2025 Design, Automation & Test in Europe Conference (DATE 2025) – Focus Session

Security Assertions for Trusted Execution Environments

Hasini Witharana, Hansika Weerasena and Prabhat Mishra
Department of Computer & Information Science & Engineering
University of Florida, Gainesville, Florida, USA

***Abstract*—Trusted Execution Environment (TEE) provides a secure and isolated execution environment for sensitive applications. In order to design secure and trustworthy TEE-based systems, it is crucial to verify the trustworthiness of TEE implementations. Property checking is a promising avenue to guarantee that the TEE implementation satisfies the security properties. In the presence of a vulnerability, property checking will fail and provide a counterexample that can be utilized to fix the vulnerability. A major challenge in TEE property checking is that it relies on manual definition of the security properties, which can be cumbersome and error-prone. In this paper, we propose an efficient framework for automated generation and verification of TEE specific properties. Specifically, we leverage Finite State Machine (FSM) analysis to automatically derive and validate security properties utilizing templates. The effectiveness of the proposed method is demonstrated through experimental evaluation of Intel Trust Domain Extension (TDX), highlighting its potential for verifying security and trustworthiness of modern trusted execution environments.**

***Index Terms*—Security Verification, Property Checking, Trusted Execution Environment, Confidential Computing**

I. Introduction

Trusted execution is an important mechanism in modern computing systems that offers an isolated and secure environment for processing sensitive data and critical operations, as shown in Figure 1. By creating a protected environment that is shielded from potentially untrusted components (i.e. Operating system), Trusted Execution Environments (TEEs) ensure the confidentiality, integrity, and authenticity of data in all forms including data in storage, in transit, and in use. TEE solutions are used in many domains such as financial transactions, cryptographic operations, cloud computing, etc.

TEE systems are complex due to the interactions between multiple subsystems and layers of software, firmware, and hardware. For example, a specific security guarantee may involve interactions between secure enclaves, virtual machines, trusted application manager, operating system, key manager, and secure storage. Verifying the correctness of each subsystem and its interactions becomes a non-trivial and challenging task. There are promising formal methods, such as theorem proving, equivalence checking, and property checking, for verifying trustworthiness of TEEs [1], [2], [3], [4], [5], [6].

Property checking is one of the most widely used form of formal verification. The security and functional correctness of TEE can be verified by checking security properties. Specifically, property checking helps identify and address security vulnerabilities, verify compliance with specifications,

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

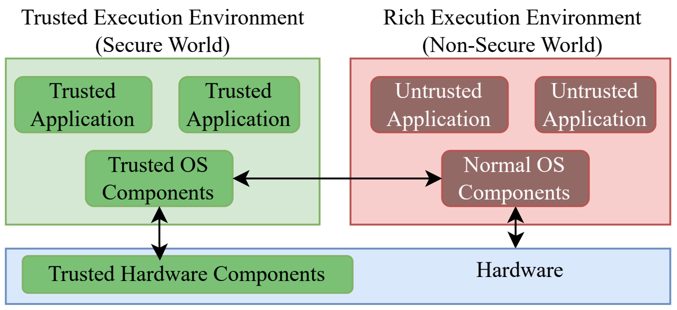

Fig. 1: Overview of classical trusted execution environments.

and ensure that the TEE operates as intended. The TEEs are designed to protect sensitive data and code, and therefore, their security requirements are complex. Verifying these security requirements using properties often involves capturing and expressing complex security protocols. Ensuring that properties accurately capture the desired security behaviors can be challenging and requires extensive manual effort from security experts [7], [8], [9]. The manual property generation can be cumbersome and error-prone. In this paper, we propose a framework for automated generation of security properties.

Figure 2 shows an overview of our proposed framework. Our automated property generation framework consists of four steps. The first step involves the extraction of Finite State Machine (FSM) from the TEE specification documents. These FSMs capture the behavior of individual subsystems of a TEE architecture. In the second step, the extracted FSMs are integrated by combining their states and transitions to form a unified FSM model that represents the collective behavior of the subsystems. In the third step, we automatically generate properties utilizing the FSMs and property templates. Finally, we perform property checking of the TEE implementation using the generated security properties. Specifically, this paper makes the following major contributions:

- We extract FSMs from subsystems using decomposition analysis of the TEE specification.
- We combine FSMs to construct an integrated FSM that represents the TEE as a system
- Our framework automatically generates properties based on FSM analysis utilizing property templates.
- We perform model checking of the TEE implementation using the generated properties (assertions).

This paper is organized as follows. Section II surveys related efforts. Section III describes our proposed property generation framework. Section IV presents the experimental results. Finally, Section V concludes the paper.

II. Related Work

Prior TEE verification efforts can be broadly divided into three categories: static analysis [10], simulation-based testing [11], and formal verification [1], [2], [3], [4], [12], [5], [6]. A recent security assessment by Google [10] on Intel TDX employed static analysis, uncovering 81 attack vectors and confirming 10 security issues. Also, Google conducted simulation-based testing on AMD Secure Encrypted Virtualization (SEV), employing various test vectors and manually deriving security invariants [11]. Security vulnerabilities in the code were also identified using fuzzing-based techniques. *The security assessment by Google [10] highlighted the need for formal verification of TEE implementations.* Moat [1] introduced a formal abstraction for the Intel Software Guard Extensions (SGX) model, forming the basis for theorem-based confidentiality verification. Subramanyan et al. extend this work to verify integrity, confidentiality, and secure measurement for Intel SGX [2]. Similarly, ARM TrustZone [3], [4] and Intel TDX [5] utilize formal verification approaches to verify different security behaviours. In these works properties and theorems are derived manually. A property checking framework with design abstraction and manual property generation was proposed for verifying VM-based TEEs [6]. *To the best of our knowledge, there are no prior efforts in automatically generating security properties (assertions) for TEE-specific architectures.*

III. Automated Generation of TEE Properties

Figure 2 presents an overview of our approach for automatic property generation to verify TEEs. The process involves four key steps: (1) extracting local FSMs from TEE specifications, (2) composing a global FSM by identifying external relations, (3) generating properties using the global FSM and property templates, and (4) performing property checking on TEE implementations. This section begins with key concept definitions, followed by a detailed explanation of each step.

A. Definitions and Preliminaries

Definition 1: An FSM is a mathematical model that consists of a finite set of states (Q), a finite set of input symbols (Σ), an initial state from which the machine starts ($q_0 \in Q$), a transition function that maps the current state and input symbol to the next state ($\delta : Q \times \Sigma \rightarrow Q$), a finite set of output symbols (Γ), and an output function that maps each state to an output symbol ($\lambda : Q \rightarrow \Gamma$). Formally, an FSM can be represented as a tuple $(Q, q_0, \Sigma, \delta, \Gamma, \lambda)$.

Definition 2: A global FSM is constructed by composing n local FSMs ($FSM_1, FSM_2, \ldots, FSM_n$), where each FSM_i is defined as $(Q_i, q_{0i}, \Sigma_i, \delta_i, \Gamma_i, \lambda_i)$. The global FSM behavior is determined by interconnecting the local FSMs. The inter-FSM transitions connecting local FSMs are defined using the transition function δ_{global} based on external factors or the behavior of other local FSMs. The inter-FSM transition function is defined as: $\delta_{\text{global}}(q_{\text{global}}, s_{\text{global}}, \textit{external}) = q'_{\text{global}}$, where q_{global} is the current state of the global FSM, s_{global} is an input symbol from the input alphabet Σ_{global} of the global FSM, *external* represents any external context or influence on the inter-FSM transition, and q'_{global} is the next state of the global FSM after the transition.

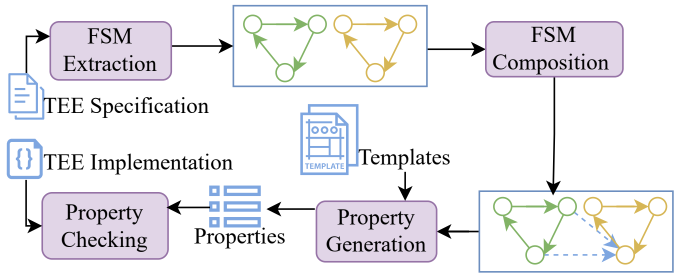

Fig. 2: Overview of our automated property generation framework that consists of four major steps: (1) Finite State Machine (FSM) extraction, (2) FSM composition, (3) property generation using templates, and (4) property checking.

Definition 3: In the context of FSMs, an *atomic proposition*, denoted as *ap*, is a logical statement that evaluates to true or false based on whether a certain condition holds regarding the FSM's current state. For example, an atomic proposition could be defined as ap = *"FSM.state is state A"*, where ap evaluates to true (T) if the FSM's current state is indeed state A, and it evaluates to false (F) otherwise.

Definition 4: We consider the following four attributes while generating a TEE-specific security property φ.

- G (Globally): $G\varphi$ means that φ must hold true for all future time steps.
- F (Finally): $F\varphi$ means that φ will become true at some point in the future.
- U (Until): For the property ψ, $\varphi U \psi$ means that φ must hold true until ψ becomes true.
- X (Next): $X\varphi$ means φ must hold true in the next step.

B. FSM Extraction

FSM extraction from the TEE specification involves the systematic analysis and abstraction of the TEE's behavior into a structured representation. There are typically several local FSMs or subsystems that work together to ensure the security and functionality of the TEE. These local FSMs are responsible for various aspects of TEE operation and security. The behavior of FSMs can vary depending on the TEE subsystems. For instance, in a typical TEE we can extract FSMs for the core subsystems, such as the security monitor that is responsible for overseeing the overall security posture, secure boot which ensures a trusted boot process, TEE application lifecycle which manages the lifecycle of trusted applications within the TEE, key management system that manages keys to ensure confidentiality, secure storage management which governs secure data storage and access control, cryptographic operations handle cryptographic functions, the attestation process is essential for providing evidence of the TEE's security state to external entities, etc. For all these sub-systems, the local FSMs can be extracted by analyzing

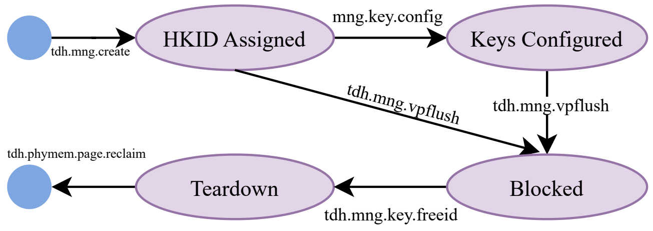

Fig. 3: Local FSM for Trust Domain (TD) life cycle

the specification. These local FSMs work together to create a secure and controlled environment within the TEE.

Figure 3 shows an example of a local FSM of a TEE application lifecycle. This is extracted from the Intel TDX specification [13]. TDX possesses the ability to instantiate hardware-isolated virtual machines, known as trust domains (TD). Intel TDX employs a combination of multi-key total memory encryption (MKTME) and hashing techniques to ensure the confidentiality and integrity of code and data within TDs. Every key that MKTME supports is uniquely identified by a Host Key Identifier (HKID). The local FSM presents the TD life cycle. There are four TD life cycle states: (1) HKID assigned, (2) keys configured, (3) blocked, and (4) teardown. HKID assigned state is reached through the use of the tdh_mng_create API, where a new TD is created. In this state, the hypervisor first flushes the cache, ensuring there are no modified cache lines for the physical pages. Subsequently, it generates the Trust Domain Root (TDR) and creates a random ephemeral key for the TD. An HKID is generated, and the KeyID Ownership Table (KOT) for each package is updated. The majority of the TD's lifetime is spent in the keys configured state. The TD's ephemeral key is configured in the Key Encryption Table (KET). In the event of an interruption or fault, the TD transitions into the blocked state. Access to the TD's private memory is blocked, and relevant caches are flushed. The $tdh\_mng\_vpflushdone$ API is employed to verify whether all cache lines associated with the address or HKID have been flushed. During the teardown state, the host Virtual Machine Monitor (VMM) reclaims the HKID and performs TLB and cache flushes. All TD private and control pages are removed using $tdh\_phymem\_page\_reclaim$ API.

C. *FSM Composition*

FSM composition integrates multiple local FSMs that collectively represent the behavior and interactions of different components within the TEE. In a TEE, various subsystems, such as secure boot, memory protection, and cryptographic operations, operate concurrently but in a coordinated manner to ensure security and isolation. FSM composition enables the creation of a global FSM that captures the synchronized behavior of these subsystems. Each subsystem within the TEE can be abstracted as a local FSM with its own states, transitions, inputs, and outputs. FSM composition brings these individual FSMs together, combining their states and transitions to form a unified representation of the TEE's behavior.

The inter-FSM transition function denoted as δ_{global}, serves as the mechanism for defining how transitions between states in the global FSM occur based on external factors and the behavior of other local FSMs. It takes into account the following components: q_{global}, s_{global}, *external*, and q'_{global}. As outlined in Definition 2, q_{global} represents the current state of the global FSM. It is crucial because it reflects the current state of the TEE as a whole. Similarly, s_{global} symbolizes an input or event originating from the global input alphabet Σ_{global}. These inputs could include external commands, sensor data, communication signals, or other events that trigger transitions in the global FSM. Likewise, *external* captures any external factors, such as security events, system interrupts, communication with external entities, or environmental conditions. q'_{global} denotes the TEE's state after processing the inputs and external factors.

A simplified version of the global FSM for Intel TDX is shown in Figure 4. The global FSM is composed using two local FSMs corresponding to the TD life cycle and key management in TDX architecture. FSM_1 can be defined as $(Q_1, q_{01}, \Sigma_1, \delta_1, \Gamma_1, \lambda_1)$. Q_1 represents the states in FSM_1 where Q_1 = (q_0, HKID Assigned, Keys Configured, Blocked, Teardown). The initial state of the FSM is represented by q_{01} and the initial state of Q_1 by q_0. Σ_1 have the input symbols which are the API functions that are used to transition from one state to another (Σ_1 = (create, config, vpflush, freeid, reclaim)). Γ_1 shows the output symbols such as whether an HKID is reserved or not. If the reserved bits of HKID are all zeros, that HKID is inactive. If the reserved bits of HKID are not all zeros, that HKID is active. Therefore, Γ_1 = (Active, Inactive). Similarly, FSM_2 can be defined as $(Q_2, q_{02}, \Sigma_2, \delta_2, \Gamma_2, \lambda_2)$ where Q_2 = (Free, Assigned, Flushed), q_{02} = Free, Σ_2 = (config, vpflush, freeid) and Γ_2 = (Active, Inactive). The global FSM combining FSM_1 and FSM_2 can be represented by $(Q_{\text{global}}, q_{0\text{global}}, \Sigma_{\text{global}}, \delta_{\text{global}}, \Gamma_{\text{global}}, \lambda_{\text{global}})$ where Q_{global} = (q_0, HKID Assigned, Keys Configured, Blocked, Teardown, Free, Assigned, Flushed), $q_{0\text{global}}$ = (q_0, Free), Σ_{global} = (create, config, vpflush, freeid, reclaim) and Γ_{global} = (Active, Inactive).

D. *Property Generation using Templates*

The composed FSM provides a comprehensive view of how different components within the TEE interact and respond to various inputs and events. Based on this understanding, properties can be derived that express critical conditions such as safety, liveness, reachability, concurrency, confidentiality, and integrity. Property generation involves translating high-level security goals into temporal logic, often using Linear Temporal Logic (LTL) or Computation Tree Logic (CTL). The composed FSM serves as the foundation for specifying these properties, as it accurately represents the coordination and interactions of TEE components. We utilize templates to derive properties from the composed FSM.

Table I presents the templates for six types of properties that are generated using the FSM analysis. Each row represents a specific property type along with its associated template and an explanation. In the templates, $q_1, q_2, \ldots, q_n \in Q$, $\sigma_1, \sigma_2 \in \Sigma$, and $\gamma_1 \in \Gamma$. Note that all the phrases in the template are atomic propositions $(q_1, q_2, \sigma_1, \sigma_2, \gamma_1 \in ap)$. *Safety*

TABLE I: Templates for different types of properties.

Property	Template	Explanation
Safety	$G(\mathrm{q_1} \rightarrow \mathrm{X}(\neg(\mathrm{q_2 U}\sigma_1)))$	In state q_1, input σ_1 should never lead to state q_2.
Liveness	$G((\mathrm{q_1} \ \& \ \sigma_1) \rightarrow \mathrm{F\ q_2})$	In state q_1, input σ_1 should eventually lead to state q_2.
Concurrency	$G((\mathrm{q_1} \ \& \ \mathrm{q_2} \ \& \ \cdots \ \& \ \mathrm{q}_n) \rightarrow \gamma_1)$	Concurrent execution of states $q_1, q_2, \dots, q_n$ should result in output γ_1.
Reachability	$G(\mathrm{q_1} \rightarrow \mathrm{q_2})$	There exists a valid path to reach q_2 starting from q_1.
Confidentiality	$G(\mathrm{q_1} \rightarrow \mathrm{X}(\neg(\mathrm{q_2 U}\sigma)))$	In state q_1, input σ should never lead to state q_2, where q_2 represents an data leakage (unauthorized access) state.
Integrity	$G(\mathrm{q_1} \rightarrow \mathrm{X}(\neg(\mathrm{q_2 U}\sigma)))$	In state q_1, input σ should never lead to state q_2, which indicates a compromised/tampered integrity state.

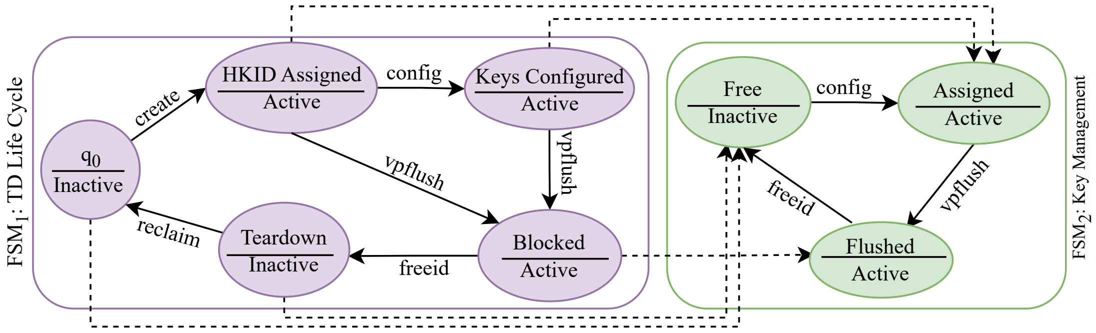

Fig. 4: A sample global FSM for TDX by composing the TD life cycle FSM with key management FSM

property, represented by the template $G(\mathrm{q_1} \rightarrow \mathrm{X}(\neg(\mathrm{q_2 U}\sigma_1)))$, ensures that in state q_1, input σ_1 should never lead to state q_2. It emphasizes system safety and prevents unauthorized execution or unwanted behavior. *Liveness* property, given by $G((\mathrm{q_1} \ \& \ \sigma_1) \rightarrow \mathrm{F\ q_2})$, guarantee that in state q_1, input σ_1 should eventually lead to state q_2. Liveness properties focus on the assurance of desirable future states or behaviors. *Reachability* property, captured by $G(\mathrm{q_1} \rightarrow \mathrm{q_2})$, asserts that there exists a valid path to reach state q_2 starting from q_1. It addresses the accessibility of specific states. The template $G((\mathrm{q_1} \ \& \ \cdots \ \& \ \mathrm{q}_n) \rightarrow \gamma_1)$ describes *concurrency* properties, where concurrent execution of states $q_1, q_2, \dots, q_n$ should result in output γ_1. The concurrency properties ensure parallelism without conflicts.

Confidentiality properties, captured by the template $G(\mathrm{q_1} \rightarrow \mathrm{X}(\neg(\mathrm{q_2 U}\sigma)))$, ensure that in state q_1, input σ should never lead to an unauthorized access state q_2. These properties focus on preventing unauthorized entities from accessing sensitive data. Integrity properties, also represented by $G(\mathrm{q_1} \rightarrow \mathrm{X}(\neg(\mathrm{q_2 U}\sigma)))$, ensure that in state q_1, input σ should never lead to state q_2, which indicates a compromised integrity state.These properties ensure that TEE data and code remain untampered by unauthorized processes. Note that safety, confidentiality, and integrity properties share the same template, however, the template's meaning depends on the specific states and transitions being modeled. While the logical structure remains consistent, the context of the states (e.g., preventing unauthorized access for confidentiality or ensuring data immutability for integrity) differentiates the properties.

Algorithm 1 shows the automated property generation using templates. The algorithm iterates through each state in the global FSM, starting from the initial state (q_0). For each state, the algorithm identifies its next local states (LS), which

Algorithm 1 Automated generation of security properties

```
Require: FSM $(Q, q_0, \Sigma, \delta, \Gamma, \lambda)$
Ensure: Set of properties of various types: $P$
1:  $P \leftarrow \emptyset$
2:  for $q \in Q$ do
3:      $LS \leftarrow q$.next_local_states ($LS \in Q$)
4:      for $ls \in LS$ do
5:          $lt \leftarrow$ local_transition($q$, $ls$)
6:          $P \leftarrow$ liveness_property($q$, $ls$, $lt$)
7:          $IT \leftarrow \Sigma \setminus \{lt\}$
8:          for $it \in IT$ do
9:              $P \leftarrow P \cup$ safety_property($q$, $ls$, $it$)
10:             $P \leftarrow P \cup$ confidentiality_property($q$, $ls$, $it$)
11:             $P \leftarrow P \cup$ integrity_property($q$, $ls$, $it$)
12:         end for
13:     end for
14:     $GS \leftarrow q$.next_global_states ($GS \in Q$)
15:     $o \leftarrow q$.output ($o \in \Gamma$)
16:     $P \leftarrow P \cup$ concurrency_property($q$, $GS$, $o$)
17:     $VS \leftarrow Q \setminus \{q\}$
18:     $P \leftarrow P \cup$ reachability_property($q$, $VS$)
19: end for
20: Return $P$
```

represent potential future states within the same local FSM. It calculates the local transition (lt) between the current state and each next local state. Subsequently, it generates liveness properties for these transitions, ensuring that the FSM will eventually reach the identified next local states. Next, we calculate a set of invalid transitions (IT) excluding the current local transition (lt). Then we iterate through the remaining invalid transitions in IT and generate safety properties for

TABLE II: Sample properties generated for the FSM presented in Figure 4.

Type	# Prop.	Example Property
Safety	36	$G((\text{FSM}_1.\text{state} == \text{HKID Assigned}) \rightarrow \text{X}(\neg((\text{FSM}_1.\text{state} == \text{Keys Configured})\text{U}(\text{transition} == \text{create}))))$
Liveness	9	$G(((\text{FSM}_1.\text{state} == \text{HKID Assigned})\ \&\ (\text{transition} == \text{config})) \rightarrow (\text{FSM}_1.\text{state} == \text{Keys Configured}))$
Reachability	56	$G((\text{FSM}_1.\text{state} == \text{HKID Assigned}) \rightarrow (\text{FSM}_1.\text{state} == \text{Blocked}))$
Concurrency	6	$G(((\text{FSM}_1.\text{state} == \text{HKID Assigned})\ \&\ (\text{FSM}_2.\text{state} == \text{Assigned})) \rightarrow \text{Active})$
Confidentiality	8	$G(((\text{FSM}_1.\text{state} == \text{HKID Assigned})\ \&\ (\text{transition} == \text{read})) \rightarrow \text{FSM}_1.\text{KOT} == \text{Accurate})$
Integrity	7	$G(((\text{FSM}_1.\text{state} == \text{TDR Finalized}) \rightarrow \neg((\text{FSM}_1.\text{state} == \text{INIT}) \vee (\text{FSM}_1.\text{state} == \text{FATAL})))$

each one. While still focused on the current state, the algorithm determines the next global states (GS) from the current state, accounting for transitions that may lead to other FSMs in a global context. Simultaneously, it retrieves the output symbol (o) associated with the current state. Using this information, the algorithm generates a concurrency property for the current state, ensuring that its output allows for concurrent execution with other states in the system. Finally, the algorithm creates a set of all other valid local states (VS) in the FSM, excluding the current state. It generates reachability properties for the current state to verify the existence of valid paths to all other states in the local FSM. These reachability properties confirm that the FSM can access all its states.

E. Property Checking

The properties are converted to assert statements and included in the implementation. Then, we use bounded model checking to verify specified properties. Bounded model checking explores all execution paths up to a defined bound, constructing Boolean formulas for both the implementation and properties. An Satisfiability Modulo Theories (SMT) solver validates these formulas, confirming property verification or generating counterexamples for debugging in case of failure. This approach helps identify security vulnerabilities in TEE implementations.

IV. Experiments

A. Experimental Setup

We use state-of-the-art TEE architecture, the implementation of the Intel TDX module [13]. Table III shows the number of files and line of code for different languages used in the TDX implementation. Total of 8741 files and 867303 number of lines are considered for the evaluation platform. The FSM analysis and FSM composition are conducted utilizing the TDX specification documentations [13]. Then the global FSM is encoded in the formal verification model using Python and the property generation algorithm is applied to the global FSM model. For property checking, we use CBMC [14] bounded model checking with its built-in SMT solver. Our experiments were conducted on a machine with Intel i7-5500U @ 3.0GHz CPU with 16GB RAM. For the property verification, our primary focus was verification of firmware properties, given the public release of the TDX module code.

TABLE III: Intel TDX module implementation

	C	Assembly	C++	Python	Total
No. of Files	7725	711	281	24	**8741**
Line of Code	575937	238098	51073	2195	**867303**

B. Property Generation Results

Algorithm 1 generated 122 properties for TDX module (Figure 4) that includes 36 safety properties, 9 liveness properties, 56 reachability properties, 6 concurrency properties, 8 confidentiality properties, and 7 integrity properties. Table II shows one example property for each property type. The example safety property ensures that if the state is "HKID Assigned", it is guaranteed that the state will not transition to "Keys Configured" until a specific condition ("create" transition) is met. The example liveliness property states that in FSM_1, if the state is "HKID Assigned" and a "config" transition occurs, it is guaranteed that the state will eventually reach "Keys Configured". The example reachability property asserts that in FSM_1, if the state is "HKID Assigned", the FSM can reach the state "Blocked". The example concurrency property states that in FSM_1 and FSM_2, if "HKID Assigned" and "Assigned" states are concurrently active, an "Active" state should be maintained. The example confidentiality property ensures that once the state is "HKID Assigned" and a "read" transition occurs, the HKID's state remains confidential and accurately reflects its assigned status within the Key Ownership Table (KOT). This ensures that the system does not transition to an inaccurate or exposed state in relation to the HKID assignment. The example integrity property asserts that once the state is "TDR Finalized" and a "finalize" transition occurs, the TDR's lifecycle state cannot return to "INIT" or "FATAL". This guarantees that the integrity of the TDR lifecycle is maintained and that the finalized state cannot be compromised by reverting to an initial or fatal state.

Listing 1 Assertion for the concurrency property in Table II

```
assert((global_data->kot[hkid].state==ASSINED)
&& (tdr->lifecycle_state==HKID_ASSIGNED)
&& (hkid.reserved!=0));
```

The generated properties are converted to assertions and placed in appropriate APIs in the TDX module implementation for verification. Listing 1 shows an assertion that represents the concurrency property shown in Table II. This assertion is placed in the *tdh_mng_create* API after TD creation. The assertion checks the TDR life cycle is in the "HKID_Assigned" state and KOT entry is "Assigned" at the same time. Also, it checks whether the HKID is active concurrently with the other two conditions. Similarly, all 122 properties are converted to assert statements and inserted in the implementation. Next, we used CBMC [14] to verify the assertions.

TABLE IV: Property Generation and Verification Results for Intel TDX. The presented numbers are average for each property.

Type	# of Properties	Verification Results per API			
		Avg. Line Coverage	Avg. Functional Coverage	Avg. Time (s)	Avg. Memory (MB)
Safety	36	62.34%	63.56%	5.64	244.1
Liveness	9	78.07%	79.82%	7.05	305.5
Reachability	56	74.09%	75.37%	6.08	457.8
Concurrency	6	54.62%	48.39%	6.34	312.6
Confidentiality	8	66.26%	67.69%	6.02	361.5
Integrity	7	67.56%	68.57%	5.54	322.7

C. Property Checking Results

Table IV presents an overview of the property verification results. The first column shows the property type from the six templates. The second column indicates the number of properties generated within each property type for TDX module. The third and fourth columns reveal the average line coverage and functional coverage achieved during the verification. The fifth and sixth columns provide insights into the computational resources required during verification, with the average time and memory consumption using the CBMC bounded model checker [14]. These results collectively demonstrate the effectiveness of the generated properties and their ability to assess various aspects of the Intel TDX implementation.

We consistently achieve an average line coverage of around 70%, indicating that a significant portion of the code has been effectively examined. This level of coverage not only ensures that critical code paths are analyzed but also provides a high degree of confidence in the correctness of the API's implementation. Furthermore, the functional coverage, which measures how well the functionality of the API has been explored, also attains an average of approximately 70%. This demonstrates that our properties have the ability to delve deep into the API's behavior. Both the average verification time and memory consumption are notably low.

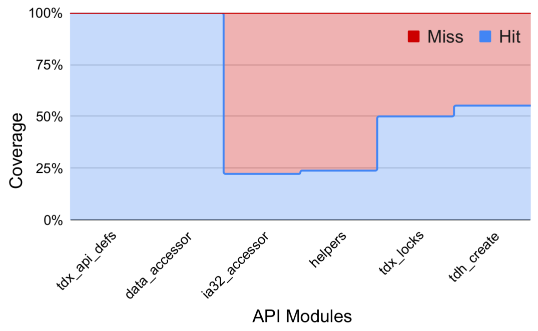

Fig. 5: Total line coverage for different API modules for the assertion shown in Listing 1.

The activation of an assertion not only contributes to the coverage of the specific API within which the assertion is embedded but can also serve to verify other modules associated with the functionality. In Figure 5, we present the coverage results for various modules in the TDX implementation resulting from the activation of the assertion depicted in Listing 1. The assertion activation triggered activity in six different modules. The figure illustrates two key metrics: the percentage of lines that were activated (hits), shown in blue, and the percentage of lines that remained untouched despite the assertion activation (miss), indicated in red. The results highlight the effectiveness of the assertion activation, achieving an average line coverage of 58% across the six modules. This highlights the effectiveness of the generated properties in comprehensive testing of TEE implementations.

V. Conclusion

TEEs safeguard sensitive data and critical operations, making their security verification essential. However, verifying TEEs is challenging due to the complex interactions between their subsystems. Existing TEE verification approaches rely on manual generation of security properties. In this paper, we proposed a framework for automated property generation as well as efficient verification of TEE implementations. We derived local FSMs from the TEE specification and composed the local FSMs to construct a global FSM representing the behaviour of TEEs. The properties are automatically generated from the global FSMs based on templates representing various scenarios, including safety, liveness, reachability, concurrency, confidentiality, and integrity. Our experimental results using Intel TDX architecture demonstrate that our framework can achieve high line and functional coverage with low time and memory requirements.

References

[1] R. Sinha *et al.*, "Moat: Verifying confidentiality of enclave programs," in *ACM Conference on Computer and Communications Security*, 2015.
[2] P. Subramanyan *et al.*, "A formal foundation for secure remote execution of enclaves," in *CCS*, 2017, pp. 2435–2450.
[3] Y. Ma *et al.*, "Formal verification of memory isolation for the trustzone-based tee," in *Asia-Pacific Software Engineering Conference*, 2020.
[4] H. Sun and H. Lei, "A design and verification methodology for a trustzone trusted execution environment," *IEEE Access*, vol. 8, 2020.
[5] M. Sardar, S. Musaev, and C. Fetzer, "Demystifying attestation in intel trust domain extensions via formal verification," *IEEE access*, 2021.
[6] H. Witharana *et al.*, "Formal verification of virtualization-based trusted execution environments," *IEEE Trans. on CAD*, 2024.
[7] ——, "A survey on assertion-based hardware verification," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–33, 2022.
[8] ——, "Automated generation of security assertions for rtl models," *ACM JETC*, vol. 19, no. 1, pp. 1–27, 2023.
[9] ——, "Directed test generation for activation of security assertions in rtl models," *ACM TODAES*, vol. 26, no. 4, pp. 1–28, 2021.
[10] "Intel TDX Security Review," https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf.
[11] "AMD SEV," https://storage.googleapis.com/gweb-uniblog-publish-prod/documents/AMD_GPZ-Technical_Report_FINAL_05_2022.pdf.
[12] D. C. G. Valadares *et al.*, "Formal verification of a trusted execution environment-based architecture for iot applications," *IEEE IoT*, 2021.
[13] "Intel TDX," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.
[14] D. Kroening and M. Tautschnig, "Cbmc–c bounded model checker," in *TACAS*, 2014, pp. 389–391.