Runtime Verification for High-Level Security Properties: Case Study on the TPM Software Stack

Yani Ziani[1,2] [0009−0000−8540−1273], Nikolai Kosmatov[1] [0000−0003−1557−2813], Frédéric Loulergue[2] [0000−0001−9301−7829], and Daniel Gracia Pérez[1] [0000−0002−5364−8244]

[1] Thales Research & Technology, Palaiseau, France
{yani.ziani,nikolai.kosmatov,daniel.gracia-perez}@thalesgroup.com
[2] Univ. Orléans, INSA Centre Val de Loire, LIFO EA 4022, France
frederic.loulergue@univ-orleans.fr

Abstract. The Trusted Platform Module (TPM) is a cryptoprocessor designed to provide hardware-based secure storage and protect integrity of modern computers. Communications with the TPM go through the TPM Software Stack (TSS), a popular implementation of which is the open-source library *tpm2-tss*. It is thus crucial to ensure that no leak of sensitive data may occur in the TSS during communications between the host platform and the TPM. Recent work on deductive verification of functional and safety properties for this library faced several challenges. The purpose of this case study paper is to focus on high-level security properties, and to propose an alternative validation approach for such properties on the library by using runtime verification. We describe the proposed approach, apply it to specify and verify at runtime some key security properties using the Frama-C verification platform and report on our first evaluation results.

1 Introduction

The *Trusted Platform Module* (TPM) [21] has become a key security component in modern computers. The TPM defines a standard for a cryptoprocessor designed to ensure security properties such as the integrity of the platform it is a component of, or the secure storage of encryption keys inside it.

Operating systems and applications can communicate with the TPM through a standard API provided by a software specification known as the *TPM Software Stack* (TSS). A widely-used implementation of the TSS is the open-source library *tpm2-tss*. Safety and security of operations involving the TPM thus rely on both the component itself and its access API, and are highly critical in modern computers. Therefore, ensuring the security of the library is crucial, making it a key target for verification and validation.

Verification of security properties was the purpose of several recent case studies. A large-scale formal verification of global security properties on the C code

of the JavaCard Virtual Machine for Common Criteria certification was successfully conducted in [8]. Our recent work [24] used the Frama-C verification platform [11] and its deductive verification plug-in Wp in order to verify safety and functional properties on a subset of high-level functions of the tpm2-tss library involved in the secure storage of an encryption key on the TPM. This target was highly complex, based on complex data structures, low-level code, dynamic memory allocation and recursive data structures. It highlighted several current tool limitations, which required workarounds and minor yet necessary code rewriting, and make the verification of larger subsets of code — and in particular lower-level functions — very challenging.

The main motivations of this work are to extend previous verification efforts to high-level security properties, and to offer an alternative validation approach for such properties on the library by using runtime verification, complementary — but of course less rigorous — with respect to deductive verification.

In this paper we extend the previous case study [24] on formal verification of tpm2-tss using the Frama-C verification platform. We use the MetAcsl plug-in [17,18] of Frama-C, which provides an extension of the ACSL specification language [3] to define high-level security properties, and translates them into regular ACSL specifications. The runtime verification plug-in E-ACSL [12] then allows us to translate such ACSL specifications into instrumented C code, to be compiled and executed in order to verify annotations at runtime. However, much like the Wp plug-in, E-ACSL faces its fair share of limitations, for example, its capacity to handle logic definitions such as those used for linked list specification [4], direct comparisons between complex structures and unions, and various other features of the C language. On the other hand, runtime verification presents opportunities to handle more easily dynamic allocations, byte-level assignments, and varying representations of complex data structures, which presented important challenges for deductive verification [24].

We focus here on the subset of functions necessary to store an encryption key on the TPM, from which we exclude calls to cryptographic operations from external libraries, as well as linked list manipulations. We use such simplifications to fit with runtime verification. We also modify a few of the lowest-level functions, some to intentionally induce TSS errors, and some to avoid the necessity to directly use a TPM. The code is available online as a companion artifact[3].

Contributions. The contributions of this paper include:

- a modeling approach relying on specific data structures, usable for the specification and verification of high-level security properties;
- specification and runtime verification in Frama-C of high-level security properties on a representative simplified subset of functions of tpm2-tss;
- presentation of the main issues we faced;
- evaluation results including detection of simulated errors.

[3] Available (with all necessary annotations) on `https://doi.org/10.5281/zenodo.12773113`.

Outline. The rest of the paper is organized as follows. Section 2 presents Frama-C and the plug-ins used. Section 3 briefly introduces the TPM, and the tpm2-tss library, an implementation of its software stack. Sections 4 and 5 present our approach to model sensitive data and to define high-level security properties on it. Section 6 describes our evaluation and verification results. Finally, Sects. 7 and 8 present related work and a conclusion with necessary tool improvements.

2 The Frama-C Verification Platform and its Wp, E-ACSL and MetAcsl plug-ins

Frama-C [2,11] is a framework for the analysis and verification of ANSI/ISO C programs organized as a set of plug-ins around a common kernel. The mean of communication between the various plug-ins is the specification language ACSL [3,6] (ANSI/ISO C Specification Language). ACSL is basically a typed first-order logic embedding C types and pure expressions as well as logic types and built-in predicates.

There are four main plug-ins used for annotating and verifying programs:

- RTE [10] generates ACSL annotations that, if proved, ensure that there are no runtime errors in the analyzed program.
- EVA [5] is an automatic abstract interpretation based value analysis. It performs a whole program analysis.
- Wp [2] offers deductive verification. In this case, the verification is modular: the contract and annotations of each function are verified independently from the other functions, using only the contracts of the callees. Wp generates verification conditions that can be proved by external automatic or interactive provers via Why3 [9].
- E-ACSL [12] provides a way to dynamically verify ACSL annotations. A subset of ACSL, named E-ACSL, is translated into C code which allows the annotations to be verified at runtime. This is *runtime assertion checking* (RAC) [7,20]. Of course, unlike EVA and Wp, E-ACSL cannot, in general, guarantee the absence of errors for all possible executions but detects and precisely locates errors for a concrete input, thus helping to find their reason.

There are other plug-ins in the Frama-C distribution for understanding the code (for example by code metrics) and for simplifying the analyzed source code (for example by slicing), as well as external plug-ins. In particular, MetAcsl [16] is not part of the Frama-C distribution but is an actively maintained plug-in. MetAcsl considers high-level properties expressed at the global level. These properties are especially suited to state security properties. High-level properties can be proved using other high-level properties by a specific prover, but most of the time they are translated into additional local annotations in function contracts and bodies.

Annotation generators (like RTE and MetAcsl) are naturally combined with verification plug-ins. But analyzers can be combined too, which makes Frama-C even more powerful. For example, EVA was used [14] to detect pieces

of code for which there may exist a vulnerability and E-ACSL to generate a monitor to check at runtime that these potential vulnerabilities are not exploited.

3 Overview of the TPM and its Software Stack

The TPM[4] is a standard conceived by the Trusted Computing Group (TCG)[5] for a passive secure cryptoprocessor designed to protect hardware from software-based threats. It is usually implemented as a discrete chip and designed to perform cryptographic operations, but it can however also be implemented as part of the firmware of a regular processor, or as a software component.

Nowadays, the TPM is well known for its usage in consumer computers to ensure integrity and the secure storage of the keys used by *Bitlocker* or *dm-crypt* to encrypt the disk, but it may also be used to provide other cryptographic services to the Operating System (OS) and applications. The TCG defines the TPM Software Stack (TSS), a set of specifications providing standard APIs to access the functionalities and commands of the TPM, regardless of the environment.

The TSS provides different levels of complexity for its APIs, from the Feature API (FAPI) (for simple and common cryptographic services), to the System API (SAPI) (for a one-to-one mapping to the TPM commands) and the TPM Command Transmission Interface (TCTI) (for connection with the TPM). Data on the SAPI and the TCTI is stored in the form of buffers of bytes. Between the FAPI and SAPI lies the Enhanced System API (ESAPI), which provides functionalities similar to that of SAPI, with a higher level of abstraction but slightly limited flexibility. Other APIs such as the Marshaling/Serialization API complete the previous ones for common operations like data formatting.

The TSS, as any software component or the TPM itself, can have vulnerabilities[6] that attackers can exploit to recover sensitive data communicated with the TPM or to take control of the system. We center our study on the verification of the tpm2-tss library, an open-source and popular implementation of the TSS.

More precisely, we focus on the import of a sensitive information (for example, an encryption key) onto the TPM, without parameter encryption, from the ESAPI layer. This can be done using the `Esys_Create` function of the ESAPI, corresponding to the `Tss2_Sys_Create` function of the SAPI and the `TPM2_Create` TPM command. For each involved layer (including the TCTI), the TSS relies on a notion of context containing all the data the layers need to store between calls, and to transfer data from one layer to the next (and vice-versa). This is done by design, so that the TSS does not need to maintain a global state or global variables. The imported sensitive data is transferred from one layer to the next through these contexts, written in high-level structures in the ESAPI context, and low-level buffers in the SAPI and TCTI contexts.

We focus here on the security of sensitive data, when going through the TSS, as it is transformed between layers to be sent to the TPM for import.

[4] See the TPM specification [21] and reference books as [1] for more detail.
[5] `https://trustedcomputinggroup.org/`
[6] Like CVE-2023-22745 and CVE-2020-24455, documented on `www.cve.org`.

4 Companion Memory Modeling for Sensitive Data

Function subset. To construct our target subset of functions, we simplified an integration test of the library for the `Esys_Create` function, to import an object onto the TPM without using parameter encryption.

To do so, we remove from the TSS dependencies to other function calls to TPM commands for authorizations, parameter encryption (and thus dependencies to an external library such as OpenSSL), and we replace the command transmission interface with a dummy that receives the command buffer sent to the TPM, but replies to higher layers of the TSS with an empty response buffer. We do not change anything else, and keep each dynamic memory allocation, freeing, and memory manipulation operation as is.

Thus, we consider the import of a single object acting as sensitive data, that is statically allocated in the test function. In this scenario, the call to `Esys_Create` will succeed in sending the TPM command to the TCTI, and will fail at the very end when receiving the invalid TPM response buffer. Regardless of the failure or success of the ESAPI call to the `TPM2_Create` command, it is expected that the TSS should not retain any sensitive data at the end of the integration test.

Examples overview. To illustrate our verification approach, we provide our modeling of the memory for sensitive data with Figs. 1–2. We isolate with Figs. 3–6 a few small examples of code and structure definitions from our subset representative of the manipulation of the imported sensitive information. Fig. 7 provides the security properties we verify in this work. Figs. 1–2 will be described step-by-step in this section, and Figs. 3–7 will be explained in Sec. 5. As it is often done, some ACSL notation (e.g. `\forall`, `integer`, `==>`, `<=`, `!=`) is pretty-printed (resp., as $\forall$, $\mathbb{Z}$, $\Rightarrow$, $\leq$, $\neq$).

Memory representation. Lines 1–4 of Fig. 1 show our companion memory representation for sensitive data defined as global variables. All sensitive data shall be uniquely represented by its memory location and its size. Thus, we define `_all_sens` on line 3 as an array of `char *` pointers to be used to store the addresses of all the possible representations and copies of the imported object and other pieces of sensitive data throughout the layers of the TSS. With the `_len_sens` array of `int` on line 2, we couple each of these addresses with the size (in bytes) of the associated sensitive data. For convenience, the macro on line 1 defines `MAX_SENS`, a maximum number of sensitive data to be modeled. The `_nb_sens` variable on line 4 serves to determine whether it is possible to add another piece of sensitive data to our representation: it stores the number of stored pointers (which gives also the index of the next available one). Lines 7–10 define indices we use to record where each type of sensitive information is located in our model. We define `_sens_A_B_idx` as the index of `A` or information `A` in location `B`, where `A` is the name of the original variable referred to by `_all_sens[_sens_A_B_idx]`, and `B` describes whether the original is a local variable in a specific layer, or stored in the ESAPI context, or in a command buffer.

```
1  #define MAX_SENS 100
2  int _len_sens[MAX_SENS];
3  char * _all_sens[MAX_SENS];
4  int _nb_sens;
5
6  // define _sens_##vartype##_##usage##_idx here
7  int _sens_inSensitive_esys_Create_idx = -1;
8  int _sens_inSensitiveData_esys_Create_ctx_idx = -1;
9  int _sens_outPrivate_esys_Create_idx = -1;
10 int _sens_inSens_sys_Create_cmdbuff_idx = -1;
11
12 bool is_sens(void *ptr, int size, int idx)
13 {
14   if (0 > idx ∨ idx ≥ MAX_SENS){return false;}
15   else if (0 ≤ idx ∧ idx < MAX_SENS){
16     return(_all_sens[idx] == ptr ∧ _len_sens[idx] == size);
17   }
18 }
19
20 void remove_sens(int idx){_len_sens[idx] = 0;}
21
22 int add_as_sens(void *ptr, int size)
23 {
24   int idx;
25   if(_nb_sens ≥ MAX_SENS ∨ _nb_sens < 0){idx =-1;}
26   else {
27     _all_sens[_nb_sens] = (char*) (ptr);
28     _len_sens[_nb_sens] = size;
29     idx = _nb_sens++;
30   }
31   return idx;
32 }
```

Fig. 1. Companion memory representation for sensitive data

Consequently, we consider that `_sens_A_B_idx` refers to sensitive data if we have `0≤_sens_A_B_idx<MAX_SIZE` and `0<_len_sens[_sens_A_B_idx]`. The fact that the corresponding memory block can be safely read and written, that is, `\valid(_all_sens[_sens_A_B_idx] + (0.._len_sens[_sens_A_B_idx]-1))`, is supposed to be true in this case (it will be verified in the global invariant given in Fig. 2, discussed below).

In particular, we define `is_sens` in lines 12–18, `remove_sens` in line 20, and `add_as_sens` in lines 22–32. Given a pointer `ptr`, a size `size` and an index `idx`, `is_sens` is used to determine whether the memory block of size `size` at address `ptr` corresponds to the sensitive data at index `idx` of the representation, returns `true` in this case, and `false` otherwise. The `remove_sens` function in line 20 sets the size `_len_sens[idx]` of the sensitive data at index `idx` of our model to `0`, to express that `idx` is not considered to refer to sensitive data anymore.[7] `remove_sens` is typically used, for instance, when dynamically allocated sensitive data is freed, overwritten with non-sensitive data, or when we exit the scope of automatically allocated variables (such as local variables) with sensitive data.

[7] For simplicity, when an element is removed, we do not shift the following array elements to the left. This feature can be easily added and particularly useful for long-running programs, where pieces of sensitive data are frequently added and removed.

```
1 /*@ meta \prop, \name(valid_sensitive),
2   \targets(\ALL),
3   \context(\strong_invariant),
4   ∀ int i; 0 ≤ i < _nb_sens ⇒ 0 < _nb_sens ≤ MAX_SENS ⇒
5     0 < _len_sens[i] ⇒
6     \valid(_all_sens[i] + (0 .. _len_sens[i] - 1)); */
```

Fig. 2. Global invariant for the validity of memory accesses of modeled sensitive data

We use the `add_as_sens` function defined in lines 22–32 to add a new piece of sensitive data to the representation, given a pointer `ptr` and a size `size`. `add_as_sens` first checks with line 25 if `_nb_sens` is between `0` and `MAX_SENS`, that is to say if it is possible to add new data to the representation. If it is, `ptr` (resp. `size`) is added to the `_all_sens` array (resp. `_len_sens` array) at index `_nb_sens` (our tracking index), which is then incremented by 1. The new value of `_nb_sens` is then returned, to be used to set the indices defined in lines 7-10.

We use `is_sens` as a way to determine whether a piece of sensitive data exists in the representation before removing it with `remove_sens`, thus to determine whether `remove_sens` is safe to call.

Validity invariant. In order to ensure that all sensitive data added to our representation is always safe for the TSS to read and write, we define as a MetAcsl meta-property a global validity invariant `valid_sensive` in Fig. 2. Line 2 defines the set of target functions in which this property should be satisfied, which in this case comprises all the defined functions of our representative subset. The `\context(\strong_invariant)` clause in line 3 indicates that the property must hold at every point of each target function. Subsequently, with lines 4–6, we express that for any index `i` for which sensitive data is defined (that is to say, for all indices within `[0, MAX_SIZE - 1]` such that `_len_sens[i]>0`) we must have `\valid(_all_sens[i] + (0.._len_sens[i] - 1))` (that is to say, the memory block at address `_all_sens[i]` of size `_len_sens[i]` bytes must be safe to read and write).

This property is expected to always be verified. Any violation of this invariant would imply that sensitive data was deleted or freed from the program before it was removed from our representation, which can only happen in one of two situations: either we neglected to remove sensitive data from the model when it should have been, or such data is freed too early.

5 Defining and Verifying High-level Security Properties

This section presents how we use the definitions introduced in Sec. 4 to define and verify high-level security properties such as the integrity and the confidentiality of sensitive data (in our case, the object sent to the TPM). We also identify issues related to the limited support of ACSL clauses in E-ACSL related to logic labels, function pointers, and which currently prevent some of the possible

```
1  /** Store sensitive data inside the ESYS_CONTEXT */
2  static void store_input_parameters (
3      ESYS_CONTEXT *esysContext,
4      const TPM2B_SENSITIVE_CREATE *inSensitive)
5  {
6    if (inSensitive == NULL) {esysContext->in.Create.inSensitive = NULL;}
7    else {
8      esysContext->in.Create.inSensitiveData = *inSensitive;
9      esysContext->in.Create.inSensitive =
10       &esysContext->in.Create.inSensitiveData;
11   /*For E-ACSL*/
12     /* We add the part of context containing the imported sensitive
13        object to our representation of sensitive data. */
14
15      _sens_inSensitiveData_esys_ctx_idx =
16       add_as_sens(esysContext->in.Create.inSensitive,
17              (int) sizeof(esysContext->in.Create.inSensitiveData));
18
19   /*end E-ACSL*/
20   }
21 }
```

Fig. 3. Adding sensitive data from ESAPI to the representation

extensions of the pool of verifiable security properties. In this section, we detail Figs. 3–7.

Adding sensitive data on higher-level layers. Figure 3 provides an example of addition of sensitive data from the ESAPI layer to our representation. More specifically, we focus on the `store_input_parameters` function in charge of storing the data from `inSensitive` inside the ESAPI context `esysContext`, before it can be sent to the lower layers. The ESAPI context as defined in tpm2-tss has specific fields to store the sensitive data to be imported, and its address.

The sensitive information passed as argument of `store_input_parameters` here is assumed to have already been added to the modeling in callers. The function checks with line 6 whether `inSensitive` is a `NULL` pointer, and sets the corresponding field in the context to `NULL` if such is the case. Otherwise, with lines 8–10, it adds a copy of the pointed memory — that is to say, the sensitive data itself — to the context (in the `in.Create.inSensitiveData` field), and sets its address within the context (in the `in.Create.inSensitive` field).

This new instance of the imported data is different from the one passed as argument of the function. As it refers to a memory location different from that of `inSensitive`, we need to add it to our representation as well. In lines 14–18, we add the new instance of the sensitive data, using its address given by the `in.Create.inSensitive` field, and its size defined as the `sizeof` of the `in.Create.inSensitiveData` field. The reason we can use `sizeof` is because the `TPM2B_SENSITIVE_CREATE` only contains fields of fixed size, as shown in Fig. 4. More specifically, this type is defined in lines 6–9 with two fields, an `int`-like field `size`, and a `sensitive` field of type `TPMS_SENSITIVE_CREATE` defined in lines 2–5 and comprised solely of two subfields `userAuth` and `data`, each defined with an `int`-like field and a fixed-size array, following the template in line 1.

```
1 typedef struct {UINT16 size;BYTE buffer[TYPE_MAX_SIZE];} TPM2B_TYPE;
2 typedef struct  {
3   TPM2B_AUTH userAuth;            //TPM2B_TYPE-like type
4   TPM2B_SENSITIVE_DATA data;      //TPM2B_TYPE-like type
5 } TPMS_SENSITIVE_CREATE;
6 typedef struct  {
7   UINT16 size;    // ignored by marshal according to the TCG specifications
8   TPMS_SENSITIVE_CREATE sensitive;
9 } TPM2B_SENSITIVE_CREATE;
```

Fig. 4. Partial type definition used for `inSensitive`

While knowing the precise type definition is not paramount to handle sensitive data on high-level layers, knowing it is essential in understanding how such data is transferred to lower-level layers, and how to handle sensitive data on SAPI and TCTI. As the latter acts as a transmission layer to send the command buffer formed on the SAPI to the TPM (resp. to send the response buffer from the TPM to the SAPI), sensitive data is stored in the same way in both layers.

Adding sensitive data on low-level layers. Figure 5 presents an example of addition of sensitive data, to our model, in a function of SAPI, a lower-level layer of the library. We focus here on the `Tss2_Sys_Create_Prepare` (line 1) library function which partially constructs the SAPI command for the Create command by copying and saving data passed as argument to the `cmdBuffer` of the `sysContext` context. It is typically used to transfer information such as `inSensitive` (used as argument in line 4) from higher layers to lower layers.

The snippet of code in lines 28–37 is in charge of such an operation. In particular, lines 29 checks whether the `inSensitive` pointer is `NULL` or not. If such is the case, the value `0` is copied byte per byte to the command buffer to indicate that the command will not have an imported object. Otherwise, it will copy the relevant content of `inSensitive` to the `ctx->cmdBuffer` buffer, starting at the `ctx->nextData`-th byte.

More specifically, the function call in lines 33–34 is used to read the subfields of `inSensitive`, and write byte per byte to position `ctx->nextData` of the command buffer, in this order (and as defined in Fig. 4), the size `userAuth.size`, the first `userAuth.size` bytes of the `userAuth.buffer` array, the size `data.size`, and the first `data.size` bytes of the `data.buffer` array. This function is a marshal/serialization function, which translates a data structure (in this case, objects of type `TPM2B_SENSITIVE_CREATE`) into a lower-level representation fit for data transmission (in this case, a buffer of bytes). This marshal function does not write `inSensitive->size` inside the command, but computes the size in bytes of the written area, and writes it. We define the total size in bytes of the written area in the command buffer with the `sys_sensitive_size` defined in lines 41–45. We base this definition on the behavior described in the official TSS SAPI Specification, rather than the behavior of this marshaling function as it is implemented in the tpm2-tss library. With line 28, we store in the `inSens_offset` variable the offset where the marshaling stores the transformed sensitive data in the command buffer. We then add to our model the SAPI command buffer

```
1  TSS2_RC Tss2_Sys_Create_Prepare(
2    TSS2_SYS_CONTEXT *sysContext,
3    TPMI_DH_OBJECT parentHandle,
4    const TPM2B_SENSITIVE_CREATE *inSensitive,
   ...
8  {
   ...
28   size_t inSens_offset = ctx->nextData;
29   if (!inSensitive) {
30     rval = Tss2_MU_UINT16_Marshal(0, ctx->cmdBuffer,
31                       ctx->maxCmdSize,&ctx->nextData);
32   } else {
33     rval = Tss2_MU_TPM2B_SENSITIVE_CREATE_Marshal(
34       inSensitive, ctx->cmdBuffer, ctx->maxCmdSize, &ctx->nextData);
35   }
36   if (rval)
37     return rval;
38   /* We add the part of the command buffer containing the imported
39     sensitive object to our representation of sensitive data. */
40
41   int sys_sensitive_size = (int) sizeof(UINT16) +
42                 (int) sizeof(inSensitive->sensitive.userAuth.size) +
43                 (int) inSensitive->sensitive.userAuth.size +
44                 (int) sizeof(inSensitive->sensitive.data.size)+
45                 (int) inSensitive->sensitive.data.size;
46   _sens_inSens_sys_Create_cmdbuff_idx =
47       add_as_sens(ctx->cmdBuffer + inSens_offset, sys_sensitive_size);
   ...
96 }
```

Fig. 5. Adding sensitive data from SAPI to the representation

instance of `inSensitive`, using its address `ctx->cmdBuffer + inSens_offset` and its size `sys_sensitive_size`, with lines 46–47.

Removing sensitive data and handling its lifetime. In order to express high-level security properties on sensitive data in this subset, it is necessary to determine the lifetime of such data, so as to ascertain when such properties need to be verified. Thus, if we take for example the integrity of the imported key, for which we need to ensure it is only written or modified when it is supposed to be (as will be explained later on with Fig. 7), we need to precisely define for every instance of the key when they are freed or overwritten, and thus when their representation should be removed from the companion model.

We illustrate this with the two snippets of code in Fig. 6 (lines 387–395 and lines 494–502 refer to the same file in the considered subset of code, available in the companion artifact). Lines 389–392 allocate memory to be used to store sensitive data `outPrivate` returned by the TPM about the imported key, before the TSS receives the response buffer (which is empty in our example). Similarly to what we presented in Figs. 3 and 5, we add it to the companion model with lines 393–394. Lines 494-502 provide a typical usage, in the TSS, of dynamic deallocation of memory previously allocated for command or response parameters. The `if` statements in lines 494 and 495 correspond to checks to ensure the call to `free` line 499 is safe, before setting the corresponding pointer to `NULL` with line 500.

```
387   /* Allocate memory for response parameters */
388   if (outPrivate ≠ NULL) {
389     *outPrivate = calloc(sizeof(TPM2B_PRIVATE), 1);
390     if (*outPrivate == NULL) {
391       return_error(TSS2_ESYS_RC_MEMORY, "Out␣of␣memory");
392     }
393     _sens_outPrivate_esys_Create_idx =
394       add_as_sens(*outPrivate, (int) sizeof(TPM2B_PRIVATE));
395   }
   ...
494   if (outPrivate ≠ NULL){
495     if((*outPrivate) ≠ NULL) {
496       /*remove the sensitive data from the model before freeing*/
497       if(is_sens(*outPrivate, size, (int) sizeof(TPM2B_PRIVATE)))
498         remove_sens(_sens_outPrivate_esys_Create_idx);
499       free((void*) (*outPrivate));
500       (*outPrivate)=NULL;
501     }
502   }
```

Fig. 6. Usage example for `remove_sens`

Defining security properties. To define high-level security properties over specific pieces of sensitive data (in our case, the key to be imported), we first need to add such data to the companion model as previously described in this section. Consequently, because our representation relies on both the address and the size in bytes to represent and describe sensitive data, we can verify high-level properties over all relevant memory locations by specifying and verifying the properties in question using their companion representation. Moreover, using this intermediary model is essential for two main reasons:

- Using a common global view of sensitive information leads to a smaller specification effort compared to using each instance of sensitive data separately, provided they all are added to the model when they are defined, and removed when they are freed or overwritten.
- The MetAcsl plug-in is mostly used to express properties over data that is visible at a global level. Therefore, it mostly relies on global variables to express properties, of which the library avoids the usage. Although it *is* possible to use the `\formal` operator to refer to arguments of target functions, it is limited in that it is not possible to use it to refer to existing memory locations unrelated to function arguments.

Figure 7 shows the meta-properties expressing the integrity and the confidentiality of sensitive data represented by the companion model. We define two additional arrays with lines 1–2, `_write_sens` and `_read_sens`, that we use in the properties defined in lines 4–32 to help determine whether a sensitive data should be read or written. Line 4 provides `all_sens_data_integrity` as a name for our property, and lines 6–9 provide the set of target functions for the property, that is to say the set of functions in which the property shall be verified. We define it as our full subset of functions, from which we exclude the set defined in lines 7–9. The functions in line 7 are excluded due to parsing issues with MetAcsl, and the functions lines 8–9 are removed from the tar-

```
1 int _write_sens[MAX_SENS];
2 int _read_sens[MAX_SENS];
3
4 /*@ meta \prop, \name(all_sens_data_integrity),
5   \targets(
6     \diff(\ALL,
7       \union({tcti_fake_init, doLog, doLogBlob, getLogFile, // F-C crash
8         tcti_proxy_transmit_fake_tpm, tcti_fake_receive,
9         tcti_proxy_receive_fake_tpm }))), //function pointers support
10  \context(\writing),
11  ∀ int i; 0 ≤ i < _nb_sens ⇒ 0 ≤ _nb_sens ≤ MAX_SENS ⇒ //idx within bounds
12    0 < _len_sens[i] ⇒ //sensitive data exists
13    _write_sens[i] ≠ 1 ⇒ //sensitive data should not be writable
14    \separated(\written, (char*)_all_sens[i]+(0..(size_t)(_len_sens[i]-1)));*/
15
16 /*@ meta \prop, \name(all_sens_data_confidentiality),
17   \targets(
18     \diff(\ALL,
19       \union({tcti_fake_init, doLog, doLogBlob, getLogFile, // F-C crash
...
27         tcti_proxy_receive_fake_tpm}))),
28  \context(\reading),
29  ∀ int i; 0 ≤ i < _nb_sens ⇒ 0 ≤ _nb_sens ≤ MAX_SENS ⇒
30    0 < _len_sens[i] ⇒
31    _read_sens[i] ≠ 1 ⇒ //sensitive data should not be readable
32    \separated(\read, (char*)_all_sens[i]+(0..(size_t)(_len_sens[i]-1)));*/
```

Fig. 7. Meta-property for the integrity and confidentiality of sensitive data

get because they are called using function pointers, whose support is limited in E-ACSL. Line 10 establishes a writing context with `\context(\writing)` for `all_sens_data_integrity`, meaning that the predicate defined in lines 11–14 must hold whenever a variable or a memory location is written, where `\written` refers to the written location. The predicate lines 11–14 states that, provided the tracking index `_nb_sens` is within bounds, for any index `i` referring to existing sensitive data, *if* the data stored at index `i` is not supposed to be readable (line 13), then the written location must be separated from any sensitive data. While MetAcsl does provide a clause to locally relax strong invariants, it is currently not natively possible to locally relax other types of meta-properties without excluding entire functions. In order to do so, for specific data stored at index `idx` in the model, we manually set `_write_sens[idx]` to `1` in parts of code where the data should be written, before it is written, and we set it back to `0` afterwards. Similarly, we may define the confidentiality of sensitive data `all_sens_data_confidentiality` with lines 16–32, using the `\reading` context, and by stating any read location must be separated from existing sensitive data in the target set of functions defined with lines 19–27. We manually set `_read_sens[idx]` for data of index `idx` to `1` to locally relax confidentiality, for instance in functions like `store_input_parameters` of Fig. 3 and `Tss2_Sys_Create_Prepare` in Fig. 5, as they can read already existing sensitive data to create new instances and copy of it.

MetAcsl translates the meta-properties defined in Fig. 2 and Fig. 7 into local ACSL assertions at each relevant program point. For example, for the integrity, an assertion of the defined separation will be added to the code before

```
#include <string.h>
/*@ assigns ((char*)dest)[0..n - 1] \from ((char*)src)[0..n-1];*/
void *memcpy_e_acsl (void * dest, const void *src, size_t n)
{return memcpy(dest, src, n);}
```

Fig. 8. `memcpy` wrapper function

each writing operation, where `\written` will be replaced by the written location. For the validity invariant, `valid_sensitive`, an assertion is generated at every program point of the target functions. Once generated, the resulting annotated code can be given as input to E-ACSL to be instrumented, and to convert the logical ACSL assertions into executable assertions. E-ACSL can then use GCC to compile this version of the code. The meta-properties are verified if all the generated executable assertions are verified at runtime.

Handling standard memory manipulation functions. By default, Frama-C uses its own C standard library, providing a number of its functions with ACSL contracts to specify their behaviors. It is however possible for an analyzed program to use libc functions that have not been defined (yet) in the Frama-C libc, for some of the defined types to mismatch those of the system libc, or even for other plug-ins to generate code that may be unusable by or incompatible with E-ACSL. In order to instrument and compile a program with E-ACSL, it can in turn be necessary to force the usage of the standard library of the system. Our subset of functions (and tpm2-tss in general) represent a case of such obligation.

However, some of these contracts include information about which parts of the memory may be modified by the corresponding function, through the use of the `assigns` clauses. In particular, `assigns A, B \from C, D;` specifies that the memory locations referred by variables `A` and `B` *can* be modified by the corresponding function, which reads variables `C` and `D` to do so. While E-ACSL does not currently support this clause, MetAcsl can rely on such annotations to generate assertions for meta-properties using the `\writing` or the `\reading` contexts. More specifically, if we consider our integrity (resp. confidentiality) meta-property applied to a function `f` specified with a minimal `assigns A \from C` clause, whenever `f` is called, MetAcsl will generate an assertion so that the written (resp. read) memory location referred by `A` is separated from any defined sensitive data. Without these specifications, MetAcsl will not be able to cover all situations where some meta-properties should be checked.

Thus, for functions of the libc such as `memcpy` that may read or write (non-local) variables, and whose contracts are unavailable to be used by MetAcsl for compatibility reasons with E-ACSL, we define wrapper functions with minimal contracts to describe such memory manipulation as shown in Fig. 8.

6 Evaluation and Key Lessons

The evaluation of our approach addresses the following research questions:

- **RQ1 (Expressiveness)**: To what extent does this verification approach allow for the expression and the verification of high-level security properties on real-life code?
- **RQ2 (Effectiveness)**: To what extent is this verification approach *effective* for the verification of high-level security properties on real-life code?
- **RQ3 (Efficiency)**: To what extent is this verification approach *efficient* for the verification of high-level security properties on real-life code?

Verification target. To answer the proposed research questions, we use a subset of 86 functions of the tpm2-tss library, of which 50 consist in marshal functions for different data types. The remaining 36 functions correspond to unique TSS operations involved in the import operation (without parameter encryption) defined by the `Esys_Create` function/the `TPM2_Create` command. The entire subset consists of approximately 20k lines of code, about 12k of which corresponding to interfaces and the remaining 8k corresponding to actual function implementations, with marshal functions defined as non-unfolded macros.

We construct the target using one of the integration tests of the library, and we remove from the subset cryptographic operations and their dependencies, and dependencies to libraries other than the C standard library. We also implement a few different versions of the code where we introduce faults designed to locally break the integrity and the confidentiality as they are defined in Fig. 7.

Tools Used. We use Frama-C v28.1(Nickel) with the corresponding version of E-ACSL, and MetAcsl v0.6. We use MetAcsl to parse the defined high-level security properties and generate corresponding low-level ACSL assertions on each version of the code (with and without introduced errors). We then instrument the generated annotated code with E-ACSL to translate the annotations into C code to allow their verification at runtime. The ouput code is compiled by GCC through E-ACSL.

Verification Environment. For conducting our evaluation, we parse, instrument, compile and run each version of the code. All evaluations are performed on a desktop computer running Ubuntu 20.04.6 LTS, with an Intel(R) Core(TM) i5-6600 CPU @ 3.30 GHz, featuring 4 cores and 4 threads, with 16GB RAM.

RQ1: Expressiveness. The MetAcsl plug-in allows users to define and express properties over data that are visible at a global level, thus mostly relying on global variables as targets. However, software specifications such as the TSS — and therefore library API code such as that of the tpm2-tss library — may require to avoid the usage of global state variables. Our approach allows rendering such data visible at a global level, and thus usable as target variables for the definition of high-level properties with MetAcsl, to be verified with other Frama-C plug-ins such as Wp or E-ACSL. In this way, it *extends* the capacities of MetAcsl, whose usage to handle data not visible at a global level is very challenging.

In our work, we focus on two high-level security properties, integrity and confidentiality of sensitive data, verified at runtime with E-ACSL. We do not define properties such as invariants relying directly on the values stored in modeled data between two program points, as their verification can appear difficult due to the current limitations of E-ACSL, and since lightweight runtime verification of security properties is an important goal by itself. However, such properties should be provable with Wp, but would require a bigger specification effort.

RQ2: Effectiveness. In our approach, we use the E-ACSL plug-in to verify at runtime the assertions generated by MetAcsl for each security property. Using this method requires a much smaller specification effort on such real-life code than that of deductive verification for properties defined in ACSL.

Indeed, we define the integrity and the confidentiality of sensitive data by using the `\separated` clause to express memory separations, which are typically hard to prove with Frama-C on C codes relying on more dynamic management of the memory, as the Wp plug-in does not currently rely on any aspect of separation logic; proving assertions about memory separations can require a lot of intermediary specifications, as highlighted in [13] and [24].

To ensure the integrity of sensitive data as defined in Fig. 7, MetAcsl generates about 800 low-level assertions on our subset of functions, and approximately 4200 low-level assertions for the confidentiality.

We did not detect any breach of integrity or confidentiality of sensitive data in our subset "as is". To ensure our approach is effective, we applied it to a few variations of the code in which we introduced errors to detect. For instance, we modified a few marshal functions to wrongly report where sensitive data was written. We also imagined a scenario in which a command buffer sent to the TPM could have been temporarily stored in a global variable for reissuing. We detected the resulting breaches of integrity in the former, and of confidentiality in the latter. In addition, we tested and detected situations where errors in specifications lead to sensitive data being added several times to the model, and the RTE safeguard checks performed by E-ACSL for each separation clause allowed for the detection of situations when sensitive data was wrongly removed from the representation after it had been deleted in the code.

RQ3: Efficiency. Parsing, instrumentation, compilation and execution times were similar for the main subset and its variations. Generating low-level assertions from high-level properties with MetAcsl took only about 8 seconds. Instrumentation with E-ACSL and compilation with GCC took much longer, reaching up to 15 minutes for the instrumentation, and up to 5 minutes for the compilation. Running a produced executable takes less than a second. Hence, the execution time remains compatible with executing numerous test cases on the same version of the code, as it should be typically required for a rigorous dynamic verification.

On average, the processing and evaluation of a version of the code takes approximately 19 minutes. We believe that this is considerably shorter than what would probably be required for deductive verification with Wp, and requires

much less time and effort for specification (from our experience with deductive verification). This should be confirmed with a more rigorous study focusing on comparing directly the usage of Wp and E-ACSL for verifying the proposed properties. Additionally, the complete processing of a single version of the code is single-threaded, and thus can be further accelerated by running in parallel (for various versions of the code, or for different test cases).

6.1 Threats to Validity

Internal validity. To verify the low-level specifications generated by MetAcsl from the high-level properties, we used E-ACSL to check them at runtime. The plug-in has however its fair share of limitations, including but not limited to the support of labels and logic definitions (which are supported by Wp), or the limited support of function pointers. Moreover, although our verification approach should not modify the behavior of the original code as is, we can not guarantee it to be on the same level as if using ghost code. Indeed, we have identified (and reported to developers) errors in ghost code instrumentation with E-ACSL.

External validity. The conclusions about the expressiveness, effectiveness and efficiency of our approach might not hold for a complete TSS call to a TPM command, other programs using linked data structures, or with dependencies to external libraries. However, it should still be possible to conduct at least a partial verification by excluding problematic parts of the code.

6.2 Key Lessons

Verification Methodology. Our approach has allowed us to suggest a methodology for the verification of high-level security properties (such as integrity and confidentiality) of sensitive data on real-life code. We believe it to be relatively easy to apply for a user who does not possess in-depth prior knowledge of the target code. The methodology can be broken down into the following steps:

1. Define the memory representation to be used for sensitive data, as shown in Fig. 1.
2. Identify at a high level of abstraction the pieces of sensitive data whose security has to be ensured, and add them into the model.
3. Define security properties over sensitive data. Defining the integrity and the confidentiality as shown in Fig. 7 makes for a good starting point, and should help refine the approach in the following steps.
4. Parse, instrument, compile and execute the target code with the defined properties:
   (a) If the code lacks an entry point, it is up to the user to define a `main` function and to ensure the code can be compiled.
   (b) Parse the code and the properties with MetAcsl, instrument the resulting annotated code and compile the output with E-ACSL, and execute.

5. If possible, use the output of the executed code to refine the previous definitions:
   (a) A detected violation of confidentiality should indicate either an "illegal" read of sensitive data, or that a sensitive data should not be considered as sensitive at the reported program point, or a "legal" read of sensitive data not yet handled by the current definitions. In the latter case, the user should locally relax the property by setting the corresponding element of `_read_sens` to `1` before the read, and setting it back to `0` after data has been read.
   (b) Similarly, a detected violation of integrity should indicate either an "illegal" write of sensitive data, or that a sensitive data should not be considered as sensitive at the reported program point, or a "legal" write of sensitive data not yet handled by the current definitions. In the latter case, the user should locally relax the property by setting the corresponding element of `_write_sens` to `1` before the write, and setting it back to `0` after data has been written.
6. Repeat steps 4 and 5 until either no violation of properties are detected, or all reported violations correspond to breaches of integrity or confidentiality.

7 Related Work

TPM related safety and security. Various case studies centered around functionalities of the TPM itself have emerged over the last decade. A recent formal analysis of the key exchange primitive of TPM 2.0 [23] provides a security model to capture TPM protections on keys and protocols. Authors of [22] propose a security model for the cryptographic support commands in TPM 2.0, proved using the CryptoVerif tool. A model of TPM commands was used to formalize the session-based HMAC authorization and encryption mechanisms [19]. Authors of [15] conducted a survey and established a study environment for the usability and security of TPM library APIs, including but not limited to the tpm2-tools library (which relies on tpm2-tss). They "conducted the first qualitative study targeting TPM library APIs and found that they are not developer-friendly".

To the best of our knowledge, none of the previously published works aim at verifying any implementation of the TSS.

Formal verification of high-level properties and real-life code. In Djoudi et al. [8] the authors present a large scale case study where deductive verification is applied on real-life C code to prove security properties.

Authors of [17] used MetAcsl to define meta-properties on two illustrative case studies, that they verified firstly with deductive verification using Wp, then secondly with testing using E-ACSL. The runtime verification of such properties in this work differs from ours in that they targeted a smaller code size, which relied on a more static management of the memory and global state variables, with unified representations, making it much easier to define meta-properties "out-of-the-box". In contrast, our approach provides the means to handle objects that conceptually contain the same information, but have different levels

of representation in the code. Moreover, it provides a high level of automation, requiring very little specification work for *the chosen properties*.

We also validate the authors' claims that their "study demonstrates that it is easy to check meta-properties at runtime without extra annotation effort thanks to the combination of MetAcsl and E-ACSL, *as long as the specified properties are supported by the tools*".

8 Conclusion and Future Work

We presented in this paper preliminary work on how to define and verify security properties over sensitive data in real-life code. We targeted the tpm2-tss library, a popular implementation of the TPM Software Stack. As the communication layer between the TPM and the host platform or applications, this library is highly critical: to ensure the security of sensitive data to be stored on the TPM, it is highly desired to guarantee that such data can never be recovered without authorization while it is passing through the software stack. The library code is very different from more industrial codes, very complex, and challenging for verification tools.

We presented our verification result for a subset of 86 functions, on which we have verified, at runtime, that the target sensitive data is never modified or read when it is not supposed to. We have described some limitations of the tool and temporary solutions we used to address them. The real-life code was slightly simplified to remove dependencies to external libraries, but the logical behavior of the code corresponding to the function subset was maintained.

Future Work. Currently, only two high-level security properties over (some, but not all) pieces of sensitive data were verified at runtime, on a relatively simplified subset of the tpm2-tss library. In the future, we plan to perform a more complete verification work by reintroducing cryptographic capabilities of the TSS, and running the annotated code paired with a real TPM. Extending the range of security properties and improving the automation of this approach is another point of improvement, as well as establishing a method to perform a more extensive evaluation of this approach, and to allow for a careful comparison with deductive verification. Another interesting perspective of future work is to combine the deductive verification of Wp with the E-ACSL runtime verification to perform a more thorough and complete verification of high-level security properties.

Acknowledgment. Part of this work was supported by ANR (grants ANR-22-CE39-0014, ANR-22-CE25-0018) and French Ministry of Defense (PhD grant of Yani Ziani). We thank the anonymous referees for many helpful comments.

References

1. Arthur, W., Challener, D.: A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security. Apress, USA, 1st edn. (2015)

2. Baudin, P., Bobot, F., Bühler, D., Correnson, L., Kirchner, F., Kosmatov, N., Maroneze, A., Perrelle, V., Prevosto, V., Signoles, J., Williams, N.: The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. Commun. ACM **64**(8), 56–68 (2021). https://doi.org/10.1145/3470569
3. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, http://frama-c.com/acsl.html
4. Blanchard, A., Kosmatov, N., Loulergue, F.: Logic against ghosts: Comparison of two proof approaches for a list module. In: Proc. of the 34th Annual ACM/SIGAPP Symposium on Applied Computing, Software Verification and Testing Track (SAC-SVT 2019). pp. 2186–2195. ACM (2019). https://doi.org/10.1145/3297280.3297495
5. Blazy, S., Bühler, D., Yakobowski, B.: Structuring abstract interpreters through state and value abstractions. In: Verification, Model Checking, and Abstract Interpretation (VMCAI). LNCS, vol. 10145, pp. 112–130. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52234-0_7
6. Burghardt, J., Gerlach, J., Lapawczyk, T.: ACSL by Example (2016), http://www.fokus.fraunhofer.de/download/acsl_by_example
7. Clarke, L.A., Rosenblum, D.S.: A historical perspective on runtime assertion checking in software development. SIGSOFT Softw. Eng. Notes **31**(3), 25–37 (May 2006). https://doi.org/10.1145/1127878.1127900
8. Djoudi, A., Hána, M., Kosmatov, N.: Formal Verification of a JavaCard Virtual Machine with Frama-C. In: Formal Methods (FM 2021). LNCS, vol. 13047, pp. 427–444. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_23
9. Filliâtre, J.C., Paskevich, A.: Why3 - where programs meet provers. In: Programming Languages and Systems (ESOP). LNCS, vol. 7792, pp. 125–128. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
10. Herrmann, P., Signoles, J.: RTE: Runtime Error Annotation Generation (2024), https://frama-c.com/download/frama-c-rte-manual.pdf
11. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7
12. Kosmatov, N., Signoles, J.: A lesson on runtime assertion checking with Frama-C. In: Runtime Verification (RV). LNCS, vol. 8174, pp. 386–399. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40787-1_29
13. Loulergue, F., Blanchard, A., Kosmatov, N.: Ghosts for lists: from axiomatic to executable specifications. In: Proc. of the 12th International Conference on Tests and Proofs (TAP 2018). LNCS, vol. 10889, pp. 177–184. Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_11
14. Pariente, D., Signoles, J.: Static Analysis and Runtime Assertion Checking: Contribution to Security Counter-Measures. In: Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC) (Jun 2017)
15. Rao, S.P., Limonta, G., Lindqvist, J.: Usability and security of trusted platform module (TPM) library APIs. In: Chiasson, S., Kapadia, A. (eds.) Eighteenth Symposium on Usable Privacy and Security, SOUPS 2022, Boston, MA, USA, August 7-9, 2022. pp. 213–232. USENIX Association (2022), https://www.usenix.org/conference/soups2022/presentation/rao
16. Robles, V., Kosmatov, N., Prevosto, V., Rilling, L., Gall, P.L.: MetAcsl: Specification and verification of high-level properties. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019). LNCS, vol. 11427, pp. 358–364. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_22

17. Robles, V., Kosmatov, N., Prevosto, V., Rilling, L., Le Gall, P.: Tame your annotations with MetAcsl: Specifying, testing and proving high-level properties. In: Proc. of the 13th International Conference on Tests and Proofs (TAP 2019). LNCS, vol. 11823, pp. 167–185. Springer (2019). https://doi.org/10.1007/978-3-030-31157-5_11
18. Robles, V., Kosmatov, N., Prevosto, V., Rilling, L., Le Gall, P.: Methodology for specification and verification of high-level properties with MetAcsl. In: 9th IEEE/ACM International Conference on Formal Methods in Software Engineering (FormaliSE 2021). pp. 54–67. IEEE (2021). https://doi.org/10.1109/FormaliSE52586.2021.00012
19. Shao, J., Qin, Y., Feng, D.: Formal analysis of HMAC authorisation in the TPM2.0 specification. IET Inf. Secur. **12**(2), 133–140 (2018). https://doi.org/10.1049/iet-ifs.2016.0005
20. Signoles, J.: The E-ACSL perspective on runtime assertion checking. In: ACM International Workshop on Verification and mOnitoring at Runtime EXecution (VORTEX). pp. 8–12. ACM, New York (2021). https://doi.org/10.1145/3464974.3468451
21. Trusted Computing Group: Trusted Platform Module Library Specification, Family "2.0", Level 00, Revision 01.59 – November. https://trustedcomputinggroup.org/work-groups/trusted-platform-module/ (2019), last accessed: May 2023
22. Wang, W., Qin, Y., Yang, B., Zhang, Y., Feng, D.: Automated security proof of cryptographic support commands in TPM 2.0. In: Information and Communications Security (ICICS). LNCS, vol. 9977, pp. 431–441. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50011-9_33
23. Zhang, Q., Zhao, S.: A comprehensive formal security analysis and revision of the two-phase key exchange primitive of TPM 2.0. Comput. Networks **179** (2020). https://doi.org/10.1016/j.comnet.2020.107369
24. Ziani, Y., Kosmatov, N., Loulergue, F., Pérez, D.G., Bernier, T.: Towards formal verification of a TPM software stack. In: Integrated Formal Methods (iFM). LNCS, vol. 14300, pp. 93–112. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-47705-8_6