A Case Study in Firmware Verification

Applying Formal Methods to Intel TDX Module

Dirk Beyer[1], Po-Chun Chien[1], Bo-Yuan Huang[2]
Nian-Ze Lee[3,1], and Thomas Lemberger[1]





[1]LMU Munich · [2]Intel INT31 · [3]National Taiwan University

TACAS 2026-04-13

SoSy-Lab Software Systems

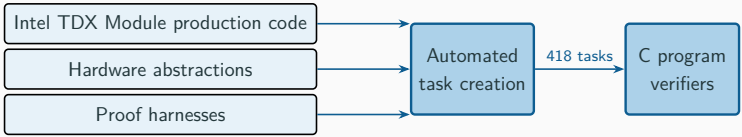
LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

INT31

This Talk

Applying automatic verifiers for C programs to industry-scale firmware

Extend a code-level model-checking methodology [1] with **hardware abstractions**



Evaluating representative verifiers (e.g., CBMC, KLEE) covering diverse techniques

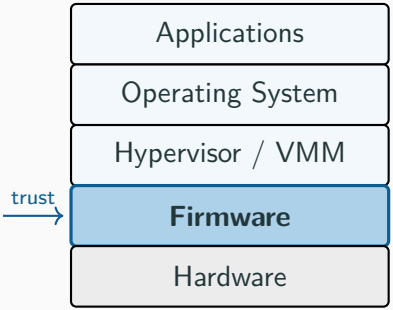
Only **59 %** tasks solved by *any* of the evaluated tools

Project webpage

INT31 blog post

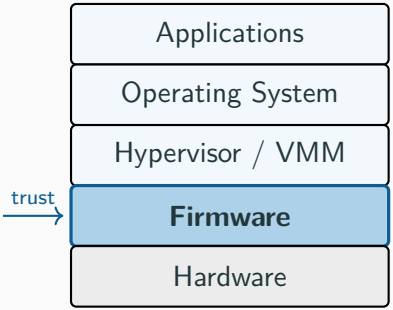
Why Firmware Verification?

Firmware: Foundation of System Security



- Orchestrates **boot**, **hardware initialization**, and **security enforcement**
- Operates at **lowest, most privileged** layer

Firmware: Foundation of System Security



- Orchestrates **boot**, **hardware initialization**, and **security enforcement**
- Operates at **lowest, most privileged** layer

The trustworthiness of all upper layers rests on firmware correctness.

How Industry Verifies Firmware Today

- **Manual code review** by security experts
- **Fuzzing** and handcrafted test suites

How Industry Verifies Firmware Today

- **Manual code review** by security experts
- **Fuzzing** and handcrafted test suites

Example: security reviews of **Intel TDX 1.5** by Google [2] and Microsoft [3]

- *Security Assessment of Intel TDX with Support for Live Migration*
  - Published in February 2026, five Google researchers over six months

How Industry Verifies Firmware Today

- **Manual code review** by security experts
- **Fuzzing** and handcrafted test suites

Example: security reviews of **Intel TDX 1.5** by Google [2] and Microsoft [3]

- *Security Assessment of Intel TDX with Support for Live Migration*
  - Published in February 2026, five Google researchers over six months

Effective but **labor-intensive** and **hard to scale** – and the *negative space* (adversarial, unforeseen inputs) is hard to cover by testing.

Automatic Verification for C Programs

- Firmware: mostly written in **C** (plus assembly)
- Mature ecosystem of automatic verifiers and success stories

  CBMC, ESBMC, KLEE, CPACHECKER, MOPSA, UAUTOMIZER, ... (cf. SV-COMP [4])
- Yet scarce application to firmware

Automatic Verification for C Programs

- Firmware: mostly written in **C** (plus assembly)
- Mature ecosystem of automatic verifiers and success stories

  CBMC, ESBMC, KLEE, CPACHECKER, MOPSA, UAUTOMIZER, ... (cf. SV-COMP [4])
- Yet scarce application to firmware

Research question: How can automatic verifiers for C programs be applied to **efficiently** and **effectively** check specification conformance and detect vulnerabilities in firmware?

Key Challenges and Why Formal Methods Fit?

- **Hardware Dependency**
  Firmware relies on specialized instructions to manage hardware resources

Key Challenges and Why Formal Methods Fit?

- **Hardware Dependency**
  Firmware relies on specialized instructions to manage hardware resources
  → Formal abstraction models (before hardware becomes available)

Key Challenges and Why Formal Methods Fit?

- **Hardware Dependency**
  Firmware relies on specialized instructions to manage hardware resources
  → Formal abstraction models (before hardware becomes available)
- **Negative Space**
  Must behave correctly under adversarial or unforeseen inputs

Key Challenges and Why Formal Methods Fit?

- **Hardware Dependency**
  Firmware relies on specialized instructions to manage hardware resources
  → Formal abstraction models (before hardware becomes available)
- **Negative Space**
  Must behave correctly under adversarial or unforeseen inputs
  → Exhaustive reasoning over input space

Key Challenges and Why Formal Methods Fit?

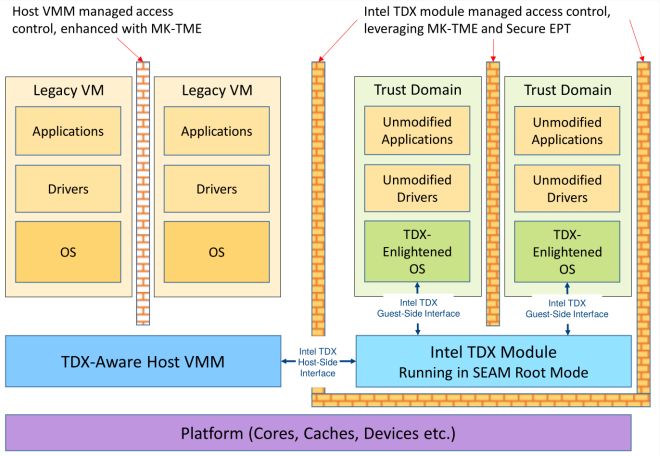
- **Hardware Dependency**
  Firmware relies on specialized instructions to manage hardware resources
  → Formal abstraction models (before hardware becomes available)
- **Negative Space**
  Must behave correctly under adversarial or unforeseen inputs
  → Exhaustive reasoning over input space
- **Stateful Analysis**
  Behaviors depend on firmware and hardware states

Key Challenges and Why Formal Methods Fit?

- **Hardware Dependency**
  Firmware relies on specialized instructions to manage hardware resources
  → Formal abstraction models (before hardware becomes available)
- **Negative Space**
  Must behave correctly under adversarial or unforeseen inputs
  → Exhaustive reasoning over input space
- **Stateful Analysis**
  Behaviors depend on firmware and hardware states
  → Symbolic initialization + constraining (for modular analysis)

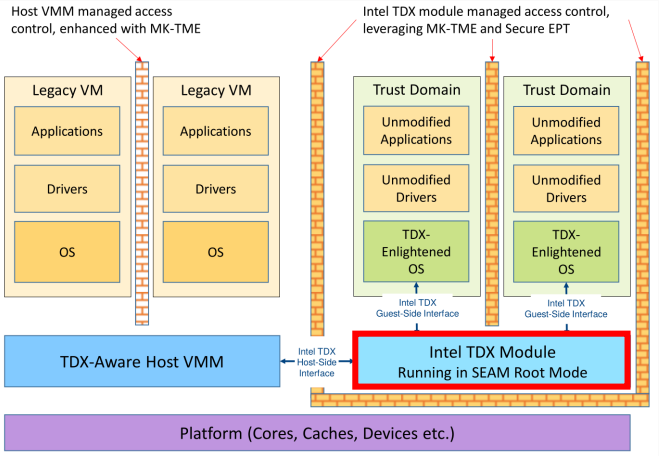
Designing the Case Study

Intel TDX Module: Hardware-Enforced VM-Level Isolation



Source: Fig. 2.1 in Intel TDX Module v1.5 Base Architecture Spec. [5]

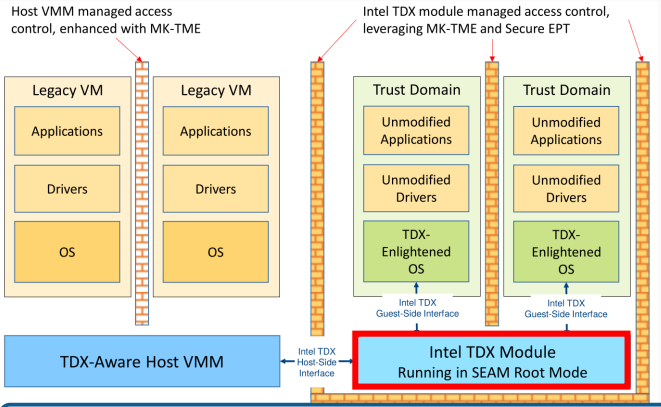
Intel TDX Module: Hardware-Enforced VM-Level Isolation



Source: Fig. 2.1 in Intel TDX Module v1.5 Base Architecture Spec. [5]

- Host: VMM
- Guest: TD
- Communicate via ABI to protect “data in use”

Intel TDX Module: Hardware-Enforced VM-Level Isolation



- Host: VMM
- Guest: TD
- Communicate via ABI to protect "data in use"

Goal: Verify the ABI of TDX Module (a prime attack surface), assuming VMM and TDs can call any interface function with arbitrary inputs.

Hardware Abstraction

Enable C verifiers to reason about firmware–hardware interactions

- **Standard x86 instructions** (e.g., `MOV`, `CMPXCHG`)
  → Precise modeling via **shadow C code**

Hardware Abstraction

Enable C verifiers to reason about firmware–hardware interactions

- **Standard x86 instructions** (e.g., `MOV`, `CMPXCHG`)
  → Precise modeling via **shadow C code**
- **Architectural extensions** (e.g., attestation: produce a measurement report of a TD)
  → Overapproximation based on function signature, refined via **CEGAR**

Hardware Abstraction

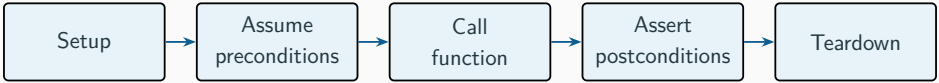
Enable C verifiers to reason about firmware–hardware interactions

- **Standard x86 instructions** (e.g., `MOV`, `CMPXCHG`)
  → Precise modeling via **shadow C code**
- **Architectural extensions** (e.g., attestation: produce a measurement report of a TD)
  → Overapproximation based on function signature, refined via **CEGAR**
- **Memory-mapped regions** (e.g., SEAMLDR-initialized module state)
  → Replace with **mock variables**

```
tdx_module_local_t *get_local_data(void) {
#ifdef TDXFV_NO_ASM
  return &local_data_fv; // mock
#else
  uint64_t local_data_addr;
  _ASM_("movq %%gs:%c[local_data], %0\n\t"
        : "=r"(local_data_addr)
        : [local_data] "i"(offsetof(tdx_module_local_t, local_data_fast_ref_ptr)));
  return (tdx_module_local_t *)local_data_addr;
#endif
}
```

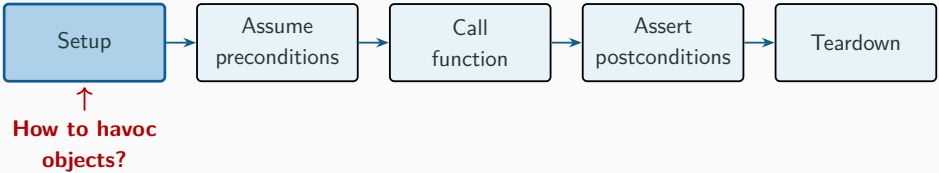
Proof-Harness Design

Five-step harness design for code-level model checking [1]



Proof-Harness Design

Five-step harness design for code-level model checking [1]

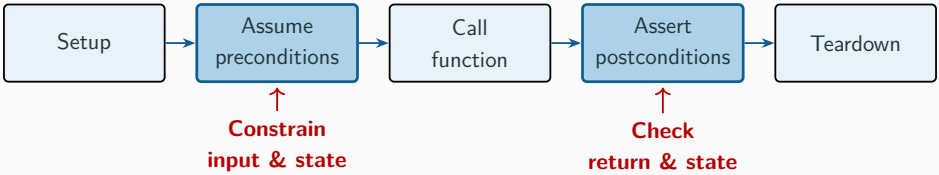


Havocking strategies for symbolic initialization:

- **Memory havocking**: byte by byte
- **Object havocking**: field by field
- **Tool-specific**: `__CPROVER_havoc_object`, `klee_make_symbolic`, ...

Proof-Harness Design

Five-step harness design for code-level model checking [1]



Verification properties (manually derived from TDX specification):

- **Functional correctness**: correct execution or error handling
- **State integrity**: registers not modified unexpectedly
- **Cover points**: inserted after precondition, function call, and teardown

Selection of Interface Functions

- Selection criteria:
  - At least one function from each functional class
  - At least one function permitted in each lifecycle stage
  - All functions that trigger lifecycle transitions

Selection of Interface Functions

- Selection criteria:
  - At least one function from each functional class
  - At least one function permitted in each lifecycle stage
  - All functions that trigger lifecycle transitions
- 16 of 77 host-side functions (`TDH.*`)
- 6 of 31 guest-side functions (`TDG.*`)

Study Results and Diagnosis

Evaluation Results

Evaluated verifiers:
CBMC [6], ESBMC [7], KLEE [8],
CPACHECKER [9], MOPSA [10], UAUTOMIZER [11]

Computing resources per task:

- 60 min CPU time
- 30 GB memory

Evaluation Results

Evaluated verifiers:
CBMC [6], ESBMC [7], KLEE [8],
CPACHECKER [9], MOPSA [10], UAUTOMIZER [11]

Computing resources per task:

- 60 min CPU time
- 30 GB memory

59 % of tasks encoded with tool-specific havocking primitives were solved.

	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
Host-side TDH	EXPORT.RESTORE	10	3	0	7
	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

Evaluation Results

Evaluated verifiers:
Cbmc [6], Esbmc [7], Klee [8],
CPAchecker [9], Mopsa [10], uAutomizer [11]

Computing resources per task:

- 60 min CPU time
- 30 GB memory

59 % of tasks encoded with tool-specific havocking primitives were solved.

	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
Host-side TDH	EXPORT.RESTORE	10	3	0	7
	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.RECLAI	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

Challenges for Evaluated Software Verifiers (1/4)

- Extensive use of nested type punning → frontend failures for several tools

```
union {
    struct {
        uint32_t field_code : 24;
        uint32_t reserved_0 : 8;
    }; // default field code
    struct {
        uint32_t element : 1;     // 0
        uint32_t subleaf : 7;     // 1-7
        uint32_t subleaf_na : 1; // 8
        uint32_t leaf : 7;        // 9-15
        uint32_t leaf_bit31 : 1; // 16
        uint32_t reserved : 15;  // 17-31
    } cpuid_field_code;
};
```

Challenges for Evaluated Software Verifiers (2/4)

- Initialization and constraining of complex structures
  - Tool-specific havocking primitives yield better results

Challenges for Evaluated Software Verifiers (2/4)

- Initialization and constraining of complex structures
  - Tool-specific havocking primitives yield better results
  - Loop-based constraints on arrays could overwhelm verifiers
    → need standardized annotations (e.g., universal quantifiers)

Challenges for Evaluated Software Verifiers (2/4)

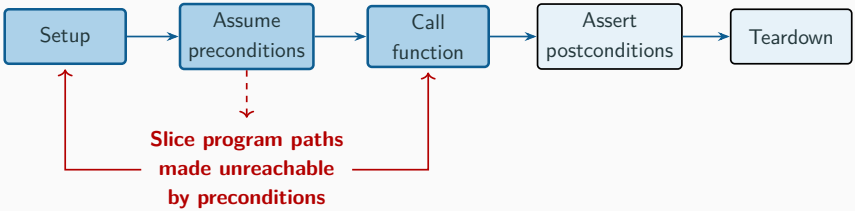
- Initialization and constraining of complex structures
  - Tool-specific havocking primitives yield better results
  - Loop-based constraints on arrays could overwhelm verifiers
    → need standardized annotations (e.g., universal quantifiers)

SV-COMP 2026 introduced a new primitive to havoc memory.

Challenges for Evaluated Software Verifiers (3/4)

Limited effectiveness for negative-space safety properties

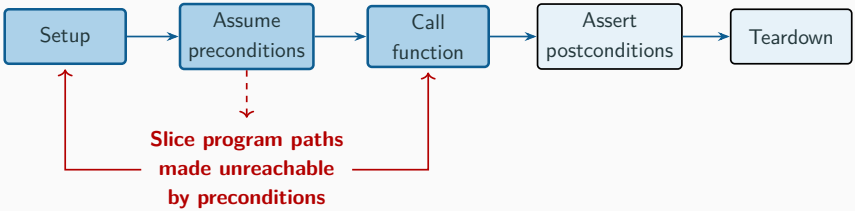
→ Require "smarter" slicing of unreachable logic and unused variables



Challenges for Evaluated Software Verifiers (3/4)

Limited effectiveness for negative-space safety properties

→ Require "smarter" slicing of unreachable logic and unused variables



Verification should go hand in hand with harness design!

Challenges for Evaluated Software Verifiers (4/4)

- Precise control over memory layout imperative for firmware
  - E.g., ensure sensitive data only written to hardware-protected memory range

Challenges for Evaluated Software Verifiers (4/4)

- Precise control over memory layout imperative for firmware
  - E.g., ensure sensitive data only written to hardware-protected memory range
  - Compiler attributes `aligned(n)`, `__packed__` common in firmware struct
  - Not widely supported by off-the-shelf verifiers

Achieved Impact So Far

- Found nine issues in Intel TDX Module and its specification
  all confirmed and fixed

Achieved Impact So Far

- Found nine issues in Intel TDX Module and its specification

  all confirmed and fixed

- Identified nine issues in total in four different software verifiers

  CBMC #8443 (incorrect handling of `__attribute__((packed))`)

  CPAchecker #818 (sound handling of `__attribute__(...)`)

Achieved Impact So Far

- Found nine issues in Intel TDX Module and its specification
  all confirmed and fixed
- Identified nine issues in total in four different software verifiers
  CBMC #8443 (incorrect handling of `__attribute__((packed))`)
  CPAchecker #818 (sound handling of `__attribute__(...)`)
- Produced a publicly available benchmark set for firmware verification
  part of SV-Benchmarks and used in SV-COMP 2026

Achieved Impact So Far

- Found nine issues in Intel TDX Module and its specification
  all confirmed and fixed
- Identified nine issues in total in four different software verifiers
  CBMC #8443 (incorrect handling of `__attribute__((packed))`)
  CPAchecker #818 (sound handling of `__attribute__(...)`)
- Produced a publicly available benchmark set for firmware verification
  part of SV-Benchmarks and used in SV-COMP 2026
- Motivated the addition of a new primitive for memory havocking in SV-COMP 2026 (`__VERIFIER_nondet_memory`)

Conclusion

A **case study**: six software verifiers applied to **TDX Module**

(22 interface functions, 418 tasks)

- Many verification tasks unsolved
  - Firmware-specific C patterns
  - Symbolic initialization/constraining of complex objects
  - Tools' lack of awareness of harness design
  - Memory layout not well understood by evaluated tools

Future work:

- Property-directed slicing
- Richer properties (e.g., multiple TDs, cross-function)
- LLM-assisted harness generation

Project webpage

INT31 blog post

References i

[1] 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). doi:10.1002/spe.2949

[2] 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

[3] 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

[4] Beyer, D., Strejček, J.: Evaluating software verifiers for C, Java, and SV-LIB (Report on SV-COMP 2026). In: Proc. TACAS (2). pp. 461–501. LNCS 16506, Springer (2026). doi:10.1007/978-3-032-22749-2_23

References ii

[5] Intel Trust Domain Extensions. https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html, accessed: 2025-07-08

[6] Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS. pp. 168–176. LNCS 2988, Springer (2004). doi:10.1007/978-3-540-24730-2_15

[7] 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). doi:10.1145/3238147.3240481

[8] 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

[9] Beyer, D., Keremoglu, M.E.: CPACHECKER: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). doi:10.1007/978-3-642-22110-1_16

References iii

[10] 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). doi:10.1007/978-3-030-41600-3_1

[11] Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. LNCS 8044, Springer (2013). doi:10.1007/978-3-642-39799-8_2

Excerpted Specification of TDH.MNG.KEY.CONFIG

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

Excerpted Specification of TDH.MNG.KEY.CONFIG

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

Functional correctness

Excerpted Specification of `TDH.MNG.KEY.CONFIG`

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

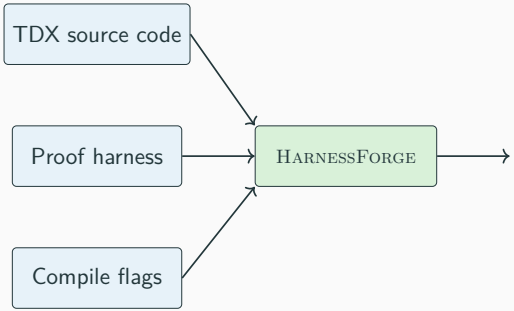
State integrity

Excerpted Specification of TDH.MNG.KEY.CONFIG

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

Reachability

Assembling of Verification Tasks




```
[...]

// auto-generated task entry point
void main() {
  hmkc__setup();
  hmkc__invalid_input_rcx__precond();
  hmkc__function_call();
  hmkc__invalid_input_rcx__postcond();
  hmkc__teardown();
}
```

single-file, self-contained

HarnessForge: to appear at FSE 2026