A Case Study in Firmware Verification: Applying Formal Methods to Intel® TDX Module (Appendix)

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

[1] LMU Munich, Munich, Germany
[2] Intel INT31, USA
[3] National Taiwan University, Taipei, Taiwan

A An Example Proof Harness

Figure 1 shows an example proof harness for a host-side interface function `TDH.MNG.KEY.CONFIG`. Setup and teardown procedures are shown at line 2 and line 7, respectively. They highlight that only the state `tdr` is explicitly allocated and deallocated, as specified in the excerpted specification [1, Table 1], in addition to the commonly required states `module_state` and `tdcs`.

This harness demonstrates a pre- and postcondition pair for the functional property of correct error handling in the negative space. Specifically, it examines the interface function `TDH.MNG.KEY.CONFIG` when the input argument passed via the register `RCX` is invalid. The precondition at line 13 assumes that the register `RCX` contains an invalid value, while all other state variables conform to expected constraints. The postcondition at line 20 asserts that the completion status code stored in the register `RAX` is `TDX_INVALID_OPERAND`, correctly signaling the captured negative-space error condition, as required by the specification [1, Table 1].

```
1  // proof harness components
2  void hmkc_setup() {
3   fv_setup_module_state();
4   fv_setup_tdr();
5   fv_setup_tdcs();
6  }
7  void hmkc_teardown() {
8   fv_teardown_tdcs();
9   fv_teardown_tdr();
10  fv_teardown_module_state();
11 }
12 void
13 hmkc_invalid_input_rcx_precond() {
14  ASSUME(
15   !is_valid_hmkc_input_rcx() &&
16   is_valid_hmkc_state_metadata() &&
17   is_valid_hmkc_state_lifecycle());
18 }
19 void
20 hmkc_invalid_input_rcx_postcond() {
21  ASSERT(
22   get_local_data()->vmm_regs.rax ==
23    api_error_with_operand_id(
24     TDX_OPERAND_INVALID,OPERAND_ID_RCX));
25 }
26 // auto-generated task entry
27 void main() {
28  hmkc_setup();
29  hmkc_invalid_input_rcx_precond();
30  hmkc_function_call();
31  hmkc_invalid_input_rcx_postcond();
32  hmkc_teardown();
33 }
```

Fig. 1: Example proof harness

The `main` function at line 27 is automatically generated by HarnessForge and will be used as the entry point of the created verification task. HarnessForge will

This document is an appendix to the TACAS 2026 paper [1].

automatically collect the required definitions of data structures and subroutines from the production TDX code base, override relied-upon assembly instructions with their shadow C implementations or overapproximation, slice off syntactically unreachable code, and finally produce a self-contained C file ready for off-the-shelf software verifiers.

B Identified Verifier Issues

Table 1 summarizes the issues we identified in the evaluated verifiers and includes the links to the corresponding issue trackers.

Table 1: Summary of identified issues in the evaluated verifiers

Verifier	Issue description
CBMC	Incorrect handling of `packed` in `struct` members (#8443)
CPA.	Anonymous `struct`/`union` in designated initializers not supported (#1239) Lack support for cast-to-union extension (#1289) Lack support for compiler attributes such as `packed` and `aligned` (#818)
ESBMC	Error during encoding of verification conditions (#2850, #2851) Error during program unrolling (#2852)
UAUT.	Anonymous `struct`/`union` not supported by C-to-Boogie translator (#272) Lack padding model for unpacked data structures (#417)

C Examples Showing Firmware's Unique Characteristics

C.1 Type Punning Using `union`

Figure 2 demonstrates the use of `union` for type punning in TDX Module. This type defines various interpretations of the register `RAX`, which all interface functions use to store completion status. It can be interpreted as a 64-bit unsigned integer for general status codes (e.g., `TDX_SUCCESS` and `TDX_OPERAND_BUSY`) or differently depending on the function. For instance, functions triggering TD exits use the lower 32 bits to encode detailed exit reasons, while the upper 32 bits encode seven distinct fields of varying bit widths.

C.2 Memory Layout with `__packed__`

Figure 3 shows how the type definition of the TD attestation measurement report uses the attribute `__packed__`. Without such an attribute, for many compiler frontends, the 239-byte field `tee_tcb_info` and the following 17-byte `reserved` field would have paddings added for easy alignment. Note that there is an accompanying compile-time check, using GNU C's macro `_Static_assert`, to further ensure that the overall size of the TD measurement report is exactly 1024 bytes. (`_Static_assert`s are omitted in our verification tasks since it is a compile-time check, not a runtime assertion that a verifier typically handles.)

```
typedef union api_error_code_u {
  struct {
    union {
      uint32_t operand;
      uint32_t details_l2;
      struct {
        uint16_t details_l2_low;
        uint16_t details_l2_high;
      };
    };
    uint32_t details_l1                : 8,
             clas                      : 8,
             reserved                  : 12,
             host_recoverability_hint  : 1,
             fatal                     : 1,
             non_recoverable           : 1,
             error                     : 1;
  };
  uint64_t raw;
} api_error_code_t;
```

Fig. 2: The type definition of the interface-function completion-status code that uses nested `union` for type punning

```
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);
```

Fig. 3: The type definition of the TD measurement report that uses compiler attributes for precise memory-layout control

C.3 Reimplementation of `memcpy` Using Inline Assembly

Figure 4 presents `tdx_memcpy`, a reimplementation of the standard function `void memcpy(void *dest, void *src, size_t nbytes)`. `tdx_memcpy` introduces an additional parameter, `dst_bytes`, used for a sanity check.

```
_STATIC_INLINE_ void tdx_memcpy(
  void * dst, uint64_t dst_bytes,
  void * src, uint64_t nbytes
) {
  volatile uint64_t junk_a, junk_b;
  tdx_sanity_check (dst_bytes >= nbytes,
      ↪ SCEC_HELPERS_SOURCE, 1);
  _ASM_VOLATILE_ (
    "rep; movsb;"
    :"=S"(junk_a), "=D"(junk_b)
    :"c"(nbytes), "S"(src), "D"(dst)
    :"memory"
  );
}
```

Fig. 4: TDX Module's internal implementation of the function `memcpy`, named `tdx_memcpy`, using inline assembly

References

1. 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. In: Proc. TACAS. LNCS, Springer (2026)