

A Case Study in Firmware Verification: Applying Formal Methods to Intel[®] TDX Module

Dirk Beyer¹ , Po-Chun Chien¹ , Bo-Yuan Huang² ,

Nian-Ze Lee^{3,1} , and Thomas Lemberger¹ 

¹ LMU Munich, Munich, Germany

² Intel INT31, USA

³ National Taiwan University, Taipei, Taiwan

Abstract. Firmware underpins system security but remains challenging to verify due to hardware dependency, specialized coding idioms, and limited open-source examples. Manual verification approaches, while common in industry, are labor-intensive and difficult to scale. This paper presents a detailed case study on applying automatic formal methods for software to a security-critical firmware component in Intel[®] Trust Domain Extensions (TDX), known as TDX Module. In this study, we employ six state-of-the-art C-program analyzers on the production TDX Module firmware, leveraging techniques ranging from bounded model checking and symbolic execution to abstract interpretation. Our empirical evaluation identifies obstacles unique to firmware, highlights harness-design decisions essential for verifying industry-scale code bases, and demonstrates opportunities in advanced slicing for more scalable verification. Although the case study focuses on TDX Module, the findings are broadly applicable to large-scale, low-level programs and have already influenced the software-verification community, such as standardizing nondeterministic object initialization. All verification tasks and proof harnesses are publicly released to foster reproducible research and future tool development.

Keywords: Firmware verification, Software verification, Industrial verification methodology, Trusted execution environment, Intel TDX

1 Introduction

Firmware lies at the foundation of modern computing platforms, orchestrating the boot process, hardware initialization, and security enforcement. Operating at the lowest layer of the stack in privileged and opaque environments, any defect in firmware can compromise the trustworthiness of the entire system. Despite its critical role, firmware verification in practice remains largely manual, relying on code reviews and dynamic testing through handcrafted test suites or fuzzing. For instance, Microsoft [37] and Google [3] recently conducted security reviews of Intel[®] Trust Domain Extensions (TDX) [1], which provides a hardware-based

An appendix to this article is available on our supplementary webpage [11].

isolation mechanism to protect *data in use* at the virtual-machine (VM) level. Such expensive and laborious reviews highlight the pressing need for automated, rigorous formal reasoning for firmware.

1.1 Problem Statement: Applying Software Verification to Firmware

We aim to transform today’s largely manual firmware-verification process into a systematic, tool-supported workflow. Since firmware is mostly written in C [44], recent advances and standardization in verification techniques and tools for C programs [17] provide a strong basis for this effort. This case study examines *how to apply automatic software-verification techniques to firmware to efficiently and effectively check specification conformance and detect vulnerabilities*.

While many success stories on applying automatic C-program verifiers to industry-scale code bases have been reported [7, 16, 22, 23, 47], and several projects worked towards streamlining the transformation from code projects to verification tasks [4, 14, 29, 47, 51, 59, 65, 74, 76], their application to firmware remains scarce. We discuss the key challenges of firmware verification and opportunities offered by formal methods in the following:

Hardware Dependency. Arguably the most referenced challenge in analyzing firmware programs, hardware dependency breaks the levels of abstraction that automatic tools often depend on. Dynamic testing tools, for example, cannot be deployed until late in the development cycle after the hardware has been fabricated. Operational virtual prototypes and formal abstraction models help left-shift the process [35, 41, 42, 61, 64, 73], but they incur non-trivial engineering costs. Planning and assessing such costs is a key part of firmware verification to ensure that early modeling efforts align with the system’s design considerations.

Negative Space. Security-critical firmware, such as TDX Module, must be robust against adversarial inputs and unexpected scenarios. Beyond verifying correct behavior under expected conditions, the *positive space*, security assurance requires rigorous input validation and error handling in adversarial or unforeseen cases, known as the *negative space*. While the positive space is typically addressed through manually crafted test cases during development, the negative space is harder to cover due to the difficulty to enumerate and the sheer size of the space. Compared to fuzzing or dynamic testing, the symbolic nature of formal verification provides the exhaustive rigor demanded by security-critical firmware, motivating our investigation into its application to firmware analysis.

Stateful Analysis. The stateful nature of firmware adds another layer of complexity to its verification. The behavior of a firmware component may depend not only on its inputs but also on the firmware and hardware states. This state-dependent behavior remains an open challenge for automatic testing techniques, which often require user-defined transition models, a suite of predefined input sequences, or design-specific code instrumentation [5, 54]. In contrast, formal methods enable modular verification of individual procedures by symbolically initializing and constraining the state variables accessed by the procedure under verification. No additional setup sequences or user-defined transition models are required.

1.2 Intel Trust Domain Extensions (TDX)

Intel TDX [1] is Intel’s latest confidential computing technology [62, 68] that provides an isolation mechanism to protect *data in use* at the virtual-machine (VM) level, in addition to the process-level isolation provided by Intel SGX [28, 57]. It is a hardware-based trusted execution environment (TEE) [33, 45, 63, 67] that facilitates the deployment of hardware-isolated VMs, called *Trust Domains* (TDs), and provides memory and CPU-state confidentiality and integrity as well as address-translation integrity. Moreover, it isolates TDs from the host virtual-machine manager (VMM), hypervisor, and other non-TD software to further reduce the trusted computing base of a TD.

Intel TDX introduces several new architectural elements, including TDX Module, a security service module running in a new, most privileged, CPU execution mode called Secure Arbitration Mode (SEAM). TDX Module enables controlled interactions between the host VMM and TDs through a set of application binary interface (ABI)

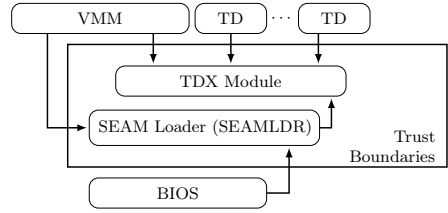


Fig. 1: Trust boundaries of TDX Module

functions. For example, host-side interface functions allow the host VMM to initialize TDX Module, configure keys, and create and manage TDs. Meanwhile, guest-side interface functions allow a TD to access platform metadata, create attestation-measurement reports, request host VMM services, etc. Besides supporting the core operations, more importantly, TDX Module is responsible for enforcing the security policies of the platform. Even more so, the ABI presents a prime attack surface as it receives untrusted inputs from outside of the trust boundaries, as shown in Fig. 1. This makes verifying TDX Module essential for the overall security assurance of Intel TDX.

The challenges outlined in Sect. 1.1, hardware dependency, negative space, and stateful analysis, are inherent in TDX Module: For example, hardware dependency can be seen in the use of the extended instruction set for measurement and remote attestation, which require appropriate modeling. Meanwhile, as TDX Module’s interface functions receive untrusted inputs from outside the trust boundary, this necessitates rigorous validation to cover the negative space. Nonetheless, the execution semantics of these functions are stateful, conditioned on the values of registers and control structures at the time of execution.

1.3 Overview of the Adopted Methodology

To enable the application of off-the-shelf software analyzers to industrial firmware, we adopt a common verification methodology [23] and tailor it to address the unique challenges of firmware. Our methodology comprises three key techniques, each targeting one of the challenges identified in Sect. 1.1.

To handle hardware dependency, we employ a manual CEGAR-like [24] approach for abstracting the underlying hardware: Firmware-hardware interactions (e.g., memory-mapped I/O and SEAM operations) are first coarsely overapprox-

mated based on their signatures, returning nondeterministic values and havocking⁴ mutable arguments. These overapproximations are refined on demand only when they cause false alarms, minimizing modeling effort while maintaining soundness.

To enable effective negative-space verification, we develop systematic proof-harness engineering practices that reflect industrial verification workflows [23]. Each verification task is automatically extracted from unmodified production code using a lightweight proof harness that (i) sets up the execution context through lazy symbolic initialization, (ii) encodes preconditions capturing the negative-space scenario of interest, and (iii) specifies postconditions as verification targets. Harness components are compositional and allow for automatic task creation, enabling systematic management and reuse across verification tasks.

With symbolically initialized state variables, we define modular, type-based constraints that are reusable invariants over firmware data structures. Rather than exhaustively enumerating all possible system states, these constraints characterize valid states declaratively by type, enabling type-based initialization that instantiates and constrains only the state variables accessed by the function under verification. This compositional approach [23] provides modular verification of individual interface functions.

Together, these three techniques transform the firmware-verification problem into a form tractable for existing software-verification tools while maintaining the rigor required for security-critical systems.

1.4 Case Study: Insights from Intel TDX Module Verification

We validate this verification methodology through a detailed case study of Intel TDX Module, developing hardware abstractions, proof harnesses, and verification properties for 22 representative interface functions to enable the application of software-verification techniques implemented in off-the-shelf C analyzers. In total, 418 verification tasks are derived from the unmodified TDX Module code base. Each task is a single-file, self-contained C program, automatically extracted using HARNESSFORGE,⁵ a tool that performs syntactic program slicing on complex code bases and keeps only relevant components for the given target functions. The total human effort for this case study was approximately ten to twelve person-months.

Evaluation with six state-of-the-art tools (CBMC [25], ESBMC [32], KLEE [20], CPACHECKER [15], MOPSA [46], and UAUTOMIZER [39]) reveals fundamental insights across three dimensions. First, design decisions in proof-harness engineering critically impact effectiveness, with lazy initialization improving success rate and tool-specific havocking outperforming generic methods. These findings have motivated the adoption of a standardized mechanism for object havocking in SV-COMP 2026.⁶ Second, the capabilities of off-the-shelf software verifiers remain limited for firmware: Only 59% of the tasks were solved by any of the evaluated tools. Carefully inspecting the unsolved tasks indicates that *property-guided slicing* (details in Sect. 6) presents a key direction for improvement by considering

⁴ In program analysis, “havoc” means assigning a nondeterministic value to a variable.

⁵ <https://gitlab.com/sosy-lab/software/harnessforge>

⁶ https://gitlab.com/sosy-lab/sv-comp/bench-defs/-/merge_requests/544

the semantics of the property under verification to slice the code base. Such optimization is particularly effective for negative-space verification of security-critical programs like TDX Module. Third, firmware-specific characteristics such as nested type punning and compiler memory-layout attributes break software verifiers’ frontends and hinder applicability. In our evaluation, half of the tools failed on all tasks largely due to these patterns.

Our study highlights the importance of viewing verification workloads as a combination of the target-under-verification and proof harness, emphasizing the need for type-based modularization and tool standardization to enhance verification engineering and drive technical innovations.

Contributions. In the following, we summarize the contributions of our paper:

1. We adapt an established methodology for code-level model checking and tailor it to firmware by integrating hardware abstractions and modular, type-based property and constraint development. This approach streamlines the application of formal methods to industry-scale firmware (details in [Sect. 3](#)).
2. We validate the adapted methodology on Intel TDX Module, focusing on negative-space verification of its interface functions. The study develops formal abstractions for architectural extensions relied upon by TDX, engineers proof harnesses that emulate industrial verification workloads, and derives type-based constraints from the official specification [1] (details in [Sect. 4](#)).
3. Using the developed infrastructure, we automatically extract 418 SV-COMP-compliant [17] single-file verification tasks from the production TDX Module code. Six state-of-the-art analyzers featuring representative techniques, including symbolic execution, bounded model checking, and abstract interpretation, are applied, which constitutes the first large-scale assessment of automatic formal software verification on commodity firmware (details in [Sect. 5](#)).
4. We analyze how firmware-specific characteristics, property types, and harness-design decisions affect verification performance. The study identifies persistent obstacles and distills practical insights to guide future industrial and academic verification efforts (details in [Sect. 6](#)).
5. The developed hardware abstractions, properties, proof harnesses, and the derived verification tasks are publicly released to support reproducible research and foster future development in this critical domain.

2 Related Work

Our study builds on previous work on formal verification of low-level programs, including device drivers, boot loaders, and embedded firmware [8, 42, 48, 50, 61, 64, 66]. The formal modeling of hardware abstraction is especially underpinned by the foundational studies in interfacing and reasoning about software-hardware interactions [35, 40, 41, 43, 56, 73]. We first review previous efforts in formalizing and verifying TEE architecture designs, and then discuss the related work on verifying the security monitor implementation in other TEE platforms.

High-Level TEE Formalization and Verification. The research community has been formalizing TEE systems at the design level to prove functional

correctness and key security properties [36, 55, 58, 69, 72, 75, 77]. On the commodity TEE side, Intel developed DVF, iPave, and Accordion that provide the languages and tools to formally model and verify Intel SGX [31, 34, 52]. Also, addressing VM-based TEE, recent work has formally modeled the attestation protocol of Intel TDX and proved the confidentiality of the shared secret and authentication of reports [70, 71].

This existing body of research in formalization and verification of hardware-assisted TEE is essential in ensuring a correct and secure design. However, the actual implementations of TEE can still have security bugs even if their high-level models have been formally verified. We emphasize the importance of directly verifying the actual implementation; for the same reason, our study considers properties that are low-level and implementation-oriented, in contrast to the higher-level properties discussed in the literature.

TEE Security Monitor Verification. Compared to design-level formalization, few works target implementation-level verification of TEEs. The Verification Infrastructure for Armv9-A (VIA) verifies an early prototype of the Realm Management Monitor (RMM) within the Arm Confidential Compute Architecture [53]. The approach is based on the interactive theorem prover Coq [2, 27], and thus, while highly expressive, requires significant manual efforts. In contrast, automatic tools CBMC and ESBMC have been applied to the reference implementation of RMM [30, 79], where the RMM implementation is automatically instrumented with pre- and postconditions derived from a machine-readable specification.

Our case study builds on learnings from these two success stories. While they rely on certain tool-specific customizations in their workflows, we focus on generating tool-agnostic verification tasks that comply with widely adopted conventions in the SV-COMP [17] and Test-Comp [9] communities. This allows leveraging the advances in these communities and unlocks verification techniques beyond bounded model checking.

3 Methodology for Automated Firmware Verification

We adapt the established methodology for code-level model checking that reflects industry best practices, similar to the approach developed at Amazon Web Services for verifying cloud software [23]. Given a function under verification, this methodology introduces a proof harness that initializes data structures used by the function symbolically, assumes preconditions over the data structures declaratively with type-based constraints, calls into the function, and asserts postconditions after the function returns. The following subsections detail key components we introduce to tailor this methodology for firmware verification.

3.1 Hardware Abstraction

Firmware interacts directly with hardware through low-level mechanisms such as inline assembly and memory-mapped I/O. Capturing these interactions, i.e., modeling the states and semantics of the operations, is essential for the complete verification of firmware. Industrial firmware code bases usually use dedicated wrap-

per functions to interface with hardware-dependent components. This modular design allows clean override of these functions with abstraction models.

Regarding inline assembly, uses of standard instructions usually have well-defined semantics and can be directly mapped to C implementations. For architectural extensions, we begin with a coarse overapproximation based on the signature of a wrapper function. The abstraction returns a nondeterministic value and updates all mutable inputs with nondeterministic values as well. We refine the abstraction only when it causes false alarms in verification results, similar to the CEGAR approach [24].

Firmware programs access reserved memory regions that map to hardware registers, data initialized by earlier stages of the boot process, etc. Efficiently handling these regions is crucial for firmware verification and often requires specialized tool support. For example, CBMC [25] provides a hook for memory-mapped I/O [26]. In our case study, we introduce mock variables to represent the states of the reserved memory regions accessed by TDX Module.

3.2 Proof-Harness Engineering

To enable effective negative-space and stateful verification of firmware, we develop a proof harness to create the verification context for each target function under verification. The proof harness contains a setup procedure that prepares the verification context by allocating and initializing the necessary state variables and input arguments. During this step, assumptions are given to relevant input arguments and state variables for capturing the negative space of the function and reflecting the stateful nature of firmware. On the other hand, we develop a corresponding teardown procedure that deallocates the state variables after the verification task is finished. Below we describe the crucial considerations for the setup and teardown procedures.

Allocation and Deallocation. While some state variables are defined as global variables in the code, others—such as control structures initialized by earlier booting stages—are opaque to the program. For these, we explicitly allocate memory in the setup procedure and deallocate during teardown.

To keep the proof harness lightweight and verifier-friendly, we allocate only the state variables accessible by the function under verification based on the specification. This *lazy* approach not only simplifies the verification context but also helps to identify potential mismatches between the implementation and the specification in terms of the accessible states.

Symbolic Variable Havocking. To cover the entire state space, the setup procedure symbolically initializes and havoc state variables and input arguments. Our proof harnesses support three styles for symbolically havocking variables with non-primitive, complex data structures.

1. *Memory havocking.* Each byte of the variable is assigned a nondeterministic value of type `unsigned char`.
2. *Object havocking.* Based on the type definition, boilerplate code is automatically synthesized to havoc the variable. Fields of primitive types are assigned

nondeterministic values of the corresponding type, while fields of non-primitive types are recursively traversed till primitive types are reached.

3. *Tool-specific havocking*. Besides the above two generic approaches, our proof harnesses can be configured to use tool-specific primitives for symbolic havocking, such as the method `__CPROVER_havoc_object()` in CBMC [25].

Note that if a non-primitive type includes a pointer field, assigning a nondeterministic value to the pointer does not yield a valid pointed-to object. Therefore, we need to allocate and initialize the pointed-to object and assign its address to the corresponding pointer field.

Hoare-Style Preconditions and Postconditions. After symbolically initializing the state and input variables, Hoare-style preconditions and postconditions specific to the property under verification are derived from the specification and encoded in the proof harness. While table-like data structures often need conditions over individual elements, which can naturally be expressed with universal quantification, we use loops to iterate over elements in the proof harnesses as off-the-shelf verifiers typically do not support quantifiers.

3.3 Type-Based Constraints

Specifying constraints over variables is one of the most labor-intensive aspects of developing a proof harness. For reusability and composability, we define these constraints in a type-based and declarative manner [23]. They fall into two main categories: (1) *value conditions*, such as requiring the state variable of a finite-state machine to belong to a predefined set or an address to lie within the physical-address range; and (2) *structural conditions*, such as ensuring no overlap between memory regions recorded in a metadata structure. These type-based constraints play a crucial role in defining inductive invariants of the system state, which are assumed in the setup procedure and checked in the teardown procedure for their inductiveness.

While type-based constraints capture a significant portion of the required conditions, some constraints cannot be expressed in a type-based manner. This is especially prominent for conditions that involve relationships between multiple variables. We specify such inter-variable relationships separately in the proof harnesses.

3.4 Automatic Verification-Task Generation

Fully automatic software verifiers for C programs typically assume self-contained, single-file verification tasks with all necessary data structures and function definitions explicitly included [17, 19]. This limits direct application of these verifiers to industrial code bases. Specifically, extracting the relevant components and assembling them into verification tasks for off-the-shelf tools is non-trivial and error-prone without systematic methodology.

We use the tool `HARNESFORGE` to assemble a standalone verification task from the proof harnesses and the firmware code base under verification. `HARNESFORGE` takes as input a YAML configuration file that defines a verification task with (1) a sequence of method calls and (2) any overrides of original project source code.

This enables automatic generation of verification tasks by specifying the sequence of calls to harness components and the hardware abstractions as overrides.

From this configuration, `HARNESSFORGE` computes the relevant project sources, applies the source-code overrides, and creates a verification entry that calls the specified methods in the given order. Moreover, `HARNESSFORGE` performs a dependency analysis based on the method call tree to remove all methods, global variables, and type definitions that are irrelevant to the verification task.

4 Design of Case Study on TDX Module

We apply the methodology described in [Sect. 3](#) to TDX Module [1], a security-critical, open-source firmware component of the Intel TDX technology. We focus on TDX Module’s ABI functions, which receive untrusted inputs from callers outside the trust boundaries and thus warrant comprehensive negative-space verification. Our goal is to investigate how formal software verifiers can be leveraged with the adapted verification methodology to achieve high coverage and assurance for these security-critical interface functions.

This section describes the design of the case study, including the developed hardware abstractions, the selection of target interface functions, and the derivation of verification properties.

4.1 Abstraction of TDX Module’s Hardware Dependencies

The hardware dependencies of TDX Module fall into three categories:

1. *Executing standard x86 instructions.* TDX Module uses inline assembly to execute standard `x86` instructions in scenarios such as reimplementing standard library functions (cf. [Sect. 6](#)), ensuring atomic operations, or handling wide bit-width data types that would otherwise be inefficient in C. While specific to the hardware platform, these instructions are normally design-agnostic.
2. *Architectural extensions.* As part of its core function, TDX Module manages the underlying hardware and leverages its computational capabilities by accessing model-specific registers and utilizing extended instruction sets. For example, to support measurement and attestation, it invokes the newly introduced instructions `SEAMDB_REPORT` and `SEAMVERIFYREPORT` to generate and verify reports, respectively. These interactions tend to be highly design-specific and require domain expertise for precise modeling.
3. *Accessing reserved or memory-mapped regions.* Data accesses in firmware are not always explicitly defined in its source code. For example, memory-mapped regions may not be declared by the firmware program, yet their accesses are valid and managed by the memory controller. In TDX Module, such “undefined” data accesses occur when accessing data initialized by `SEAMLDR`, the bootloader module in Intel TDX, which determines the physical addresses of TDX Module code and data based on the BIOS configuration and updates the corresponding dedicated registers.

The TDX Module code base uses dedicated wrapper functions to interface with hardware-dependent components, allowing clean override of these functions with

abstraction models. As described in Sect. 3.1, we translate standard x86 instructions to shadow C implementations directly. For the design-specific extensions leveraged by TDX Module, we overapproximate their behaviors based on the signatures of the wrapper functions by returning nondeterministic values of the expected return types and havocking all mutable inputs. For reserved or memory-mapped regions, mock variables are introduced to represent their states; constraints on these variables are added as needed to reflect the verification context.

In this case study, we observe that most of the first-attempt signature-based abstractions are *sufficient* for proving our selected properties and do not require further refinements.

4.2 Selection of Target Interface Functions

TDX Module’s ABI includes 77 host-side and 31 guest-side interface functions. All functions are security-critical and warrant comprehensive verification; in this case study, we focus on a representative subset to investigate the effectiveness of the adapted verification methodology. The selection is guided by the following criteria and objectives:

- **Interface-function class:** TDX Module ABI functions are grouped into classes based on their functionalities, such as physical-memory management, migration supports, and measurement and attestation. Each class performs distinct operations, managing different parts of the system state and following a unique program flow. To ensure broad coverage, our case study includes interface functions from each class.
- **Module and TD lifecycle:** TDX Module enforces platform security policies by ensuring that the lifecycles of both the module and TDs comply with the defined protocols. These protocol state machines guide and track these lifecycles through stages such as initialization, operation, and shutdown. Different sets of interface functions are permitted at different lifecycle stages. Our case study covers functions permitted in each stage and includes all functions that can trigger lifecycle-stage transitions.

Using the above criteria, we include 16 host-side and 6 guest-side interface functions into the case study (cf. Table 2).

4.3 Selection and Derivation of Verification Properties

Security assurance is a multi-faceted effort. For complex systems like Intel TDX, it requires a combination of verification techniques applied at various abstraction levels and targeting different classes of properties. This case study focuses on a subset of properties that (1) can be verified modularly at the individual interface-function level and (2) are practically tractable using off-the-shelf software analyzers. This results in safety properties, which address *functional correctness* and *state integrity*, as well as reachability properties, which are introduced to cover different execution paths within a function.

For additional developer-defined properties, such as debug assertions and sanity checks embedded in the TDX Module source code, we added preprocessor

Table 1: Excerpted specification of the interface function `TDH.MNG.KEY.CONFIG`, showing its input and output operands, access information on state operands, and possible completion-status codes

Operand	Description
Input	
RAX	SEAMCALL instruction leaf number and version
RXC	The physical address of a TDR page (HKID bits must be 0.)
Output	
RAX	SEAMCALL instruction return code
Other registers	Unmodified
State Information	
Trust Domain Root (TDR) page	Explicitly accessed, Read/Write permission, 4KB alignment
Key Encryption Tables (KETs)	Implicitly accessed
Completion-Status Codes (Excerpted)	
<code>TDX_OPERAND_INVALID</code>	An input operand of the interface function is invalid.
<code>TDX_LIFECYCLE_STATE_INCORRECT</code>	System is in an incorrect lifecycle state for this interface function.
<code>TDX_KEY_GENERATION_FAILED</code>	CPU failed to generate a random key.

directives to distinguish them from the above properties. They are omitted from the evaluation in [Sect. 5](#) due to space constraints. Other classes of properties, e.g., liveness, noninterference, and information flow, which arise in sequences of interface-function invocations or concurrent interaction between multiple TDs, are important for comprehensive guarantees of data confidentiality and integrity, but left for future work as they do not enjoy the same level of tool readiness.

We derive the properties based on the TDX Module ABI specification [1], applying best-effort interpretation in cases of ambiguity or missing details. As an example, [Table 1](#) provides an excerpt of the specification for the interface function `TDH.MNG.KEY.CONFIG`, which is used to configure a TD’s private key. For each interface function, the specification defines the input and output operands, along with their expected conditions and values. In addition to register-based input and output operands, it documents the firmware and hardware states that the interface function may access or modify. It also lists the possible completion-status codes, accompanied by descriptions of the corresponding failure conditions.

Functional-Correctness Properties. TDX Module ABI functions receive untrusted inputs from callers outside the trust boundaries. As a result, they perform *precondition checks* and *error handling* before executing their core logic. In this case study, we derive functional properties that capture such behavior, with a focus on the negative space, i.e., cases where preconditions are violated. Specifically, the properties verify that the interface functions detect invalid inputs or states and respond accordingly, such as clearing sensitive data and returning the designated error code. For each interface function, we enumerate detailed preconditions based on the available information in the specification [1]. An example proof harness for invalid input handling in `TDH.MNG.KEY.CONFIG` is shown in [Appendix A](#) [11].

State-Integrity Properties. As shown in [Table 1](#), the specification defines the set of registers and states that an interface function may access or modify. We include properties asserting the integrity of registers that should not be modified by a given interface function. Integrity properties for states with complex data structures are excluded, as no available tools were found capable of handling them.

Reachability Properties. Based on the specification, we derive reachability properties to check that the target interface function can indeed complete with different status codes. These codes represent various precondition-check failures or convey distinct outcomes upon successful completion. The reachability properties guide the analysis of different execution paths within the interface function.

Besides the above properties derived from the specification, we also systematically generate *cover tasks* by inserting reachability labels at various control points within a task. Specifically, we insert labels at the completion of (1) the preconditions, (2) the target function call, and (3) the teardown procedure. Cover tasks provide an approximate measure of how effectively a software analyzer explores the verification context. Inserting them is a common industrial practice since fully converged proofs (i.e., properties under verification proved or disproved) are not always attainable, and cover tasks provide evidence of analysis progress, analogous to coverage metrics in testing.

5 Results and Effects of Harness-Engineering Decisions

This section presents the outcomes of applying our verification methodology to TDX Module and an analysis of how different initialization and havocking strategies in the proof-harness design influenced the effectiveness of the evaluated software analyzers. We discuss the key challenges encountered and the underlying reasons for the limited effectiveness of current verification tools.

5.1 Verification Setup

Automatic Analyzers. We leveraged six state-of-the-art software analyzers for C programs, encompassing different classes of verification techniques. The tools include the industrial-strength bounded model checkers CBMC (version 6.6.0) [25, 49] and ESBMC (version 7.7.0) [32, 78], the symbolic-execution engine KLEE (version 3.1) [20, 21], and three top-ranking participants from SV-COMP 2025 [17]: CPACHECKER (version 4.0) [6, 15] and MOPSA (version 1.0) [46, 60], the first- and second-place winner in category *SoftwareSystems*, respectively; and UAUTOMIZER (version 0.3.0-d790fccc) [38, 39], the overall competition winner.

Among these tools, CBMC, ESBMC, and KLEE offer built-in primitives for symbolic initialization of complex data structures, which we leveraged for tool-specific havocking. We executed each tool with the configuration used in its most recent competition participation. For CBMC and KLEE, we adapted their latest competition wrapper scripts to ensure compatibility with their current releases.

Verification Tasks. We derived the verification tasks from version 1.5.05 of TDX Module using HARNESFORGE version 1.1. These include 22 TDX Module ABI functions (6 guest-side (TDG) and 16 host-side (TDH) interface functions), listed in [Table 2](#), resulting in 418 verification tasks.

Table 2: Results for each interface function under different havocking strategies

	Interface function	#tasks	Mem. havoc.	Obj. havoc.	Tool-spec. havoc.
Guest-side TDG	MR.REPORT	10	0	0	5
	SERVTD.WR	10	0	0	7
	SYS.RD	10	0	0	8
	VM.WR	17	0	0	14
	VP.ENTER	18	0	0	12
	VP.VMCALL	10	0	0	8
	EXPORT.RESTORE	10	3	0	7
Host-side TDH	IMPORT.ABORT	9	2	0	7
	MNG.ADDCX	33	5	2	11
	MNG.CREATE	18	3	0	17
	MNG.INIT	33	5	0	11
	MNG.KEY.CONFIG	18	4	0	16
	MNG.KEY.FREEID	17	2	0	16
	MNG.VPFLUSHDONE	21	4	0	19
	MR.FINALIZE	29	3	0	7
	PHYMEM.PAGE.RECLAIM	25	0	0	5
	SYS.CONFIG	29	0	0	8
	SYS.INIT	17	0	0	13
	SYS.KEY.CONFIG	13	10	0	13
	SYS.SHUTDOWN	17	0	0	13
	SYS.UPDATE	25	0	0	19
	VP.ENTER	29	0	0	9
Overall		418	41	2	245

Execution Environment. All experiments were conducted on machines running Ubuntu 24.04 (64 bit) with Linux kernel version 6.8.0. Each machine was equipped with an Intel Xeon E3-1230 v5 CPU (3.4 GHz, 8 logical cores) and 33 GB of RAM. Each verification task was allocated 2 CPU cores, a CPU time limit of 60 min, and a memory limit of 30 GB.

5.2 Verification Results

Table 2 shows the numbers of verification tasks derived from each of the selected TDX Module interface functions and the numbers of tasks correctly solved by any of the tools, under different havocking strategies. Overall, tool-specific havocking outperformed generic byte-wise memory havocking and field-wise object havocking, making about 59% of the generated tasks correctly solved. We further conducted several targeted analyses to understand the effects of the harness-engineering decisions on verification performance.

Havocking Strategies. Tool-specific havocking not only improved effectiveness but also the resource efficiency in terms of CPU time and memory usage. One key reason for the inefficiency of generic havocking is its reliance on potentially large loops. In contrast, tool-specific primitives allow verifiers to havoc the entire object internally in their memory model, avoiding the need to explicitly encode such loops in the tasks. Our results highlight the importance of supporting native havocking primitives in both verification-task encoding and verifier development. Notably, this insight has motivated the adoption of a standardized mechanism for symbolically havocking non-primitive data types in SV-COMP 2026.

State-Variable Constraints. About 30% of the cover points at function entry (cf. Sect. 4.3) were unreachable by any tools. All these tasks involve preconditions that constrain the elements in the physical-address metadata table. Currently, such constraints are implemented by iterating through each element in a loop, a method that tends to be ineffective for the evaluated tools. While these element-wise constraints can be more naturally and efficiently expressed using universal quantifiers, such constructs are not commonly supported by fully automatic verification tools.

5.3 Limited Tool Performance

Despite adapting a well-established industry verification workflow (cf. Sect. 3) and exploring key harness-engineering decisions to improve performance, the overall verification success rate remains limited. In our case study, while KLEE and CBMC delivered most of the successful results (237 and 117, respectively), CPACHECKER, UAUTOMIZER, and MOPSA were unable to solve any verification tasks. Many of the failed analyses stemmed from frontend parsing (identified tool issues summarized in Appendix B [11]). For example, UAUTOMIZER failed on every task due to its incomplete support for nested, anonymous `union` types, a prevalent pattern in firmware for type-punning and memory reinterpretation (more details in Sect. 6). These frontend issues prevented the tools from starting the actual verification process, indicating that frontend robustness is a fundamental limitation in applying off-the-shelf verifiers to firmware programs, despite the sophistication of their underlying algorithms.

CPACHECKER similarly failed to parse 141 tasks, primarily because of anonymous `union` members within `struct` types. For the remaining tasks, CPACHECKER encountered internal errors or timeouts. Upon investigation, we found that these errors were caused by the lack of support for a C extension that allows casting a scalar type to a `union` type,⁷ leading to the failure of 60 object-havocking and 27 memory-havocking tasks.

Although MOPSA’s frontend was able to parse all verification tasks, it ran into timeouts on all task variants using object havocking. For task variants using memory havocking, MOPSA managed to finish its analysis on 76 tasks but was unable to construct concrete execution paths, and hence returned `unknown` as the results.

ESBMC encountered segmentation faults or other internal errors on 264 tasks using its own havocking primitive. Most failures occurred during the encoding of verification conditions, while a few arose at an earlier program-unrolling stage.

Driven by the limited performance of the evaluated tools, we manually investigated the verification tasks and tool outputs to identify firmware-specific code patterns that hinder effective analysis and envision potential improvements to scalable verification, as explained in Sect. 6.

Comparisons to Results at SV-COMP and Test-Comp 2026. The 418 verification tasks derived from TDX Module have been integrated into the 2026 editions of SV-COMP and Test-Comp. Readers interested in the performance of a wider array of tools on these tasks are referred to the official competition

⁷ <https://gcc.gnu.org/onlinedocs/gcc/Cast-to-Union.html>

reports [10, 18]. Our evaluation differs from the competition setups in three aspects: (1) Computing resources: We provided higher computing resources (Sect. 5.1) per task than the competitions. (2) Tool versions: We used SV-COMP 2025 versions for CPACHECKER, MOPSA, UAUTOMIZER, and ESBMC; for CBMC and KLEE, which are not active competition participants, we used more recent versions and updated their wrapper scripts. (3) Havocking strategies: The tasks submitted to the competitions utilize `__VERIFIER_nondet_memory()`, the newly introduced primitive for object havocking inspired by this case study. This evaluation did not include tasks using the primitive as the assessed tools had not yet supported it.

6 Bottlenecks and Opportunities

Prior research on firmware verification has primarily focused on modeling software-hardware interactions. However, as shown in our experimental results in Sect. 5, even with available hardware abstractions, applying off-the-shelf software analyzers to firmware remains far from a straightforward, push-button process. While firmware shares certain similarities with higher-level C programs, its unique characteristics often pose significant challenges on existing tools. In this section, we uncover these distinguishing features and discuss how they hinder the effective use of off-the-shelf tools. We also explore potential directions for developing new approaches tailored to the specific demands of firmware verification.

Extensive Use of Nested Type Punning. Type punning allows the reinterpretation of a data object’s memory representation as a different type. In C programs, it is usually achieved through pointer-type castings or `union` types, which allow multiple fields of different types to occupy the same memory region. It enables programmers to write data as one type and read them later via another, and vice versa.

Firmware interacts directly with hardware, writing input arguments to registers for hardware to consume and retrieving output after the operations complete. Unlike virtualized environments, only a limited, finite number of registers are available on hardware to be shared across different operations, making type punning an essential way to efficiently interface between different operational contexts in firmware (see the Appendix C.1 [11] for an example).

We found that off-the-shelf software analyzers often do not have a robust frontend to handle deeply nested, and potentially anonymous, uses of `union`. To make it worse, pointer-type casting and `union` are often used interchangeably in firmware. This can lead to misinterpretations of data structures, causing quiet incorrect verification results or tool crashes.

Explicit and Static Memory Management. Precise control over the exact memory layout of data structures is imperative for firmware. Such control is essential to ensure that variables stored in registers and data payloads in memory can be correctly processed by hardware. This is more than consistent and correct interpretation within the program level. To achieve this goal, firmware commonly uses compiler attributes to explicitly define the memory layouts, particularly for data structures that interface directly with hardware. For example, firmware uses

the attribute `aligned(n)` to specify a minimum alignment of `n` bytes, and the attribute `__packed__` to ensure no padding is added between fields. An example is provided in [Appendix C.2 \[11\]](#).

We found that off-the-shelf software analyzers often lack support for compiler attributes related to precise memory-layout control. In particular, we observed that CBMC, CPACHECKER, and UAUTOMIZER do not correctly handle these constraints (see [Appendix B \[11\]](#) for more information). This can lead to incorrect interpretation of program behavior, particularly in cases where memory accesses rely on base addresses and offsets computed under the assumption of a correct memory layout.

On the other hand, firmware’s static memory allocation also provides an opportunity for more efficient analysis. Tools that recognize such a no-heap assumption could simplify their memory models, for example, by eliminating alias analysis for heap pointers. However, we did not yet observe tools benefiting from this characteristic in our study. Before reaching a stage where memory-model simplifications would matter, the evaluated tools encountered more fundamental obstacles, such as parsing failures on nested union types and encoding errors on complex data structures, as discussed above.

Standard Libraries and System Calls. Firmware operates below the operating system and is often subject to stricter size constraints due to limited embedded memory. Unlike higher-level C programs, it lacks access to standard libraries and system calls. As a result, rather than relying on external dependencies, firmware often reimplements some selected subset of these functionalities internally.

While firmware avoids issues related to external dependencies, its reimplementation introduces an emerging obstacle: State-of-the-art software analyzers typically develop their own formal-friendly stubs for common operations in standard libraries. However, firmware’s custom reimplementations prevent these tools from identifying such functions, limiting their ability to apply built-in optimizations. Furthermore, such reimplementations may require additional effort to mock as they often include platform-specific instructions written in inline assembly. A reimplementation of the function `memcpy` in TDX Module is shown in [Appendix C.3 \[11\]](#).

To address this emerging challenge, we advocate for a standardized annotation scheme to label the reimplementations of commonly used standard-library functions, such as `memcpy`, `memset`, `memcmp`, and string functions. Such annotations would enable verification tools to reliably recognize reimplemented functions in firmware (and higher-level C programs in general) as *semantically equivalent* to their standard-library counterparts, and thus facilitate the substitution of built-in, formal-friendly models of the tools without parsing the inline assembly.

Property-Directed Slicing. Most slicing techniques implemented in state-of-the-art tools consider a given verification task homogeneously, i.e., treating preconditions (assumptions) equally as the control flow of the target function without leveraging it for slicing. Our case study highlights the opportunity for *property-directed slicing* through a clearer structure of the proof-harness design.

Each verification task consists of five components in sequence: (1) setup, (2) preconditions, (3) target function, (4) postconditions, and (5) teardown.

While components (1) and (5) depend on the target function (3), components (2) and (4) are specific to the property of each verification task. The property-under-verification, e.g., correct error-handling under invalid input arguments, can significantly influence the program paths exercised within (3), and thus the state variables required for setup and teardown in (1) and (5), respectively.

In our case study, we found property-directed slicing especially effective for negative-space verification tasks, where precondition violations lead to early returns from the target function. For example, applying this technique to an input-validation task of a guest-side interface function `TDG.VP.VMCALL` turns the task unsolvable within an hour into one solved in seconds. This orders-of-magnitude speedup demonstrates that incorporating property semantics into program slicing—beyond the syntactic dependency analysis performed by existing tools—is a promising direction for more scalable verification.

Moreover, in our case study, we often observed that many fields in a large, complex data structure are irrelevant to the property under verification, but explicitly havocking the entire object includes the irrelevant fields in the analysis, thus increasing the verification complexity unnecessarily. Therefore, finer-grained, *field-level slicing* that initializes only the accessed fields in a large, complex data structures is another important direction for future work.

7 Conclusion

Firmware underpins system security, yet its verification remains largely manual. The absence of open-source examples and tool support has long hindered the systematic use of formal methods at this layer. In this case study, we investigated how mature techniques for automatic software verification can be transferred to firmware verification. Our methodology emphasizes a clear separation between the target function and the auxiliary proof harness that constructs the verification context, together with modular, type-based management of verification artifacts. This structure enables automated extraction of verification tasks suitable for off-the-shelf analyzers and facilitates the reuse of harness components.

Our study over 418 verification tasks from 22 interface functions reveals current limitations and future opportunities: Frontend fragility remains a bottleneck in applying software analyzers to firmware. Tool-specific havocking primitives significantly improve performance, motivating SV-COMP 2026’s adoption of a standardized havocking mechanism. We further identified the potential of property-directed slicing to achieve orders-of-magnitude speedup on negative-space properties. By releasing our verification tasks, hardware abstractions, and proof harnesses, we provide realistic firmware verification challenges that enable reproducible research and future tool development.

Data-Availability Statement. The source code of TDX module, the developed proof harnesses, and the derived verification tasks are archived on Zenodo [12]. An artifact for reproducing the results presented in Sect. 5 is also available on Zenodo [13]. In addition, a supplementary webpage is hosted at <https://www.sosy-lab.org/research/tdx-module-firmware-verification>.

Funding Statement. This project was funded by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY) and 536040111 (Bridge), and a research gift from Intel.

References

1. Intel Trust Domain Extensions. <https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html>, accessed: 2025-07-08
2. The Coq proof assistant, version 8.16.0 (2022). <https://doi.org/10.5281/zenodo.7313584>
3. Aktas, E., Cohen, C., Eads, J., Forshaw, J., Wilhelm, F.: Intel Trust Domain Extensions (TDX) security review. Tech. rep., Google Project Zero (2023). https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf
4. Alglave, J., Donaldson, A.F., Kröning, D., Tautschnig, M.: Making software verification tools really work. In: Proc. ATVA. pp. 28–42. LNCS 6996, Springer (2011). https://doi.org/10.1007/978-3-642-24372-1_3
5. Ba, J., Böhme, M., Mirzamomen, Z., Roychoudhury, A.: Stateful greybox fuzzing. In: Proc. USENIX Security. pp. 3255–3272. USENIX Association (2022). <https://www.usenix.org/conference/usenixsecurity22/presentation/ba>
6. Baier, D., Beyer, D., Chien, P.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Spiessl, M., Wachowitz, H., Wendler, P.: CPACHECKER 2.3 with strategy selection (competition contribution). In: Proc. TACAS (3). pp. 359–364. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_21
7. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Commun. ACM **54**(7), 68–76 (2011). <https://doi.org/10.1145/1965724.1965743>
8. Ball, T., Rajamani, S.K.: The SLAM project: Debugging system software via static analysis. In: Proc. POPL. pp. 1–3. ACM (2002). <https://doi.org/10.1145/503272.503274>
9. Beyer, D.: Advances in automatic software testing: Test-Comp 2025. In: Proc. FASE. pp. 257–274. LNCS 15693, Springer (2025). https://doi.org/10.1007/978-3-031-90900-9_13
10. Beyer, D.: Evaluating tools for automatic software testing: Test-Comp 2026. In: Proc. FASE. LNCS 16504, Springer (2026)
11. Beyer, D., Chien, P.C., Huang, B.Y., Lee, N.Z., Lemberger, T.: A case study in firmware verification: Applying formal methods to Intel® TDX Module (Appendix) (2026). <https://www.sosy-lab.org/research/tdx-module-firmware-verification/tacas26-appendix.pdf>
12. Beyer, D., Chien, P.C., Huang, B., Lee, N.Z., Lemberger, T.: The Intel TDX Module benchmark set. Zenodo (2025). <https://doi.org/10.5281/zenodo.16547223>
13. Beyer, D., Chien, P.C., Huang, B., Lee, N.Z., Lemberger, T.: Reproduction package for TACAS2026 article ‘A case study in firmware verification: Applying formal methods to Intel TDX Module’. Zenodo (2026). <https://doi.org/10.5281/zenodo.18371342>
14. Beyer, D., Grunske, L., Kettl, M., Rosenfeld, M.L., Raselimo, M.: P3: A dataset of partial program patches. In: Proc. MSR. pp. 123–127. ACM (2024). <https://doi.org/10.1145/3643991.3644889>
15. Beyer, D., Keremoglu, M.E.: CPACHECKER: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16

16. Beyer, D., Petrenko, A.K.: Linux driver verification. In: Proc. ISoLA. pp. 1–6. LNCS 7610, Springer (2012). https://doi.org/10.1007/978-3-642-34032-1_1
17. Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS (3). pp. 151–186. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_9
18. Beyer, D., Strejček, J.: Evaluating software verifiers for C, Java, and SV-LIB (report on SV-COMP 2026). In: Proc. TACAS (2). LNCS 16506, Springer (2026)
19. Beyer, D., Strejček, J.: SV-Benchmarks: Benchmark set for software verification (SV-COMP 2025). Zenodo (2025). <https://doi.org/10.5281/zenodo.15012096>
20. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI. pp. 209–224. USENIX Association (2008). <https://dl.acm.org/doi/10.5555/1855741.1855756>
21. Cadar, C., Nowack, M.: KLEE symbolic execution engine in 2019 (competition contribution). Int. J. Softw. Tools Technol. Transf. **23**(6), 867 – 870 (December 2021). <https://doi.org/10.1007/s10009-020-00570-3>
22. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O’Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM. pp. 3–11. LNCS 9058, Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1
23. Chong, N., Cook, B., Eidelman, J., Kallas, K., Khazem, K., Monteiro, F.R., Schwartz-Narbonne, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Code-level model checking in the software development workflow at Amazon Web Services. Softw. Pract. Exp. **51**(4), 772–797 (2021). <https://doi.org/10.1002/spe.2949>
24. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). <https://doi.org/10.1145/876638.876643>
25. Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS. pp. 168–176. LNCS 2988, Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
26. Cook, B., Khazem, K., Kröning, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Model checking boot code from AWS data centers. Formal Methods Syst. Des. **57**(1), 34–52 (2021). <https://doi.org/10.1007/S10703-020-00344-2>
27. Coquand, T., Huet, G.P.: The calculus of constructions. Inf. Comput. **76**(2/3), 95–120 (1988). [https://doi.org/10.1016/0890-5401\(88\)90005-3](https://doi.org/10.1016/0890-5401(88)90005-3)
28. Costan, V., Devadas, S.: Intel SGX explained. IACR Cryptol. ePrint Arch. pp. 86:1–86:118 (2016). <http://eprint.iacr.org/2016/086>
29. Fink, X., Berger, P., Katoen, J.P.: Configurable benchmarks for C model checkers. In: Proc. NFM. pp. 338–354. LNCS 13260, Springer (2022). https://doi.org/10.1007/978-3-031-06773-0_18
30. Fox, A.C.J., Stockwell, G., Xiong, S., Becker, H., Mulligan, D.P., Petri, G., Chong, N.: A verification methodology for the Arm® Confidential Computing Architecture: From a secure specification to safe implementations. Proc. ACM Program. Lang. **7**(OOPSLA1), 376–405 (2023). <https://doi.org/10.1145/3586040>
31. Fraer, R., Keren, D., Khasidashvili, Z., Novakovsky, A., Puder, A., Singerman, E., Talmor, E., Vardi, M.Y., Yang, J.: From visual to logical formalisms for SoC validation. In: Proc. MEMOCODE. pp. 165–174. IEEE (2014). <https://doi.org/10.1109/MEMCOD.2014.6961855>
32. Gadelha, M.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: ESBMC 5.0: An industrial-strength C model checker. In: Proc. ASE. pp. 888–891. ACM (2018). <https://doi.org/10.1145/3238147.3240481>

33. Geppert, T., Deml, S., Sturzenegger, D., Ebert, N.: Trusted execution environments: Applications and organizational challenges. *Frontiers Comput. Sci.* **4**, 930741:1–930741:6 (2022). <https://doi.org/10.3389/FCOMP.2022.930741>
34. Goel, A., Krstic, S., Leslie, R., Tuttle, M.R.: SMT-based system verification with DVF. In: *Proc. SMT. EPIc Series in Computing*, vol. 20, pp. 32–43. EasyChair (2012). <https://doi.org/10.29007/59RN>
35. Große, D., Kühne, U., Drechsler, R.: HW/SW co-verification of embedded systems using bounded model checking. In: *Proc. GLSVLSI*. pp. 43–48. ACM (2006). <https://doi.org/10.1145/1127908.1127920>
36. Guo, Y., Wang, Z., Zhong, B., Zeng, Q.: Formal modeling and security analysis for intra-level privilege separation. In: *Proc. ACSAC*. pp. 88–101. ACM (2022). <https://doi.org/10.1145/3564625.3567984>
37. Hania, B., Villard, M., Netzer, Y., Kodalapura, N., Nguyen, T.: Technical report of joint security review by Microsoft and Intel on Intel TDX1.5. Tech. rep., Intel Corporation and Microsoft Corporation (August 2024). https://www.intel.com/content/dam/www/public/us/en/security-advisory/documents/intel_tdx_joint_security_review_with_microsoft.pdf
38. Heizmann, M., Bentele, M., Dietsch, D., Jiang, X., Klumpp, D., Schüssele, F., Podelski, A.: ULTIMATE AUTOMIZER and the abstraction of bitwise operations (competition contribution). In: *Proc. TACAS* (3). pp. 418–423. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_31
39. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: *Proc. CAV*. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
40. Herdt, V., Große, D., Drechsler, R.: Enhanced Virtual Prototyping: Featuring RISC-V Case Studies. Springer (2021). <https://doi.org/10.1007/978-3-030-54828-5>
41. Horn, A., Tautschnig, M., Val, C., Liang, L., Melham, T., Grundy, J., Kroening, D.: Formal co-validation of low-level hardware/software interfaces. In: *Proc. FMCAD*. pp. 121–128. IEEE (2013). <https://doi.org/10.1109/FMCAD.2013.6679400>
42. Huang, B.Y., Ray, S., Gupta, A., Fung, J.M., Malik, S.: Formal security verification of concurrent firmware in SoCs using instruction-level abstraction for hardware. In: *Proc. DAC*. pp. 91:1–91:6. ACM (2018). <https://doi.org/10.1145/3195970.3196055>
43. Huang, B.Y., Zhang, H., Subramanyan, P., Vizel, Y., Gupta, A., Malik, S.: Instruction-level abstraction (ILA): A uniform specification for system-on-chip (SoC) verification. *ACM Trans. Des. Autom. Electron. Syst.* **24**(1), 10:1–10:24 (2018). <https://doi.org/10.1145/3282444>
44. ISO/IEC JTC 1/SC 22: ISO/IEC 9899-2018: Information technology — Programming Languages — C. International Organization for Standardization (2018), <https://www.iso.org/standard/74528.html>
45. Jauernig, P., Sadeghi, A., Stapf, E.: Trusted execution environments: Properties, applications, and challenges. *IEEE Secur. Priv.* **18**(2), 56–60 (2020). <https://doi.org/10.1109/MSEC.2019.2947124>
46. Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: *Proc. VSTTE*. pp. 1–18. LNCS 12031, Springer (2019). https://doi.org/10.1007/978-3-030-41600-3_1
47. Khoroshilov, A.V., Mutilin, V.S., Petrenko, A.K., Zakharov, V.: Establishing Linux driver verification process. In: *Proc. Ershov Memorial Conference*. pp. 165–176. LNCS 5947, Springer (2009). https://doi.org/10.1007/978-3-642-11486-1_14
48. Krstić, S., Yang, J., Palmer, D.W., Osborne, R.B., Talmor, E.: Security of SoC firmware load protocols. In: *Proc. HOST*. pp. 70–75. IEEE (2014). <https://doi.org/10.1109/HST.2014.6855571>

49. Kröning, D., Tautschnig, M.: CBMC: C bounded model checker (competition contribution). In: Proc. TACAS. pp. 389–391. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_26
50. Lal, A., Qadeer, S.: Powering the static driver verifier using Corral. In: Proc. FSE. pp. 202–212. ACM (2014). <https://doi.org/10.1145/2635868.2635894>
51. Lee, H., Kim, S., Cha, S.K.: Fuzzle: Making a puzzle for fuzzers. In: Proc. ASE. ACM (2022). <https://doi.org/10.1145/3551349.3556908>
52. Leslie-Hurd, R., Caspi, D., Fernandez, M.: Verifying linearizability of Intel® software guard extensions. In: Proc. CAV. pp. 144–160. LNCS 9207, Springer (2015). https://doi.org/10.1007/978-3-319-21668-3_9
53. Li, X., Li, X., Dall, C., Gu, R., Nieh, J., Sait, Y., Stockwell, G.: Design and verification of the Arm Confidential Compute Architecture. In: Proc. OSDI. pp. 465–484. USENIX Association (2022). <https://www.usenix.org/conference/osdi22/presentation/li>
54. Ma, R., Wang, D., Hu, C., Ji, W., Xue, J.: Test data generation for stateful network protocol fuzzing using a rule-based state machine. *Tsinghua Science and Technology* **21**(3), 352–360 (2016). <https://doi.org/10.1109/TST.2016.7488746>
55. Ma, Y., Zhang, Q., Zhao, S., Wang, G., Li, X., Shi, Z.: Formal verification of memory isolation for the TrustZone-based TEE. In: Proc. APSEC. pp. 149–158. IEEE (2020). <https://doi.org/10.1109/APSEC51365.2020.00023>
56. Malik, S., Subramanyan, P.: Invited - Specification and modeling for systems-on-chip security verification. In: Proc. DAC. pp. 66:1–66:6. ACM (2016). <https://doi.org/10.1145/2897937.2911991>
57. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative instructions and software model for isolated execution. In: Proc. HASP. ACM (2013). <https://doi.org/10.1145/2487726.2488368>
58. Miao, X., Chang, R., Zhao, J., Zhao, Y., Cao, S., Wei, T., Jiang, L., Ren, K.: CVTEE: A compatible verified TEE architecture with enhanced security. *IEEE Transactions on Dependable and Secure Computing* **20**(1), 377–391 (2023). <https://doi.org/10.1109/TDSC.2021.3133576>
59. Moloney, C., Dyer, R., Sherman, E.: Demonstrating ARG-V’s generation of realistic Java benchmarks for SV-COMP. In: Proc. TACAS. LNCS 16506, Springer (2026)
60. Monat, R., Ouadjaout, A., Miné, A.: MOPSA-C with trace partitioning and autosuggestions (competition contribution). In: Proc. TACAS (3). pp. 229–235. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_17
61. Mukherjee, R., Purandare, M., Polig, R., Kroening, D.: Formal techniques for effective co-verification of hardware/software co-designs. In: Proc. DAC. pp. 35:1–35:6. ACM (2017). <https://doi.org/10.1145/3061639.3062253>
62. Mulligan, D.P., Petri, G., Spinale, N., Stockwell, G., Vincent, H.J.M.: Confidential computing - A brave new world. In: Proc. SEED. pp. 132–138. IEEE (2021). <https://doi.org/10.1109/SEED51797.2021.00025>
63. Muñoz, A., Rios, R., Román, R., López, J.: A survey on the (in)security of trusted execution environments. *Comput. Secur.* **129**, 103180:1–103180:26 (2023). <https://doi.org/10.1016/J.COSE.2023.103180>
64. Nguyen, M.D., Wedler, M., Stoffel, D., Kunz, W.: Formal hardware/software co-verification by interval property checking with abstraction. In: Proc. DAC. pp. 510–515. ACM (2011). <https://doi.org/10.1145/2024724.2024843>
65. Novikov, E., Zakharov, I.S.: Towards automated static verification of GNU C programs. In: Proc. PSI. pp. 402–416. LNCS 10742, Springer (2017). https://doi.org/10.1007/978-3-319-74313-4_30

66. Ray, S., Ghosh, N., Masti, R.J., Kanuparthi, A., Fung, J.M.: Formal verification of security critical hardware-firmware interactions in commercial SoCs. In: Proc. DAC. pp. 43:1–43:4. ACM (2019). <https://doi.org/10.1145/3316781.3323478>
67. Sabt, M., Achemlal, M., Bouabdallah, A.: Trusted execution environment: What it is, and what it is not. In: Proc. TrustCom. pp. 57–64. IEEE (2015). <https://doi.org/10.1109/TRUSTCOM.2015.357>
68. Sardar, M.U., Fetzer, C.: Confidential computing and related technologies: A critical review. *Cybersecur.* **6**(1), 10:1–10:7 (2023). <https://doi.org/10.1186/S42400-023-00144-1>
69. Sardar, M.U., Faqeh, R., Fetzer, C.: Formal foundations for Intel SGX data center attestation primitives. In: Proc. ICFEM. pp. 268–283. Springer (2020). https://doi.org/10.1007/978-3-030-63406-3_16
70. Sardar, M.U., Fossati, T., Frost, S., Xiong, S.: Formal specification and verification of architecturally-defined attestation mechanisms in Arm CCA and Intel TDX. *IEEE Access* **12**, 361–381 (2023). <https://doi.org/10.1109/ACCESS.2023.3346501>
71. Sardar, M.U., Musaev, S., Fetzer, C.: Demystifying attestation in Intel Trust Domain Extensions via formal verification. *IEEE Access* **9**, 83067–83079 (2021). <https://doi.org/10.1109/ACCESS.2021.3087421>
72. Sardar, M.U., Quoc, D.L., Fetzer, C.: Towards formalization of enhanced privacy ID (EPID)-based remote attestation in Intel SGX. In: Proc. DSD. pp. 604–607. IEEE (2020). <https://doi.org/10.1109/DSD51259.2020.00099>
73. Schmidt, B., Villarraga, C., Bormann, J., Stoffel, D., Wedler, M., Kunz, W.: A computational model for SAT-based verification of hardware-dependent low-level embedded system software. In: Proc. ASP-DAC. pp. 711–716. IEEE (2013). <https://doi.org/10.1109/ASPDAC.2013.6509684>
74. Steffen, B., Isberner, M., Naujokat, S., Margaria, T., Geske, M.: Property-driven benchmark generation: Synthesizing programs of realistic structure. *Int. J. Softw. Tools Technol. Transfer* **this volume** (2014)
75. Subramanyan, P., Sinha, R., Lebedev, I., Devadas, S., Seshia, S.A.: A formal foundation for secure remote execution of enclaves. In: Proc. CCS. pp. 2435–2450. ACM (2017). <https://doi.org/10.1145/3133956.3134098>
76. Westhofen, L., Berger, P., Katoen, J.P.: Benchmarking software model checkers on automotive code. In: Proc. NFM. pp. 133–150. LNCS 12229, Springer (2020). https://doi.org/10.1007/978-3-030-55754-6_8
77. Wu, P., Shen, Q., Deng, R.H., Liu, X., Zhang, Y., Wu, Z.: OblidC: An SGX-based oblivious distributed computing framework with formal proof. In: Proc. Asia CCS. pp. 86–99. ACM (2019). <https://doi.org/10.1145/3321705.3329822>
78. Wu, T., Li, X., Manino, E., Menezes, R., Gadelha, M., Xiong, S., Tihanyi, N., Petoumenos, P., Cordeiro, L.: ESBMC v7.7: Efficient concurrent software verification with scheduling, incremental SMT and partial order reduction (competition contribution). In: Proc. TACAS (3). pp. 223–228. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_16
79. Wu, T., Xiong, S., Manino, E., Stockwell, G., Cordeiro, L.C.: Verifying components of Arm[®] Confidential Computing Architecture with ESBMC. In: Proc. SAS. pp. 451–462. LNCS 14995, Springer (2024). https://doi.org/10.1007/978-3-031-74776-2_18

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

