Intel TDX Module Benchmark Set for Firmware Verification

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

SV-COMP Workshop 2026-03-17

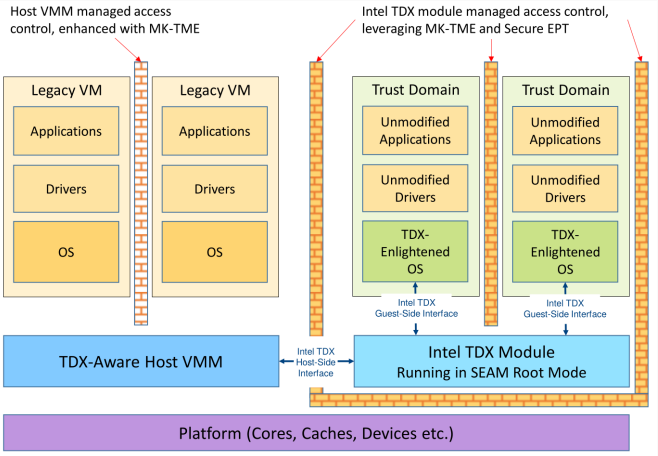
This work is supported by a research gift from Intel

Intel TDX Module Benchmark Tasks

- Own SoftwareSystems category
- Property: `unreach-call`
- Directory: `intel-tdx-module/`
- Derived from unmodified TDX Module source code
- + configurable proof harnesses
- Assembled automatically by HarnessForge

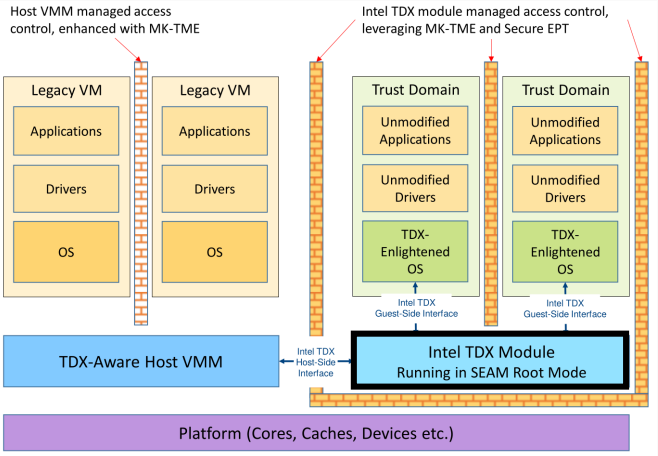
ETAPS 2026 Distinguished Paper Award: *Beyer, Chien, Huang, Lee, Lemberger: A Case Study in Firmware Verification. Applying Formal Methods to Intel TDX Module, TACAS 2026.*

Intel TDX: Overview



Source: Fig. 2.1 in Intel TDX Module v1.5 Base Architecture Spec. [1]

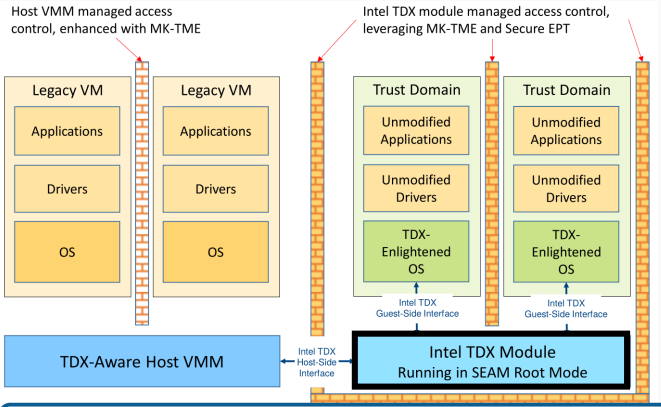
Intel TDX: Overview



Source: Fig. 2.1 in Intel TDX Module v1.5 Base Architecture Spec. [1]

- Host: VMM
- Guest: TD
- Communicate via application binary interfaces (ABIs)

Intel TDX: Overview



- Host: VMM
- Guest: TD
- Communicate via application binary interfaces (ABIs)

Goal: Verify ABIs of TDX module (implemented as C code plus assembly), assuming VMM and TDs can call any ABI with arbitrary inputs

Key Aspects in Verification of TDX Module

- **Hardware Dependency**
  FW relies on specialized instructions for managing HW resources
  → Formal model for hardware implemented in C
- **Positive + Negative Space**
  Ensure correct state after any valid and any invalid state
  → Properties for functional correctness
  → Different scenarios split into multiple tasks
- **Stateful Analysis**
  Behaviors of interface functions depend on FW & HW state
  → Symbolic initialization + preconditions

Structure of Tasks

- Verification tasks for valid input/state (`*_expected_*`)
- Verification tasks for specific invalid input/state (`*_invalid_X_*`)
- Verification tasks for any combination of invalid input/state (`*_unexpected_*`)

```
static inline bool_t all_conditions_valid() {
    return input_tdr_pa_is_valid() &&
        state_td_state_is_valid() &&
        state_lifecycle_is_valid();
}
void foobar__unexpected__precond() {
    tdh_mng_init__common_precond();
    TDXFV_ASSUME(!all_conditions_valid());
}
```

Verification Properties

- **Safety** (`*_safety_requirement*`)
  - **Functional correctness**: correct error handling
  - **State integrity**: registers not modified unexpectedly
- **Reachability**: different status codes are reachable (`*_success*` and `*_unsuccess*`)
- **Developer-defined**: debug assertions and sanity checks
- **Cover points**:
  - `reach_error()` after last precondition (`*_cover_entry*`)
  - `reach_error()` before first postcondition (`*_cover_exit*`)
  - `reach_error()` after last postcondition (`*_cover_proof*`)

Initialization Methods

- Element-wise initialization (`*_havoc_object*`)

```
void init_foo(foo_t *foo) {
    init_bar(foo->bar);
}
void init_bar(bar_t *bar) {
    bar->x = __VERIFIER_nondet_int();
    bar->y = __VERIFIER_nondet_char();
}
```

- Or initialization of memory region(`*_havoc_memory`)

Initialization Methods

- Element-wise initialization (`*_havoc_object*`)
- Or initialization of memory region(`*_havoc_memory`)

```
void init_foo(foo_t *foo) {
    __VERIFIER_nondet_memory(foo, sizeof(foo_t));
}
```

Evaluation Results

		Correct		Incorrect	
Tool	**Total**	**True**	**False**	**True**	**False**
CBMC	836	0 (+0)	0 (+118)	0	0
CPACHECKER	836	8 (+0)	55 (+ 83)	0	0
ESBMC	836	2 (+4)	0 (+ 29)	0	0

- __VERIFIER_nondet_memory: All of the above (except for 2 unconfirmed)
- BUBAAK: Frontend failed (uint128)
- GOBLINT: Timeout/Error(verify)
- SYMBIOTIC: OOM/Frontend failed (uint128)
- THETA: Frontend failed
- UAUTOMIZER: Frontend failed (field not in struct)

Challenges for Current Software Verifiers

- Extensive use of nested type punning

```
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 Current Software Verifiers

- Compiler attributes for memory-layout control

  (e.g., `__packed__` and `aligned(n)`)

```
typedef struct __attribute__((__packed__))
  td_report_s {
  report_mac_struct_t report_mac_struct;
  tee_tcb_info_t      tee_tcb_info;
  uint8_t             reserved[17];
  td_info_t           td_info;
} td_report_t;
_Static_assert(
  sizeof(td_report_t) == 1024,
  td_report_t
);
```

Challenges for Current Software Verifiers

- Path explosion (bounded, but long and complex control flow)
- Initialization and preconditions for complex structures
  - Initialization of large arrays (of defined size)
  - Tool-specific havocking primitives yield better results
- Limited effectiveness for negative-space safety properties
  → Require “smarter” slicing of unreachable core logic

Conclusion

- 2x 418 verification tasks
- Low-hanging fruits:
  - Support __VERIFIER_nondet_memory
  - Language support for uint128, nested types
- Show that your verifiers work on original source code
- We have strong access to domain knowledge

Project webpage

References i

[1] Intel Trust Domain Extensions, https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html, accessed: 2024-05-01