Formal Verification of Virtualization-based Trusted Execution Environments

Hasini Witharana, *Student Member, IEEE,* Hansika Weerasena, *Student Member, IEEE,* and Prabhat Mishra, *Fellow, IEEE,*

***Abstract*—Trusted Execution Environments (TEEs) provide a secure environment for computation, ensuring that the code and data inside the TEE are protected with respect to confidentiality and integrity. Virtual Machine (VM)-based TEEs extend this concept by utilizing virtualization technology to create isolated execution spaces that can support a complete operating system or specific applications. As the complexity and importance of VM-based TEEs grow, ensuring their reliability and security through formal verification becomes crucial. However, these technologies often operate without formal assurances of their security properties. Our research introduces a formal framework for representing and verifying VM-based TEEs. This approach provides a rigorous foundation for defining and verifying key security attributes for safeguarding execution environments. To demonstrate the applicability of our verification framework, we conduct an analysis of real-world TEE platforms, including Intel's Trust Domain Extensions (TDX). This work not only emphasizes the necessity of formal verification in enhancing the security of VM-based TEEs but also provides a systematic approach for evaluating the resilience of these platforms against sophisticated adversarial models.**

***Index Terms*—Property Checking, Confidential Computing, Confidentiality, Integrity, Trusted Execution Environments**

I. Introduction

As the nature of computing evolves, ensuring the security and trustworthiness of sensitive data and critical applications has become an important concern. With the rapid growth of cloud computing, edge devices, and the Internet of Things (IoT), the need for robust security measures has never been more critical. Trusted Execution Environments (TEEs) emerge as a promising solution to improve security by providing a secure environment for the execution of sensitive code with sensitive data and the protection of confidential information [1]. TEEs offer a secure execution environment that is isolated from the rest of the system, safeguarding against various threats such as malicious software, unauthorized access, and hardware-based attacks. TEEs utilize hardware and software components to establish a secure environment where cryptographic operations, key management, and other critical and confidential tasks can be performed with security assurance.

TEEs come in various forms, each tailored to meet specific requirements and challenges. Figure 1 shows three types of TEEs: Enclave-based TEEs (e.g., Intel Software Guard Extensions (SGX) [2], Sanctum [3]), Virtual Machine (VM)-based TEEs (e.g., Intel Trust Domain Extensions (TDX) [4], AMD Secure Encrypted Virtualization (SEV) [5]), ARM Confidential Computing Architecture (CCA), and TEEs for embedded systems (e.g., Keystone [6]). Enclave-based TEEs leverage hardware-supported isolation to create secure enclaves within a processor. These enclaves are isolated regions of memory resistant to external tampering and surveillance, ensuring the integrity and confidentiality of the code and data. Intel SGX is a prime example of an enclave-based TEE, allowing developers to create secure enclaves for the execution of sensitive operations without revealing the data to the underlying system. Virtual Machine-based TEEs take advantage of virtualization technologies to create secure execution environments within virtual machines. Intel TDX and AMD SEV are some examples of VM-based TEEs. They extend security to the virtualization layer by protecting against attacks even in the presence of compromised hypervisors. Embedded system-based TEEs are designed to cater to the unique constraints and requirements of embedded systems and IoT devices. For example, Keystone integrates with RISC-V architectures to provide hardware-enforced memory protection and secure execution environments, making it well-suited for resource-constrained embedded systems.

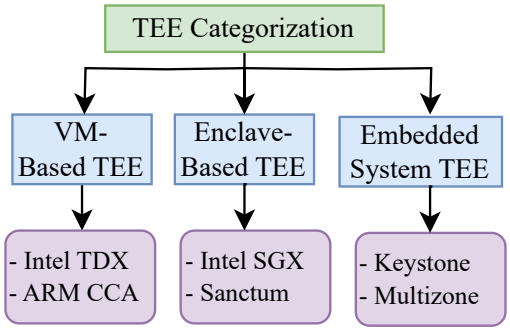

Fig. 1: Categories of trusted execution environments (TEE).

VM-based TEEs are important in cloud computing due to their ability to offer scalable and flexible security solutions that are well-suited to the dynamic nature of cloud services. Unlike enclave-based TEEs, which are designed for securing small pieces of sensitive code and data within tightly controlled memory regions, VM-based TEEs can secure entire virtual machines, offering a broader and more flexible approach to isolation and security in cloud environments. In this paper, we present a formal verification framework for VM-based TEEs for confidentiality and integrity.

Figure 2 presents an overview of our formal verification framework for security verification of TEE architectures. We

H. Witharana, H. Weerasena, and P. Mishra are with the Department of Computer & Information Science & Engineering, University of Florida, Gainesville, Florida, USA.

first conduct an abstraction of the TEE architecture that accurately represents the TEE behavior by following the specification. This abstraction phase simplifies the specification, focusing on the essential aspects relevant to confidentiality and integrity. Next, we develop a formal model for VM-based TEE architectures based on the abstraction. Then, we derive properties related to confidentiality and integrity from the TEE specification. Finally, we perform property checking to verify whether the TEE formal model satisfies the specified properties, ensuring that the TEE architecture meets the predefined security criteria on confidentiality and integrity. Specifically, this paper makes the following major contributions:

1) We present a comprehensive formal model that defines the security boundaries of confidential VMs, explicitly considering the capabilities of adversaries with access to advanced attack vectors.
2) We formally model confidentiality and integrity properties, tailoring them for VM-based trusted execution environments.
3) We introduce a detailed formal model for the Intel TDX architecture, developed using the Rosette language.
4) We formally verify the confidentiality and integrity of Intel TDX for code and data in use.

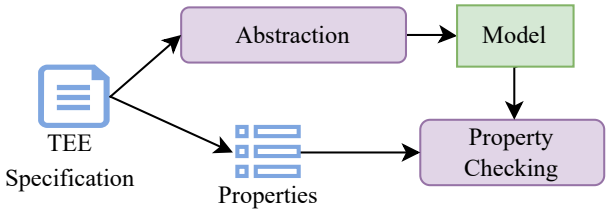

Fig. 2: Overview of our formal verification framework.

This paper is organized as follows. Section II provides relevant background and surveys related efforts. Section III defines the formal model for virtual machine and adversary. Section IV provides formal definitions for both confidentiality and integrity. Section V conducts a security analysis of Intel TDX for confidentiality and integrity. Section VI and VII provide details of formal modeling of TDX architecture and cache. Section VIII discusses the results of the formal analysis. Finally, Section IX concludes the paper.

II. Background and Related Work

This section first provides relevant background on VM-based TEEs. Next, it surveys related efforts in security verification of TEE architectures.

A. Background: VM-based TEEs

We first introduce virtual machines (VM). Next, we discuss confidential VMs. Finally, we provide an overview of Intel TDX, which implements confidential VM architecture.

1) Virtual Machine (VM): A VM enables software-based emulation of physical computers. This technology allows for running an operating system (OS) and applications within an isolated and encapsulated environment. VMs facilitate multiple OSes to operate concurrently on the same physical hardware. This is achieved through virtualization, which significantly enhances resource utilization, flexibility, and isolation in computing environments. At the heart of a VM lies the hypervisor, or Virtual Machine Monitor (VMM), a critical component tasked with managing and allocating the physical resources among VMs. Hypervisors come in two varieties: Type 1 (bare-metal), which operates directly on the host hardware, and Type 2 (hosted), which functions on top of an existing OS.

The VMM orchestrates access to hardware components such as CPUs, memory, storage, and network interfaces, enabling the seamless and concurrent operation of multiple VMs on a single physical machine. The core virtualization concept: hardware abstraction allows each VM to operate as though it has its own dedicated hardware. Each VM hosts its own guest operating system, providing an independent operating environment that interacts with the virtualized hardware, ensuring that applications run in a manner that is both efficient and isolated from the host system and other VMs. Even though VMs are isolated from each other, they are not entirely separate entities in terms of security. The shared use of the hypervisor, underlying hardware, memory subsystem, and other components of the virtualization stack introduces potential vulnerabilities. These shared resources can become attack vectors, where a malicious entity might exploit one VM or host system to gain unauthorized access to or influence over others or the host system itself. This inherent risk highlights the critical need for confidential VMs.

2) Confidential Virtual Machines: Confidential VMs are designed to protect against threats, including malicious insiders, compromised hypervisors, and other potential vulnerabilities in the virtualization stack. Confidential VMs use memory encryption with a unique key for each VM to protect the contents of the VM's memory from unauthorized access. This ensures confidentiality by ensuring the memory contents remain inaccessible and secure from external threats and internal attackers gaining access to physical memory. Furthermore, they also provide integrity protection mechanisms to verify that data and code have not been tampered with. The creation of confidential VMs often utilizes hardware-based security features offering a level of protection that extends even to the host hypervisor. Confidential VMs often use secure boot mechanisms and attestation processes. A secure boot ensures that only authenticated and trusted code is executed during the VM's startup. This works against malicious software and rootkits that might attempt to load during the boot process. Attestation verifies the integrity and authenticity of the VM for external entities. Specifically, attestation allows a third party to confirm that the VM is running the expected software stack.

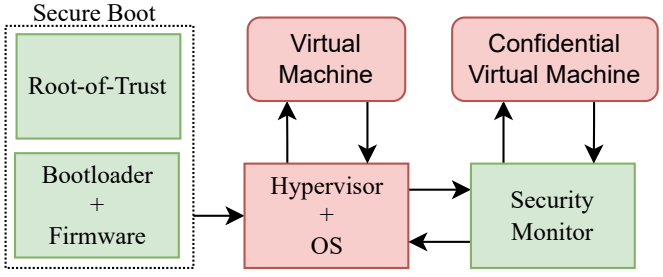

Fig. 3: Overview of VM-based confidential computing.

Figure 3 shows the basic building blocks required for VM-based TEE architecture. It starts with a secure boot process; the system relies on a foundational security mechanism known as the Root of Trust (RoT), complemented by the principle of a chain of trust. The RoT is pivotal for ensuring that only authenticated and integrity-verified firmware and software are loaded for execution. It achieves this through the provision of essential cryptographic functions and services. Initially, the RoT verifies the integrity and authenticity of the bootloader and establishes the first link in the chain of trust. Once the bootloader is authenticated, it securely loads and verifies the firmware, setting the stage for the execution environment. Hypervisor can create and manage VMs. In a type 2 hypervisor configuration, a host OS will work alongside the hypervisor, while type 1 will have only a hypervisor. A fundamental element of a VM-based confidential computing framework is the security monitor. It operates at a low level, closely interacting with the hypervisor and host operating system to monitor and control access to resources, manage permissions, and ensure isolation between different confidential VMs. The security monitor's primary objectives include preventing unauthorized access to sensitive data, ensuring that software components cannot interfere with each other maliciously, and enforcing compliance with security protocols. Intel TDX module [4] is one of the examples of a secure monitor.

3) Intel TDX: Intel TDX [4] is an example of a confidential VM architecture. TDX provides the infrastructure to create hardware-isolated virtual machines known as trust domains (TDs), which are designed to operate securely within a system, separate from the virtual machine monitor or hypervisor and any other unrelated software entities. To protect the confidentiality and integrity of the code and data within a TD, Intel TDX uese technologies such as Multi-Key Total Memory Encryption (MKTME) and hashing techniques. Figure 4 shows an overview of the Intel TDX architecture. Intel TDX is engineered to function within a Secure Arbitration Mode (SEAM), which is an extension to the prior Virtual Machine Extension (VMX) architecture. SEAM introduces a new VMX root operation mode, referred to as SEAM root, specifically constructed to support CPU-attested modules that establish Trust Domains for virtual machine guests. The SEAM operation is divided into two logical modes: TDX non-root mode for the TD guest operations and TDX root mode, which is reserved for host-side activities.

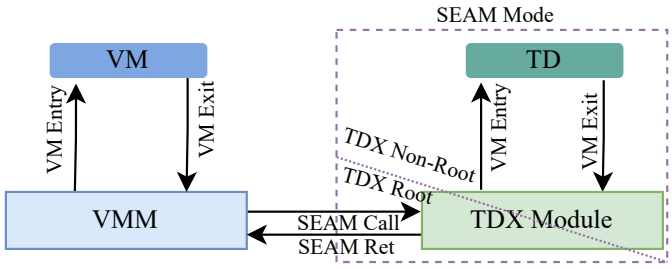

Fig. 4: An overview of Intel TDX architecture.

B. Related Work

This section surveys related efforts, including static analysis, simulation-based testing, and formal verification.

1) Static Analysis: Google's security review of Intel TDX employed static analysis tools to uncover numerous attack vectors and security issues [7]. Security review discovered 81 potential avenues for attacks, confirmed 10 security flaws, and made 5 modifications to enhance the code's defense mechanisms. The review assessed four components of Intel TDX, including the MCHECK mechanism in BIOS, the non-persistent SEAM loader, the persistent SEAM loader, and the design of the TDX module. However, this approach did not provide formal security guarantees. In fact, this security review highlights the need of formal verification.

2) Simulation-based Validation: Simulation-based testing methodologies have been used to evaluate the security of TEEs. Google's examination of AMD SEV technology through simulation-based testing uncovered critical vulnerabilities [8]. This hands-on approach allows for a practical assessment of TEE security. Simulation-based verification faces the exponential input space complexity to cover all possible scenarios [9], [10]. Nevertheless, the lack of formal security guarantees limits the ability of simulation-based testing to guarantee the security properties of TEE systems.

3) Formal Verification: ProveriT [11] provides a theorem proving solution to formally verify Global Platform TEE common criteria. The authors in [12] develop a formal model for memory isolation that includes a detailed formalization of the ARMv8 architecture's hardware components associated with memory isolation, as well as the formalization of a TrustZone monitor that facilitates switching between secure and non-secure worlds. A recent study [13] introduces a verification methodology for ARM TrustZone using property checking techniques. Sardar et al. [14], [15] formally specifies the attestation mechanism using ProVerif's specification language. This work only focuses on the attestation process, whereas our work focuses on memory confidentiality and integrity of Intel TDX. The authors in [16] present a methodology for formally modeling and proving the security of a security monitor, which is a key component of VM-based confidential computing systems.

While there are promising efforts for formally verifying different types of TEE architectures, including verification of Intel SGX [17], verification of ARM Trustzone [12], and verification of RISC-V based TEEs [16]), existing formal verification solutions cannot be directly applied to the TDX architecture due to their inherent differences in the implementation of the TEE architecture.

III. Formal Modeling of VM and Adversary

In this section, we first define a formal model for virtual machine, including VM state, VM inputs, and VM outputs. Next, we define a formal model for the adversary. Throughout the paper, the symbol $=$ is used to denote intensional equality, which asserts that two expressions or variables are equivalent in all respects, including their state, value, or configuration. Similarly, the symbol $\Leftrightarrow$ represents an equivalence relation or extensional equality in our context.

A. Formal Model for Virtual Machines

A VM is initiated with a specific allocation of resources, including CPU cores, memory, and storage. The virtualization platform often uses a unique identifier or configuration snapshot of the foundational state, enabling users to guarantee that the VM was initialized according to the predefined settings. The VM's configuration includes the boot sequence with OS image, and the virtual hardware components assigned to the VM, such as memory size, and disk space.

VM: A user deploys a virtual machine denoted as v. The attributes of this virtual machine include a unique identifier for the VM ($v.id$), the VM's OS image ($v.os$), VM's virtual address list ($v.valist$), VM's memory size ($v.mem$), and VM's data and code pages ($v.data$). These attributes define the configuration of the virtual machine, enabling it to perform designated computing tasks within a virtualized environment.

VM State: At any given moment, the host machine exists in a certain state (m). The state of the VM, $S_v(m)$, can be seen as a specific instance of the overall system state, capturing key operational data. This includes the virtual memory mapping *Vmem* : *Va* $\rightarrow$ *W*, which represents a function from virtual addresses (*Va*) to their corresponding values in machine words (*W*); a set of general-purpose registers *regs* : $\mathbb{N} \rightarrow$ *W* indexed by natural numbers; the program counter *pc* : *Va*, indicating the VM's current execution point; and the VM's attributes, which are established at the VM's creation and remain unchanged during its operation. The initial state $init_v$ defines the starting condition of the VM's memory (*Vmem*) at the time of its instantiation. For simplicity, $init_v(S_v(m_v))$ denotes that $S_v(m_v)$ is in its initial state, as configured before any operations have been executed within the VM.

VM Inputs: The inputs to the VM, $I_v(m) = (I_v^D(m) + I_v^R(m))$, can be categorized into two main types: external inputs and internal inputs. External Inputs ($I_v^D(m)$) can change the state of a VM, such as initializing it to runnable. These inputs are received from the external environment and, given the VM operates in a potentially hostile environment, may come from sources under adversarial control. However, they consist of a pre-defined set of instructions, making the impact of these inputs deterministic. On the other hand, internal inputs ($I_v^R(m)$) run inside the VM, such as code and data, where the impact of the internal inputs can be variable.

VM Outputs: The outputs of the VM, $O_v(m)$, project the machine state of the VM. VM output can have both encrypted data in memory and decrypted data inside the processor. Specifically, $O_v(m)$ focuses on memory elements while data is in use.

VM Execution: The execution of a VM is modeled as a deterministic process with respect to input $I_v(m)$, where the next state of the VM, is a function of its current state, $S_v(m)$, and its inputs, $I_v(m)$. We assume that one virtual CPU for each VM and single thread is used per applications in the VM. We also assume that code and data running inside the VM is not malicious. Therefore, it is safe to assume that $I_v^R(m)$ does not lead to non-deterministic process. Given the deterministic assumption, the VM's execution at any step can be defined by the transitive closure of the transition relation $m_i \rightsquigarrow m_j$, indicating that the VM can transition from state m_i to state m_j based on the operational semantics of its instruction set. This transition relation implies a set of all possible states that can be reached from a given state, directly or indirectly, through multiple steps or transitions. It essentially expands the basic transition relation to include not just direct successors but all reachable states. This transition process involves first identifying the next instruction to execute based on the current state of the virtual memory (*Vmem*) and the program counter (*pc*). Following this, the identified instruction is executed, which may involve bitvector operations, memory accesses, and interactions with VM-specific primitives for security, randomness, and I/O operations.

B. Formal Model for Adversary

A confidential VM operates under the assumption of a privileged adversary who has compromised all software layers except for the confidential VM platform itself (security monitor). This section defines the adversary's potential actions and their implications for the VM.

Adversary State: The privileged adversary is capable of pausing the VM at any moment, executing arbitrary instructions that can modify the adversary's state ($A_v(m)$), the VM's inputs ($I_v(m)$), and can initiate or terminate VM instances. We show the adversary's influence through the *attack* relation over pairs of states: $(m_1, m_2) \in$ *attack* if the attacker can transition the system's state from m_1 to m_2. A key constraint is that the *attack* operation cannot alter the confidential VM state, ensuring $S_v(m_1) = S_v(m_2)$. This maintains the integrity of the VM despite the adversary's actions.

Here, *attack* is a subset of the transition relation $\rightsquigarrow$, indicating that an adversary's actions are confined to utilizing the platform's instructions to alter the system's state. Furthermore, the *attack* relation is reflexive, denoting that the adversary might choose not to alter the state: $\forall m.(m, m) \in$ *attack*. This model allows the adversary to operate concurrently with the VM, with the capability to modify the machine's state before the VM's launch and to alter the VM's initial state.

Adversary Monitoring: In a confidential VM environment, untrusted software, including potential adversaries, may observe aspects of the VM's execution. These observations are contingent on the confidentiality protections enforced by the VM platform. While explicit outputs are invariably observable, adversaries might also detect patterns through indirect means such as side channels, including memory access patterns and computational timing.

The capability for an adversary to make observations is formalized through the execution of arbitrary instructions or the utilization of platform primitives. These actions allow the adversary to monitor the effects of their operations on the VM's state. Let $monitor_v(m)$ denote the result of an observation for the machine state m. For instance, an attacker that only observes outputs enjoys the monitor function $monitor_v(m) \doteq O_v(m)$. Observations by an adversary may include explicit data produced by the VM's computational results intended

for external consumption. Also the observations may include indirect information that can be inferred from the VM's operation, such as timing information, power consumption patterns, or memory access patterns. These observations require more sophisticated analysis and may reveal sensitive information without direct access to the VM's data.

VM Execution with an Attacker: An execution trace of the VM is an unbounded-length sequence of states, denoted by $\sigma = (m_0, m_1, \ldots, m_n)$, satisfying the condition $\forall i. m_i \rightsquigarrow m_{i+1}$; here, $\sigma[i]$ refers to the i-th element of the trace. Considering the ability of the attacker to pause and resume the VM at any time, we define the VM's execution as the sequence of states from σ where the VM is actively executing.

To identify when the VM is executing, we use the function $curr(m)$ to denote the current mode of the platform, with $curr(m) = v$ if the platform is executing the VM (v) in state m. Using this function, we can extract the steps in σ where the VM is executing, resulting in a subsequence $(m'_0, m'_1, \ldots, m'_m)$ where $init(S_v(m'_0)) \wedge \forall i. curr(m'_i) = v$. This subsequence represents the VM's execution trace, including inputs, execution states, and outputs at each step. Given the VM's execution trace, the attacker may perform attack actions between any two consecutive steps, represented as $\forall i.(m'_i, m'_{i+1}) \in attack$. This action effectively introduces uncertainty in the VM's state and inputs, providing the VM with potentially fresh inputs at each step.

The semantics of a VM, denoted by $[v]$, is defined as the set of all possible finite or infinite execution traces, capturing every possible input sequence. Formally:

$$[v] = \{(I_v(m'_0), S_v(m'_0), O_v(m'_0)), \ldots | init(S_v(m_0))\}$$

This model accounts for all potential input sequences because the VM may receive any value of input at any step. Furthermore, $[v]$ is prefix-closed, acknowledging that the attacker can pause and terminate the VM's execution at any time. The determinism of the VM's program means that a specific sequence of inputs uniquely identifies a trace from $[v]$ and determines the expected execution trace under that sequence of inputs.

Table I provides a summary of notation used in defining formal models for both virtual machines and adversary.

TABLE I: Table of notations for defining formal models for virtual machines and adversary.

Symbol	Description
v	A virtual machine instance.
$v.id$	Unique identifier for the VM.
$v.os$	The operating system image used by the VM.
$v.valist$	List of virtual addresses assigned to the VM.
$v.mem$	The allocated memory size for the VM.
$v.data$	The data and code pages within the VM.
$S_v(m)$	The state of the VM at a given moment m.
$I_v(m)$	Inputs to the VM at moment m, comprising external ($I_v^D(m)$) and internal ($I_v^R(m)$) inputs.
$O_v(m)$	Outputs from the VM at moment m.
$\rightsquigarrow$	The transition relation for the VM's execution.
$A_v(m)$	The state of the adversary with respect to VM v at moment m.
$monitor_v(m)$	The result of an adversary's observation at machine state m.
σ	An execution trace of the VM.
$curr(m)$	A function denoting the current execution mode of the platform at state m.
$[v]$	The semantics of a VM, representing the set of all possible execution traces.

IV. Formal Modeling of Confidentiality and Integrity Properties

In this section, we provide the formal definition for confidentiality and integrity properties with respect to VM-based trusted execution.

Let $\lambda(v)$ denote the measurement of a VM instance v, computed upon its launch. This measurement process guarantees:

$$\begin{aligned}\forall m_1, m_2.\ &\text{init}_{v1}(S_{v1}(m_1)) \wedge \text{init}_{v2}(S_{v2}(m_2)) \\ &\Rightarrow \lambda(v1) = \lambda(v2) \\ &\Leftrightarrow S_{v1}(m_1) = S_{v2}(m_2)\end{aligned}$$

This measurement process involves computing a cryptographic hash of the VM's initial content and configuration, providing a unique identity for the VM that serves as the basis for authenticating its legitimacy. This hash serves as a fingerprint of the VM at a particular point in time. The measurement process asserts that any two VM instances with the same measurement must have identical initial states, ensuring that any deviation from the expected VM program is detectable by the user. This assertion is based on the cryptographic property of collision resistance, which implies that it is computationally infeasible to find two distinct inputs (in this case, VM states) that result in the same hash output.

A. Confidentiality

Confidentiality ensures that a privileged software attacker cannot distinguish between the executions of two VMs, except for what is revealed through observable outputs. An attacker cannot gain information about the VM's execution state or internal processes beyond what is explicitly allowed through the monitoring function, denoted as *monitor*. This function provides all observations, including initial configurations, outputs to non-VM memory, and any potential side channel leakages. To formally assert the confidentiality guarantee, we propose the following:

$$\begin{aligned}&\forall \sigma_1, \sigma_2. \Big(A_{v1}(\sigma_1[0]) = A_{v2}(\sigma_2[0]) \wedge \\ &\forall i. \big(\text{curr}(\sigma_1[i]) = \text{curr}(\sigma_2[i]) \wedge I_{v1}(\sigma_1[i]) = I_{v2}(\sigma_2[i])\big) \wedge \\ &\forall i. \big(\text{curr}(\sigma_1[i]) = v\big) \Rightarrow \\ &\text{monitor}_{v1}(\sigma_1[i+1]) = \text{monitor}_{v2}(\sigma_2[i+1])\Big) \\ &\Rightarrow \Big(\forall i. A_{v1}(\sigma_1[i]) = A_{v2}(\sigma_2[i])\Big)\end{aligned}$$

This formulation implies that for any two traces, σ_1 and σ_2, that exhibit equivalent attacker operations and observations (as permitted by monitor) but may differ in their private VM states and internal executions, the observable outcome to the attacker must be identical. Here $\sigma[i]$ means the i-th index of the execution trace. The input that is responsible for i-th state in the trace is denoted as $I(\sigma[i])$ and the corresponding

output is denoted as $O(\sigma[i])$. By adhering to this model, a VM platform ensures that all potential traces of VM execution, which may yield the same observable outputs but originate from distinct internal states, remain indistinguishable to an external observer. This guarantees that the VM's confidentiality is preserved, preventing attackers from leveraging observable information to infer sensitive internal states or execution paths.

Figure 5 shows the confidentiality property. Let's assume that the two traces start with an equivalent state and differ from state m_1. This is because the two VMs can perform different computations. Adversary monitoring is assumed to be the same in both traces. Also, adversary actions are assumed to be the same in both traces. This should lead to the adversary state being identical in each step. The confidentiality property implies that the adversary state only depends on the adversary's actions and the initial state. Therefore, whatever the VM state is, it should not affect the adversary state. This shows that the adversary can only know information through the monitor function and not more.

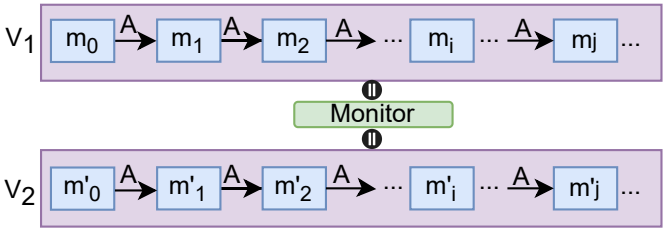

Fig. 5: An overview of confidentiality property.

B. Integrity

The integrity property states that the execution trace of the VM is solely determined by the sequence of inputs, independent of any interference by privileged software attackers beyond the provision of inputs. The integrity of a VM execution ensures that the VM's operational sequence and its resultant states and outputs are determined solely by its sequence of inputs. Operations by an attacker, such as manipulation of I/O peripherals or execution of privileged instructions, should not deviate the VM's execution from its intended path. The integrity property can be formalized as:

$$\forall \sigma_1, \sigma_2. \Big(S_{\mathrm{V}}(\sigma_1[0]) = S_{\mathrm{V}}(\sigma_2[0]) \wedge$$
$$\forall i. \big(\mathrm{curr}(\sigma_1[i]) = \mathrm{V}\big) \Leftrightarrow \big(\mathrm{curr}(\sigma_2[i]) = \mathrm{V}\big) \wedge$$
$$\forall i. \big(\mathrm{curr}(\sigma_1[i]) = \mathrm{V}\big) \Rightarrow I_{\mathrm{V}}(\sigma_1[i]) = I_{\mathrm{V}}(\sigma_2[i]) \Big)$$
$$\Rightarrow \Big(\forall i. S_{\mathrm{V}}(\sigma_1[i]) = S_{\mathrm{V}}(\sigma_2[i]) \wedge O_{\mathrm{V}}(\sigma_1[i]) = O_{\mathrm{V}}(\sigma_2[i]) \Big)$$

This states that if two execution traces, σ_1 and σ_2, begin with identical initial states and receive the same sequence of inputs, then despite any differences in the attacker's operations across the traces, the VM's state transitions and outputs will remain consistent across both traces.

This integrity model emphasizes a crucial aspect of VM security: the system's ability to maintain a predictable and reliable execution path, even when under adversarial influence. It ensures that the VM's computation integrity is preserved, thereby guaranteeing that the execution outcomes are solely the result of the provided inputs and the VM's deterministic behavior. The determinism and equivalence of VM execution can be formalized as:

$$\forall \sigma_1, \sigma_2. \Big(S_{v1}(\sigma_1[0]) = S_{v2}(\sigma_2[0]) \wedge$$
$$\forall i. \big(\mathrm{curr}(\sigma_1[i]) = v1\big) \Leftrightarrow \big(\mathrm{curr}(\sigma_2[i]) = v2\big) \wedge$$
$$\forall i. \big(\mathrm{curr}(\sigma_1[i]) = v1\big) \Rightarrow I_{v1}(\sigma_1[i]) = I_{v2}(\sigma_2[i]) \Big)$$
$$\Rightarrow \Big(\forall i. S_{v1}(\sigma_1[i]) = S_{v2}(\sigma_2[i]) \wedge O_{v1}(\sigma_1[i]) = O_{v2}(\sigma_2[i]) \Big)$$

This formalism establishes that if two VM instances start with the same initial state and receive identical input sequences, then their execution traces, including state transitions and outputs, will be equivalent. This equivalence emphasizes the determinism property of the VM platform's execution model, ensuring that VM programs operate predictably and securely even in the presence of potential attackers.

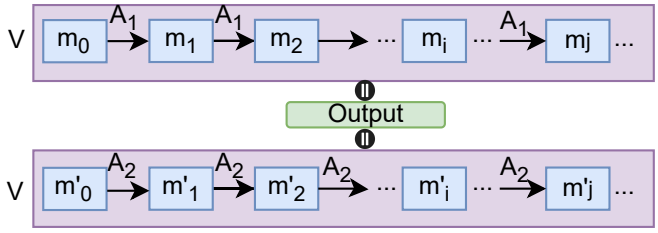

Fig. 6: Overview of integrity property.

Figure 6 shows the integrity property. Actions by the adversary are marked as A_1 and A_2. Let's assume that the VM's inputs and actions remain consistent across both traces. Similarly, the initial conditions of the VMs are identical. The adversary's actions are specified through the *attack* function, allowing for possible variations between the traces. The integrity verification necessitates demonstrating that the state and outputs of the VM remain unchanged in spite of these differences. The assumption that the adversary operates for an equal number of steps in both traces does not limit their capability, as any attack necessitating a variable number of steps across traces can be replicated within this model by extending the shorter trace of the adversary with a series of non-operative steps. According to this theorem, under the specified assumptions, the state and outputs of the VM at every step are guaranteed to be the same across both traces.

V. Analysis of TDX Architecture

This section provides a security analysis for data confidentiality and integrity of Intel TDX [4] architecture, which is an example of a VM-based TEE architecture.

A. Intel TDX: Ensuring Data Confidentiality

Intel TDX safeguards the confidentiality of Trusted Domain (TD) data across memory, the processor, and the bus by encrypting data during transmission from the processor back to memory. This encryption uses the MKTME (Multi-Key Total

Memory Encryption) system, employing AES-XTS with 128-bit encryption for each cache line. The unique keys for each TD, identifiable through a Host KeyID (HKID), are generated and managed securely, with encryption keys stored internally and not disclosed to unauthorized entities.

Upon activation of TDX, physical memory is partitioned into secure (private) and normal (shared) regions, with the former designated for sensitive TD data and the latter for interactions with non-trusted entities. The allocation to either region is determined by the state of the highest order bit of the Guest Physical Address (GPA), ensuring a clear separation and safeguarding of confidential data.

The life cycle of a TD includes several key states, from creation and key configuration to potential blocking and eventual teardown, each facilitated by specific API calls. This process begins with the creation of a new TD and the generation of an ephemeral key, followed by its configuration and operational management through the Key Encryption Table (KET) and KeyID Ownership Table (KOT), ensuring secure and efficient key management throughout the TD's existence.

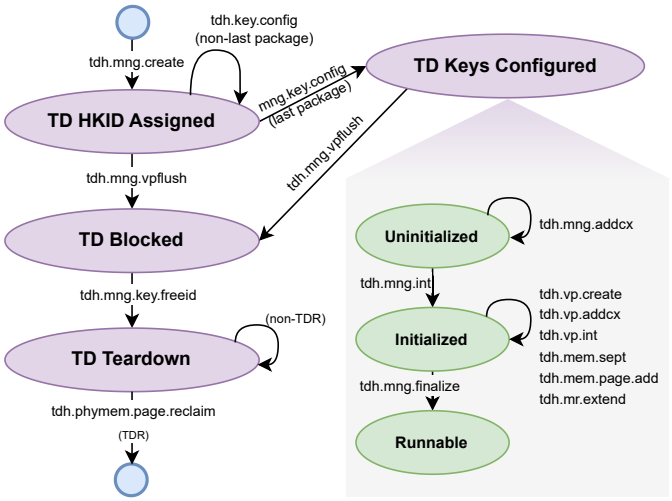

Fig. 7: Trust domain (TD) life cycle state diagram with host keyID (HKID) states.

In this section, we briefly describe the functionality of different HKID states in Figure 7.

1) *HKID Assigned State*: Achievable through the tdh_mng_create API, this state marks the initialization of a new Trust Domain (TD). Initially, the hypervisor ensures that any changes in the cache related to the TD's physical pages are committed. Following this, it establishes the Trust Domain Root (TDR) and creates a unique, temporary key for the TD. An HKID is generated and recorded in the KOT for each involved package.
2) *Keys Configured State*: This state occupies the majority of a TD's operational lifespan. The TD's temporary key is set up in the Key Encryption Table, with a secondary state machine managing the TD's activities. The TD transitions through several sub-states: from 'uninitialized' to 'initialized', and finally to 'runnable'. To incorporate the necessary Trust Domain Control Extension (TDCX) pages, the tdh_mng_addcx API is utilized. The TD's state within the TDR is initialized using tdh_mng_init, and achieving the 'runnable' state is finalized with $tdh\_mng\_finalize$.
3) *Blocked State*: Any interruptions or faults prompt the TD to move into the blocked state, during which access to the TD's private memory is suspended, and related caches are cleared. The $tdh\_mng\_vpflushdone$ API checks for the complete flushing of cache lines associated with the TD's address or HKID.
4) *Teardown State*: In this final phase, the host's Virtual Machine Monitor reclaims the HKID and clears both the Translation Lookaside Buffer (TLB) and cache. It proceeds to remove all private and control pages of the TD through $tdh\_phymem\_page\_reclaim$, with $tdh\_phymem\_page\_wbinvd$ being employed to ensure any modified cache lines are flushed.

These states highlight the dynamic and secure management of TDs within the TDX framework, emphasizing data confidentiality through stringent key control and memory encryption practices, as shown in Figure 8.

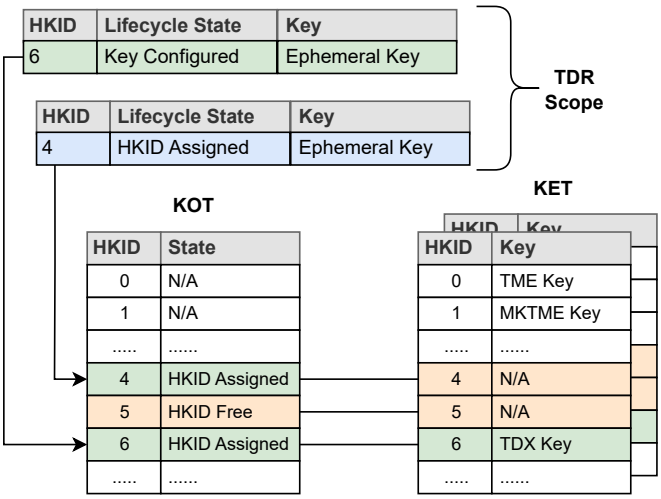

Fig. 8: Key management with trust domain root (TDR), KeyID ownership table (KOT), and key encryption table (KET).

B. Intel TDX: Guaranteeing Memory Integrity

Intel TDX maintains memory integrity via a dual approach, incorporating a TD owner bit and a Message Authentication Code (MAC), both embedded within ECC memory. A 128-bit MAC key is created during system initialization, with a 28-bit MAC generated for each memory write. This MAC, alongside the TD owner bit, aids in verifying data integrity during reads, with discrepancies indicating potential integrity breaches and resulting in the marking of compromised cache lines.

The TD owner bit serves as a gatekeeper, controlling access based on whether a physical address is associated with a private HKID. This mechanism ensures that only authorized SEAM mode operations can access secured memory segments, with all other requests being denied and returned as null, thereby preserving the integrity of sensitive data.

Furthermore, any attempt to write to a protected memory segment outside of SEAM mode triggers the reset of the corresponding TD owner bit, marking the segment as poisoned. This serves as a critical fail-safe, triggering a TD exit and,

if necessary, transitioning the TD to a fatal state for security, thereby emphasizing the robust measures in place to maintain memory integrity within the TDX architecture.

VI. Formal Modeling of Intel TDX architecture

The formal model is developed following the Intel TDX module specification. We model the Intel TDX architecture using Rosette [18] to enable symbolic simulation of the complex mechanisms, including TDX tables, Application Binary Interfaces (ABIs), and other configurations essential for ensuring memory confidentiality and integrity within trusted domains. This section details the formal modeling, highlighting only the key components that form the backbone of TDX security.

A. Defining TDX Tables

The TDX specification has various tables, each serving a unique purpose in the security architecture. Among these, the Extended Page Table (EPT), Key Encryption Table (KET), and KeyID Ownership Table (KOT) are foundational elements for confidentiality.

1) Extended Page Table (EPT): The EPT maps Guest Physical Addresses (GPAs) to Host Physical Addresses (HPAs) and maintains the page state, incorporating a shared bit to differentiate between secure and shared memory spaces. This mapping is crucial for memory isolation and confidentiality.

Listing 1: Extended Page Table

```
(define sec_ept (make-hash))
(define-struct sec_ept_entry (hpa gpa_shared state))
(define/contract sec_ept-contract
    (hash/c integer? sec_ept_entry? #:flat? #t)
    sec_ept)
```

Listing 1 shows a hash table sec_ept created using $make\text{-}hash$, which serves as a repository for managing EPT entries, and a custom-defined structure sec_ept_entry, which keep track of the essential attributes of each entry, including Host Physical Address (hpa), Guest Physical Address shared status (gpa_shared), and the entry's current state (state). This approach enables efficient tracking and manipulation of memory addresses between the host and virtual machines, facilitating a streamlined mechanism to oversee the shared or exclusive access to physical memory resources.

2) Key Encryption Table (KET) and KeyID Ownership Table (KOT): The KET (Listing 2) associates each TD's ephemeral encryption key with its corresponding HKID, playing a pivotal role in encrypting memory access and safeguarding data in transit. The KOT, on the other hand, tracks the lifecycle state of each HKID, ensuring proper key management and assignment. Two hash table structures are used to represent the two tables:

Listing 2: Key encryption table and key ownership table

```
(define KET (make-hash))
(define/contract KET-contract
    (hash/c integer? bitvector? #:flat? #t)
    KET)

(define KOT (make-hash))
(define/contract KOT-contract
    (hash/c integer? integer? #:flat? #t)
    KOT)
```

B. Trust Domain Management

The management of Trust Domains (TD) includes TD creation, key configuration, handling exceptions, and teardown. We implemented structures to facilitate this, the Trust Domain Root (TDR) and Trust Domain Control Structure (TDCS), which maintain state and control information for each TD.

1) TD Creation: TDs are instantiated and assigned unique HKIDs, with their ephemeral keys generated and stored securely. This process involves interactions with the Physical Address Metadata Table (PAMT) for memory allocation and the cache to ensure confidentiality during TD operations. Listing 3 shows the Application Binary Interface (ABI) for TDH_MNG_CREATE, which manages creating and initializing a Transactional Data Handler (TDH) by mapping hpa to HKID. Initially, it checks the current state of the given HKID and the hpa in two hash tables (KOT for HKIDs and PAMT for physical addresses) to determine if the HKID is private, not yet assigned, or if the hardware page is not yet allocated or is in a non-disclosure agreement state (PT_NDA). If these conditions are met, the function proceeds to mark the HKID as assigned ($HKID\_ASSIGNED$) in the KOT table and creates a new PAMT entry with initial parameters. Finally, it initializes a new TDR with default or initial values. This setup indicates a mechanism for managing access and operations on hardware resources, ensuring data privacy and integrity through proper handling of hardware keys and memory pages.

Listing 3: TDH_MNG_CREATE ABI

```
(define (TDH_MNG_CREATE hpa HKID)
    (define hkid_state (hash-ref KOT HKID #f))
    (define page_state (hash-ref PAMT hpa #f))
    (when (and (is_hkid_private HKID)
        (or (equal? hkid_state #f)
            (equal? hkid_state HKID_FREE))
        (or (equal? page_state #f)
            (equal? page_state PT_NDA)))
        (begin
            (hash-set! KOT HKID HKID_ASSIGNED)
            (hash-set! PAMT hpa
                (make-PAMT_entry PT_TDR 0 0))
            (make-TDR #f #f 0 0 0
                HKID_ASSIGNED HKID 0 #f #f))))
```

2) Key Configuration: $TDH\_MNG\_KEY\_CONFIG$ (Listing 4) is designed to configure keys for a TDR associated with a specific hpa. It begins by retrieving the current entry for the given PAMT and determining its state, specifically checking if it matches the expected type for transactional data records (PT_TDR). It then checks the tdr for a fatal error condition (td_fatal) and its lifecycle state to ensure it is in the $HKID\_ASSIGNED$ state, indicating that a HKID has been assigned but not yet configured with keys.

Listing 4: TDH_MNG_KEY_CONFIG ABI

```
(define (TDH_MNG_KEY_CONFIG hpa tdr)
    (define page_entry (hash-ref PAMT hpa #f))
    (define page_state
        (if (PAMT_entry? page_entry)
            (PAMT_entry-PAGE_TYPE page_entry) #f))
    (define td_fatal (TDR-FATAL tdr))
    (define hkid_state (TDR-LIFECYCLE_STATE tdr))
    (when (and (equal? page_state PT_TDR)
    (not td_fatal) (equal? hkid_state HKID_ASSIGNED))
        (begin
```

```
            (hash-set! KET (TDR-HKID tdr) key_val)
            (struct-copy TDR tdr
                [LIFECYCLE_STATE KEYS_CONFIGURED])))))
```

If the page is correctly prepared for transactional data, no fatal errors are present, and the TDH is ready for key configuration, the function proceeds to update KET with the hardware key ID extracted from TDR and sets a new key (key_val). It then updates the tdr structure itself to reflect that the keys have been configured, changing its lifecycle state to $KEYS\_CONFIGURED$.

3) Handling Interrupts and Exceptions: Handling of interrupts and exceptions is crucial for the secure and stable operation of TDs. This involves saving the current TD state, scrubbing the VCPU state, and executing a cache flush to maintain data integrity.

Listing 5: TDH_MNG_VPFLUSH ABI

```
(define (TDH_MNG_VPFLUSH pa tdr)
    (define page_entry (hash-ref PAMT pa #f))
    (define page_state
        (if (PAMT_entry? page_entry)
            (PAMT_entry-PAGE_TYPE page_entry)#f))
    (define td_state (TDR-LIFECYCLE_STATE tdr))
    (define hkid_state (hash-ref KOT (TDR-HKID tdr)))
    (if (and (equal? page_state PT_TDR)
             (or (equal? td_state TD_HKID_ASSIGNED)
             (equal? td_state TD_KEYS_CONFIGURED))
             (equal? hkid_state HKID_ASSIGNED))
            (begin
                (flush_cache (TDR-HKID tdr))
                (hash-set! KOT (TDR-HKID tdr)
                HKID_FLUSHED)(struct-copy TDR tdr
                 [LIFECYCLE_STATE TD_BLOCKED]))#f))
```

The function $TDH\_MNG\_VPFLUSH$ (Listing 5) is responsible for securely flushing a virtual page from the cache. The function starts by looking up the state of the page associated with the given physical address in PAMT. It assesses the lifecycle state of the tdr to ensure it is either in the $HKID\_ASSIGNED$ or $KEYS\_CONFIGURED$ state, and verifies that the HKID related to the tdr is marked as assigned in KOT. If these conditions are met, the tdr is in an appropriate state for flushing. This is critical for maintaining data consistency and security, ensuring that no sensitive data remains in the cache that could be accessed inappropriately. After flushing the cache, the function updates the state of the HKID in the KOT to $HKID\_FLUSHED$, indicating that the flush operation has been completed. Finally, it updates the lifecycle state of the tdr to $TD\_BLOCKED$, indicating that the tdr is in a state where it cannot perform regular operations.

4) TD Teardown: $TDH\_MNG\_KEY\_FREEID$ (Listing 6) function is designed for releasing or freeing HKIDs that are no longer in use.

Listing 6: TDH_MNG_KEY_FREEID ABI

```
(define (TDH_MNG_KEY_FREEID pa tdr)
    (define page_entry (hash-ref PAMT pa #f))
    (define page_state
        (if (PAMT_entry? page_entry)
            (PAMT_entry-PAGE_TYPE page_entry)
            #f))
    (define hkid_state (TDR-LIFECYCLE_STATE tdr))
    (define hkid (TDR-HKID tdr))
    (define kot_entry (hash-ref KOT hkid #f))
    (when (and (equal? page_state PT_TDR)
    (equal? hkid_state TD_BLOCKED)
    (equal? kot_entry HKID_FLUSHED))
        (begin (hash-set! KOT hkid HKID_FREE)
            (struct-copy TDR tdr
            [LIFECYCLE_STATE TD_TEARDOWN]
                        [HKID 0]))))
```

It first verifies that the specified physical address is associated with a page prepared, that the TDR is in a blocked state ($TD\_BLOCKED$), and that the HKID has been flushed ($HKID\_FLUSHED$). This ensures the function operates under safe conditions where the data associated with the tdr and HKID has been securely managed and is ready for cleanup. Upon confirming these prerequisites, the function sets the HKID's state to $HKID\_FREE$ in the KOT, marking it available for future assignments. Additionally, it updates the tdr to reflect a teardown lifecycle state ($TD\_TEARDOWN$) and resets the HKID within the tdr, effectively clearing the association and preparing the system for new transactions. This process is essential for the secure and efficient reuse of hardware resources, ensuring that data integrity and confidentiality are maintained throughout the lifecycle of a TD.

VII. Formal Modeling of Cache for TDX Systems

When the cache is unencrypted, the data stored within remains in plaintext, posing significant risks to both data integrity and confidentiality. Such a scenario lays the groundwork for multiple security vulnerabilities, as unauthorized access to this unencrypted data can lead to information leakage or manipulation. In this section, we evaluate how the Intel TDX module addresses these critical security concerns within the context of a shared cache environment. This ensures that even in a shared cache scenario, where multiple processes or virtual machines might access the same physical cache resources, data remains secure, isolated, and impervious to unauthorized access or tampering, thereby upholding the highest standards of integrity and confidentiality. We model and formally evaluate two distinct cache types, each capable of enhancing security in TDX environments independently: HKID-tagged cache and TD-owner-bit cache.

A. Basic Cache Structure and Initialization

The cache model in Listing 7 is designed to simulate a 4-way associative cache, a common setup in modern computing systems. This setup is characterized by a finite number of cache sets, each with multiple ways to store data. The model initializes hash maps to track the validity, tag, data, and HKID of each cache line, providing a foundational structure for simulating cache operations:

Listing 7: Cache configuration

```
(define kmax-cache-set-index-t 256)
(define kmax-cache-way-index-t 4)

(define cache-valid-map (make-hash))
(define cache-tag-map (make-hash))
(define cache-data-map (make-hash))
(define cache-hkid-map (make-hash))
```

The init-cache function (Listing 8) populates these structures, initially setting all cache lines to an invalid state and

assigning default values to the tags, data, and HKIDs. This ensures a clean state from which cache operations can start:

Listing 8: Cache initialization

```
(define (init-cache)
(for ([i (in-range kmax-cache-set-index-t)])
(for ([j (in-range kmax-cache-way-index-t)])
(let ([key (cons i j)])
(hash-set! cache-valid-map key #false)
(hash-set! cache-data-map key 0)
(hash-set! cache-tag-map key 0)
(hash-set! cache-hkid-map key 0))))
```

B. HKID Tagged Cache

Integrating HKID into the cache model adds a layer of security by ensuring that cache lines are accessible only by the appropriate Trusted Domain (TD). Listing 9 shows our modeling of HKID tagged cache. This approach leverages HKIDs to tag cache lines, thus facilitating the validation of access requests based on the TD's identity. HKID tagging is implemented by extending the cache model to include a mapping of cache lines to HKIDs. This extension allows the cache to check not only the validity and tag match for cache hits but also the HKID, ensuring that only requests from the owning TD can access the cached data.

Listing 9: HKID tagged cache model

```
(define (query-cache pa repl-way hkid)
  (define set (paddr2set pa))
  (define tag (paddr2tag pa))
  (define hit-way
    (for/or ([way (in-range kmax-cache-way-index-t)])
      (let ([key (cons set way)])
        (and (hash-ref cache-valid-map key #false)
             (= (hash-ref cache-tag-map key) tag)
             (= (hash-ref cache-hkid-map key) hkid)
             way))))
  (if hit-way
      (let ([key (cons set hit-way)])
        (values #true hit-way (hash-ref
                        cache-data-map key)))
      (begin
        (let ([key (cons set repl-way)])
          (hash-set! cache-valid-map key #true)
          (hash-set! cache-tag-map key tag)
          (hash-set! cache-data-map key 0)
          (hash-set! cache-hkid-map key hkid))
        (values #false repl-way 0))))
```

C. TD Owner Bit

By using a TD Owner bit in access control, TDX enforces strict access policies, allowing only SEAM mode processes to read secure cache lines, thereby significantly mitigating the risk of unauthorized data access. Listing 10 shows our modeling of TD Owner bit cache. This cache management strategy not only enhances data security by providing fine-grained access control based on hardware-level identifiers but also introduces a flexible framework for managing cache data across multiple Trusted Domains.

Listing 10: TD owner bit cache model

```
(define (query-cache pa repl-way hkid seam-mode?)
  (define set (paddr2set pa))
  (define tag (paddr2tag pa))
  (define hit-way
    (for/or ([way (in-range kmax-cache-way-index-t)])
      (let ([key (cons set way)])
        (and (hash-ref cache-valid-map key #false)
             (= (hash-ref cache-tag-map key) tag)
             (= (hash-ref cache-hkid-map key) hkid)
             (or seam-mode? (not (hash-ref
             cache-td-owner-map key))) way))))
  (if hit-way
      (let ([key (cons set hit-way)])
        (if (or seam-mode? (not (hash-ref
                cache-td-owner-map key)))
            (values #true hit-way (hash-ref
                cache-data-map key))
            (values #true hit-way 0)))
      (begin
        (let ([key (cons set repl-way)])
          (hash-set! cache-valid-map key #true)
          (hash-set! cache-tag-map key tag)
          (hash-set! cache-data-map key 0)
          (hash-set! cache-hkid-map key hkid)
          (hash-set! cache-td-owner-map key
            (if (> hkid 0) #true #false)))
        (values #false repl-way 0))))
```

VIII. Experiments

This section demonstrates the effectiveness of our proposed VM-based TEE verification framework to verify the Intel TDX module. First, we describe our formal verification setup. Next, we present the formal verification results of our framework.

A. Experimental Setup

Our model for Intel TDX and caches and properties for formal security verification is constructed using Rosette [18] formal verification language. We used the publicly available specification as well as implementation for the TDX module [4] to derive the formal model. The Rosette model has assertions, symbolic variables, and solver-aided functions. The correctness of these elements is verified using Rosette's symbolic execution engine, which internally uses Z3 [19] SMT solver to check the feasibility of paths and the satisfaction of constraints. We ran our experiments on Intel i7-5500U @ 3.0GHz CPU with 16GB RAM machine. We have developed 15 confidentiality properties and 9 integrity properties for two cache models: HKID-tagged and TD-owner-bit-tagged cache.

B. Generation of Confidentiality Properties

The properties for confidentiality and integrity are defined based on the threat model outlined in the TDX specification. We have developed 15 confidentiality properties.

1) cP_1: Assert that any guest physical address (GPA) mapped to a specific host physical address (HPA) within the secure EPT maintains confidentiality, meaning no other GPA can map to this HPA.
2) cP_2: Assert that the ephemeral encryption key associated with a specific HKID in the KET table remains confidential and is not leaked or accessible to unauthorized entities.
3) cP_3: Assert that once an HKID is assigned, its state remains confidential and accurately reflects its assigned status within the KOT table.
4) cP_4: Assert that the lifecycle state of a Trust Domain Root (TDR) remains confidential and can only be one of

the predefined states (INIT, FATAL, RUNNING), safeguarding the state transitions from unauthorized access.

5) cP_5: Assert that the page state of any entry in the secure EPT is limited to predefined states, protecting the confidentiality of page mappings.
6) cP_6: Assert that the key configuration state for a given HKID remains confidential and accurately reflects the $TD\_KEYS\_CONFIGURED$ state, protecting the key configuration status from unauthorized changes.
7) cP_7: Assert that the finalization status of a Trust Domain Control Structure (TDCS) remains confidential and is always set to true after finalizing.
8) cP_8: Assert that the confidentiality of the shared bit status for any given entry in the secure EPT.
9) cP_9: Assert that the package configuration bitmap of a Trust Domain Root (TDR) remains confidential, ensuring that the configuration details are protected from unauthorized disclosure.
10) cP_{10}: Assert that the association between a VCPU and its corresponding HKID is kept confidential.
11) cP_{11}: Assert that querying the cache with an incorrect HKID results in a cache miss.
12) cP_{12}: Assert that after querying with the correct HKID, a subsequent query with the same HKID and address results in a cache hit.
13) cP_{13}: Assert that querying with a different HKID (assuming unauthorized access) after a cache line is populated does not provide access to the data.
14) cP_{14}: Assert that after updating a cache line with a new HKID and data, the previous HKID no longer has access.
15) cP_{15}: Assert that once data is written to a cache line, it remains unchanged unless explicitly modified through a valid cache update.

C. Generation of Integrity Properties

We have developed 9 integrity properties.

1) iP_1: Assert that the integrity of the EPT mappings by asserting that any entry mapping a GPA to a HPA cannot be in a "blocked" state.
2) iP_2: Assert that the integrity of the TDR lifecycle by asserting that once a TDR is finalized, its lifecycle state cannot be "INIT" or "FATAL".
3) iP_3: Assert that the integrity of key state transitions within the TDX module by providing consistent transitions for a HKID based on its current state. Specifically, it asserts that an HKID assigned state can only move to the keys configured state, a keys configured state can transition to either blocked or teardown, and a blocked state can only move to teardown.
4) iP_4: Assert that if a cache entry is marked as valid, it must have a corresponding tag and data in the cache.
5) iP_5: Asserts that within a single set in a set-associative cache, all valid entries must have unique tags.
6) iP_6: Asserts that each valid cache entry, the corresponding HKID is correctly mapped to the same set and way in the cache-hkid-map.
7) iP_7: Assert that any cache entry from SEAM mode marked as valid has a corresponding and correct TD owner bit set.
8) iP_8: Assert that for any two valid cache entries in the same set but in different ways, their tags must be different.
9) iP_9: Assert that if two cache entries have the same tag and are valid, they must have the same TD owner bit.

Listing 11 shows sample confidentiality property (cP_{13}) of a cache system through symbolic execution. Initially, symbolic variables pa1 and pa2 with a bit-vector size of 28 (bv28) representing physical addresses are initialized. Then symbolic integers repl-way1, reply-way2, hkid1, and hkid2 are introduced, with the latter two representing replacement cache element. Sample assertion queries the cache twice, using the same physical address (pa1) but different key identifiers (hkid1 and hkid2), and stores the results (hit flags, ways, and data) in (hit1, way1, data1) and (hit2, way2, data2), respectively. The confidentiality assertion checks if, hkid1 and hkid2 are different, both cannot have cache hits for the same physical address that could violate confidentiality. The assertion is then solved, and the result is displayed, indicating whether the cache system maintains confidentiality across the given symbolic inputs.

Listing 11: Sample confidentaility assertion

```
(define-symbolic pa1 pa2 bv28)
(define-symbolic repl-way1 reply-way2 integer?)
(define-symbolic hkid1 hkid2 integer?)

(define-values (hit1 way1 data1)
    (query-cache pa1 repl-way hkid1))
(define-values (hit2 way2 data2)
    (query-cache pa1 repl-way hkid2))

(define confidentiality-assertion
 (assert (or (not (and hit1 hit2)) (= hkid1 hkid2))))

(define result (solve (confidentiality-assertion)))
(displayln result)
```

D. Verification Results

Table II provides a summary of the verification outcomes for three distinct models: the TDX module, HKID-tagged-cache, and TD-Owner-bit-cache. It details the number of lines of code in each model, with the TDX module being the largest at 400 lines, and the HKID-tagged-cache the smallest at 100 lines. The table also indicates the number of confidentiality and integrity properties verified for each module. Verification time, measured in seconds, showcases the efficiency of the verification process for each model, demonstrating the practicality and scalability of the verification process in evaluating the reliability and robustness of the models.

Our work can be extended to other VM-based solutions. For example, if we consider AMD SEV, which uses x86 architecture similar to Intel TDX, the changes needed are minimal. Similarly, AMD SEV uses an VM address space identifier (ASID) to uniquely identify the VM addresses, which is similar to HKID used by Intel TDX. To extend our formal model to AMD SEV, we need to model the ASID, but most of the formal model of Intel TDX module can be reused.

IX. Conclusion

This paper has presented a comprehensive framework for the formal verification of VM-based TEEs, addressing the

TABLE II: Rosette models and verification results

Description	# Lines	Confidentiality	Integrity	Verification Time (s)
TDX Module	400	10 properties ($cP_1 - cP_{10}$)	3 properties ($iP_1 - iP_3$)	12.86
HKID-tagged cache	100	5 properties ($cP_{11} - cP_{15}$)	3 properties ($iP_4 - iP_6$)	5.37
TD-Owner-bit cache	150	5 properties ($cP_{11} - cP_{15}$)	6 properties ($iP_4 - iP_9$)	7.92
Total	650	15	9	26.15

critical need for robust security mechanisms in the face of evolving threats. We have developed a formalization of confidentiality and integrity for confidential virtual machines (VM), proposing a secure and verifiable model in the context of powerful adversaries. Our contributions, including the formalization of a confidential VM, the establishment of formal definitions for confidentiality and integrity within VM-based TEEs, and the development of a refinement-based methodology, underline the importance and effectiveness of formal verification in ensuring the security of VM-based trusted execution environments. Our experimental results demonstrate the applicability and resilience of our framework to analyze sophisticated attack scenarios, highlighting its potential to significantly enhance the security posture. By proving the confidentiality and integrity guarantees of the Intel TDX platform through machine-checked proofs, we not only validate our approach but also pave the way for future research in securing virtualized TEE environments.

ACKNOWLEDGMENTS

This work was partially supported by the Semiconductor Research Corporation (SRC) grant 2022-HW-3128.

REFERENCES

[1] H. Witharana, D. Chatterjee, and P. Mishra, "Verifying memory confidentiality and integrity of intel tdx trusted execution environments," in *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2024, pp. 44–54.

[2] "Intel Software Guard Extensions (SGX)," https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html.

[3] V. Costan, I. A. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation." in *USENIX Security Symposium*, 2016, pp. 857–874.

[4] "Intel Trust Domain Extensions (TDX)," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[5] "AMD Secure Encrypted Virtualization (SEV)," https://developer.amd.com/sev/.

[6] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[7] "Intel Trust Domain Extensions (TDX) Security Review," https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf.

[8] "Amd secure processor for confidential computing security review," https://storage.googleapis.com/gweb-uniblog-publish-prod/documents/AMD_GPZ-Technical_Report_FINAL_05_2022.pdf, 2023.

[9] H. Witharana, Y. Lyu, and P. Mishra, "Directed test generation for activation of security assertions in rtl models," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 26, no. 4, pp. 1–28, 2021.

[10] H. Witharana, Y. Lyu, S. Charles, and P. Mishra, "A survey on assertion-based hardware verification," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–33, 2022.

[11] J. Hu, F. Zeng, Y. Zhao, Z. Zhang, L. Zhang, J. Zhao, R. Chang, and K. Ren, "Proverit: A parameterized, composable, and verified model of tee protection profile," *IEEE Transactions on Dependable and Secure Computing*, 2024.

[12] Y. Ma, Q. Zhang, S. Zhao, G. Wang, X. Li, and Z. Shi, "Formal verification of memory isolation for the trustzone-based tee," in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2020, pp. 149–158.

[13] H. Sun and H. Lei, "A design and verification methodology for a trustzone trusted execution environment," *IEEE Access*, vol. 8, pp. 33 870–33 883, 2020.

[14] M. U. Sardar, S. Musaev, and C. Fetzer, "Demystifying attestation in intel trust domain extensions via formal verification," *IEEE access*, vol. 9, pp. 83 067–83 079, 2021.

[15] M. U. Sardar, T. Fossati, and S. Frost, "Comprehensive specification and formal analysis of attestation mechanisms in confidential computing," *ICE 2023 Pre-Proceedings*, 2023.

[16] W. Ozga, "Towards a formally verified security monitor for vm-based confidential computing," in *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2023, pp. 73–81.

[17] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2435–2450.

[18] "Rosette Language Guide," https://docs.racket-lang.org/rosette-guide/index.html.

[19] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

Hasini Witharana received her Ph.D. from the Department of Computer & Information Science & Engineering at the University of Florida. She received her B.Sc. in the Department of Computer Science and Engineering from the University of Moratuwa, Sri Lanka in 2018. Her area of research includes hardware security and assertion-based verification.

Hansika Weerasena is a Ph.D. student in the Department of Computer & Information Science & Engineering at the University of Florida. He received his B.Sc. in Department of Computer Science and Engineering from university of Moratuwa, Sri Lanka. His area of research includes cyber and hardware communication security, computer architecture, and applied machine learning.

Prabhat Mishra is a Professor in the Department of Computer and Information Science and Engineering at the University of Florida. He received his Ph.D. in Computer Science from the University of California at Irvine. His research interests include embedded and cyber-physical systems, hardware security and trust, and energy-aware computing. He currently serves as an Associate Editor of IEEE Transactions on VLSI Systems and ACM Transactions on Embedded Computing Systems. He is an IEEE Fellow, an AAAS Fellow, and an ACM Distinguished Scientist.