOT produce valid attestations as SGX signing enclave is loaded with non-production keys.

Note how the P-SEAMLDR mutex forces a single-threaded execution model for the loader.

The loader exposes a set of APIs to install and shutdown the main TDX module. The shutdown operation also clears the “P-SEAMLDR ready” flag in CR_SEAMEXTEND, thus allowing P-SEAMLDR reinstallations and upgrades.

Install initiation

The installation process is designed to run serially on all logical processors (LPs), where the work is done on the last LP that invoked P-SEAMLDR’s install API.

A two-step process is designed to ensure that no other LPs are running in SEAM mode while a module installation is in process: 1) The loader clears the “SEAM Ready” flag in CR_SEAMEXTEND. This blocks LPs from entering SEAM mode. 2) The loader tracks which LPs have called the Install API. Installation starts only when the bitmap is full, meaning all LPs have called P-SEAMLDR. Note that P-SEAMLDR cannot race with itself (P-SEAMLDR Mutex), and once an LP calls P-SEAMLDR it cannot SEAMCALL into a previously loaded TDX module (SEAM Ready is cleared).

TDX module authentication

The P-SEAMLDR accepts a seamldr_params argument that points to the TDX module binary in physical memory, and a seam_sigstruct parameter, also called a manifest, that authenticates the TDX module binary.

Authentication is as follows:

1. Loader authenticates the **RSA verification key**: SHA-384 of SIGSTRUCT.MODULUS is equal to a hard coded constant, INTEL_SIGNER_KEY_HASH, embedded in P-SEAMLDR.
2. Loader authenticates the **manifest**: manifest fields have the correct signature under the RSA-3072 modulus key. Signature scheme is EMSA-PKCS1-v1.5 with SHA-384 message digest.
3. Loader authenticates the **module**: module's SHA-384 digest matches the expected digest listed in the manifest.

The chain of trust from the NP-SEAMLDR to the dynamically loaded TDX module is preserved.

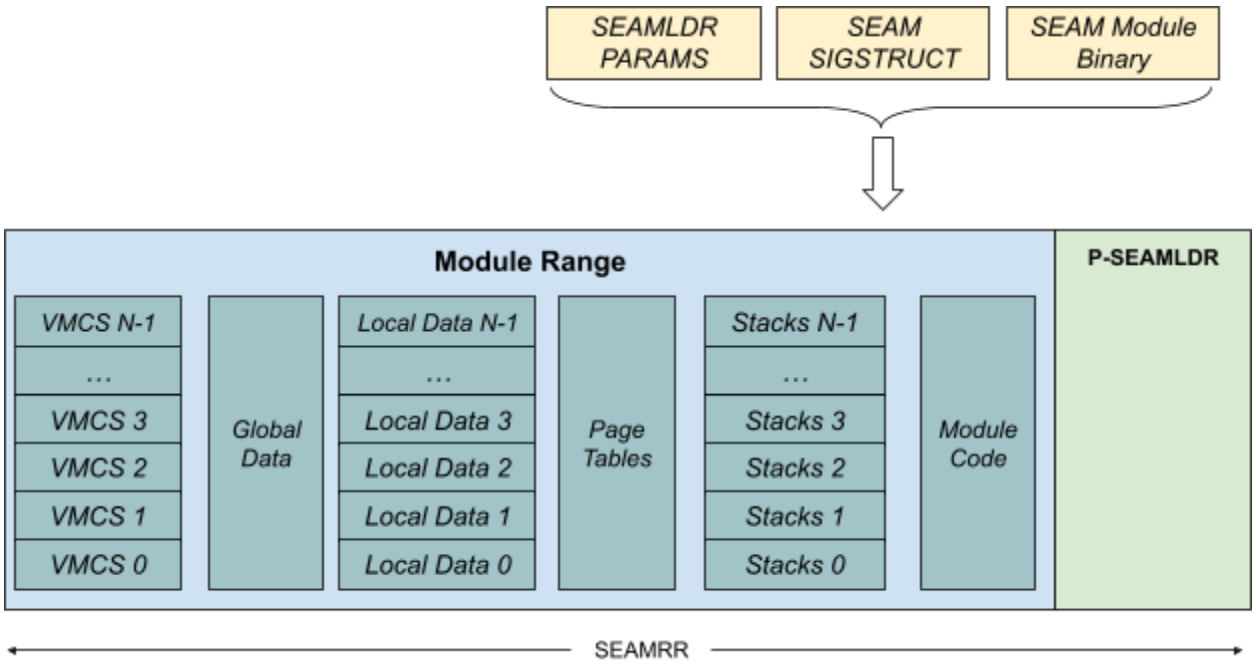

Figure 11: SEAM setup after P-SEAMLDR

We reviewed the authentication scheme for correctness and security: we confirmed the signing key is used for a single purpose, message parsing is unambiguous, and only strong, well-known primitives are being used. Sigstruct and module binary are copied from host memory to private SEAM range before being authenticated, so there's no risk of TOCTOU. Length values are sanity checked.

Finally, we tested the implementation - Intel Integrated Performance Primitives Cryptography library, or IPP - using Wycheproof test vectors. We confirmed the library correctly handles edge cases in the RSA verification code, and doesn't have implementation issues parsing the DER encoded digest.

Module installation

Similar to how the NP-SEAMLDR sets up the execution environment for the P-SEAMLDR, the latter does the same for the TDX module. The loader copies the module's code and data pages to SEAM range, prepares local data and stack regions for each of the N logical processors, prepares the N transfer VMCS, and configures the page tables. On successful installation, the loader stores the module's measurement (*sigstruct.seamhash*) in CR_SEAMEXTEND, and sets the "SEAM Ready" flag to True.

Page tables "keyhole" mechanism

The P-SEAMLDR constructs page tables that are, for the most part, **static**. The page tables map linear addresses to physical addresses such that when the module runs in 64b protected-mode, it has access to its own code and data pages. Since the mappings are static, the page tables are global, and can be safely used concurrently by different LPs.

There are flows in the module that require **dynamic** mappings. For instance, when the module reads input arguments from host physical addresses, it first needs to map it to its virtual address space. In order to support dynamic mappings, the loader reserves a region - a page table directory - for this purpose. This region, also called a "keyhole", is mapped with RW permissions to the module's address space. At runtime, the module constructs page table entries (PTE) in the keyhole page, and uses the appropriate linear address to access outside host memory. Each LP gets a dedicated keyhole space for its own dynamic mappings. This prevents race conditions between LPs.

A similar mechanism exists for the P-SEAMLDR. The NP-SEAMLDR constructs static page tables with a RW keyhole for P-SEAMLDR dynamic mappings.

We thoroughly reviewed the page table configuration and TLB management in both the loader and the TDX module. We verified they properly manage the TLB during dynamic mappings: the TLB is flushed using INVLPG if a cached entry is being reused. We identified that a crucial piece of TLB management - TLB shootdown - is missing in the code. However, we confirmed with Intel engineers that this is **safe**. Each logical processor has dedicated keyholes with a distinct set of linear addresses, so TLB and other paging structure caches are always coherent across the LPs, and there's no need for a shootdown.

Misconfiguration bugs

We identified an implementation bug in how the NP-SEAMLDR constructed the P-SEAMLDR page tables. The P-SEAMLDR layout is as follows:

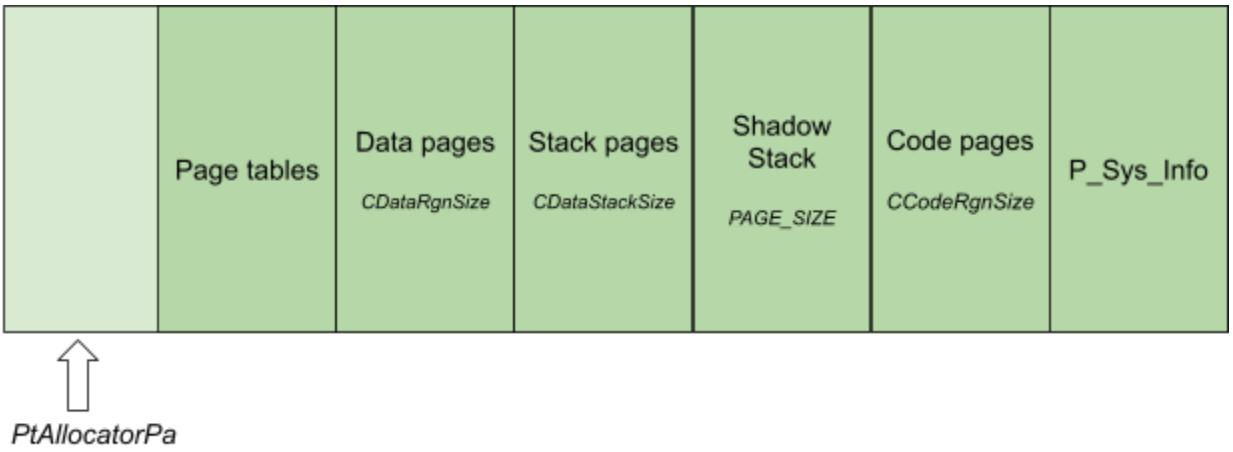

Figure 12: P-SEAMLDR layout

An internal *MapPage()* adds new static mappings and "grows" the page table to the right. It tracks the current page table size using the *PtAllocatorPa* variable. On entry, the function performs a sanity check that there's sufficient space remaining to add the new mapping. The check verifies the updated PtAllocatorPa does not overlap with the data region:

```
// if the allocator reached the data region - error
if (SeamrrPtCtx->PtAllocatorPa >=
     SeamldrData.SeamrrBase + SeamldrData.SeamrrSize -
    (C_P_SYS_INFO_TABLE_SIZE +
            SeamldrData.PSeamldrConsts->CCodeRgnSize +
            SeamldrData.PSeamldrConsts->CDataStackSize +
            SeamldrData.PSeamldrConsts->CDataRgnSize)) {
    return NULL;
}
```

Notice that the shadow stack size is not accounted for, and *MapPage()* computes an incorrect offset for the data region. If *PtAllocatorPa* crosses over the data region, MapPage builds valid page table entries that overlap with data variables. In an extreme case, P-SEAMLDR operations could overwrite valid PTEs, and potentially point them to attacker controlled data. This was fixed by adding the shadow stack size to the expression.

A second finding was in how P-SEAMLDR translated virtual addresses (VA) to physical addresses (PA) by subtracting the data region base address:

```
uint64_t offset_in_data_region = va - st_p->data_rgn_base;
// Set by NP-SEAMLDR to C_DATA_RGN_BASE | SeamldrData.AslrRand (0xFFFF800300000000)
```

This computation is correct for VA pointing at variables in the data region. However, there were flows that passed pointers to the stack region, for instance *seamldr_info()*:

```
ALIGN(256) seamextend_t       seamextend;
...
seamextend_read(&seamextend);
```

This computation `va - st_p->data_rgn_base` "blows up", since the stack's base address is set to C_STACK_RGN_BASE (0xFFFF800100000000), a much smaller value than C_DATA_RGN_BASE (0xFFFF800300000000). The fix uses a temporary buffer on the stack.

Mitigating controls

The P-SEAMLDR has a set of defensive measures that make exploitation harder. This includes the following:

- **Constrained execution model**: P-SEAMLDR mutex guarantees a single threaded execution model - the loader cannot race with itself. Furthermore, interrupts and NMIs are inhibited, so unexpected code paths are not allowed during P-SEAMLDR operation.
- **Fail-closed**: exceptions cause a triple fault and a machine shutdown.
- **State**: bitmap tracking logical processors guarantees installation sessions are serialized.
- **Heap**: no dynamic heap allocations prevent the risks commonly associated with bad object management, such as use-after-frees.
- **Input validation**: loader performs extensive input validation. Furthermore, data is copied from host memory to SEAM range before being validated.
- **ASLR**: NP-SEAMLDR randomizes the base virtual address for code and data regions of the P-SEAMLDR.
- **CET**: control-flow enforcement is enabled. This feature uses shadow stacks and indirect branch tracking, and blocks return/jump-oriented programming attacks.
- **Mappings**: most page table mappings are static. Dynamic mappings go through per-LP keyholes. TLB is flushed on SEAM transitions, and on new mappings. Entries are marked as user-owned which in combination with **SMAP** prevents an arbitrary write from modifying these keyhole mappings.
- **Secrets**: P-SEAMLDR, by design, does not hold or process any secret key material.
- **Side channels**: P-SEAMLDR enables mitigations against speculation based side channel attacks.
- **Host MSRs**: transfer VMCS masks MSRs that are controlled by the host VMM.

TDX Module

The Intel TDX module is the central privileged software component for running confidential VMs (called TDs). P-SEAMLDR installs the module into the protected SEAMRR memory range and runs in SEAM Root Mode giving it full access to the host OS and all TDs. It is responsible for creating and managing Trust Domains and enabling communication between the host VMM and TDs, while enforcing access controls. A detailed specification and source code are publicly available.

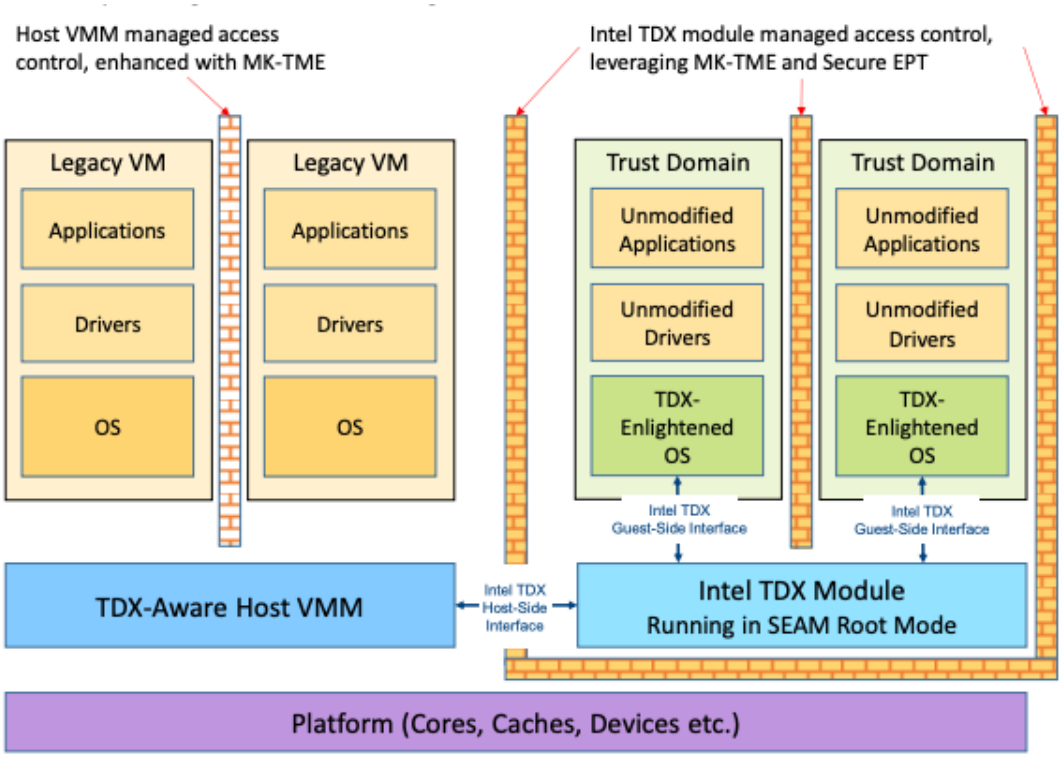

Figure 2.1: Intel® Trust Domain Extension Components Overview

Figure 13: TDX Components

The TDX module provides host-side and guest-Side APIs. The host calls into the TDX module by using the SEAMCALL instruction, the TD triggers its API using TDCALL.
The host-side interface can be used to initialize and manage the TDX module state, configure TDs and manage their private memory regions.
The guest-side interface is smaller and offers functionality for runtime attestation and measurement, and a hypercall mechanism to enable TD->VMM communication.

A core responsibility of the TDX module is physical memory management. Both TDX internal control structures and TD private memory are stored in physical memory ranges which are configured by the host VMM. This is done by configuring so-called Trust Domain Memory Region (TDMRs) which describe the physical memory space usable by TDX. Internally, the TDX module manages metadata for all used memory pages in a data structure called Physical

Address Metadata Table (PAMT). PAMT entries describe the owner and type of each physical page and are also used as part of TLB tracking.

Attack Surface

All code outside of the TDX chain of trust can be a threat to the TDX module. At a high level we can differentiate between attackers that have some level of control over the host (e.g. malicious administrators or a compromised BIOS) and attackers that only control one or more guests.

Malicious TDs

The TDX-specific attack surface reachable by a malicious TD is small but important. A malicious guest should not be able to negatively influence the hosts or other guests.The main attack surface in the TDX module is the handling of VM exits and TD calls as implemented in the src/td_dispatcher directory.

Malicious Host

TDX needs to defend against malicious or compromised hosts. While all components outside of TDX are untrusted, it still makes sense to distinguish between attacks that only require OS/VMM control, attacks that additionally require a SMM compromise and attacks involving the BIOS.
A Cloud Service Provider that wants to use TDX to protect against malicious OS administrators, would not consider attacks requiring a compromised BIOS critical as long as BIOS integrity can be verified.

The largest attack surface of the TDX module is the SEAMCALL API interface implemented in the src/vmm_dispatcher directory. In addition, several system wide components can be configured and manipulated by a malicious VMM, BIOS or SMM to indirectly impact the secure operation of the module (see the MSR and uncore sections).
Naturally, a malicious VMM can spin up an arbitrary number of cooperating malicious TDs so issues that involve attacks from both sides of the virtualization stack need to be considered as well.

Security Review

As the TDX module is a large and low level C codebase, much of our review concentrated on typical C language issues like temporal and spatial memory safety, integer truncation or overflows, concurrency issues and correct error handling.

In addition, we looked for application specific issues that could break TDX's security invariants. These include:

- Access to VMM controlled pointers. The TDX module needs to ensure that these point to VMM-owned or shared memory.

- Validation of TD VMCS state. Depending on the TD configuration (debug vs. non-debug), the VMM has limited control over parts of the TD VMCS configuration. The TDX module needs to ensure that all VMM controlled VMCS fields are sanitized. For example, VMM controlled physical pointers in the VMCS need to point into shared memory and be correctly aligned.
- Physical Memory invariants. The TDX module uses physical memory pages offered by the VMM to store both TD guest data, internal metadata and the secure EPTs of all TDs. Special care must be taken to ensure that a single physical page is always used in a single context and no “type confusions” occur. Pages that are reused in other contexts need to be correctly reinitialized. Finally, TLB tracking needs to ensure that no stale mappings are left over in the TLB.

While we did discover some issues during the review, we were impressed with the overall quality of the codebase. The APIs exposed to the VMM and TD are small, well-designed and don’t rely on overly complex input or output parameters. Security critical functionality like access to physical addresses is done using helper functions that implement all the necessary checks and locking in a single place.

While manual code review is often required to find security issues in complex code bases, fuzzing and static code analysis can be used for better coverage. In this case, fuzzing was not possible as we did not have access to suitable test systems. However we used two static analysis tools to assist with our review:

weggli

weggli is a code search tool for C and C++ codebases designed to help security researchers identify interesting code patterns. We used weggli to assist our manual review by running both generic and target specific queries over the codebase. Examples are shown below:

Find functions that use an argument as dynamic array index:

```
_ $func(_ $index) {_[$index];}
```

Find functions where the return value isn’t always checked:

```
_ = $func(_);' -p 'strict: $func(_);
```

Search for suspicious early returns (this query identified a bug in the TDX module) :

```
_ $func(_) {
goto $LABEL;
return _;
```

```
goto $LABEL;
}
```

Frama-C

Frama-C is a static analysis framework for C/C++ codebases. Through a combination of the E-ACSL, evolved value analysis, and weakest precondition plugins, Frama-C analyzes a target codebase while tracking variable value sets and attempting to prove the absence of C/C++ undefined behavior. Since we were unable to dynamically run the SEAMLDR or TDX module code, static analysis was an attractive option to attempt to find bugs missed through manual analysis.

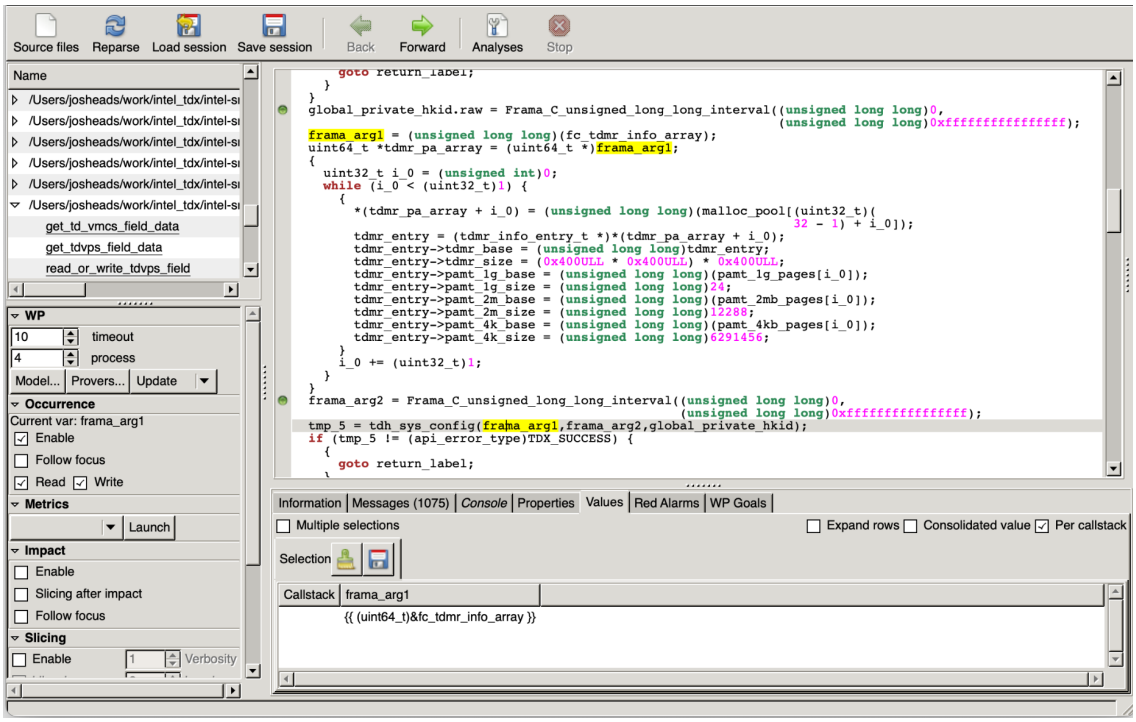

Figure 14: Screenshot of Frama-C analysis before TDH.SYS.CONFIG API call

We wrote scaffolding to simulate the TDX module simulation and then analyze the host-facing API calls with user input annotated as attacker controlled. There was limited success in running this analysis due to the nature of the TDX module and time constraints. Many of the TDX module APIs with user input deal with memory management and directly access raw pointers into memory outside the scope of the TDX module. This activity would normally indicate a bug in a program, so we had to model these memory accesses in various ways to eliminate false positives. A majority of the APIs were analyzed using this method with no true positives, but more effort is necessary to be fully confident in the results.

Given the relatively small size of the SEAMLDR and TDX module code, formal verification of part or all of these components may be worthwhile.

Discovered Issues

Incorrect loop boundary in tdh_sys_tdmr_init

The `TDH.SYS.TDMR.INIT` handler function `tdh_sys_tdmr_init` defined in `src/vmm_dispatcher/api_calls/tdh_sys_tdmr_init.c` is used to initialize parts of the Physical Address Metatable (PAMT) of a Trust Domain Memory Range (TDMR).

When `tdh_sys_tdmr_init` is called with a physical address as its argument, it needs to make sure that the address is actually part of a configured TDMR. It does so by iterating through the `tdx_global_data_ptr->tdmr_table` as shown below:

```
for (tdmr_index = 0; tdmr_index < MAX_TDMRS; tdmr_index++)
{
  if ((tdmr_pa >= tdx_global_data_ptr->tdmr_table[tdmr_index].base) &&
          (tdmr_pa < (tdx_global_data_ptr->tdmr_table[tdmr_index].base
                + tdx_global_data_ptr->tdmr_table[tdmr_index].size)))
  {
    break;
  }
}
```

Crucially, this code assumes that all array indexes smaller than MAX_TDMRs are configured correctly. However, if we look at `thd_sys_config`,the function responsible for initializing the table, we can see that callers can choose an arbitrary number of TDMRs to initialize as long as it's smaller or equal to MAX_TDMR:

```
api_error_type tdh_sys_config(uint64_t tdmr_info_array_pa,
                              uint64_t num_of_tdmr_entries,
                              hkid_api_input_t global_private_hkid)
{
    if (num_of_tdmr_entries > MAX_TDMRS)
    {
        TDX_ERROR("Num of TDMR entries %llu bigger than MAX_TDMRS (%d)\n",
num_of_tdmr_entries, MAX_TDMRS);
        retval = api_error_with_operand_id(TDX_OPERAND_INVALID, OPERAND_ID_RDX);
        goto EXIT;
    }

    if (num_of_tdmr_entries < 1)
    {
        TDX_ERROR("Num of TDMR entries %llu smaller than 1 \n",
```

```
num_of_tdmr_entries);
        retval = api_error_with_operand_id(TDX_OPERAND_INVALID, OPERAND_ID_RDX);
        goto EXIT;
    }

[...]

for(uint64_t i = 0; i < num_of_tdmr_entries; i++)
{

        ...
        update_pamt_array(tdmr_info_copy, pamt_data_array, (uint32_t)i);
// save tdmr's pamt data
}
tdx_global_data_ptr->num_of_tdmr_entries = (uint32_t)num_of_tdmr_entries;
```

A malicious VMM can exploit this issue by calling `tdh_sys_config` twice: The first call configures MAX_TDMRS entries, but triggers an error on the last entry by specifying an overlapping or otherwise invalid TDMR. The second call configures X entries with X<MAX_TDMR. tdmr_table[0..X-1] and tdmr_table[X..MAX_TDMR-1] can now contain overlapping TDMRs.
As `tdh_sys_tdmr_init` doesn't correctly limit the iteration count to X, an attacker can use this to create overlapping PAMT entries breaking one of the fundamental assumptions of the codebase.

While this bug did exist in the TDX module version shared with us during the review, it was already known to Intel. The fixed version correctly limits the loop to `tdx_global_data_ptr->num_of_tdmr_entries`

Incorrect error handling in tdh_mng_rd_wr

The TDH.MNG.RD/WR API calls can be used to read and write control structure fields of debuggable TDs.
Both variants are implemented in the function `tdh_mng_rd_wr` in `src/vmm_dispatcher/api_calls/tdh_mng_rd_wr.c`

At the beginning of the function, the code acquires a shared lock to the Trust Domain Root (TDR) of the target TD and maps its Trust Domain Control Structure (TDCS) into the virtual address space. To work correctly, all return paths from the function need to make sure that the lock is released and the page is unmapped.

In `tdh_mng_rd_wr` this was implemented using a EXIT label at the end of the function and using "goto EXIT" instead of early returns for error handling. However, one of the error cases did perform an early return instead of a goto without releasing the lock and freeing the page mappings:

```
// Check that previous value has the expected value
if (prev_value != rd_value)
{
   return api_error_with_operand_id(TDX_OPERAND_BUSY, OPERAND_ID_RDX);
}
```

The impact of vulnerabilities like this strongly depends on the underlying implementation locking and paging implementations. In the case of TDX module repeated triggering of this bug can lead to Use-After-Free issue as the Keyhole manager used for page mapping uses a 32bit integer for reference counting and the early return is missing a call to `free_la(tdr_ptr)`.

Again, this bug was part of the initial shared codebase but already known to Intel. The issue was fixed by converting the early return into a goto. Additionally, we recommended hardening the keyhole manager against similar issues by either switching to a uint64_t counter or adding checks to prevent over- and underflows.

Off-by-one in shared_hpa_check

Auditing the module's security checks for correctness was a high priority. We paid close attention to how it handles input in its boundary conditions. An important attack vector is the processing of shared memory addresses: the module sanitizes host address spaces, maps them to its virtual address space, and reads/writes to these locations.

Most security checks on a shared Host Physical Address (HPA) value take place in the function *shared_hpa_check*:

```
api_error_code_e shared_hpa_check(pa_t hpa, uint64_t size)
{
    // 1) Check that no bits above MAX_PA are set

    if (!is_pa_smaller_than_max_pa(hpa.raw))
    {
        return TDX_OPERAND_INVALID;
    }

    // 2) Check that the provided HPA is outside SEAMRR.

    uint64_t seamrr_base = get_global_data()->seamrr_base;
    uint64_t seamrr_size = get_global_data()->seamrr_size;

    if (!is_valid_integer_range(seamrr_base, seamrr_size) ||
        !is_valid_integer_range(get_addr_from_pa(hpa), TDX_PAGE_SIZE_IN_BYTES) ||
        is_overlap(get_addr_from_pa(hpa), size, seamrr_base, seamrr_size))
    {
```

```
        return TDX_OPERAND_INVALID;
    }

    // 3) Check that HKID bits in the HPA are in the range configured for shared
HKIDs (0 to MAX_MKTME_HKIDS - 1).

    if ((uint64_t)get_hkid_from_pa(hpa) > get_global_data()->max_mktme_hkids)
    {
        return TDX_OPERAND_INVALID;
    }

    return TDX_SUCCESS;
}
```

Note the boundary conditions in check #3: the module accepts HKIDs in [0, max_mktme_hkids] *inclusive*. The max value is initialized as follows:

```
tdx_global_data_ptr->max_mktme_hkids = MIN(
  tdx_global_data_ptr->plt_common_config.ia32_tme_capability.mk_tme_max_keys,
  BIT(tdx_global_data_ptr->plt_common_config.ia32_tme_activate.mk_tme_keyid_bits -
  tdx_global_data_ptr->plt_common_config.ia32_tme_activate.tdx_reserved_keyid_bits)
- 1);
```

A typo in one of the MIN sub-expressions (the “-1” subtraction should be outside the MIN expression) inadvertently computes the wrong max value, which could, if triggered, be used to bypass shared_hpa_check.

Intel engineers reviewed the implementation, and concluded that mk_tme_max_keys is always greater than mk_tme_keyid_bits - tdx_reserved_keyid_bits, therefore the MIN expressions always match the correct value (mk_tme_max_keys), and the bug **cannot be triggered** on current hardware. Nevertheless, Intel committed to re-factoring this logic, making it cleaner and more consistent.

TLB tracking

The TDX module is responsible for ensuring the processor's TLB entires for TDs do not become stale while the TDs are running. The VMM can update the guest-to-host physical page mappings of a TD during its lifetime – the TLB cache of these entries must remain accurate to maintain TD isolation. This section discusses how these guest-to-host translations are made in TDX, modifications to the EPT for TDX, how the TDX software tracks TLB freshness, and a vulnerability we discovered in the implementation.

Address translation

Page walks to shared TD pages go through the shared Extended Page Table (EPT). This table's entries map GPAs to HPAs. CPU uCode **masks** private HKID bits from the translated PA, which ensures that subsequent loads / stores to shared pages are **not** decrypted / encrypted using the TD's private encryption key.

The untrusted VMM manages this table, and therefore offers no security guarantees. For example, it's possible that a shared GPA maps to a page currently assigned to a different TD, or that two shared GPAs map to the same physical page.

Page walks to private TD pages go through a separate, Secure EPT. This table's entries also map GPAs to HPAs. CPU uCode **sets** the TD's HKID bits in the translated PA. This guarantees that subsequent loads / stores to private pages are properly decrypted / encrypted using the TD's unique private encryption key.

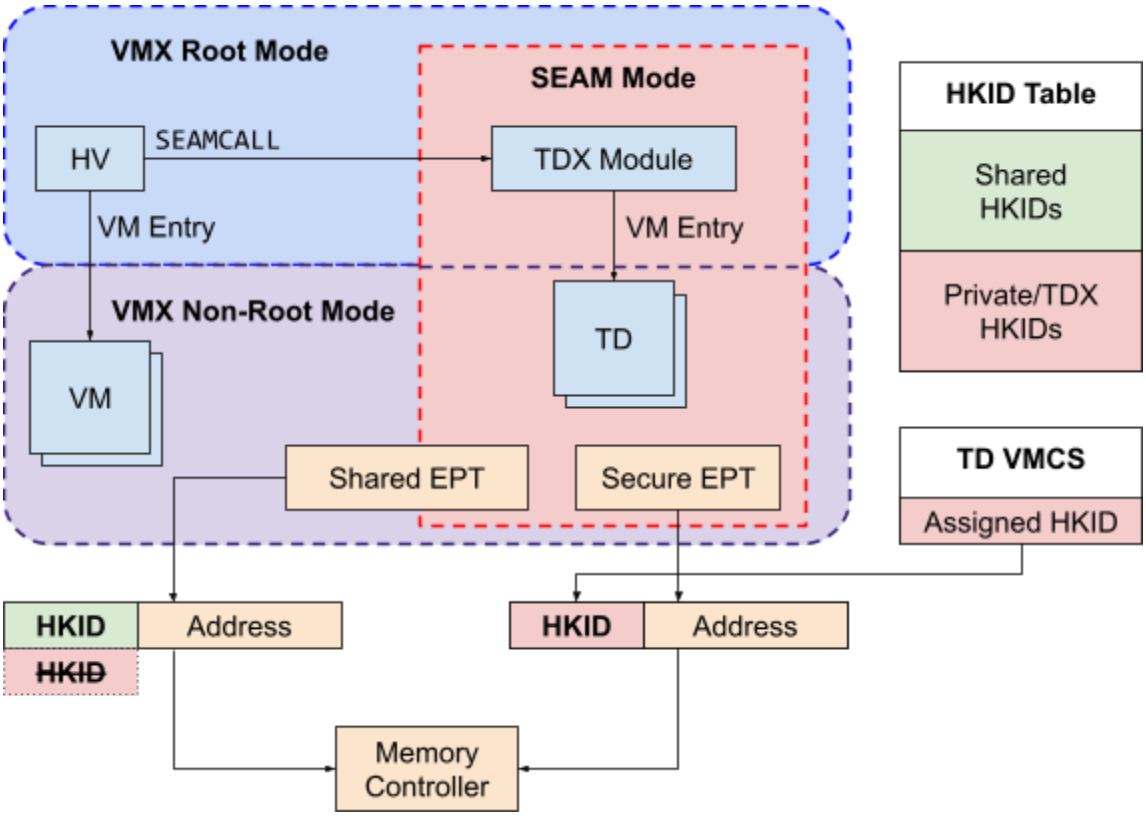

Figure 15: Diagram of the TD guest physical to host physical memory translation paths. For shared memory, the Shared EPT is used and private HKIDs are masked. For private memory, the Secure EPT is used and the assigned HKID is hardcoded.

Secure EPT

The trusted TDX module manages the secure EPT, and is responsible for maintaining its security properties:

1. Secure EPT is only accessible to the TDX module. SEPT pages are private TD pages - they are encrypted and integrity protected.
2. Mapping is consistent. A TD private page or a Secure EPT page can be mapped at most by a single guest TD GPA. Furthermore, the table is constructed to be acyclic.

There are important implementation issues the module must be aware of in order to securely manage the SEPT:

Risk	Mitigation

Race conditions with TDX module APIs.	The module exposes a set of APIs for managing the SEPT: *SEPT.ADD, SEPT.REMOVE, PAGE.ADD, etc*. The VMM can invoke any API on any logical processor. To prevent concurrent modifications to a TD's Secure EPT, the module takes a TD-scoped **lock** in each API handler.
Race conditions with TD guest page walks.	Modifications to page table entries could race with concurrent guest page walks, leading to inconsistent PTEs. To remove this risk, the module verifies, as a precondition, that the **parent** Secure EPT entry is **free** (unmapped). Furthermore, the module initializes SEPT pages and entries using atomic operations.
Stale mappings in the TLB.	Address translations are cached by the CPU in its Translation Lookaside Buffer (TLB). To prevent the risk of stale translations, the TDX module tracks the TLB state. It verifies no cached translations exist to a page before it creates a mapping to it. This mechanism is quite involved, and we cover it in the next section.

TLB Tracking Algorithm

The TDX module maintains a data structure, one per TD, called *EPOCH_TRACKING*. One field, *TD_EPOCH*, is a monotonic counter that conceptually tracks execution generations for all TD's vCPUs. Another field, *REFCOUNT*, is a fixed array that counts the number of vCPUs in each of the two most recent generations.

The logic is as follows:

On VM enter	VCPU samples the TD EPOCH and updates REFCOUNT for current epoch: 1. CUR_VCPU_EPOCH ← TD_EPOCH 2. REFCOUNT[CUR_VCPU_EPOCH &1]++ Return if this TD_EPOCH hasn't changed: 3. If VCPU_EPOCH == CUR_VCPU_EPOCH: Done Otherwise, VCPU ran on an older epoch. If VCPU was already associated with the current LP, then there's risk of cached translations: 4. If new association: call INVEPT INVEPT flushes TLB context and extended paging structure caches. Finally, store sampled TD_EPOCH, and enter guest mode. 5. VCPU_EPOCH ← CUR_VCPU_EPOCH

On VM exit	Update REFCOUNT for VCPU's epoch: 1. REFCOUNT[VCPU_EPOCH & 1]--
On *MEM.TRACK* API call	Advance TD_EPOCH if previous epoch has 0 refcount: 1. PREV_TD_EPOCH ← TD_EPOCH - 1 2. If REFCOUNT[PREV_TD_EPOCH & 1] == 0: TD_EPOCH++ This guarantees that TD EPOCH advances only when there's no vCPUs associated with the previous epoch.

The key point: the costly *INVEPT* instruction happens on VM enter, only when the VCPU advances to a new TD_EPOCH.

A second mechanism encodes the TD_EPOCH value in private pages metadata information (PAMT). This happens only on pages marked for removal or remapping:

On *MEM.RANGE.BLOCK* API call	Unmap page and label it as 'blocked': 1. Write PTE with reserved 'blocked' bit, and clear 'present' bits ('rxw'). Record page block epoch in metadata entry: 2. PAMT.BEPOCH ← TD_EPOCH

This operation blocks a TD private GPA range (SEPT page or TD private page) from creating new GPA-to-HPA address translations.

Finally, all TDX module APIs that modify the secure EPT (page demote, page promote, page relocate, page remove, range unblock and SEPT remote) consult the PAMT, and verify the block epoch is sufficiently behind the current TD EPOCH:

Is TLB tracked check	TD EPOCH advanced beyond block EPOCH: 1. PREV_TD_EPOCH ← TD_EPOCH - 1 2. If PAMT.BEPOCH < PREV_TD_EPOCH: Return True Or, block EPOCH is equal to previous TD EPOCH, and there's no

	vCPUs associated with that: 3. If PAMT.BEPOCH == PREV_TD_EPOCH && REFCOUNT[PREV_TD_EPOCH & 1] == 0: Return True The check means that all LPs that ran in TDX non-root mode during the epoch when the GPA range was blocked have since TD-exited. Otherwise, return False, since TLB state cannot be guaranteed. 4. Return False

The TLB tracking mechanism is safe and efficient: SEPT modifications are done only when TLB is guaranteed to not hold active cached translations, TLB flushes are kept at a minimum, and the tracking information is stored in an existing PAMT data structure.

To recap, TLB tracking sequence is as follows (assume TD EPOCH = X):

1. TDBLOCK on a set of private GPAs
    a. Each blocked HPA has BEOPOCH = X
2. VMM calls TDH.MEM.TRACK
    a. TDX module verifiers REF_CNT[(X-1)%2] = 0
    b. TD EPOCH = X + 1
3. VMM sends IPI on all TD VCPUs and immediately resume the TD VCPUs
    a. TD REF_CNT[X%2] = 0 (after all IPIs are served)
    b. On entry, if TDVCPU EPOCH != TD EPOCH then invalidate VCPU ASID (TLB)
4. TDX module knows that a blocked HPA has no TLB references if either
    a. HPA BEPOCH == TD EPOCH -1 && REF_CNT[(TD EPOCH-1)%2] == 0
    b. HPA BEPOCH < TD EPOCH -1

revert_tlb_tracking_state() vulnerability

We identified an implementation bug in an uncommon TLB tracking flow.

`tdh_vp_enter()`, updates REFCOUNT in `adjust_tlb_tracking_state()`:

```
bool_t adjust_tlb_tracking_state(tdcs_t* tdcs_ptr, tdvps_t* tdvps_ptr,
                                 bool_t new_association)
{
  ...
  // Sample the TD epoch and atomically increment the REFCOUNT
  uint64_t vcpu_epoch = epoch_tracking->epoch_and_refcount.td_epoch;
  _lock_xadd_16b(&epoch_tracking->epoch_and_refcount.refcount[vcpu_epoch & 1], 1);

  ...
```

```
  if (vcpu_epoch != tdvps_ptr->management.vcpu_epoch)
  {
    ...
    // Store the sampled value of TD_EPOCH as the new value of VCPU_EPOCH
    tdvps_ptr->management.vcpu_epoch = vcpu_epoch;
  }
  return true;
}
```

Next, it performs the "stepping filter" logic. This defense mechanism, implemented in `td_entry_stepping_filter()`, aims to detect "zero step" attacks. When the same single VCPU instruction raises too many EPT violations on accesses to private TD pages, the TDX module "suspects" a zero-step attack against the TD VCPU. After the number of events pass a threshold, the TDX module aborts `tdh_vp_enter()`. Notably, before it exits, it restores the TLB tracking information:

```
void revert_tlb_tracking_state(tdcs_t* tdcs_ptr)
{
  tdcs_epoch_tracking_fields_t* epoch_tracking = &tdcs_ptr->epoch_tracking;

  // Sample the TD epoch and atomically decrement the REFCOUNT
  uint64_t vcpu_epoch = epoch_tracking->epoch_and_refcount.td_epoch;
  _lock_xadd_16b(&epoch_tracking->epoch_and_refcount.refcount[vcpu_epoch & 1],
                 (uint16_t)-1);
}
```

`revert_tlb_tracking_state()` aims to revert the `adjust_tlb_tracking_state()` operations. The problem is that it uses the global `td_epoch` variable, which **may change** between the two function calls. The following sequence leads to an inconsistent `epoch_tracking` state:

	td_epoch	refcount[0]	refcount[1]
Initial state: no associated vCPUs	100	0	0
LP0: Enters TD. `adjust_tlb_tracking_state()` updates `refcount[td_epoch & 1]`	100	1	0
LP1: Calls *MEM.TRACK* API. `tdh_mem_track()` checks previous epoch is clear (`refcount[1] == 0`), and advances `td_epoch`	101	1	0
LP0: "Zero-step" detected.	101	1	0xffff

`revert_tlb_tracking_state()` updates `refcount[td_epoch & 1]`			
LP2: Enters TD. `adjust_tlb_tracking_state()` updates `refcount[td_epoch & 1]`	101	1	0

At this point, an active TD vCPU is running with epoch 101, however, it is not refcounted in the global `epoch_tracking` data structure.

We couldn't find a way to exploit this inconsistency to bypass the "is TLB tracked" check: `tdh_mem_track()` would not advance `td_epoch` since `refcount[0] > 0`, and there's no way to decrement it back to zero.

Uncore Attack Vector

Intel considers all logic outside of the CPU but within the same SoC to be the “uncore”.This is a broad term covering a large assortment of IP components and the fabric interconnect. While some portions of TDX involve new CPU architecture additions (SEAM mode, new TDX instructions, etc…), others depend on uncore components (MK-TME, MCA, etc…). The CPU-based features are typically controlled via MSRs are specialized control registers. For the uncore, there is an analogous set of registers[12] for each IP and a complex access control system.

As the uncore was not in scope for the TDX security review this report does not provide any analysis. A full security review of this area would be recommended to ensure all security concerns are identified and remediated.

ECC Disablement Vulnerability

One vulnerability did arise due to incorrect assignment of configuration controls. In this case, a privileged attacker could disable ECC and additionally weaken DRAM settings which would make Rowhammer bit flips more likely. The issue stems from Intel’s original TDX design relying on cryptographic integrity while logical integrity was later added for additional DRAM compatibility[13]. With TDX-Ci, the HMAC provides similar protections as ECC for detecting memory integrity attacks. However, with TDX-Li there is only a single bit protecting the cache line integrity. **If an attacker could disable ECC then they would only need to flip a single bit in order to bypass the TDX-Li access control checks**. This leads to TDX-Li being vulnerable to Rowhammer style attacks where a VMM tries to flip bits in memory owned by the TDX module or TDs.

This issue has been resolved in the 4th Gen Intel Xeon Scalable CPUs so that these control registers are locked before MCHECK runs and MCHECK validates that their values are configured properly before enabling TDX.

MSRs

The system MSRs present a large and complex attack vector accessible to both the unprivileged TDs and highly privileged BIOS and VMM. These registers are used for feature enumeration, monitoring, and configuration and span most features of the Intel SoC. The registers are associated with the x86 cores (unlike the similar uncore registers which control everything else on the SoC), and are scoped to either a thread, core, or the entire platform. For example, the MKTME_KEYID_PARTITIONING MSR is platform scope and sets the division between shared and private HKIDs for all cores on the socket. Alternatively, the

[12] The uncore registers are exposed through internal PCIe device configuration space and MMIO
[13] TDX-Ci (cryptographic integrity) includes a 28-bit truncated HMAC for integrity protection of every cache line. TDX-Li (logical integrity) only includes a single bit indicating TDX ownership of the cache line.

DEBUGCTLMSR MSR is thread scope and configures the debug settings uniquely per SMT thread.

In order to support isolation between the guest VMs and host VMM, a context switch must occur to ensure the low privilege guest can't affect the MSRs of the highly privileged VMM. For a few MSRs, they are automatically swapped by the hardware and stored within the VMCS during VM transitions. For all other MSRs, the VMM configures a bitmap through the VMCS which indicates for each MSR whether access should pass through to the actual register or if it should instead cause a VM exit. During the VM exit, the VMM can context switch the MSR in software and emulate behavior if needed.

A somewhat similar method is used to ensure a less robust amount of isolation between the VMM and both P-SEAMLDR and the TDX module. These modules adapt the transfer VMCS technology used in SMI Transfer Monitors which enables a similar hardware-accelerated context switch compared to the TDX-TD context switch. The key difference is that P-SEAMLDR and the TDX module can't trap or prevent VMM behavior, such as writing to MSRs. The implications of this design are discussed in more detail below.

Review Methodology

For Sapphire Rapids there are nearly 3000 MSRs, so a process was required to ensure we extract all MSR details and then filter to a manageable set to review. For the TD-to-TDX module attack vector, Intel provides a list of MSRs in section 18.1 of the TDX architecture specification which details the intercept behavior for each intercepted MSR. For all MSRs outside of this list, TD access is blocked and a #GP is returned – this can be verified in the source as well. For the VMM-to-TDX module attack surface, we need to think about how all MSRs potentially interact with the TDX module. Additionally, the MSR scope determines the type of attacks possible: for thread-scope, the MSR can only be modified before or after TDX execution; for core and platform scope, the MSR can be modified in parallel with TDX execution.

Based on these requirements, we generated a list of all MSRs for Sapphire Rapids and filtered by their scope and descriptions. MSRs directly affecting TDX can be found by reading the specification and source code. We also identified a new category of address-based MSR attacks where a register containing a programmable memory address may also pose a threat to TDX. Lastly, there are MSRs which affect memory layout, memory encryption, memory access controls, cache controls, and many other components that overlap with TDX security. Based on these filters, we arrived at a shorter list of MSRs to review. Additionally, we have worked with Intel's security team to incorporate additional tests based on our understanding of the MSR threat to TDX.

Attack Vectors

Given this background, we can divide the MSR attack vectors into several categories:

TD-to-TDX Module Attacks

In this attack, one or more malicious TDs utilize virtualized MSRs to either manipulate CPU behavior or leak information from outside of their context (TDX module, sibling TD, or VMM). This attack vector is essentially identical to the traditional VM/VMM relationship and is relatively well understood. Intel has designed the TDX module to deny MSR access by default and generated an allow-list of MSRs which it will virtualize (see Section 18.1 in the TDX spec).

Because core-scope and platform-scope MSRs affect the sibling thread and all threads respectively, only thread-scope MSRs can be virtualized while maintaining thread isolation. We reviewed all MSRs that the TDX allows the guest to write and confirmed that only thread-scope registers are allowed. Additionally, the list of MSRs shared with TDs was reviewed to ensure none provide methods to violate TD isolation.

VMM-to-TDX Module Attacks

The MSR threat model for a malicious VMM is quite different from the malicious TD case. The VMM is running on bare metal and can program all MSRs[14] (within hardware limitations) without the TDX module's approval or awareness. When the VMM transitions to the TDX module (or P-SEAMLDR), this is done using what's known as a transfer VMCS. This transition is derived from the SMI transfer monitor technology previously developed by Intel and enables the same hardware context switching as a normal VMCS but doesn't allow the TDX code to intercept VMM execution. The result is that MSRs which are represented in the VMCS (a small subset) can be context switched before TDX execution begins; however, all other MSRs must be context switched in software if wanted.

Additionally, since the VMM has access to core-scope and platform-scope MSRs (unlike the TDs) this enables parallel MSR-based attacks while the P-SEAMLDR/TDX/TD code is running. We reviewed many of the possible MSRs an attacker could modify[15] and many that could cause harm to TDX are previously locked and validated by MCHECK. However, this is a large attack surface and extension testing should be done with actual hardware to increase confidence that a malicious VMM can't use MSRs to attack TDX.

[14] There are over 2000 MSRs on Sapphire Rapids.

[15] For example, the TDX architecture specification describes the core-scope MSR_TEST_CTRL which is not virtualizable and can create effects on the SMT sibling thread. In this case, the TD OS is responsible for anticipating these effects and handling them safely.

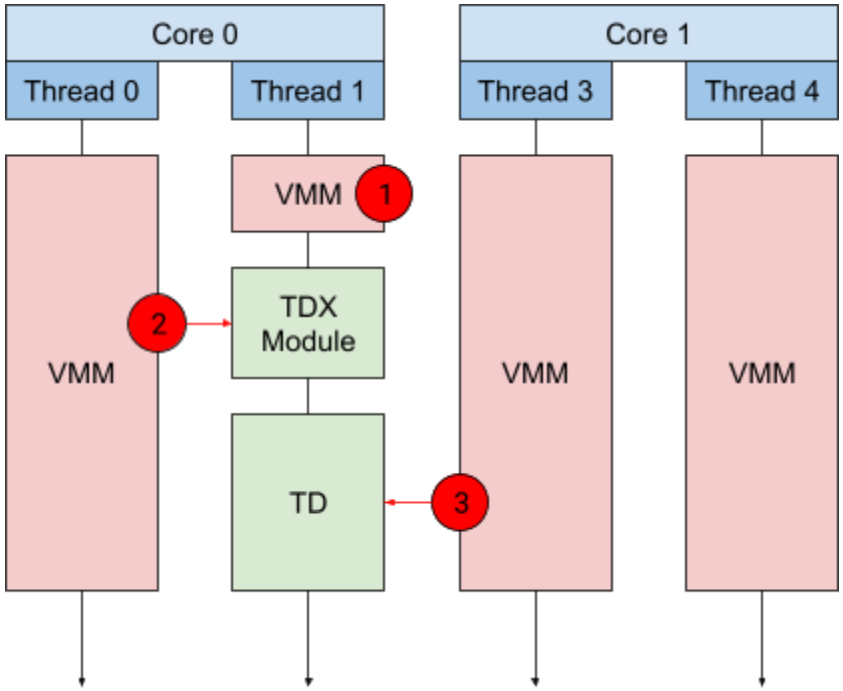

Figure 16: Timeline demonstrating VMM-initiated attacks using thread-scope (1) MSRs before TDX execution and core-scope (2) and platform-scope (3) MSRs during TDX execution

Address-Based Attacks

We identified an alternative class of attack where a malicious user programs an MSR which contains addresses. For example, the IA32_APIC_BASE MSR contains an address field which points to the base address of the APIC MMIO region. If this address can be programmed to point to protected regions (e.g., SEAMRR) or TD private memory with the HKID set then an attacker would be able to indirectly read or write to these otherwise blocked regions. Alternatively, an attacker could program the address to start just before the protected/TD-private region and overflow into the region if the MSR controls data access of a large enough size (e.g., a processor trace log).

We enumerated a list of MSRs which contain physical addresses and have worked with Intel to verify that the hardware restricts writes to these registers such that the addresses don't overlap with regions like SEAMRR or contain HKID bits.

Security Concerns

VMM-to-TDX Privilege Inversion

While the TD-to-TDX vector is largely identical to the well understood boundary that traditional VMs have with a VMM, the VMM-to-TDX boundary does not follow the same principles. In order to maintain VM isolation, the VMM must be able to prevent the VM from manipulating system state (such as MSRs) that can have an effect on itself or neighboring VMs. For TDX-enabled

systems, the SEAMLDRs and TDX module are the most privileged software running on the machines. Ideally, these privileged programs should have similar levels of isolation from unprivileged software as in the traditional VMM/VM model.

Unfortunately in the current TDX design, the VMM is able to control MSRs which may affect TDX SEAM code execution behavior or leak information back to the VMM[16]. This leads to a privilege inversion where a low privilege component gains some control over high privileged components by design. To mitigate this risk, we have reviewed many of the MSRs we and Intel deemed sensitive and confirmed that either MCHECK, locked contents, or internal design prevent these MSRs from posing an actual threat to TDX code. Further research into how to systematically audit the risk a malicious VMM with MSR control poses to TDX code would help build confidence that this attack surface is safe.

[16] This same privilege inversion also applies to the Uncore system registers that the VMM can access.

Side Channel Attacks and Mitigations

Side channel attacks (SCA) are a constant concern in secure systems, and TDX is no exception. In this section we discuss side channel attacks and mitigations, and list a set of primitives that *might* leak information about software running in SEAM mode.

TDX designers went to great lengths to prevent information leakage. Nevertheless, users of TDX technology should be aware of these risks, as they might affect TD workloads.

Speculation based side channel attacks

Transient execution (or Spectre) attacks are powerful because during speculation the CPU may temporarily *violate* program semantics by executing code that would not have been executed otherwise. These violations are discarded, and are not observable on an architectural level. However, transient execution leaves *traces* in micro-architectural components - traces a sophisticated attacker *can* observe.

Transient execution attacks

A successful Spectre attack chains several primitive gadgets[17]:

1. A speculation variant triggers speculation on a mis-predicted branch,
2. A memory load is performed using an attacker supplied input,
3. A secret is encoded in a micro-architectural covert channel.

Speculation variants

Prediction based

The CPU speculation engine is guided by a set of internal history or prediction buffers. In a “prediction based” attack, an attacker indirectly controls these buffers, and steers the CPU into *mispredicting* a branch, and executing instructions under adverse conditions.

Several speculation variants have been identified: Spectre v1 targets the Pattern History Table (PHT), Spectre v2 targets the Branch Target Buffer (BTB), and “Ret2Spec” targets the Return Stack Buffer (RSB). A recent Spectre v2 sub-variant targets the Branch History Buffer (BHB). In the Spectre v4 “Store To Load” (STL) variant, the CPU speculatively bypasses store instructions.

[17]Breakdown inspired by the *Kasper* paper. https://www.vusec.net/projects/kasper/

Fault / Assist based

In a “fault/assist based” attack, the CPU performs a (speculative) computation using cached data that belongs to a *different security domain*. This can be used to *extract* secret data from a victim domain.

Several fault attacks have been identified: Spectre v3 “Meltdown” leaks data across protection ring boundaries, “Foreshadow” (AKA “L1TF”) leaks L1 data across virtualization and SGX boundaries. Similarly, “Microarchitectural Data Sampling” (MDS) variants leak L1 data across security boundaries.

See the Intel developer site, Wikipedia page, and this website for a detailed taxonomy of transient execution attacks.

Value injection variants

These speculation gadgets perform a computation, typically a memory load, using an attacker controlled value.

For example, In Spectre v1, the CPU performs an out-of-bounds memory read using an attacker controlled index (“bounds check bypass”). In other vulnerable code patterns, the CPU speculatively reads from uninitialized / freed memory objects (“speculative use-after-free”), or from different memory objects (“speculative type-confusion”), where the memory contents are under attacker control.

Fault/assist based attacks could also be used to *inject* attacker supplied values to the victim domain. “Load Value Injection” (LVI) forces a sibling core to speculatively process unsanitized values.

Secret output variants

A final Spectre gadget encodes the secret value in a place an attacker can later retrieve.

Several micro-architectural covert channels have been identified. A well known one is a cache based covert channel, where load access times differentiate between “hot” and “cold” cache lines. “Flush+Reload” and “Prime+Probe” techniques can be used to read secrets over a cache based covert channel.
“MDS” can also be used to output secrets over the shared L1D cache.
“NetSpectre” demonstrated that secrets can be encoded over AVX instructions invocations.
Finally, “SMoTherSpectre” proved that execution port contention and timing measures can be an effective covert channel.

Covert channels vary by their accuracy (spatial / temporal resolution), bandwidth, and signal to noise ratio. Research shows that repeated measurements and techniques from machine learning can successfully reduce the noise, and improve accuracy.

Applications to TDX

Guided by the framework above, we can list *potential* transient execution attacks in the context of TDX. The space of potential threats is {attacker domains} x {victim domains} x {attacks}:

Attacker domains:	SEAM non-root: a malicious TD guest.
	Legacy VMX: the host VMM.
Victim domains:	SEAM non-root: other TD guests.
	SEAM root: SEAM loaders and the TDX module.
	Legacy VMX: a malicious TD may attempt to attack non-TD VMs or the host VMM.
Attacks:	Attacker mounts an L1TF or RIDL fault attack, and extracts victim data residing in shared L1D cache. *Prerequisites*: 1) Hardware vulnerable to L1TF / MDS. 2) Victim is running concurrently on a sibling hyper-thread, or was recently running on the same core, with no proper L1D flushes.
	Attacker poisons prediction buffers (BTB, RSB, BHB), victim mis-predicts indirect branches, executes disclosure gadgets which leak data from victim's address space. *Prerequisites*: 1) Predication buffers are shared between domains. For example, logical processors sharing a core may share indirect branch predictors. 2) Victim does not flush prediction buffers on domain transitions, 3) Victim holds secret data in its virtual address space, 4) there's a working covert channel between attacker and victim.
	Attacker injects values to L1D cache (LVI); On page faults, the victim speculatively reads from injected, unsanitized memory pointers with visible side effects. *Prerequisites*: 1) Hardware vulnerable to MDS, 2) Victim code has vulnerable read gadgets, 3) Working covert channel.
	Spectre v1: Attacker mistrains conditional branch (PHT) in victim code path. Victim mis-predicts under adverse conditions ("OOB read",

	"speculative UAF" or "speculative type confusion"), executes disclosure gadgets that leak information from victim's address space. *Prerequisites*: 1) Victim has code that processes attacker supplied input (i.e. an API handler), 2) Conditional branches don't block speculation (i.e. missing *LFENCE*), 3) Victim code has vulnerable read gadgets, or patterns that give an attacker a large degree of freedom during speculative execution. For instance, a type confusion bug may call an attacker controlled virtual function, and lead to a speculative ROP attack, 4) working covert channel.

Speculation adds another dimension to the TDX threat model. For example, the VMM may attempt to use speculative reads to leak information about private TD memory. Speculative reads from poisoned cache lines (see below) *do not generate* a machine check exception (#MCE), so a malicious VMM may attempt to leverage that, and poison a cache line until a MAC collision is found (TDX with MAC integrity has a small 26b space). Fortunately, this threat is fully mitigated in TDX: speculative reads of TDX-owned memory initiated by the VMM always return 0, and no poison indication is returned.

Mitigations

Mitigations against transient execution attacks in platforms that support TDX are extensive, and include both hardware and software solutions. Details matter, and they are listed below:

Attack variant	**Mitigation**
Spectre v1 (Bounds Check Bypass)	Mitigated in software. SEAM software places speculation barriers (*LFENCE*) at strategic places, before processing untrusted input. P-SEAM loader entrypoint, and dynamic keyhole mappings. TDX module TD-exit entrypoint, SEAMCALL entrypoint, and dynamic keyhole mappings. *Potential gaps*: Speculation barriers were manually placed at locations that were identified as potentially vulnerable, however, there's a risk the authors have missed other locations. For instance, LFENCE is placed after a *map_pa* operation, however, there could be vulnerable conditional jumps further down the code, after speculation is resumed. We could not identify such branches in our review, but we believe that developing a principled way to find all Spectre gadgets in the TDX codebase is an interesting research area. Furthermore, we believe there's an opportunity to adopt stronger compile time

	mitigations, such as LLVM SLH. Fortunately, *full* Spectre gadgets are not common in production libraries, so the likelihood of an unprotected, exploitable *Bounds Check Bypass* pattern in the TDX module is *low*.
Spectre v2 (Branch Target Injection)	Mitigated in hardware + extended software controls. Hardware supports enhanced Indirect Branch Restricted Speculation ("eIBRS"). This feature restricts speculation of indirect branches by isolating predictor modes. This feature is "on" by default, and *cannot* be disabled at runtime. IBRS supersedes another Spectre v2 mitigation called Single Thread Indirect Branch Predictors ("STIBP"), thus, STIBP is not required on TDX supported hardware. IBRS isolates predictor modes, so SEAM root predictor is isolated from SEAM non-root predictor. However, mutually distrusting TD workloads share the same non-root predictor. To help prevent cross-TD Spectre v2 attacks, the TDX module issues an Indirect Branch Predictor Barrier ("IBPB") command when associating a vCPU on a new LP; see tdh_vp_enter. Finally, to help mitigate Spectre v2 Branch-History-Buffer variant, SEAM software runs a BHB clearing sequence on SEAM entry: pseamldr_entry_point, tdx_vmm_dispatcher, tdx_td_dispatcher.
Spectre v3 ("Meltdown")	Fully mitigated in hardware. TDX module checks rdcl_no in ARCH_CAPABILITIES MSR.
Spectre v4 (Speculative Store Bypass)	SEAM software enables hardware mitigations by setting Speculative Store Bypass Disable (SSBD) bit in SPEC_CTRL MSR. P-SEAM loader in pseamldr_dispatcher. TDX module in all domain transitions: tdx_vmm_dispatcher, tdx_td_dispatcher.
"L1 Terminal Fault"	Fully mitigated in hardware. TDX module checks rdcl_no in ARCH_CAPABILITIES MSR on SYS_INIT.
"Microarchitectural Data Sampling" and variants	Fully mitigated in hardware. TDX module checks mds_no in ARCH_CAPABILITIES MSR on SYS_INIT.
"TSX Asynchronous Abort"	Fully mitigated in hardware. TDX module checks taa_no in ARCH_CAPABILITIES MSR on SYS_INIT.
Stale Data Read from Legacy xAPIC ("AEPIC leak")	Fully mitigated in hardware. On SPR, the BIOS enables and locks x2APIC, and MCHECK verifies the configuration which prevents access to legacy xAPIC.

It is the responsibility of the TD guest to protect itself from intra-TD transient execution attacks such as Spectre v1, Spectre v2 and Spectre v4. Similar responsibility falls on a regular guest OS.

On hyperthreading

TDX enabled hardware has no known CPU vulnerabilities that make Hyper-Threading, generally known as Simultaneous Multithreading (SMT), insecure. Furthermore, SEAM loaders and the TDX module do not perform any secret dependent operations, therefore, attacks such as “PortSmash” do not apply to these modules.

To clarify, SEAM range contains secrets - saved TD register state is a good example - however, SEAM software does not perform operations that depend on secret data or secret key material: no symmetric encryption or MACing, no asymmetric decryption or signing. Most cryptographic operations are done in hardware: inline memory encryption and integrity protection. The only cryptographic operations carried by SEAM software are RSA verification and SHA384 digest computations, and those do not use private key material.

It should be noted that TDX attestation includes the hyperthreading settings. Users that are wary of running TD workloads on SMT enabled hosts, may choose to do so.

Fixes to newly discovered CPU vulnerabilities will be reflected in the attestation report, either directly or indirectly via the module’s Secure Version Number (SVN).

Traditional side channel attacks

SEAM software is responsible for protecting itself from traditional side channel attacks such as *timing*, *memory access* and *power* SCA.

To mitigate the risk of a timing SCA, TDX module uses a **hardened crypto library** that runs in constant time. TD guest software is responsible for doing the same for its sensitive operations.

To mitigate the risk of a memory access SCA, TDX module uses a **hardened crypto library** that runs with constant memory patterns. TD guest software is responsible for doing the same for its sensitive operations. Architectural “access oracles” are detailed in a section below.

To mitigate the risk of a power (energy) SCA, TDX module uses **masked energy reporting**. The feature depends on a uCode mitigation called “Running Average Power Limit Energy Reporting“, which limits privileged software (root VMX) from using Running Average Power Limit

(RAPL) interfaces to monitor SEAM power activity. TDX module checks energy_filtering_ctl in ARCH_CAPABILITIES MSR on SYS_INIT, and bails out if this feature is not set. ENERGY_FILTERING_ENABLE is set internally by the CPU.

Finally, the module **virtualizes CPU's performance counters**. This method prevents the host VMM from using perfmon counters to monitor SEAM activity.

Access oracles

In this section we describe architectural "access oracles": methods available to the host VMM to gain limited visibility to TD memory activity.

Blocked private pages

TDH.MEM.RANGE.BLOCK is a VMM operation that unmaps a guest GPA range from its secure EPT. A memory access (read, write or execute) to an unmapped address causes an "EPT violation" VM exit, that is propagated back to the VMM. The faulting address, masked to a page boundary, is returned to the VMM. A blocked page can later be restored using *UNBLOCK*, making this a non-destructive access oracle.

To illustrate the impact of this, let's assume the following const-time Montgomery's ladder private key operation is running inside a TD guest. Let's further assume lines {1, 2, 3} fall on distinct pages:

```
x1 = x; x2 = x^2
for i = k - 2 to 0 do
    if ni = 0 then                  (1)
        x2 = x1 * x2; x1 = x1^2     (2)
    else
        x1 = x1 * x2; x2 = x2^2     (3)
return x1
```

By blocking the {2, 3} GPA range, and observing the faulting page address, a malicious VMM is able to infer private key bits.

Poisoned cache lines

Recall that Multi-Key Total Memory Encryption, or MK-TME, is the building block of TDX: it enables the CPU to encrypt each TD's memory with a unique AES key. In MK-TME, memory requests include the KeyID that specifies the TD's memory encryption key. Logically, the KeyID flows from the CPU through the caches to the memory controller, and every cache tag at all levels of the memory hierarchy in the system includes a KeyID. Practically, the KeyID is embedded in the *Max_PA* bits physical address:

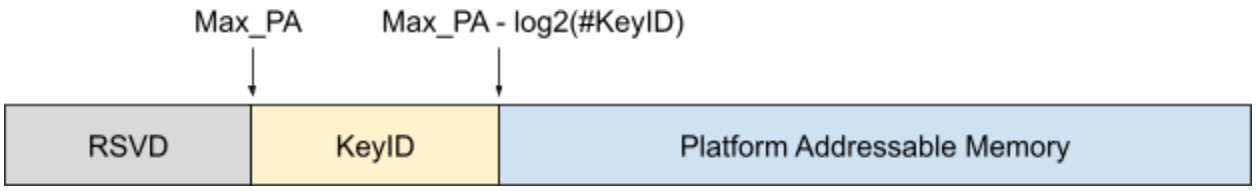

Figure 17: Physical Address with KeyID tag

The TDX architecture partitions KeyIDs between Shared (VMM) and Private (TDX) usage in order to prevent the VMM from directly accessing TD private memory. Specifically, a VMM is *prohibited* from generating requests with TDX KeyIDs and therefore cannot read/write TD private memory successfully. However, it can still access TD private memory using an *incorrect* KeyID, resulting in potential caching of 'KeyID aliases', which are copies of the same physical address decrypted using different KeyIDs.

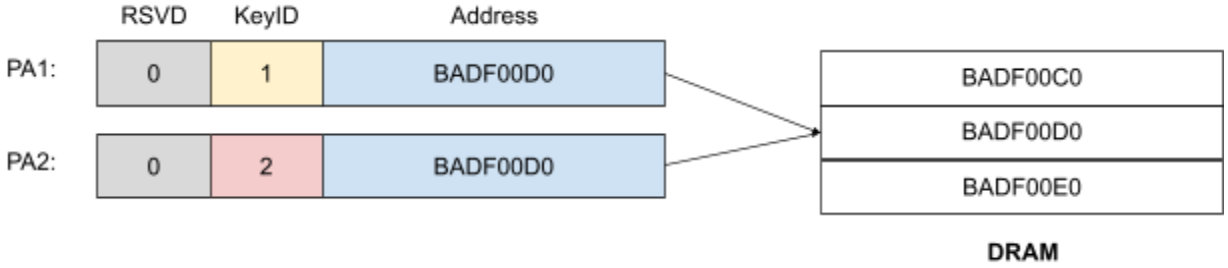

Figure 18: KeyID aliases: different PAs target the same cache line

A VMM can overwrite private TD memory using a shared KeyID: this process "poisons" the cache line - its memory contents are modified, and its TD owner metadata bit is cleared. A subsequent TD read from a poisoned cache ine generates a machine check exception, as the integrity check fails (the owner bit is included under the MAC).

A machine check exception terminates the TD: the faulting vCPU exits, and attempts to enter the TD on any other LP fail. The machine check register bank limits the information about the generated exception: RIP is masked to a page boundary and all other GPRs are cleared.

Poisoned cache lines can be used as an access oracle. Following the Montgomery's ladder example above, assume lines (2) and (3) fall on distinct cache lines. By poisoning (2), and

“single-stepping” (see below) the TD after line (1), the VMM is able to learn the TD’s execution flow, and recover a single private bit.

It should be noted that the TDX module, running in SEAM root-mode, may also map and access TD private memory. In case it reads a poisoned cache line, a machine check exception - like any vectored event (exception or interrupt) during TDX module operation - triggers a **SEAM shutdown** event. The SEAM-Ready flag is cleared - this prevents LPs from entering the TDX module. However, LPs that are currently running in SEAM mode, may continue running until they exit.

In the review we looked for vulnerable code patterns where an unexpected SEAM shutdown would leave that system in an inconsistent state. Imagine the following scenario:

1. LP0: Enter TD guest
2. LP1: Modify global state
3. LP1: Map and access TD memory
4. LP1: Cleanup global state
5. LP0: Exit TD
6. LP0: Access global state.

A MCE in step 3 would skip the “cleanup global state” step, leaving it in an inconsistent state for LP0 in step 6. For example, pamt_promote accesses mapped TD memory inside a for loop. A MCE mid-loop would leave the PAMT promotion in a half-baked state. Fortunately, the locking mechanism (*promoted_pamt_entry->entry_lock* in this case) protects global data structures, and prevents LPs from using it mid-operation, even if this operation throws an exception. tdx_sanity_check in pamt_get_block is also an interesting candidate, since the VMM can fail this assertion, and trigger a SEAM shutdown, for VMM controlled PAs. Though an interesting attack vector, we could not identify any vulnerable code pattern.

In addition, we explored how the system reacts when a poisoned cache line is consumed during operations outside of explicit memory reads/writes. For example, when the EPT page walker consumes poisoned SEPT entries, or when the VM enter uCode consumes poisoned VMCS entries. We confirmed with Intel engineers that these flows lead to an unbreakable shutdown.

MONITOR and *MWAIT*

Hardware has a mechanism to put the CPU in a low-power state until a condition is true - sort of an optimized polling loop (“`while (*cond == 0) {}`”). This is useful for implementing spinlocks and condition variables.

Behind the scenes, the address monitoring hardware detects writes to a target physical address. The MONITOR instruction arms the monitoring unit with the target address, and the

MWAIT instruction waits for its modification. Note, MONITOR takes a linear address as input, however, the translation to a physical address is fully under the control of the VMM.

Close examination identified the following gap: the monitoring unit ignores the KeyID portion of incoming physical address snoop requests. This means the host VMM can monitor memory writes to aliased TD private pages.

This access oracle has cache line precision, can monitor up to ~40 addresses (on SPR) in parallel, and impacts both TD private memory *and* TDX control structures. To illustrate the impact, note that TD's SEPT entries are modified ('accessed' and 'dirty' bits) at runtime, as the vCPU executes its logic and references guest memory. By monitoring writes to the SEPT entries, the VMM is able to recover partial information about the TD's execution flow.

Boosting cache based side channel attacks

An aliased access (say with KeyID K2) to an existing cache line (say with KeyID K1), forces the eviction of the currently cached (K1) copy. After eviction, the processor fetches the same line with the new KeyID (K2):

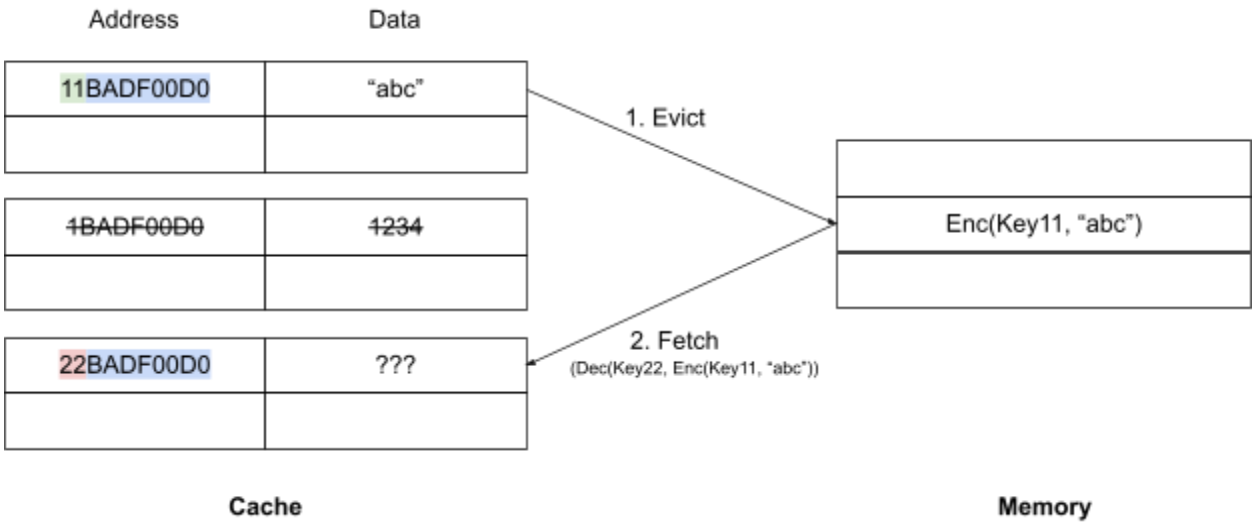

Figure 19: KeyID related cache coherence flow

A side effect of this cache coherence flow is improved cache based side channel attacks. Typically, an untrusted VMM cannot directly generate memory requests (and hence, cause flushes) with TDX KeyIDs. However, using aliased access, the VMM can force flushes of cached data. This enables cache line precision cache based attacks such as Flush+Reload and Flush+Flush. These attacks have higher precision than other types of cache side-channel attacks that rely on cache set congruence such as Prime+Probe or Evict+Reload.

Cache Allocation Technology (CAT) is a new feature on server platforms. CAT partitions the cache, and restricts L2/L3 cache range. By isolating cache activity to a single CPU, CAT can be used to improve cache SC signal-to-noise ratio.

We explored whether modifying the partition size at runtime would prevent cached writes from being committed to DRAM, and confirmed with Intel engineers that this is *not* the case.

CAT demonstrates how an unrelated CPU technology might negatively interact with TDX. CPUs grow in complexity, and new CPU features present additional risk to confidential computing technology.

Zero step / Single step mitigations

Side-channel observations may improve by repeatedly resuming TD execution at a faulting instruction. As a faulting instruction widens the speculation window, resuming execution at such "replay anchors" causes the CPU to execute the same speculative window with the same stimulus. The TDX module attempts to thwart this "**zero step**" attack by blocking the VMM from resuming a TD when repeated SEPT violations are detected at the same GPA. Execution resumes only when faulting addresses are properly mapped in the Secure EPT. A TD can be notified of such attacks via a special #VE exception.

The VMM may interfere with TD execution by injecting interrupts, NMI, SMI and INIT. SGX-Step, a previously published attack on SGX, demonstrated how high-frequency APIC timer interrupts cause an SGX workflow to exit after *every* instruction - effectively single stepping through the code. This "**single step**" attack improves side-channel attacks on confidential workloads.
The TDX module detects frequent interrupts, and when it suspects a single step attack, it continues TD VCPU execution for a small random number of instructions before the interruption is delivered to the host VMM.

Baseboard Management Controller

Baseboard Management Controller (BMC) is a specialized microprocessor embedded on the system board of a server, managing the interface between system management software and the platform hardware. The platform operator controls the BMC firmware, therefore, this component is outside TDX's trust boundary.

Modern BMCs are capable of monitoring the server's power consumption. In addition, they may dynamically control the server's power source. This introduces the following risks: 1) a BMC based power side channel attack, 2) a BMC based power glitching attack. A glitching attack is

effective against security sensitive components such as MCHECK and the SEAM loaders, as it could cause the CPU to skip over security critical instructions.

We did not research BMC based attacks any further. We note there's an opportunity to adopt defense-in-depth measures against possible glitching attacks, such as repeated instructions / majority tests.

We confirmed with Intel engineers that *Plundervolt* software based glitching attack is mitigated on SPR: the overclocking voltages control interface is disabled. Older server parts have an additional microcode that clears SGX keys when the overclocking interface is used.

Conclusions

The space of side channel attacks is vast and evolving. TDX module protects itself and TD workloads against a broad range of publicly known transient execution attacks and traditional side channel attacks. Platform security state is reflected in the attestation report.

The VMM may be able to monitor partial TD activity, as "access oracles" are architecturally possible.

It is the TD guest responsibility to follow best practices, and adopt defensive measures for its sensitive operations.

TDX Logical Integrity and Memory Corruption Attacks

Intel TDX supports two different mechanisms for enforcing memory integrity. Cryptographic Integrity (TDX-CI) and Logical Integrity (TDX-LI). Whereas TDX-CI uses a 28-bit MAC to protect each cache line, TDX-LI relies on a single bit indicating TDX ownership. While this can still provide security guarantees in the absence of software bugs, it makes TDX-LI prone to rowhammer attacks. As part of our review we investigated potential corruption targets for rowhammer attacks.

We are interested in two attacker models: First, an attacker that has full control over all components outside of the TDX TCB. This includes the VMM, BIOS and cooperating TDs.

Second, a less powerful attacker that only controls the VMM and cooperating TDs, but can not influence BIOS operations.

Our attacker is also able to randomly corrupt arbitrary 16-byte[18] long, 16-byte aligned physical memory chunks without breaking the logical integrity check. In practice this can be achieved using rowhammer, but faulty hardware or a software bug in the TDX module that leads to a mismatched-HKID memory write would also result in the same capability.

While row hammering cleartext memory gives an attacker the ability to randomly flip bits in the targeted row, a bit flip in the AES-XTS encrypted memory used by TDX will corrupt a full 16-byte block, making the attack equivalent to a fully random corruption.

In addition, rowhammer attacks can potentially flip the TDX-ownership bit stored within DRAM metadata, giving attackers read and write access to encrypted memory which may be used to mount ciphertext rollback attacks.

The attacker is interested in compromising TDs already running on the host and compromising TDs created in the future. The best way to achieve this capability is by compromising SEAM mode itself, making this the main goal.

Corruption Targets

Potential corruption targets can be split into three categories:

[18] MK-TME encryption uses AES-XTS with a 128-bit block size. Thus, any bit-flip within a 16-byte chunk will corrupt the entire block after encryption/decryption.

TD-memory: Private memory of a running TD encrypted with its Private HKID
TDX-control: TDX module control structures stored in VMM allocated memory either encrypted with the Global Private HKID (TDR Page, PAMT) or a TD Private HKID (TDCS, TDVPS, SEPT).
SEAM-range: Code and data of the TDX module and the Intel P-SEAMLDR module stored in the SEAM memory range.

Both **TD-memory** and **TDX-control** offer a high level of control to the attacker. They can be mapped and remapped by the VMM to arbitrary physical pages on a single page granularity, which can potentially help with Rowhammer attacks. The **SEAM-range** itself offers slightly less control as it can't be remapped and needs to be a continuous 32MB aligned memory block. Targeting **TD-memory** comes with a number of downsides: Attacks have to be TD specific (e.g target software running inside the TD), there is no escalation path into a full SEAM compromise and TDs can defend against blind attacks by randomizing the guest-physical location of high-risk data structures.

Randomly corrupting code of the TDX or P-SEAMLDR modules is difficult, as the 16-byte chunk size is large enough to make blind corruptions hard to pull off[19]. Therefore the bulk of our analysis concentrated on VMM allocated TDX control structures and global data stored in the SEAM-range.

Good corruption targets give an attacker a way to escalate their privileges, while being robust against random corruption so that a wrong corruption result does not lead to a system crash. The following is an incomplete list of promising targets we identified during our review

VMM allocated control structures

TDCS: Root control structure of a TD. Encrypted with the TD private HKID.
Contains the MSR_BITMAP page for intercepting TD MSR accesses. Corrupting this bitmap can lead to unrestricted host MSR access by a malicious TD. Corrupting the "wrong" bits won't have any negative side effects (besides a potential TD crash) so this is a very useful target.
Also contains the executions_ctl_fields.attributes.debug flag to enable or disable debug mode. Neighboring fields seem to be robust against corruption. Easy vector to compromise a running TD, but can't be used for a SEAM compromise. The included epoch_tracking.epoch_and_refcount 16 byte field could be corrupted to bypass TLB tracking.

SEPT: Secure EPT entries. Encrypted with the TD private HKID.
Corrupting SEPT entries gives a clear path to a SEAM compromise. An attacker can corrupt an SEPT entry repeatedly until they get a valid RW entry pointing to an arbitrary physical page.

[19] The most realistic attack vector seems to be the injection of an early 'retn' instruction in a security critical function. This results in a roughly 1/256 success rate, with most attempts resulting in a random code change and a system shutdown.

Adding this page to the Secure EPT (TDH.MEM.SEPT.ADD) of a cooperating TD will give the TD read/write access to its own secure page tables. The TD can then get read-write access to its own VMCS by modifying a SEPT entry and pointing it to tdvps_ptr->management.tdvps_pa[TDVPS_VMCS_PAGE_INDEX]. Again, the VMCS is encrypted with the TD HKID so arbitrary read-write access is possible. VMCS write access can then be escalated to SEAM code execution via various means (for example by modifying the host-state area registers)

PAMT Encrypted with the TD private HKID. Corruption of PAMT entries makes “type confusion” attacks against VMM allocated pages possible if an attacker can mark a PAMT as unused. An attacker can turn this primitive into a SEPT corruption by remapping a SEPT page as a TD private page while it’s still in use. Alternatively, we can also remap the TDCS or TDVPS pages directly. In practice, this attack seems too hard to exploit successfully: Only 1 in 256 bit flips will result in pamt.pt == NDA. In addition, the entry_lock field will be corrupted so acquiring an exclusive pamt_entry lock will fail and an attacker needs to trigger the corruption between lock acquisition and use.

SEAM Range

TDX module global state
At first glance, the most valuable target is the tdmr_table member. By corrupting TDMR entries, an attacker can achieve a similar primitive to the PAMT reuse discussed above. However, random 16-byte corruptions only rarely result in usable values so a successful attack is hard. Corrupting num_of_tdmr_entries can lead to a similar issue as discussed above could potentially be escalated into a SEAM compromise.

P-SEAMLDR
P-SEAMLDR is loaded at a known physical location (end of SEAM range - P_SYS_INFO table), which simplifies attacks against SEAM range components.. pseamldr_data.system_info has size and mask fields that are used in shared_hpa_check security checks. A corruption in one of those fields could lead to shared_hpa_check bypass, which directly impacts SEAM range integrity.

Mitigations

In the above attacks, the attacker has a primitive which corrupts the physical DRAM contents. If the corruption targets TDX private memory (encrypted at rest with MK-TME) on a system using TDX-LI then this results in the aligned 16-byte chunk of plaintext to be corrupted when read by the TDX module or a TD. Modern server-class DDR is partitioned into both a data region and a metadata region – this is where the TDX integrity bit (for TDX-LI) or MAC (for TDX-CI) is stored. If the corruption targets the metadata region then this could potentially flip a TDX private row into a shared row (only TDX-LI with its single bit is a realistic target).

Figure 20: Difference in probabilistic complexity between attacking TDX-CI MAC, TDX-LI without ECC, and TDX-LI with ECC. The attacker can target both the data and metadata regions.

The metadata region is also where the error correction codes are stored if ECC is enabled for the platform. These codes are used to detect and potentially correct errors in both the data and metadata region. In this case, the attacker must now both flip the precise bits required for the previous attacks but also bypass the error detection that will occur on the next read of the data.

In the case of rowhammer, these bit flips are probabilistic (although can be somewhat fine-tuned) so any additional constraints on the flip results will significantly increase the likelihood of success. Critically, the attacker must flip enough bits that both correctable error (CE) and detected but uncorrectable (DUE) are not triggered by the memory controller – this leads to the desired silent data corruption. A miss will trigger either a TDX security violation or an ECC violation on the TD private memory – both of which will prevent further execution of the TD.

Due to this, we believe that ECC is a sufficient mitigation against integrity attacks when using TDX-LI. Therefore, the TDX trusted components must prevent ECC from being disabled (see related vulnerability and remediation).

Attestation

Customers eager to run trusted workloads on a cloud service provider's TDX solution require confidence that their VMs are running on and booted from the expected platform configuration. TDX provides a cryptographic attestation solution that is designed to solve this problem even with an adversarial CSP. The measurement and attestation of TDX is covered in detail in Section 11 of the TDX architecture specification and builds on their existing Data Center Attestation Primitives using SGX. For this security review we focused on the portions of attestation handled by the SEAMLDRs and TDX module, leaving the SGX quote generation out of scope. In particular, we wanted to ensure that previous (potentially vulnerable) versions of the SEAMLDRs and TDX module can't be run without the customer knowing.

At a high level, a TDX attestation report is generated when a customer TD issues the TDG.MR.TDREPORT API call to the TDX module. This API gathers several key measurements from the TD's lifetime, signs them with an ephemeral HMAC key, and returns the signed report to the guest. The guest now passes this signed report to the VMM which in turn hands it to the SGX TD quoting enclave. The SGX enclave is previously provisioned with a signing key from Intel and after verifying the HMAC-signed TD report (it has access to the ephemeral key) the enclave creates and signs a quote. This signed quote can now be passed back to the TD and presented to a 3rd party server which can verify the signature and decide if the TD is running in a trusted state.

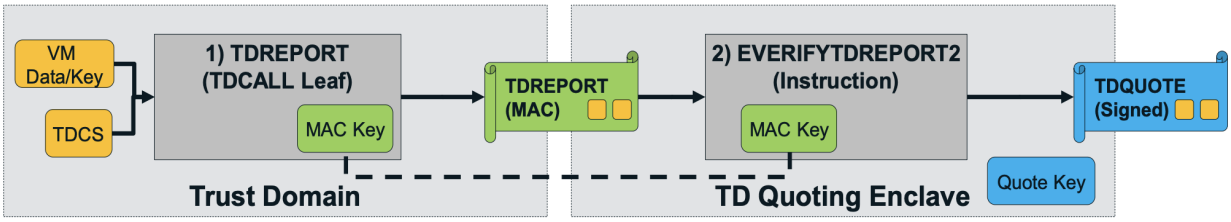

Figure 21: High level TD attestation flow (omitting VMM involvement)[20]

Measurements

The contents of the TD report are critical for 3rd party customers to determine whether a TD is in a trusted state. During TDX initialization and TD initialization, multiple measurements are taken and firmware versions recorded which are incorporated into this report. The coverage of TDX attestation can be broken down into categories of information required to verify a TD is trusted:

- That the code is running as a TD on a TDX-enabled machine
- That the code is running on the expected platform, including FW/ucode versions
- That the code booted the guest firmware (UEFI) they expect
- That the code booted the disk contents (OS, kernel, …) they expect

[20] Taken from Section 2.7 of the TDX architecture specification

The following diagram taken from Intel's TDX architecture specification shows all of the data which is included (either directly or by hash) in the final signed quote.

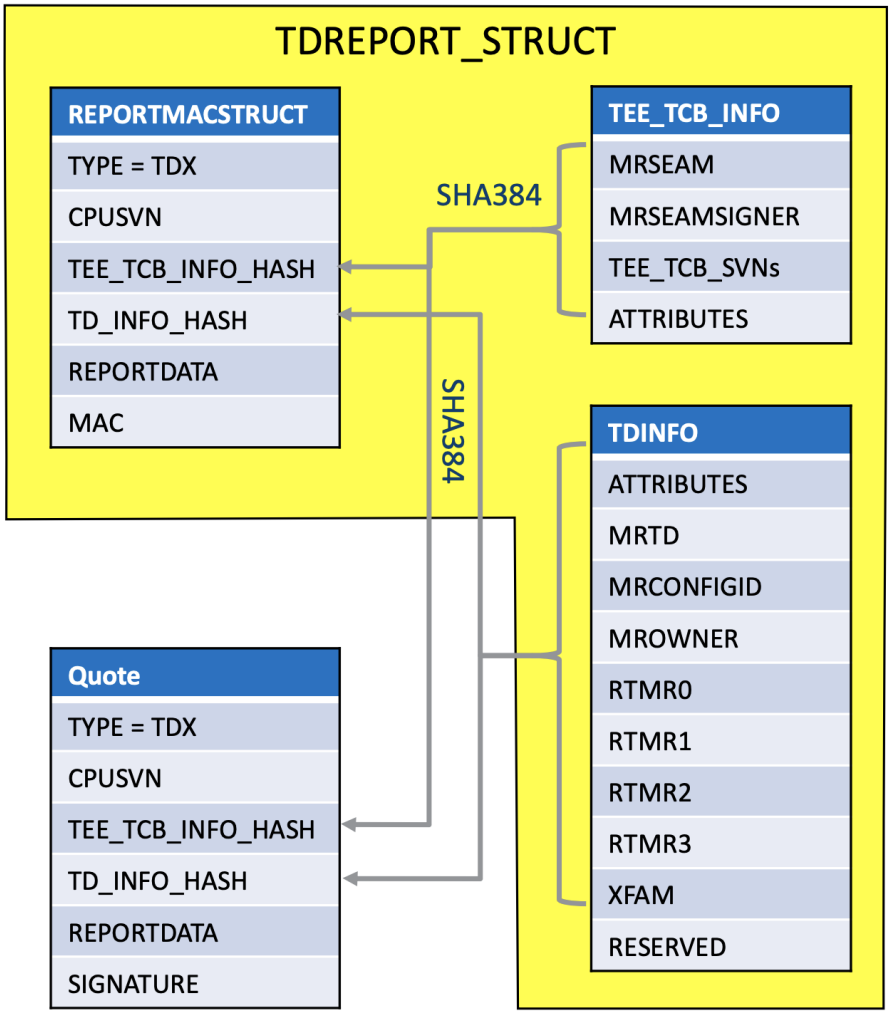

Figure 22: Measurements contained within a TDX attestation quote[21]

The TEE_TCB_INFO attests the CPU and TDX firmware the machine booted with. The MRSEAM field measures the TDX module's contents and the MRSEAMSIGNER measures the SEAM module signer's key (reserved for potential future use with 3rd party SEAM modules). The security-version-number (SVN) fields describe the firmware versions of various CPU components that are critical for TDX and SGX integrity. Additionally, the hardware prevents loading firmware with a lower SVN than currently loaded. Within CPUSVN the processor microcode SVN and NP-SEAMLDR SVN are recorded. Within TEE_TCB_SVN the current and previous (if TDX module was updated during power cycle) SVN of the TDX module are recorded. With this information, TD owners can use the attested data to be confident of the range of versions for microcode and TDX-related code during the current power cycle.

[21] Taken from Section 11.2 of the TDX architecture specification

In addition to the direct measurements listed above, there are a small number of other measurements that the SGX TD quoting enclave includes. Within the SGX PCK certificate, there are fields which indicate additional SVNs and whether the machine is running with SMT enabled or not.

Debug Security

There are various situations where a developer or user of TDX may want to debug a portion of the system. For TDX developers, they require a method for running debug builds with extra instrumentation while also preventing these from running on production hardware or reporting that they did during attestation. Additionally, Intel or the cloud service provider may need to use a JTAG hardware debugger to investigate issues on production machines – the system must ensure that JTAG debugging is impossible after TDX attestation and must prevent production keys from being used or accessed before attestation. Lastly, the customers who own a TD may want to debug their workloads. The TDX system provides several debugging capabilities to address many of these requirements.

For each of these debugging flows, the TDX security properties around confidentiality and integrity protections must either continue to hold or the system must attest otherwise. Additionally, these debugging capabilities must be immutable for a given TD after attestation has occurred.

TD Debugging

Intel categorizes TD debugging into On-TD Debug and Off-TD Debug. The On-TD debug flow involves a TD owner using existing software (e.g., debuggers) and hardware capabilities (e.g., tracing) to enable debugging of software within the TD. This is the default behavior of all TDs and utilizes the limited number of hardware features (BTS, LBR & PT if allowed in XFAM) exposed to the guest.

The Off-TD debug flow requires that the TD be created with the ATTRIBUTES.DEBUG bit. The ATTRIBUTES field is included in the TD attestation report and thus the TD owner can verify this is set according to expectation. When DEBUG is enabled, several additional TDX APIs are exposed to the host VMM and several have modified behavior[22]:

TDH.MNG.(RD|WR)	Several fields in the TDCS and TDR are now readable or writable which were previously not.
TDH.MEM.SEPT.RD	
TDH.VP.(RD|WR)	Secret state within the TDVPS is now potentially readable and writable
THD.MEM.(RD|WR)	All TD guest memory is readable and writable (using the assigned HKID key)

[22] See Section 14.3 from the Intel TDX Architecture Specification for details

TDH.PHYMEM.PAGE.RDMD	

There are additional alternative paths in the TDX module dependent on the ATTRIBUTES.DEBUG field. We reviewed these debug-only paths and did not find any methods to bypass the checks (e.g., reach debug-only APIs on a non-debug TD) – however the DEBUG bit is an attractive target for memory corruption vulnerabilities.

Overall, these relaxed APIs allow the host VMM to completely control the guest TD and **the TD shouldn't be trusted by the TD owner if ATTRIBUTES.DEBUG is set in the attestation**. The ATTRIBUTES field for a given TD is immutable and therefore a TD owner can be confident that if the attestation report states DEBUG is disabled that it was never previously enabled for the current boot cycle.

TDX System Debugging

There are various methods in which the host system may be debugged, either during TDX development or to diagnose a production issue. In any of these cases, TDX must take action to ensure that the debug state is reflected in the attestation and potentially remove access to any production keys.

If JTAG or SGX debugging is enabled by DFx unlock, the TDX module detects this by reading the SGX_DEBUG_MODE MSR and sets the SeamUnderDebug bit in the internal SEAMEXTEND structure. The contents of this structure is later used by the SEAMOPS[SEAMREPORT] microcode which generates the HMAC-signed report that is then presented to the SGX quoting enclave to create a signed attestation quote. From this, the TD owner can determine whether SGX debugging was enabled or not. Additionally, when SGX debug is enabled the production keys are cleared which will lead to the TD quote having an invalid signature.

The security of the DFX system itself (unlocking procedure and attack vectors) was outside the scope of this review. However, there has been a string of research into this area for previous Intel architectures including work that has enabled access to internal CPU tracing and microcode modifications on older Goldmont CPUs. Intel's threat model for TDX places Intel insiders (i.e., DFX access) outside of the TCB, meaning that even if DFX is unlocked it should not degrade any TDX security properties. While there are no publicly known vulnerabilities in the Sapphire Rapids DFX system, its presence adds some potential security risk due to the high level of system access granted assuming undiscovered implementation flaws may exist that violate Intel's design. .

Security Review Results

This report contained the results from the TDX security review. The review process was undertaken over a 9 month period in 2022. It encompassed documentation, design and code review steps to identify the widest range of security issues. Some areas of review were out of scope. For example, uncore and the MCHECK implementation were only reviewed based on their externally visible behavior.

The review focussed on discovering any security issues which would compromise the security of the secure TD VM, leak sensitive information to untrusted code running outside of the TCB or deny service to the entire system. Where possible tooling was used to improve the comprehensivity of the review, such as Wycheproof for testing cryptography or weggli to aid in source code review and analysis.

The review covered 81 potential attack vectors and resulted in 10 confirmed security issues and 5 defense in depth changes[23]. Of the 10 confirmed security issues, 9 were fixed in the TDX code. The final issue necessitated a change to the guide for writing a BIOS to support TDX so that the issue would not be present in a production system. Where possible the review performed variant analysis of any discovered issues to determine if the same pattern could be identified in other areas of the code base. All confirmed issues were mitigated before the production release of the 4th gen Intel Xeon Scalable processors.

The 9 issues which resulted in code changes were resolved before the final product release. Although these issues were not assigned CVE identifiers, Intel internally assigned CVSS v3.1 scores to gauge the severity of the issues if they were present in a production system. The most serious issue discovered was the Exit Path Interrupt Hijacking when returning from ACM mode. This was assigned a CVSS score of 9.3 which indicates the serious nature of the issue which would allow arbitrary code in the privileged ACM execution mode.

It's positive to note that of the security issues discovered only 2 would be considered memory safety issues. By far the most common class of security issues discovered were logical bugs due to the complexity of Intel processors generally, and the TDX feature specifically. For example the Exit Path Interrupt Hijacking issue was a result of the complex set of steps necessary to switch between the privileged ACM mode and normal operating mode. Completely eliminating these logical issues is much more difficult than moving to a memory safe language such as Rust.

The review met its expected goals and was able to ensure significant security issues were resolved before the final release of Intel TDX. Overall the review provided Google with a better understanding of how the TDX feature functions which can be used to guide deployment. In conclusion the security review team found that the design and implementation of Intel TDX as deployed on the 4th gen Intel Xeon Scalable processors meets a high security bar.

[23] Not all discovered issues are described in this document for brevity.

Acknowledgments

The review team acknowledges the contributions of the following people for making the review possible through dealing with technical questions, giving access to necessary resources and providing security expertise.

We would like to extend our gratitude to the following Intel engineers: **Arie Aharon, Baruch Chaikin, Boaz Tamir, Dhinesh Manoharan, Dror Caspi, Fahimeh Razaei, Nagaraju Kodalapura, Truc Nguyen**.

Finally, we would also like to thank the following Google engineers who provided technical support and guidance throughout this security assessment: **Andres Lagar-Cavilla, Christian Ludloff, Arthur Wongtschowski**.

Appendix A - MSRs of Interest

The following table lists all public MSRs we identified that contain address fields. See the MSR section for details on why we reviewed these.

Register Type	Register Offset	Name	Address Type	Write Access
MSR	0x0000001b	IA32_APIC_BASE	Physical	BIOS + OS
MSR	0x000017d0	HW_FEEDBACK_PTR	Physical	BIOS + OS
MSR	0x00000984	IA32_TME_EXCLUDE_BASE	Physical	BIOS + OS
MSR	0x00000793	EXTENDED_MCG_PTR	Physical	BIOS + OS
MSR	0x000001f5	PRMRR_MASK	Physical	BIOS
MSR	0x00000200	MTRR_PHYS_BASE_n	Physical	BIOS + OS
MSR	0x000002a0	PRMRR_BASE_0	Physical	BIOS
MSR	0x00000560	RTIT_OUTPUT_BASE	Physical	BIOS + OS
MSR	0x00000572	RTIT_CR3_MATCH	Physical	BIOS + OS
MSR	0x000001f2	SMRR_BASE	Physical	BIOS
MSR	0x000001f6	SMRR2_BASE	Physical	BIOS
MSR	0x000006a8	INTERRUPT_SSP_TABLE	Physical	BIOS + OS
MSR	0x00001400	SEAMRR_BASE	Physical	BIOS
MSR	0x00001401	SEAMRR_MASK	Physical	BIOS
MSR	0x00001402	SEAM_EXTEND	Physical	SEAM only
MSR	0xc0000100	FS_BASE	Virtual	BIOS + OS
MSR	0xc0000101	GS_BASE	Virtual	BIOS + OS
MSR	0xc0000102	KERNEL_GS_BASE	Virtual	BIOS + OS

Appendix B - Public Resources

Intel have published architectural specifications, white papers and source code for the TDX feature on their website. The following is a list of the most important information the review team has determined would be useful for further public research on the security of TDX. The links are correct at the time of publication.

- Trust Domain Extensions Whitepaper
- Architecture Specification: TDX Module v1.0
- TDX Loader v1.0 Source Code
- TDX Module v1.0 Source Code

The following is a non-exhaustive list of public tooling that the team used for the security review process.

- Project Wycheproof
- Weggli
- Frama-C