# Verifying Memory Confidentiality and Integrity of Intel TDX Trusted Execution Environments

Hasini Witharana*, Debapriya Chatterjee[†] and Prabhat Mishra*
*University of Florida, Gainesville, Florida, USA
[†]IBM, Austin, Texas, USA

***Abstract*—Encryption is widely used to protect the confidentiality of data when it is in transit or at rest. However, encryption itself is not enough to guarantee the confidentiality of data-in-use. A Trusted Execution Environment (TEE) provides assurance of three key properties: data confidentiality, data integrity, and code integrity for data-in-use. This is done by enabling isolation and secure storage within a system. It is a major challenge to verify the security requirements in a TEE since isolation and secure storage require complex interaction between different components such as the operating system, hypervisor, firmware, and hardware. In this paper, we perform a comprehensive security analysis of the Intel Trust Domain Extensions (TDX) for memory confidentiality and integrity. Specifically, we analyze the Intel TDX architecture to derive invariants. We generate security assertions (properties) for the invariants based on Finite State Machine (FSM) analysis. We utilize model checking to verify security properties. We also perform static analysis of TDX code to check for various vulnerabilities, including buffer overflow, pointer safety, and arithmetic issues. Extensive experimental evaluation demonstrates that our proposed framework is suitable for verifying Intel TDX architecture. We are also able to identify inconsistency in the TDX specification.**

## I. Introduction

Data confidentiality needs to protect data that can exist in three possible states: transit (transferred through network), rest (stored), and in use (utilized during computation). Traditional encryption can protect the confidentiality of data when it is in 'transit' or 'rest'. While homomorphic encryption can process encrypted data (protect when in 'use'), it can introduce significant area, power and performance overhead. In order to provide trustworthy computing, we need to provide not only data confidentiality but also other important guarantees such as data integrity, attestation, availability, and recoverability. Modern computing systems utilize Trusted Execution Environment (TEE) to fulfill these requirements [1], [2].

TEEs provide a secure execution environment by isolating the execution from other non-secure world interactions. There are many commercial TEE solutions, including ARM TrustZone [3], AMD Secure Encrypted Virtualization (SEV), Intel Software Guard Extensions (SGX) [4], Intel Trust Domain Extensions (TDX) [5], and IBM-Z Secure Execution [6]. There are also academic [7] and open-source [8], [9] TEEs. Since trustworthy computing relies on the guarantees provided by the TEEs, it is critical to verify the trustworthiness of the TEEs. Specifically, we need to verify that a given TEE implementation provides data confidentiality, data integrity and code integrity under all possible execution scenarios.

This work was partially supported by the SRC grant 2022-HW-3128.

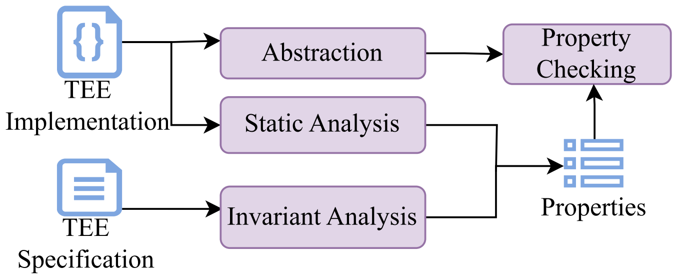

Fig. 1: Overview of our proposed TEE verification framework.

It is a major challenge to perform TEE verification since it involves complex interactions between different subsystems consisting of hardware, software, and firmware. There are TEE verification approaches, such as static analysis [10], simulation-based testing [11], and formal verification [12], [13], [14], [15], [16], [17], [18]. For example, there is a recent security review on Intel TDX [10] based on static analysis.

In this paper, we present a framework for formally verifying the memory confidentiality and integrity properties of Intel TDX architecture. Figure 1 shows an overview of our proposed formal verification framework that consists of four major steps: invariant analysis, static analysis, abstraction, and property checking. The first two steps analyze the TDX specification [5], [19], [20], [21] to generate properties. The abstracted model (third step) is used to verify the generated properties. Our approach decomposes the overall guarantee in terms of the correctness of the operation of individual subsystems. Specifically, this paper makes the following five major contributions:

1) Derives invariants for memory confidentiality and integrity for Intel TDX architecture.
2) Derives security properties for the invariants based on automated analysis of finite state machines and decomposes them into hardware and firmware properties.
3) Performs static analysis of the TDX implementation to derive security properties related to buffer overflow, pointer safety, and arithmetic issues.
4) Performs abstraction of the TDX implementation, enabling effective analysis and scalable formal verification.
5) Performs model checking of the generated properties using both invariant and static analysis.

The remainder of the paper is organized as follows. Section II surveys the related efforts. Section III presents the threat model. Section IV describes our framework for verifying TDX memory confidentiality and integrity. Section V presents the experimental results. Finally, Section VI concludes the paper.

## II. Background and Related Work

We first provide relevant background on Intel TDX and how memory confidentiality and integrity are achieved. Next, we survey existing TEE verification efforts.

### A. Intel Trust Domain Extensions (TDX)

Intel TDX is a TEE based on secure virtualization [5]. TDX has the capability to deploy hardware isolated virtual machines called trust domains (TD). TDs are isolated from virtual machine manager (VMM)/hypervisor and any other software which are not related to TD as shown in Figure 2. This strong isolation provides the required security guarantees for a TEE system. Moreover, Intel TDX uses multi-key total memory encryption (MKTME) and hashing to maintain the confidentiality and integrity of the code and data in TD. Intel TDX module is designed to run in Secure Arbitration Mode (SEAM). SEAM introduces an expansion to the Virtual Machine Extension (VMX) architecture, establishing a fresh VMX root mode known as SEAM root. This SEAM root mode serves as the platform for accommodating a CPU-attested module designed for generating virtual machine (VM) guests termed Trust Domains (TD). SEAM mode can be used as two logical modes: TDX non-root mode and TDX root mode. TDX root mode is used for Host side operations and non-root mode is used for TD guest operations.

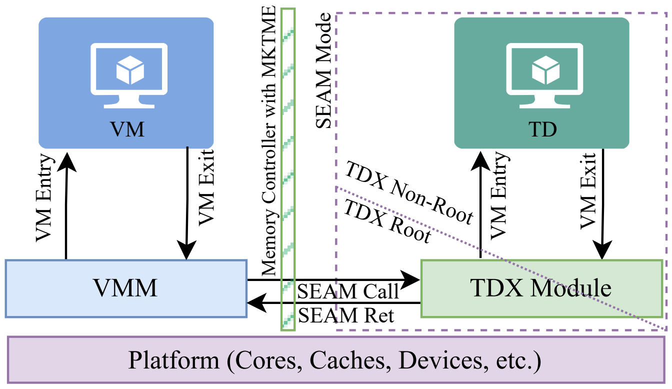

Fig. 2: Overview of Intel TDX architecture [21].

### B. Intel TDX: Memory Confidentiality

The data of a TD can reside in memory, processor or bus. When data is in the processor, it is stored in clear text. The processor encrypts the data when the data is transmitted back to the main memory from the processor. The processor uses a unique key for the TD that is only shown to the processor. TDX uses MKTME for memory encryption. Each cache line is encrypted by AES-XTS 128-bit memory encryption. Each key supported by MKTME is identified by a key identifier, Host KeyID (HKID). The processor provides a PCONFIG instruction used by the Intel-TDX module to assign a unique ephemeral AES-XTS 128-bit key to each HKID. MKTME stores the ephemeral key in the internal memory and it is not revealed to any unauthorized parties. When TDX is enabled, the physical memory is divided into two parts: secure (private) and normal (shared) memory. Private memory is used for TD's confidential data whereas shared memory is used for communication with untrusted parties. The highest order bit of the Guest Physical Address (GPA) is considered as the shared bit and it is used to distinguish whether GPA maps to a private (shared bit is false) or shared memory (shared bit is true). We briefly discuss two major components of Intel TDX.

**TD Creation Life Cycle with HKID:** Figure 3 shows the state diagram for TD life cycle with the TDX key management states for MKTME. There are four TD life cycle states:

1) *HKID Assigned State*: This state can be achieved using $tdh\_mng\_create$ API. This creates a new TD. First, the hypervisor flushes the cache if there are any modified cache lines for the physical pages. Next, it creates the Trust Domain Root (TDR) and generates a random ephemeral key for the TD. It generates an HKID and updates the KeyID Ownership Table (KOT) for each package.
2) *Keys Configured State*: Most of the TD lifetime is spent in this state. TD's ephemeral key is configured in Key Encryption Table (KET). A secondary-level state machine controls the TD operations in this state. The secondary states are: uninitialized, initialized and runnable. $tdh\_mng\_addcx$ API is used to add the required number of Trust Domain Control Extension (TDCX) pages. $tdh\_mng\_init$ is used to initialize the TD state in TDR. At this point, the TD is initialized by issuing $tdh\_mng\_finalize$, the 'runnable' state can be reached.
3) *Blocked State*: When there is an interruption or fault, the TD transitions into the blocked state. The TD private memory access gets blocked and the relevant caches get flushed. $tdh\_mng\_vpflushdone$ is used to check whether all the cache lines related to the address or HKID has been flushed.
4) *Teardown State*: The host VMM reclaims the HKID and flushes TLB and cache. It removes all the TD private and control pages using $tdh\_phymem\_page\_reclaim$. $tdh\_phymem\_page\_wbinvd$ is used to flush the modified cache lines.

**TD Key Management:** TDX architecture maintains several tables for key management and HKID processes at different levels as shown in Figure 4. All of these tables are only accessible by the TDX module through API calls. KOT keeps track of the pool of HKIDs and assigns HKIDs for TDs. KOT is internal to the TDX module. Similarly, TDR control structure has a key management field to maintain the HKID information at the TD level. The key management field consists of the key state, HKID, corresponding ephemeral key, and key configuration information. TD's ephemeral key is configured in KET. KET is a hardware table used for memory encryption engine configuration. This table is also indexed by the HKID.

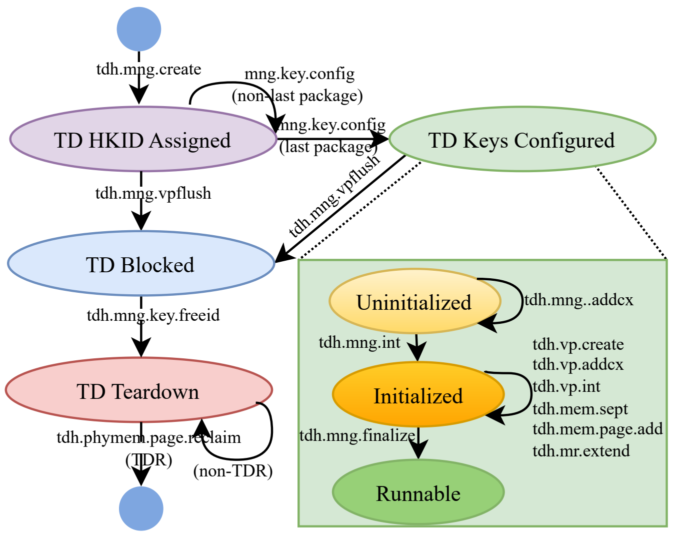

Fig. 3: TD life cycle state diagram with HKID states [21].

KET consists of both shared and private HKID values. For private HKID entry configuration, $tdh\_mng\_key\_config$ is used by host VMM. For the shared HKID configuration, PCONFIG instruction is used by software.

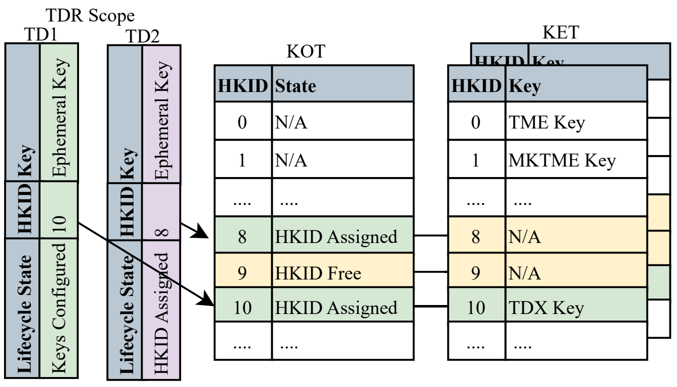

Fig. 4: Key management using trust domain root (TDR) structure, KeyID ownership table (KOT) and key encryption table (KET) [21].

### C. Intel TDX: Memory Integrity

In TDX architecture, memory integrity is maintained using a TD owner bit and Message Authentication Code (MAC). Both of them are stored in the Error Correction Code (ECC) memory. During the system initialization process, the processor generates a 128-bit MAC key. During each write, the memory controller uses this key to compute a 28-bit MAC for data in the corresponding cache line. The ciphertext, TD owner bit, 128-bit MAC key and AES-XTS tweak values are calculated for the MAC value. When reading, the memory controller calculates the MAC value separately and checks with the MAC value from the ECC memory. If there is a mismatch, integrity is violated and the cache line will be marked as poisoned.

TD owner bit is used to prevent unauthorized encryption reads. During writes, the memory controller sets the TD owner bit to 1 if the physical address corresponds to a private HKID. Similarly, it sets the TD owner bit to 0 if the address does not have a private HKID. For each cache line read, the memory controller enforces the access control by checking the TD owner bit. For all the read requests that run through the memory controller, only the SEAM mode requests get access to read from a memory where the TD owner bit is set to 1. When accessing a memory with the TD owner bit set to 1, the memory controller only permits read requests originating from SEAM mode. Any other read requests are rejected. Instead of providing actual data, the memory controller returns a result consisting of all zeros.

When non SEAM mode entity, writes to a segment that belongs to the private memory, it results in the corresponding TD owner bit being set to 0. Subsequently, when the TD or the TDX module reads this segment, it is flagged as poisoned. If the reader happens to be the TD, this poisoned marking will trigger a TD exit. The TDX module can detect this TD exit and transition the TD into a fatal state which leads tearing down of the TD. If the TDX module itself reads the poisoned content, both the TDX module and the TDX's hardware extension within the processor are marked as disabled.

### D. Related Work

Verifying the trustworthiness of a trusted execution environment is a complex task due to the interactions between different systems involved in a TEE [22]. Prior verification efforts can be broadly classified into three categories: static analysis [10], simulation-based testing [11], and formal verification [13], [14], [15], [16], [17], [18], [23]. Simulation based testing approach has been conducted for AMD SEV by Google [11]. The verification framework had an AMD SEV set up such that they are able to conduct simulations for different test vectors. They have derived different security invariants for different use cases as well. Moreover, fuzzing-based techniques were conducted to identify security vulnerabilities in the code. These invariants were derived manually for AMD SEV. Existing simulation-based methods does not provide any guarantees about the security properties. In our work, we utilize FSM-based analysis to automatically generate security properties to verify the invariants that can provide confidentiality and integrity guarantees.

Recently Google published a security review on Intel TDX [10] based on static analysis, where they have found 81 attack vectors, 10 confirmed security issues, and 5 defence in depth changes to the code. Specifically, this review covers four components implemented by Intel: (1) MCHECK implementation used in BIOS, (2) non-persistence SEAM loader, (3) persistence SEAM loader, and (4) TDX module implementation. For TDX module verification, Google used static analysis of the C code using "Weggli" [24] and "Frama-C" [25]. Weggli is used to identify interesting design patterns that could be harmful to the implementation. Frama-C is used as a static analysis framework for the TDX module codebase. However, the Google review [10] does not use any formal methods to give any guarantees about any security-related properties. In fact, the Google security review has highlighted the need for formal verification. Our proposed verification framework is complementary to [10] since we are using property checking to give a formal guarantee for Intel TDX memory confidentiality and integrity.

Assertion-based verification [26], [27], [28], [29], [30] and formal methods, including property checking, theorem proving [13], [14], and equivalence checking are promising avenues to provide security guarantees in TEEs. Moat [13] provides a formal abstraction for the Intel SGX model and it is used as the basis of theorem proving for confidentiality property of SGX. This abstraction has been extended by [14] for Intel SGX for more properties including integrity, confidentiality, and secure measurement. Formal verification has been used by other TEEs such as ARM TrustZone [15], [16] and Intel TDX [18], [31]. Sradar et al. formally verify the attestation process of Intel TDX using ProVerif's specification language [18]. This work only focuses on the attestation process, whereas our work focuses on memory confidentiality and integrity of Intel TDX. Existing TEE architectures can be broadly divided into three categories: VM-based [5], [32], enclave-based [4], [3], and TEEs for RISC-V based embedded systems [8]. ARM TrustZone and Intel SGX belong to enclave-based category, whereas Intel TDX is a VM-based TEE. Therefore, existing theorem proving [14], [15] based solutions for ARM TrustZone cannot be directly applied to the TDX architecture. *To the best of our knowledge, there are no prior efforts in formally verifying memory confidentiality and integrity of Intel TDX using invariant analysis and property checking.*

## III. Threat Model

Figure 5 shows the threat model for our proposed framework. Our threat model assumes that an adversary will have physical or remote access to a machine where TDX is enabled. The adversary has the capability of getting control over BIOS, System Management Mode (SMM), host operating system, hypervisor, all non-TD software, and peripheral devices. The objective of the adversary is to leak confidential data through a TD or compromise the integrity of the data associated with a TD. In our threat model, only the TDX module and the Intel hardware are trusted.

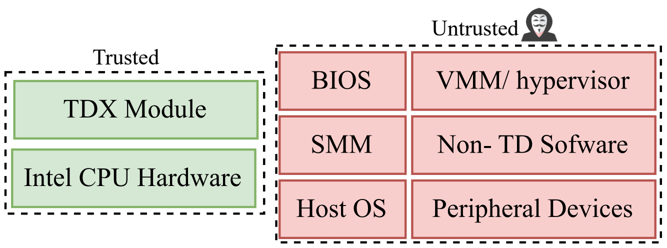

Fig. 5: Threat model considered for the proposed framework

Our threat model assumes that the adversary has the capability to invoke the TDX module using host-side functions. Therefore, the adversary can create, interrupt, and teardown a TD. The adversary can also control computer resources allocated to a TD. Specifically, we consider the resources related to confidentiality and integrity in this paper. Moreover, the adversary can change input data to a TD using valid/invalid inputs via the TDX API calls. Side-channel attacks and denial-of-service attacks are not considered in this threat model.

## IV. Confidentiality and Integrity Verification

Figure 6 shows an overview of our proposed approach to validate Intel TDX module using property checking. We have used the TDX module implementation (C code) that is publicly available [5]. Different abstraction techniques are conducted on top of the implementation to minimize false negative results. Using the publicly available specifications [5], [19], [20], [21], we have derived different security invariants that are applicable to memory confidentiality and integrity of the TDX architecture. These invariants are decomposed into different sub-properties/assertions and are included in the implementation. Moreover, buffer overflow, pointer safety and arithmetic issues-related properties are also derived by performing static analysis of the TDX module implementation. Finally, model checking is used for verifying generated properties (assertions). The remainder of this section describes the four major steps in our framework: invariant analysis, static analysis, abstraction, and model checking.

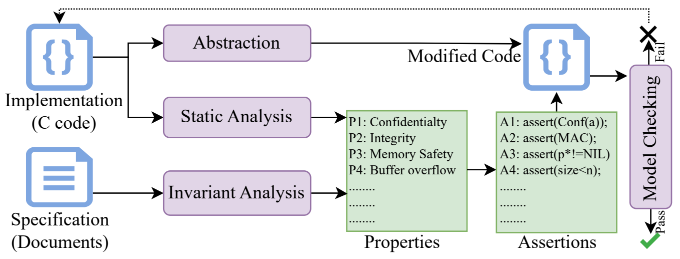

Fig. 6: Verifying TDX module using property checking

### A. *Invariant Analysis*

A security invariant is a property that is defined to prevent security-related issues in a system. Memory confidentiality and integrity are two major security guarantees provided by TDX. As shown in Figure 8, the key components that we are considering for memory confidentiality are HKID and ephemeral key. Similarly, we are considering the TD owner bit and the MAC for integrity. We have explored security invariants that are required to maintain memory confidentiality and integrity in TDX architecture. These invariants can be extended to similar architectures that is used in TDX. Some of the invariants were derived manually by going through publicly available specifications [21], papers [19], and the TDX module code [5]. Other invariants were derived automatically using Finite State Machine (FSM) analysis. Moreover, these invariants needs to be decomposed into several hardware and firmware properties for compositional security verification. The remainder of this section describes the invariants we derived for TDX memory confidentiality and integrity and the decomposition of the invariants into hardware and firmware properties.

Automated property generation using FSM analysis is the process of generating formal properties for the TEE system by analyzing its behavior using FSMs. First, we model the

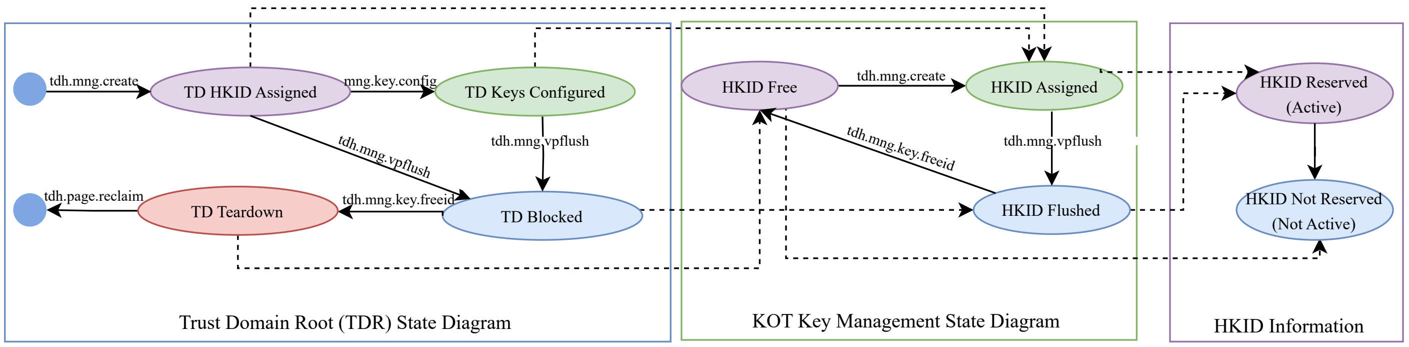

Fig. 7: Finite state machine combining TDR, KOT and HKID

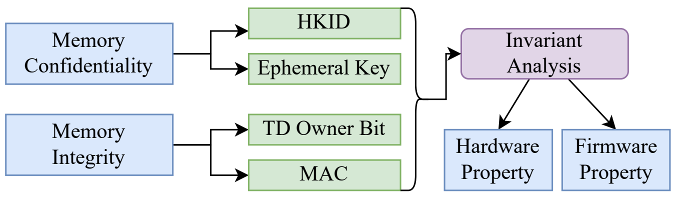

Fig. 8: Invariant analysis for confidentiality and integrity

system as an FSM. This step is conducted manually by going through the specifications [33], [21]. Figure 7 shows the states and transitions that are relevant for the confidentiality of TDX which consists of states and transitions between TDR, KOT and HKID states. When deriving the FSM presented in Figure 7, we found a documentation inconsistency in one of the Intel specifications [21]. It says that a transition between "TD HKID Assigned" to "TD Teardown" state is possible using $tdh.mng.key.freeid$. This should not be allowed because there can be pages allocated to that HKID. First, the "HKID Assigned" state should be transitioned into a blocked state before proceeding to the tear-down state. This is correctly represented in the architecture specification [33] and also in the implementation. We suggest that Intel specification [21] should be modified to reflect the correct transitions.

As shown in Figure 7, there are two types of transitions. Transitions between the same FSM (intra-FSM transitions) are represented in solid lines whereas transitions between FSMs (inter-FSM transitions) are represented in dashed lines. Listing 1 shows three properties derived based on FSM analysis. Specifically, the first two properties are generated by traversing intra-FSM transitions whereas the third property is generated using traversal of inter-FSM transitions. The first property asserts that for all time steps, if the state of TDR is Teardown, then it should lead to a situation where the state of KOT is HKID_Free, and the information associated with HKID is not reserved. The third property asserts that at some point in time, the state of TDR becomes Blocked, and this Blocked state continues to hold until a subsequent point in time when the state of TDR transitions to Teardown.

We have derived properties for the invariants using both manual and FSM analysis. Combining these approaches can enhance the comprehensiveness and rigor of the verification process, leading to more reliable and dependable TEE system designs. The rest of this section shows the six invariants we derived for memory confidentiality and integrity of Intel TDX.

**Listing 1** Sample properties generated using FSM analysis

```
// Properties generated using intra-transitions.
1) A(TDR.state == Teardown|->(KOT.state ==
   HKID_Free & HKID.info == Not Reserved))
2) A(TDR.state == Keys_Configured|->
   (KOT.state == HKID_Assigned & HKID.info
   == Reserved))
// Properties generated using inter-transitions.
3) (TDR.state == Blocked) U (TDR.state
   == Teardown)
```

1) *Invariant 1: TEE should not allow assigning an active private HKID to two different TDs.* An active private HKID maps to an ephemeral key. Therefore, if this HKID is shared with two different TDs, it will lead into a shared ephemeral key, which will allow shared access to encrypted data. Since the other TDs are not trusted, this could breach memory confidentiality. The TDX API that is responsible for the HKID allocation is $tdh\_mng\_create$. This API is used to create a TD and assign an HKID as discussed in Section II-B. HKID information (hkid_info) is stored as a 64-bit number where the first 16 bits are considered as HKID and the next 48 bits are for the reserved state. Note that the reserved field of an HKID consists of the reserved bits for future extension.

The property for the invariant 1 can be derived by analysing the FSM shown in Figure 7. By following the intra-FSM transitions from HKID not reserved state, we were able to derive an assertion as shown in Listing 2. This assertion is placed in the $tdh\_mng\_create$ API functionality. The assertion checks whether the reserved state is 0 to proceed with the HKID. Next, when assigning an HKID, the status of it is checked in the KOT to ensure that the HKID is in a free state. Finally, the key management field of TDR is checked before assigning the HKID. In TDX architecture, invariant 1 is implemented by these three security checks as well. All of these checks are implemented in a thread-safe manner. Even if the hypervisor can maliciously change the reserved state of an active HKID as not reserved, the KOT entry for that HKID will not be changed, as only TDX has access to KOT.

**Listing 2** Assertion for Invariant 1

```
assert((hkid_info.reserved==0) && (global
_data->kot.entries[td_hkid].state ==
KOT_STATE_HKID_FREE) && (tdr->management
_fields.lifecycle_state != (HKID_ASSIGNED
| TD_KEYS_CONFIGURED | BLOCKED)));
```

*2) Invariant 2: TEE should not allow to reassign private HKID when plaintext data is still in the CPU cache for that HKID.* The CPU cache is tagged with HKID. If an HKID is reassigned to another TD by a malicious VM, while some data is still residing in the cache for that HKID, the plaintext in the cache can be leaked by writing back the values to the memory by the malicious TD. Therefore, before assigning the HKID value to a new TD, previous cache lines associated with the HKID have to be flushed.

**Listing 3** Assertion for Invariant 2

```
assert(global_data->kot.entries[HKID].
state == HKID_FLUSHED)  && (tdr->management
_fields.lifecycle_state ==  BLOCKED));
```

The specification ensures invariant 2 behaviour by allowing the host VMM/hypervisor to flush the cache by using two APIs: $tdh\_phymem\_page\_wbinvd$ and $tdh\_phymem\_cache\_wb$. Since the hypervisor is not trusted, TDX implementation has added a cache line flush (*zero_area_cacheline*) before assigning the HKID value in $tdh\_mng\_create$ API functionality. This invariant can be decomposed into a hardware property to check whether the cache line got flushed before proceeding. Invariant 2 can be extended as a firmware property when we consider assigning an HKID value that has been blocked. When an HKID is blocked, the cache line is flushed and the state is updated in the KOT. We can use the assertion shown in Listing 3 derived by the FSM analysis to check whether the cache line is flushed indirectly by checking the state of the KOT entry. This assertion can be placed in the $tdh\_mng\_key\_freeid$ API call. Note that this assertion does not directly capture the condition that data is still residing in the cache. It checks whether the HKID is in the correct state after flushing the cache.

*3) Invariant 3: TEE should not allow any software or device which is not a TD to use the private HKID values.* If any application other than TD can get access to a private HKID, that application will be considered a TD and will gain access to the ephemeral key associated with that HKID value. This can compromise memory confidentiality.

**Listing 4** Assertion for Invariant 3

```
assert((is_private_hkid(HKID)) && (check_lock_
and_map_tdr(tdr_pa) == TDX_SUCCESS));
```

HKID can be either private or shared. Shared HKID values can be assigned to non-TD applications as well. However private HKID values can only be assigned to a TD. For a TD, there is a TDR structure to maintain the metadata. This TDR can be checked to see if a request is issued by a TD using *check_lock_and_map_tdr*. This function will return a success only if the TDR is mapped to a TD. The invariant 3 can be decomposed into a firmware assertion as shown in Listing 4. This assertion can be placed in the $tdh\_mng\_create$ API call.

*4) Invariant 4: TEE should not allow a TD to run without an HKID configured with TD's ephemeral key.* If the TD runs with an HKID that has not been configured with an ephemeral key, this would lead to a breach of memory confidentiality. TD should only start executing the operation after an ephemeral key is assigned by the TDX.

**Listing 5** Assertion for Invariant 4

```
assert((global_data->kot.entries[HKID].
state == HKID_ASSIGNED) && (tdr-> management_
fields.lifecycle_state == TD_KEYS_CONFIGURED));
```

TDX implementation achieves the invariant 4 by using a separate state of keys configured before moving to the runnable state. Key configuration is conducted using the $tdh\_mng\_key\_config$ API call. This API is used to update the KET per memory controller. Once the ephemeral key is generated using the MKTME engines that reside in memory controllers, the KET table is updated with HKID and corresponding key values as discussed in Section II-B. Invariant 4 can be decomposed into firmware and hardware properties. In each memory controller, a hardware property can be added to check whether the key is configured in the KET. The firmware property is shown in Listing 5, to check the state of the HKID value is key configured in TDR before proceeding to TD's runnable state.

*5) Invariant 5: TEE should not allow for a TD/TDX to read decrypted private data of the TD if the data is identified as corrupted.* An adversary can try to write data to a TD memory without having the proper privileges such as SEAM mode. If this data was read by the TD, integrity will be violated and also it can lead to an active adversarial attack based on the data written by the malicious entity. Therefore, it is important to identify the corrupted data before reading the private data.

**Listing 6** Assertions for Invariant 5

```
assert((Mode == SEAM) |-> (TD_Owner_Bit == 1));
assert((MAC.current == ECC.MAC));
```

Invariant 5 is implemented in the TDX module by checking the TD owner bit and MAC before each memory read by the memory controller. For each write, the memory controller checks whether the request is from the SEAM mode. According to the threat model, an adversary can create a legitimate

TD. Therefore, if the attack was conducted using a malicious TD, the TD owner bit will still remain 1. However, the MAC value will be changed. Therefore, the memory controller will be able to identify the integrity violation. If the adversary acts in a non-SEAM mode, the TD owner bit itself will be enough to identify the integrity violation. This invariant can be decomposed into hardware properties where we can check the value of the TD owner bit and the value of MAC. Sample hardware assertions are shown in Listing 6. This assertion should be added before reading from memory.

6) *Invariant 6: TEE should not allow the use of an HKID if the corresponding TD of that HKID has an integrity violation.* When an integrity violation is identified by a TD, the keys related to the HKID of that TD should not be used for further executions until the TD is torn down and HKID is again in the free state. If the TD is not torn down after the violation, it can lead to a breach of the integrity of the TDX architecture.

Implementation of the invariant 6 in the TDX module consists of TDX identifying that TD's execution is interrupted and sending the TD from a runnable state to a blocked state and then to a teardown state. We can have a property in the $tdh\_mng\_key\_free\_id$ to check whether the key is in the free state in both KOT and TDR data structure. This invariant can be decomposed into both hardware and firmware properties. In the hardware level, we can add a property to check the integrity violation. At the firmware level, we can check whether the HKID is getting freed after the required API calls. Listing 7 shows the firmware assertion to check the HKID state after $tdh\_mng\_key\_free\_id$ API call.

**Listing 7** Assertion for Invariant 6

```
assert((global_data->kot.entries[curr_hkid]
.state == HKID_FREE) && (tdr->management_
fields.lifecycle_state == TD_TEARDOWN));
```

## B. Static Analysis

Applying static analysis techniques in TEEs presents a range of unique challenges. TEEs have several priorities such as security and privacy. Therefore, the static analysis techniques must ensure that the analysis process itself does not undermine the core security goals of TEEs. Moreover, TEEs rely heavily on isolation mechanisms to protect sensitive information, requiring static analysis tools to appropriately handle interactions between code executed within and outside the TEE. The resource-constrained nature of TEE environments also poses a challenge, demanding that static analysis techniques be both accurate and efficient within these limited computational resources.

We perform static analysis to explore the TDX code implementation to figure out potential bugs. In this work, we are considering three potential bugs as shown in Figure 9. The three categories that we are considering are buffer overflow, pointer safety issues, and arithmetic issues. We generate assertions that have the capability of checking any potential vulnerabilities of these three classes. Our static analysis approach is different from the Google security review on Intel TDX [10], since we are generating properties/assertions using static analysis that can be used to give a formal guarantee. The rest of this section describes the three vulnerability classes.

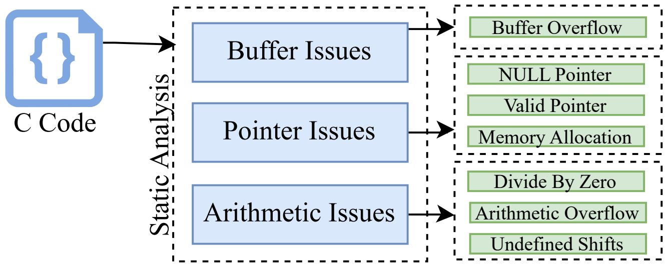

Fig. 9: Overview of assertion generation using static analysis

1) *Buffer Overflow:* A buffer overflow occurs when the size of data that is written to a buffer exceeds the actual storage capacity of the buffer. This would result in writing to adjacent memory locations of the buffer hence leading to unexpected behaviors such as memory crashes. Buffer overflows can be introduced by using invalid input patterns or by improper memory allocation. Adversaries can exploit buffer overflow issues by overwriting a memory of a corresponding application which can lead to different execution paths including arbitrary code execution. To check any potential buffer overflow vulnerabilities, we need to check all the buffers/arrays in the implementation and add assertions to check if any upper and lower bounds can be violated. A sample assertion for a buffer overflow is shown in Listing 8.

**Listing 8** Assertion for Buffer Overflow

```
assert(input_length >= 0 &&
input_length < sizeof(buffer));
```

2) *Pointer Safety Issues:* Pointer safety issues occur when the pointers are not handled properly in terms of allocation and usage. There can be several pointer safety issues. If a pointer is initialized to NULL and then this is dereferenced without checking for NULL could lead to unexpected behaviors and system crashes. Similarly, if the pointer is dereferenced to invalid inputs, this can cause some security violations as well. Improper memory allocations and deallocations of pointers can also lead to memory leaks. To check for the potential pointer safety issues in the implementation, we use static analysis to identify the pointers and add assertions related to pointer safety. Sample assertions are shown in Listing 9.

**Listing 9** Assertions for Pointer Safety Issues

```
assert(pointer != NULL);
assert(validPointer(pointer));
```

3) *Arithmetic Issues:* Arithmetic issues can occur due to various reasons such as arithmetic overflow, undefined shifts,

division by zero errors, etc. Arithmetic overflow can occur during an arithmetic operation. If the results of an arithmetic operation do not match the range of the return type declared in the implementation, it will lead to arithmetic overflow. Another cause for arithmetic issues is division by zero. This can lead the program to unexpected behavior and possible system crashes. When using floating point numbers, computation results can lead to a not-a-number state which also can lead to security vulnerabilities. When performing shift operations, it can lead to undefined states if the shifts have excessive distance. Adversaries can exploit arithmetic issues to change the behavior of the programs and bypass security checks. To check any potential arithmetic issues, we add assertions for arithmetic operations. Sample assertions are shown in Listing 10. For example, the assertion for the overflow check will use two types of inputs: data type and numerical values for the operation. The output of the assertion should be true if there is an overflow after performing the arithmetic operation.

**Listing 10** Assertions for Arithmetic Issues

```
assert(divisor != 0);
assert(!overflow(signed_int,val_1,val_2));
```

### C. Abstraction

Formal verification mathematically proves the correctness of a system with respect to the specification. However, it can lead to state space explosion when dealing with large and complex designs. Abstraction is a promising mechanism to reduce state space [34], [35]. As shown in Table I, abstraction can lead to false positive (verification passes when there are vulnerabilities) as well as false negative (identifies issues when there are none) results. As explained next, false positive results are not possible in our framework, but there can be false negative results.

TABLE I: False positive and false negative results in assertion-based verification.

| | No vulnerability in the design | Security vulnerability in the design |
|---|---|---|
| Assertion Pass | True Positive | False Positive |
| Assertion Fail | False Negative | True Negative |

Compositional verification is particularly useful for dealing with large and complex systems such as TEEs, where verifying the entire system at once might be impractical or computationally intensive. By breaking the verification process into smaller parts, each of which can be handled more easily, compositional verification helps manage complexity and improve the scalability of the verification process. We perform module-level model checking to reduce the complexity of the verification process. Specifically, we consider one API function at a time to perform property checking. This effectively removes implementation details that are not relevant for the API under consideration and focuses on the essential behaviors of an API. This requires us to abstract the interactions with other APIs. For example, if we can prove that assertions work for all possible values of HKID, in the original implementation (no approximation), it is guaranteed to work since it operates with a subset of values. If assertion fails for a specific value in the abstracted design, it may not fail in the original design since a TDX module may not access that value (e.g., only HKIDs inside the range of $max\_activated\_mktme\_hkid$ to $max\_activated\_hkids$ are considered as private HKIDs). For example, private HKID (16-bit) values can be tested with all possible patterns ($2^{16}$). This leads to over-approximation. As a result, there is no possibility of false positives in our framework, however, it can lead to false negatives. For example, HKID value 0 is reserved for total memory encryption engine, therefore, if there is an assertion failure for HKID 0, this is not a valid vulnerability in the original design.

False negatives are reduced by applying refinement. We are providing initial values and configurations during module-level evaluation. For example, private HKID value will never get the value 0. Also, only HKIDs inside the range of $max\_activated\_mktme\_hkid$ to $max\_activated\_hkids$ are considered as private HKIDs. We have used such constraints to refine the values which will reduce the over-approximation. We have used the two main functions (*get_global_data and get_local_data*) of the TDX implementation to get the configuration values that have been set up by the initialization process of the architecture. When we try the property checking at the API level, these global and local values are not set to the valid inputs. This can lead to false negative results. To address this issue, we abstracted the values for global and local data by allocating memory and mocking the behavior such that testing values will be in the range of valid inputs. Sample memory allocation mocking values of *get_local_data* are shown in Listing 11.

**Listing 11** Abstraction of Local Data

```
local_t* local_data = malloc(sizeof(local_t));
uint64_t rcx;
local_data_ptr->vmm_regs.rcx = rcx;
```

### D. Model Checking

During invariant analysis, we decompose the TEE architecture into two subsystems: firmware and hardware. We utilize assume-guarantee reasoning [36] to prove that derived properties through invariants are capable of collectively guaranteeing system-level assurance. Assume-guarantee reasoning involves decomposing a system under analysis into subsystems and analyzing them individually. By combining the results of these individual analyses, it becomes possible to determine whether the entire system satisfies the original property. This reduces the state space explosion problem of providing guarantees for an entire system. For a given subsystem we need to make assumptions regarding the environment of the subsystem to verify the property at that subsystem level. The verification problem can be represented as a triple, $<A>S<P>$, such that $S$ is the subsystem, $P$ is the property, and $A$ is an assumption of the environment of $S$. The formula $<A>S<P>$ is true, if $S$ under the assumption $A$ satisfies $P$. For our framework, $A$ can be the assumption that the environment is set for TDX SEAM mode, $S$ can be the firmware subsystem and $P$ can be the firmware property for invariant 6.

Suppose $S_1$ and $S_2$ are two of the subsystems that are being analyzed. The following rule can be applied to verify whether a property $P$ holds true for a system consisting both $S_1$ and $S_2$ denoted by $S_1||S_2$:

$$\begin{array}{rl} \text{Premise 1:} & <A>S_1 <\text{P}> \\ \text{Premise 2:} & <true>S_2 <\text{A}> \\ \hline & <true>S_1||S_2 <\text{P}> \end{array}$$

According to the rule, if subsystem $S_1$, operating under assumption $A$, satisfies property $P$, and subsystem $S_2$ satisfies assumption $A$, then the combined system $S_1||S_2$ also satisfies property $P$. Property $P$ can be verified on the composite system $S_1||S_2$ without the necessity of examining a unified model of the entire system. We are considering two major components in the TEE: hardware and firmware. For the hardware component, the boundary defines the interface through which it interacts with the firmware and the external environment. This includes hardware registers, memory-mapped I/O, communication protocols, and interrupt mechanisms. Assumptions about the hardware component might include its correct initialization with the root of trust and proper handling of hardware events such as interrupts or power fluctuations. These assumptions ensure that the firmware can rely on the hardware behaving as expected. The boundary of the firmware component is the interface through which it interacts with the hardware and potentially other firmware components. This includes function calls, system calls, and interactions with the hardware component. Assumptions about the firmware might include proper execution environments, memory safety, and correct handling of external events.

As the final step of our verification framework, we perform bounded model checking to verify such properties. The key idea behind bounded model checking is to explore all possible execution paths up to a certain bound. It systematically explores different paths within the bound, checking if any of them violate the specified property. The verification process in bounded model checking involves constructing a Boolean formula encoding the system behavior and property. This formula is then checked using an efficient SMT (Satisfiability Modulo Theories) solver. If the property checking passes, the property can be considered verified. However, if property checking fails, it should be checked for any false negatives which can be fixed using abstraction. After verifying that property failure is not a false negative, the generated counterexample can be used as a test case for efficient debugging and localization of a security vulnerability in the TEE design.

## V. Experiments

This section demonstrates the effectiveness of our proposed property-checking framework to verify the Intel TDX module. First, we describe our experimental setup. Next, we present the results of the property checking. Finally, we discuss applicability and limitations.

### A. Experimental Setup

For the experiments, we used the implementation that is publicly available for the TDX module [5]. Invariant analysis was conducted using the publicly available specification documentation [21]. For static analysis, we used the framework provided by CBMC [37]. For bounded model checking, we have used CBMC with its' in-built SMT solver [37]. We ran our experiments on Intel i7-5500U @ 3.0GHz CPU with 16GB RAM machine. For the property verification, we consider only the firmware properties since the firmware code is publicly available (TDX hardware implementation is not available).

### B. Experimental Results

This section presents the property checking results using CBMC [37] bounded model checker. Table II presents the property checking results for different API functions we checked in the TDX module *vmm_dispatcher* directory. This contains the largest attack surface for the TDX module. The first column shows the API name. The second column provides the number of lines in the C implementation of the API. The third, fourth and fifth columns describe the number of assertions checked for buffer issues, pointer safety and arithmetic safety, respectively. The sixth column shows the line coverage achieved by the model checking framework. The seventh and the eighth columns present the time(s) and memory(GB) taken to activate security assertions by CBMC [37]. These results demonstrate that most of the APIs have achieved more than 77% line coverage and have been checked for various vulnerability cases. The time and memory taken for the verification is also reasonable. Therefore, our proposed method can be used as an effective formal verification framework to identify vulnerabilities and give functional verification guarantees.

CBMC is widely used for bug hunting and verification in software systems, including TEEs. However, it is important to understand that while CBMC can uncover bugs and violations of specified properties within a bounded execution depth, it does not provide a formal guarantee of correctness in the absence of certain considerations and requirements. CBMC involves exhaustively exploring a finite state space up to a specified bound, searching for counterexamples that violate specified properties. It can effectively identify issues such as assertion violations, safety violations, and possible runtime errors within a given execution depth. In the realm of TEE formal verification, CBMC can be employed to detect security vulnerabilities, unauthorized data flows, or unexpected behaviors in TEE firmware. To move beyond CBMC's limitations and achieve more comprehensive formal guarantees in TEE verification, we have considered several approaches such as abstraction and invariant analysis.

Table III presents the property checking results for properties derived through invariant analysis. The first column shows the invariant. The second column shows whether the invariant decomposed into hardware (HW) and firmware (FW). The

TABLE II: Property checking results for different API functions

| API | No. of lines | Buffer Overflow | Pointer Safety | Arithmetic Safety | Line Coverage | Time (s) | Memory (MB) |
|---|---|---|---|---|---|---|---|
| tdh_mem_page_add | 273 | 0 | 66 | 1 | 91% | 32.31 | 298.8 |
| tdh_mem_page_aug | 197 | 0 | 50 | 10 | 90% | 20.71 | 370.6 |
| tdh_mem_page_demote | 252 | 0 | 30 | 8 | 90% | 40.18 | 721.7 |
| tdh_mem_page_promote | 257 | 3 | 32 | 1 | 91% | 161.27 | 3620.2 |
| tdh_mem_page_relocate | 259 | 1 | 30 | 3 | 87% | 499.93 | 1772.2 |
| tdh_mem_page_remove | 207 | 1 | 38 | 11 | 89% | 43.2 | 1151.2 |
| tdh_mem_range_block | 231 | 0 | 30 | 4 | 90% | 11.14 | 2027.5 |
| tdh_mem_range_unblock | 223 | 1 | 32 | 5 | 91% | 50.57 | 1085.8 |
| tdh_mem_sept_add | 208 | 1 | 42 | 7 | 73% | 20.71 | 370.3 |
| tdh_mem_sept_rd | 151 | 0 | 24 | 5 | 88% | 18.18 | 367.5 |
| tdh_mem_sept_remove | 238 | 1 | 60 | 8 | 87% | 45.47 | 729.3 |
| tdh_mem_track | 118 | 0 | 18 | 11 | 72% | 6.31 | 347.8 |
| tdh_mng_add_cx | 159 | 2 | 30 | 7 | 72% | 7.99 | 54.2 |
| tdh_mng_create | 125 | 4 | 48 | 4 | 54% | 6.47 | 32.3 |
| tdh_mng_init | 633 | 26 | 312 | 33 | 35% | 3410.5 | 8792.4 |
| tdh_mng_key_config | 119 | 0 | 24 | 0 | 96% | 21.91 | 267.8 |
| tdh_mng_key_freeid | 109 | 2 | 24 | 1 | 73% | 6.94 | 324.9 |
| tdh_mng_vpflushdone | 135 | 2 | 54 | 1 | 65% | 6.65 | 316.2 |
| tdh_mr_extend | 214 | 0 | 30 | 3 | 87% | 23.62 | 253 |
| tdh_phymem_cache_wb | 192 | 12 | 108 | 21 | 41% | 6.93 | 314.12 |
| tdh_phymem_page_rdmd | 104 | 0 | 92 | 2 | 100% | 37.44 | 894.4 |
| tdh_phymem_page_reclaim | 192 | 0 | 96 | 1 | 99% | 99.06 | 1176.6 |
| tdh_sys_config | 873 | 33 | 888 | 573 | 20% | 8.19 | 445.8 |
| Average | 237 | 3 | 93 | 31 | 77% | 199.3 | 1118.8 |

third column shows the verification status of the invariant. The fourth and fifth columns show the time(s) and memory(MB) taken for verification. We have verified all the firmware properties. Since we do not have access to the hardware implementation, we could not verify the hardware properties.

TABLE III: Property checking results for invariants.

| Invariant | Property Type | Verified | Time (s) | Memory (MB) |
|---|---|---|---|---|
| 1 | FW | Verified | 6.52 | 316.2 |
| 2 | HW + FW | FW Verified | 6.51 | 308.8 |
| 3 | FW | Verified | 6.83 | 315.9 |
| 4 | HW + FW | FW Verified | 7.12 | 325.2 |
| 5 | HW | N/A | - | - |
| 6 | HW + FW | FW Verified | 6.78 | 308.7 |

### *C. Applicability and Limitations*

In this paper, we focused on one specific TEE (Intel TDX architecture) and two properties (memory confidentiality and memory integrity). The four major steps in our proposed framework will remain the same for other TEEs since we are still checking confidentiality and integrity. In fact, the invariant analysis will also be similar for other TEE properties, such as attestation and availability. However, the invariant analysis will produce TEE- and property-specific invariants (e.g., temporal logic properties with different variable names).

A major limitation of our approach is the manual intervention in two scenarios. For example, the invariant analysis step is manual based on reading the specification (e.g., English document). However, property generation for invariants is a semi-automated process based on FSM analysis. Another limitation of our framework is that we cannot guarantee the completeness of the number of generated properties. However, the invariants that we derived are necessary for memory confidentiality and integrity. If any of the invaraints fails, it guarantees that the confidentiality and integrity is compromised. Our future research will explore mechanisms to automate these steps. One promising avenue to semi-automate the invariant analysis is to utilize TEE-specific templates. We can divide the TEEs into different categories such as VM-based, enclave-based, and TEEs for RISC-V based embedded systems. For each category, we can develop templates that can capture security properties.

## VI. Conclusion

A trusted execution environment (TEE) provides isolation and security guarantees (data confidentiality, data integrity, and code integrity) to enable trustworthy computing. It is a major challenge to verify the trustworthiness of TEEs since the security guarantees rely on complex interactions with different sub-systems, such as hardware, software, and firmware. In this paper, we present a formal property checking framework to verify Intel TDX memory confidentiality and integrity. Our proposed approach derived properties through invariant analysis of the specification, FSM analysis of the architectural components and static analysis of the implementation. We utilize abstraction as well as bounded model checking to verify the properties. Our experimental results demonstrated the effectiveness of the proposed formal verification to verify security properties. It also enabled us to find inconsistency in the TDX specification. While this work focused on confidentiality and integrity verification for Intel TDX, the steps outlined in our framework can be used as a stepping stone to formally verify security guarantees for other TEE architectures.

## References

[1] Dalton CG Valadares, Álvaro Sobrinho, Newton C Will, Kyller C Gorgônio, and Angelo Perkusich. Trusted and only trusted. that is the access! improving access control allowing only trusted execution environment applications. In *International Conference on Advanced Information Networking and Applications*, pages 490–503. Springer, 2023.

[2] Carsten Weinhold, Nils Asmussen, Diana Göhringer, and Michael Roitzsch. Towards modular trusted execution environments. In *Proceedings of the 6th Workshop on System Software for Trusted Execution*, pages 10–16, 2023.

[3] ARM TrustZone. https://www.arm.com/technologies/trustzone-for-cortex-a, 2023.

[4] Intel Software Guard Extensions (SGX). https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html, 2023.

[5] Intel Trust Domain Extensions (TDX). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, 2023.

[6] IBM Secure Execution. https://www.ibm.com/docs/en/linux-on-systems?topic=virtualization-introducing-secure-execution-linux, 2023.

[7] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016.

[8] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[9] Guerney DH Hunt, Ramachandra Pai, Michael V Le, Hani Jamjoom, Sukadev Bhattiprolu, Rick Boivie, Laurent Dufour, Brad Frey, Mohit Kapur, Kenneth A Goldman, et al. Confidential computing for openpower. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 294–310, 2021.

[10] Intel Trust Domain Extensions (TDX) Security Review. https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf, 2023.

[11] 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.

[12] Eriko Nurvitadhi, James C Hoe, Timothy Kam, and Shih-Lien L Lu. Integrating formal verification and high-level processor pipeline synthesis. In *2011 IEEE 9th Symposium on Application Specific Processors (SASP)*, pages 22–29. IEEE, 2011.

[13] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. Moat: Verifying confidentiality of enclave programs. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1169–1184, 2015.

[14] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A Seshia. A formal foundation for secure remote execution of enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2435–2450, 2017.

[15] Yuwei Ma, Qianying Zhang, Shijun Zhao, Guohui Wang, Ximeng Li, and Zhiping Shi. Formal verification of memory isolation for the trustzone-based tee. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, pages 149–158. IEEE, 2020.

[16] Haiyong Sun and Hang Lei. A design and verification methodology for a trustzone trusted execution environment. *IEEE Access*, 8:33870–33883, 2020.

[17] Dalton Cézane Gomes Valadares, Álvaro Alvares de Carvalho César Sobrinho, Angelo Perkusich, and Kyller Costa Gorgonio. Formal verification of a trusted execution environment-based architecture for iot applications. *IEEE Internet of Things Journal*, 8(23):17199–17210, 2021.

[18] Muhammad Usama Sardar, Saidgani Musaev, and Christof Fetzer. Demystifying attestation in intel trust domain extensions via formal verification. *IEEE access*, 9:83067–83079, 2021.

[19] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel tdx demystified: A top-down approach. *arXiv preprint arXiv:2303.15540*, 2023.

[20] Ravi Sahita, Dror Caspi, Barry Huntley, Vincent Scarlata, Baruch Chaikin, Siddhartha Chhabra, Arie Aharon, and Ido Ouziel. Security analysis of confidential-compute instruction set architecture for virtualized workloads. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 121–131. IEEE, 2021.

[21] Intel Trust Domain Extensions (Intel TDX) Module v1.5 Base Architecture Specification. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, 2023.

[22] Antonio Jesus Muñoz-Gallego, Ruben Rios-del Pozo, Rodrigo Roman-Castro, Francisco Javier López-Muñoz, et al. A survey on the (in) security of trusted execution environments. 2023.

[23] Pranav Gaddamadugu. *Formally verifying trusted execution environments with uclid5*. PhD thesis, MA thesis. EECS Department, University of California, Berkeley, 2021.

[24] Weggli. https://github.com/weggli-rs/weggli, 2023.

[25] Frama-C. https://github.com/Frama-C, 2023.

[26] Hasini Witharana, Yangdi Lyu, and Prabhat Mishra. Directed test generation for activation of security assertions in rtl models. *ACM Transactions on Design Automation of Electronic Systems*, 26(4), 2021.

[27] Hasini Witharana, Yangdi Lyu, Subodha Charles, and Prabhat Mishra. A survey on assertion-based hardware verification. *ACM Computing Surveys (CSUR)*, 54(11s):1–33, 2022.

[28] Hasini Witharana, Aruna Jayasena, Andrew Whigham, and Prabhat Mishra. Automated generation of security assertions for rtl models. *ACM Journal on Emerging Technologies in Computing Sys.*, 19(1), 2023.

[29] Aruna Jayasena, Binod Kumar, Subodha Charles, Hasini Witharana, and Prabhat Mishra. Network-on-chip trust validation using security assertions. *Journal of Hardware and Systems Security*, 6(3-4):79–94, 2022.

[30] Yuanwen Huang, Prabhat Mishra, and Farimah Farahmandi. *System-on-Chip Security: Validation and Verification*. Springer, 2020.

[31] Muhammad Usama Sardar, Thomas Fossati, and Simon Frost. Comprehensive specification and formal analysis of attestation mechanisms in confidential computing. *ICE 2023 Pre-Proceedings*, 2023.

[32] AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/, 2023.

[33] Architecture Specification: Intel Trust Domain Extensions (Intel TDX) Module. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, 2023.

[34] Pramod Subramanyan, Yakir Vizel, Sayak Ray, and Sharad Malik. Template-based synthesis of instruction-level abstractions for soc verification. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*, pages 160–167. IEEE, 2015.

[35] Jamieson M Cobleigh, George S Avrunin, and Lori A Clarke. Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):1–52, 2008.

[36] Thomas A Henzinger, Shaz Qadeer, and Sriram K Rajamani. You assume, we guarantee: Methodology and case studies. In *Computer Aided Verification: 10th International Conference, CAV'98 Vancouver, BC, Canada, June 28–July 2, 1998 Proceedings 10*, pages 440–451. Springer, 1998.

[37] Daniel Kroening and Michael Tautschnig. Cbmc–c bounded model checker: (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*, pages 389–391. Springer, 2014.