# Towards Formal Verification of a 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], Daniel Gracia Pérez[1][0000-0002-5364-8244], and Téo Bernier[1][0009-0003-4834-7126]

[1] Thales Research & Technology, Palaiseau, France
`{yani.ziani,nikolai.kosmatov,daniel.gracia-perez,teo.bernier}@thalesgroup.com`
[2] Université d'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 protect integrity and security 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*. Vulnerabilities in its code could allow attackers to recover sensitive information and take control of the system. This paper describes a case study on formal verification of tpm2-tss using the Frama-C verification platform. Heavily based on linked lists and complex data structures, the library code appears to be highly challenging for the verification tool. We present several issues and limitations we faced, illustrate them with examples and present solutions that allowed us to verify functional properties and the absence of runtime errors for a representative subset of functions. We describe verification results and desired tool improvements necessary to achieve a full formal verification of the target code.

## 1 Introduction

The *Trusted Platform Module* (TPM) [20] has become a key security component in modern computers. The TPM is a cryptoprocessor designed to protect integrity of the architecture and ensure security of encryption keys stored in it. The operating system and applications communicate with the TPM through a set of APIs called *TPM Software Stack* (TSS). A popular implementation of the TSS is the open-source library *tpm2-tss*. It is highly critical: vulnerabilities in its code could allow attackers to recover sensitive information and take control of the system. Hence, it is important to formally verify that the library respects its specification and does not contain runtime errors, often leading to security vulnerabilities, for instance, exploiting buffer overflows or invalid pointer accesses. Formal verification of this library is the main motivation of this work. This target is new and highly ambitious for deductive verification: the library code is very large for a formal verification project (over 120,000 lines of C code). It is also highly complex, heavily based on complex data structures (with multiple levels of nested structures and unions), low-level code, calls to external (e.g. cryptography) libraries, linked lists and dynamic memory allocation.

In this paper we present a first case study on formal verification of tpm2-tss using the Frama-C verification platform [15]. We focus on a subset of functions involved in storing an encryption key in the TPM, one of the most critical features of the TSS. We verify both functional properties and the absence of runtime errors. The functions are annotated in the acsl specification language [2]. Their verification with Frama-C currently faces several limitations of the tool, such as its capacity to reason about complex data structures, dynamic memory allocation, linked lists and their separation from other data. We have managed to overcome these limitations after minor simplifications and adaptations of the code. In particular, we replace dynamic allocation with `calloc` by another allocator (attributing preallocated memory cells) that we implement, specify and verify. We adapt a recent work on verification of linked lists [4] to our case study, add new lemmas and prove them in the Coq proof assistant [19]. We identify some deficiencies in the new Frama-C–Coq extraction for lists (modified since [4]), adapt it for the proof and suggest improvements. We illustrate all issues and solutions on a simple illustrative example while the (slightly adapted) real-life functions annotated in acsl and fully proved in Frama-C are available online as a companion artifact[3]. Finally, we identify desired extensions and improvements of the verification tool.

*Contributions.* The contributions of this paper include the following:

– specification and formal verification in Frama-C of a representative subset of functions of the tpm2-tss library (slightly adapted for verification);
– presentation of main issues we faced during their verification with an illustrative example, and description of solutions and workarounds we found;
– proof in Coq of all necessary lemmas (including some new ones) related to linked lists, realized for the new version of Frama-C–Coq extraction;
– a list of necessary enhancements of Frama-C to achieve a complete formal verification of the tpm2-tss library.

*Outline.* The paper is organized as follows. Section 2 presents Frama-C. Section 3 introduces the TPM, its software stack and the tpm2-tss library. Sections 4 and 5 present issues and solutions related, resp., to memory allocation and memory management. Necessary lemmas are discussed in Sect. 6. Section 7 describes our verification results. Finally, Sect. 8 and 9 present related work and a conclusion with necessary tool improvements.

## 2 Frama-C Verification Platform

Frama-C [15] is an open-source verification platform for C code, which contains various plugins built around a kernel providing basic services for source-code analysis. It offers acsl (ANSI/ISO C Specification Language) [2], a formal

[3] Available (with the illustrative example, all necessary lemmas and their proofs) on `https://doi.org/10.5281/zenodo.8273295`.

specification language for C, that allows users to specify functional properties of programs in the form of *annotations*, such as assertions or function contracts. A function contract basically consists of pre- and postconditions (stated, resp., by `requires` and `ensures` clauses) expressing properties that must hold, resp., before and after a call to the function. It also includes an `assigns` clause listing (non-local) variables and memory locations that *can* be modified by the function. While useful built-in predicates and logic functions are provided to handle properties such as pointer validity or memory separation for example, ACSL also supplies the user with different ways to define predicates and logic functions.

FRAMA-C offers WP, a plugin for deductive verification. Given a C program annotated in ACSL, WP generates the corresponding proof obligations (also called verification conditions) that can be proved either by WP or, via the WHY3 platform [13], by SMT solvers or an interactive proof assistant like COQ [19]. To ensure the absence of runtime errors (RTE), WP can automatically add necessary assertions via a dedicated option, and try to prove them as well.

Our choice to use FRAMA-C/WP is due to its capacity to perform deductive verification of industrial C code with successful verification case studies [7] and the fact that it is currently the only tool for C source code verification recognized by ANSSI, the French Common Criteria certification body, as an acceptable formal verification technique for the highest certification levels EAL6–EAL7 [8].

## 3 The TPM Software Stack and the tpm2-tss Library

This section briefly presents the Trusted Platform Module (TPM), its software stack and the implementation we chose to study: the tpm2-tss library. Readers can refer to the TPM specification [20] and reference books as [1] for more detail.

*TPM Software Stack.* The TPM is a standard conceived by the Trusted Computing Group (TCG)[4] for a passive secure cryptoprocessor designed to protect secure hardware from software-based threats. At its base, a TPM is implemented as a discrete cryptoprocessor chip, attached to the main processor chip and designed to perform cryptographic operations. However, it can also be implemented as part of the firmware of a regular processor or a software component.

Nowadays, the TPM is well known for its usage in regular PCs to ensure integrity and to provide a secure storage for the keys used to encrypt the disk with *Bitlocker* and *dm-crypt*. However, it can be (and is actually) used to provide other cryptographic services to the Operating System (OS) or applications. For that purpose, the TCG defines the TPM Software Stack (TSS), a set of specifications to provide standard APIs to access the functionalities and commands of the TPM, regardless of the hardware, OS, or environment used.

The TSS APIs provide different levels of complexity, 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 services and commands providing greater

[4] `https://trustedcomputinggroup.org/`

flexibility but complexifying its usage. In between lies the Enhanced System API (ESAPI) providing SAPI-like functionalities but with slightly limited flexibility. Other TSS APIs complete the previous ones for common operations like data formatting and connection with the software or hardware TPM.

The TSS APIs, as any software component or the TPM itself, can have vulnerabilities[5] that attackers can exploit to recover sensitive data communicated with the TPM or take control of the system. We study the verification of one of the implementations of the TSS, tpm2-tss, starting more precisely with its implementation of the ESAPI.

*ESAPI Layer of tpm2-tss.* The ESAPI layer provides functions for decryption and encryption, managing session data and policies, thus playing an essential role in the TSS. It is very large (over 50,000 lines of C) and is mainly split into two parts: the API part containing functions in a one-to-one correspondence with TPM commands (for instance, the `Esys_Create` function of the TSS will correspond to — and call — the `TPM2_Create` command of the TPM), and the back-end containing the core of that layer's functionalities. Each API function will call several functions of the back-end to carry out various operations on command parameters, before invoking the lower layers and finally the TPM.

The ESAPI layer relies on a notion of context (`ESYS_CONTEXT`) containing all data the layer needs to store between calls, so it does not need to maintain a global state. Defined for external applications as an opaque structure, the context includes, according to the documentation, data needed to communicate to the TPM, metadata for each TPM resource, and state information. The specification, however, does not impose any precise data structure: it is up to the developer to provide a suitable definition. Our target implementation uses complex data structures and linked lists.

## 4 Dynamic Memory Allocation

*Example Overview.* We illustrate our verification case study with a simplified version of some library functions manipulating linked lists. The illustrative example is split into Fig. 1–6 that will be explained below step-by-step. Its full code being available in the companion artifact, we omit in this paper some less significant definitions and assertions which are not mandatory to understand the paper (but we preserve line numbering of the full example for convenience of the reader). This example is heavily simplified to fit the paper, yet it is representative for most issues we faced (except the complexity of data structures). It contains a main list manipulation function, `getNode` (`esys_GetResourceObject` in the real code), used to search for a resource in the list of resources and return it if it is found, or to create and add it using function `createNode` (`esys_CreateResourceObject` in the real code) if not.

Figure 1 provides the linked list structure as well as logic definitions used to handle logic lists in specifications. Our custom allocator (used by `createNode`) is

[5] Like CVE-2023-22745 and CVE-2020-24455, documented on `www.cve.org`.

```
   ...
11 typedef struct NODE_T {
12   uint32_t      handle;   // the handle used as reference
13   RESOURCE      rsrc;     // the metadata for this rsrc
14   struct NODE_T *   next; // next node in the list
15 } NODE_T; // linked list of resource
   ...
25 /*@
26   predicate ptr_sep_from_list{L}(NODE_T* e, \list<NODE_T*> ll) =
27     ∀ ℤ n; 0 ≤ n < \length(ll) ⇒ \separated(e, \nth(ll, n));
28   predicate dptr_sep_from_list{L}(NODE_T** e, \list<NODE_T*> ll) =
29     ∀ ℤ n; 0 ≤ n < \length(ll) ⇒ \separated(e, \nth(ll, n));
30   predicate in_list{L}(NODE_T* e, \list<NODE_T*> ll) =
31     ∃ ℤ n; 0 ≤ n < \length(ll) ∧ \nth(ll, n) == e;
32   predicate in_list_handle{L}(uint32_t out_handle, \list<NODE_T*> ll) =
33     ∃ ℤ n; 0 ≤ n < \length(ll) ∧ \nth(ll, n)->handle == out_handle;
34   inductive linked_ll{L}(NODE_T *bl, NODE_T *el, \list<NODE_T*> ll) {
35     case linked_ll_nil{L}: ∀ NODE_T *el; linked_ll{L}(el, el, \Nil);
36     case linked_ll_cons{L}: ∀ NODE_T *bl, *el, \list<NODE_T*> tail;
37       (\separated(bl, el) ∧ \valid(bl) ∧ linked_ll{L}(bl->next, el, tail) ∧
38       ptr_sep_from_list(bl, tail)) ⇒
39         linked_ll{L}(bl, el, \Cons(bl, tail));
40   }
41   predicate unchanged_ll{L1, L2}(\list<NODE_T*> ll) =
42     ∀ ℤ n; 0 ≤ n < \length(ll) ⇒
43       \valid{L1}(\nth(ll,n)) ∧ \valid{L2}(\nth(ll,n)) ∧
44       \at((\nth(ll,n))->next, L1) == \at((\nth(ll,n))->next, L2);
   ...
48   axiomatic Node_To_ll {
49     logic \list<NODE_T*> to_ll{L}(NODE_T* beg, NODE_T* end)
50       reads {node->next | NODE_T* node; \valid(node) ∧
51                           in_list(node, to_ll(beg, end))};
52     axiom to_ll_nil{L}: ∀ NODE_T *node; to_ll{L}(node, node) == \Nil;
53     axiom to_ll_cons{L}: ∀ NODE_T *beg, *end;
54       (\separated(beg, end) ∧ \valid{L}(beg) ∧
55       ptr_sep_from_list{L}(beg, to_ll{L}(beg->next, end))) ⇒
56         to_ll{L}(beg, end) == \Cons(beg, to_ll{L}(beg->next, end));
57   }
58 */
59
60 #include "lemmas_node_t.h"
```

**Fig. 1.** Linked list and logic definitions.

defined in Fig. 2. Figure 3 defines a (simplified) context and additional logic definitions to handle pointer separation and memory freshness. The search function is shown in Fig. 4 and 5. As it is often done, some ACSL notation (e.g. `\forall`, `integer`, `==>`, `<=`, `!=`) is pretty-printed (resp., as ∀, ℤ, ⇒, ≤, ≠). In this section, we detail Fig. 1–3.

*Lists of Resources.* Lines 11–15 of Fig. 1 show a simplified definition of the linked list of resources used in the ESAPI layer of the library. Each node of the list consists of a structure containing a handle used as a reference for this node, a resource to be stored inside, and a pointer to the next element. The handle is supposed to be unique[6]. In our example, a resource structure (omitted in Fig. 1) is assumed to contain only a few fields of relatively simple types. The real code uses a more extensive and complex definition (with several levels of nested structures and unions), covering all possible types of TPM resources. While it

[6] This uniqueness is currently not yet specified in the ACSL contracts.

does add some complexity to prove certain properties (as some of them may require to completely unfold all resource substructures), it does not introduce new pointers that may affect memory separation properties, so our example remains representative of the real code regarding linked lists and separation properties.

In particular, we need to ensure that the resource list is well-formed — that is, it is not circular, and does not contain any overlap between nodes — and stays that way throughout the layer. To accomplish that, we use and adapt the logic definitions from [4], given on lines 26–44, 48–57 of Fig. 1. To prove the code, we need to manipulate linked lists and segments of linked lists. Lines 48–57 define the *translating function* `to_ll` that translates a C list defined by a `NODE_T` pointer into the corresponding ACSL logic list of (pointers to) its nodes. By convention, the last element `end` is not included into the resulting logic list. It can be either `NULL` for a full linked list, or a non-null pointer to a node for a *linked list segment* which stops just before that node. Lines 34–40 show the *linking predicate* `linked_ll` establishing the equivalence between a C linked list and an ACSL logic list. This inductive definition includes memory separation between nodes, validity of access for each node, as well as the notion of reachability in linked lists. In ACSL, given two pointers `p` and `q`, `\valid(p)` states that `*p` can be safely read and written, while `\separated(p,q)` states that the referred memory locations `*p` and `*q` do not overlap (i.e. all their bytes are disjoint).

Lines 26–29 provide predicates to handle separation between a list pointer (or double pointer) and a full list. `\nth(l,n)` and `\length(l)` denote, resp., the `n`-th element of logic list `l` and the length of `l`. The predicate `unchanged_ll` in lines 41–44 states that between two labels (i.e. program points) `L1` and `L2`, all list elements in a logic list refer to a valid memory location at both points, and that their respective next fields retain the same value. It is used to maintain the structure of the list throughout the code. Line 60 includes lemmas necessary to conduct the proof, further discussed in Sec. 6.

*Lack of Support for Dynamic Memory Allocation.* As mentioned above, per the TSS specifications, the ESAPI layer does not maintain a global state between calls to TPM commands. The library code uses contexts with linked lists of TPM resources, so list nodes need to be dynamically allocated at runtime. The ACSL language provides clauses to handle memory allocations: in particular, `\allocable{L}(p)` states that a pointer `p` refers to the base address of an unallocated memory block, and `\fresh{L1,L2}(p, n)` indicates that p refers to the base address of an unallocated block at label `L1`, and to an allocated memory block of size `n` at label `L2`. Unfortunately, while the FRAMA-C/WP memory model[7] is able to handle dynamic allocation (used internally to manage local variables), these clauses are not currently supported. Without allocability and freshness, proving goals involving validity or separation between a newly allocated node and any other pointer is impossible.

[7] that is, intuitively, the way in which program variables and memory locations are internally represented and manipulated by the tool.

```
62  #define _alloc_max 100
63  static NODE_T _rsrc_bank[_alloc_max];  // bank used by the static allocator
64  static int _alloc_idx = 0;  // index of the next rsrc node to be allocated
65  /*@
66    predicate valid_rsrc_mem_bank{L} = 0 ≤ _alloc_idx ≤ _alloc_max;
67    predicate list_sep_from_allocables{L}(\list<NODE_T*> ll) =
68      ∀ int i; _alloc_idx ≤ i < _alloc_max ⇒
69                          ptr_sep_from_list{L}(&_rsrc_bank[i], ll);
70    predicate ptr_sep_from_allocables{L}(NODE_T* node) =
71      ∀ int i; _alloc_idx ≤ i < _alloc_max ⇒ \separated(node, &_rsrc_bank[i]);
72    predicate dptr_sep_from_allocables{L}(NODE_T** p_node) =
73      ∀ int i; _alloc_idx ≤ i < _alloc_max ⇒ \separated(p_node, &_rsrc_bank[i]);
74  */
    ...
76  /*@
77    requires valid_rsrc_mem_bank;
78    assigns _alloc_idx, _rsrc_bank[\old(_alloc_idx)];
79    ensures valid_rsrc_mem_bank;
  ...
89    behavior allocable:
90      assumes 0 ≤ _alloc_idx < _alloc_max;
91
92      ensures _alloc_idx == \old(_alloc_idx) + 1;
93      ensures \result == &(_rsrc_bank[ _alloc_idx - 1]);
94      ensures \valid(\result);
95      ensures zero_rsrc_node_t( *(\result) );
96      ensures ∀ int i; 0 ≤ i < _alloc_max ∧ i ≠ \old(_alloc_idx) ⇒
97              _rsrc_bank[i] == \old(_rsrc_bank[i]);
98    disjoint behaviors; complete behaviors;
99  */
100 NODE_T *calloc_NODE_T()
101 {
102   static const RESOURCE empty_RESOURCE;
103   if(_alloc_idx < _alloc_max) {
104     _rsrc_bank[_alloc_idx].handle = (uint32_t) 0;
105     _rsrc_bank[_alloc_idx].rsrc = empty_RESOURCE;
106     _rsrc_bank[_alloc_idx].next = NULL;
107     _alloc_idx += 1;
108     return &_rsrc_bank[_alloc_idx - 1];
109   }
110   return NULL;
111 }
```

**Fig. 2.** Allocation bank and static allocator.

*Static Memory Allocator.* To circumvent that issue, we define in Fig. 2 a bank-based static allocator `calloc_NODE_T` that replaces calls to `calloc` used in the real-life code. It attributes preallocated cells, following some existing implementations (like the memb module of Contiki [17]). Line 63 defines a node bank, that is, a static array of nodes of size `_alloc_max`. Line 64 introduces an allocation index we use to track the next allocable node and to determine whether an allocation is possible. Predicate `valid_rsrc_mem_bank` on line 66 states a validity condition for the bank: `_alloc_idx` must always be between 0 and `_alloc_max`. It is equal to the upper bound if all nodes have been allocated. Predicates lines 67–73 specify separation between a logic list of nodes (resp., a pointer or a double pointer to a node) and the allocable part of the heap, and is used later on to simulate memory freshness.

Lines 76–99 show a part of the function contract for the allocator defined on lines 100–111. The validity of the bank should be true before and after the function execution (lines 77, 79). Line 78 specifies the variables the function is allowed to modify. The contract is specified using several cases (called *behaviors*). Typically, a behavior considers a subset of possible input states (respecting its `assumes` clause) and defines specific postconditions that must be respected for this subset of inputs. In our case, the provided behaviors are complete (i.e. cover all states allowed by the function precondition) and their corresponding subsets are disjoint (line 98). We show only one behavior (lines 89–97) describing a successful allocation (when an allocable node exists, as stated on line 90). Postconditions on lines 92–93 ensure the tracking index is incremented by one, and that the returned pointer points to the first allocable block. While this fact is sufficient to deduce the validity clause on line 94, we keep the latter as well (and it is actually expected for any allocator). In the same way, lines 96–97 specify that the nodes of the bank other than the newly allocated block have not been modified[8].

Currently, Frama-C/Wp does not offer a memory model able to handle byte-level assignments in C objects. To represent as closely as possible the fact that allocated memory is initialized to zero by a call to `calloc` in the real-life code, we initialize each field of the allocated node to zero (see the C code on lines 104–106 and the postcondition on line 95).

*Contexts, Separation Predicates and Freshness.* In the target library (and in our illustrative example), pointers to nodes are not passed directly as function arguments, but stored in a context variable, and a pointer to the context is passed as a function argument. Lines 113–116 of Fig. 3 define a simplified context structure, comprised of an `int` and a `NODE_T` pointer to the head of a linked list of resources.

Additional predicates to handle memory separation and memory freshness are defined on lines 118–132. In particular, the `ctx_sep_from_list` predicate on lines 118–119 specifies memory separation between a `CONTEXT` pointer and a logic list of nodes. Lines 120–121 define separation between such a pointer and allocables nodes in the bank.

In C, a successful dynamic allocation of a memory block implies its *freshness*, that is, the separation between the newly allocated block (typically located on the heap) and all pre-existing memory locations (on the heap, stack or static storages). As this notion of freshness is currently not supported by Frama-C/Wp, we have to simulate it in another way. Our allocator returns a cell in a static array, so other global variables — as well as local variables declared within the scope of a function — will be separated from the node bank. To obtain a complete freshness within the scope of a function, we need to maintain separation between the allocable part of the bank and other memory locations accessible through pointers. In our illustrative example, pointers come from arguments

[8] This property is partly redundant with the assigns clause on line 78 but its presence facilitates the verification.

```
113 typedef struct CONTEXT {
114   int placeholder_int;
115   NODE_T *rsrc_list;
116 } CONTEXT;
117 /*@
118   predicate ctx_sep_from_list(CONTEXT *ctx, \list<NODE_T*> ll) =
119     ∀ ℤ i; 0 ≤ i < \length(ll) ⇒ \separated(\nth(ll, i), ctx);
120   predicate ctx_sep_from_allocables(CONTEXT *ctx) =
121     ∀ int i; _alloc_idx ≤ i < _alloc_max ⇒ \separated(ctx, &_rsrc_bank[i]);
122 
123   predicate freshness(CONTEXT * ctx, NODE_T ** node) =
124     ctx_sep_from_allocables(ctx)
125     ∧ list_sep_from_allocables(to_ll(ctx->rsrc_list, NULL))
126     ∧ ptr_sep_from_allocables(ctx->rsrc_list)
127     ∧ ptr_sep_from_allocables(*node)
128     ∧ dptr_sep_from_allocables(node);
129 
130   predicate sep_from_list{L}(CONTEXT * ctx, NODE_T ** node) =
131     ctx_sep_from_list(ctx, to_ll{L}(ctx->rsrc_list, NULL))
132     ∧ dptr_sep_from_list(node, to_ll{L}(ctx->rsrc_list, NULL));
133 */
```

**Fig. 3.** Context and predicates to handle separation from a list and memory freshness.

including a pointer to a `CONTEXT` object (and pointers accessible from it) and a double pointer to a `NODE_T` node. This allows us to define a predicate to handle freshness in both function contracts.

The `freshness` predicate on lines 123–128 of Fig. 3 specifies memory separation between known pointers within the scope of our functions and the allocable part of the bank, using separation predicates previously defined on lines 120–121, and on lines 67–73 of Fig 2. This predicate will become unnecessary as soon as dynamic allocation is fully supported by Frama-C/Wp.

In the meanwhile, a static allocator with an additional separation predicate simulating freshness provides a reasonable solution to verify the target library. Since no specific constraint is assumed in our contracts on the position of previously allocated list nodes already added to the list, the verification uses a specific position in the bank only for the newly allocated node. The fact that the newly allocated node does not become valid during the allocation (technically, being part of the bank, it was valid in the sense of acsl already before) is compensated in our contracts by the freshness predicate stating that the new node — as one the allocable nodes — was not used in the list before the allocation (cf. line 310 in Fig. 4). We expect that the migration from our specific allocator to a real-life dynamic allocator — with a more general contract — will be very easy to perform, as soon as necessary features are supported by Frama-C.

Similarly, the `sep_from_list` predicate on lines 130–132 specifies separation between the context's linked list and known pointers, using predicates on lines 118–119, and on lines 28–29 of Fig 1.

## 5 Memory Management

This section presents how we use the definitions introduced in Sec. 4 to prove selected ESAPI functions involving linked lists. We also identify separation issues related to limitations of the Typed memory model of Wp, as well as a way to manage memory to overcome such issues. In this section, we detail Fig. 4–6.

*The Search Function.* Figure 4 provides the search operation `getNode` with a partial contract illustrating functional and memory safety properties we aim to verify and judge necessary for the proof at a larger scale. Some proof-guiding annotations (assertions, loop contracts) have been skipped for readability, but the code is preserved (mostly with the same line numbers). The arguments include a context, a handle to search and a double pointer for the returned node.

Lines 380–416 perform the search of a node by its handle: variable `temp_node` iterates over the nodes of the resource list, and the node is returned if its handle is equal to the searched one (in which case, the function returns 616 for success).

Lines 420–430 convert the resource handle to a TPM one, call the creation function to allocate a new node and add it to the list as its new head with the given handle if the allocation was successful (and return `833` if not). The new node is returned by `createNode` in `temp_node_2` (again via a double pointer).

Lines 435–462 perform some modifications on the content of the newly allocated node, without affecting the structure of the list. An error code is returned in case of a failure, and 1611 (with the allocated node in `*node`) otherwise. Lines 450–451, 453–454 and 461 provide some assertions to propagate information to the last return clause of the function, attained in case of the successful addition of the new element to the list.

Compared to the real-life code, we have introduced anonymous blocks on lines 380–416 and 422–452 (which are not semantically necessary and were not present in the original code), as well as two local variables `tmp_node` and `tmp_node2` instead of only one. We explain these code adaptations below.

*Contract of the Search Function.* Lines 309–375 of Fig. 4 provide a partial function contract, illustrating two behaviors of `getNode`: if the element was found by its handle in the list (cf. lines 325–326), and if the element was not found at first, but was then successfully allocated and added (cf. lines 355–359), for each of them specific postconditions are stated. For instance, for the latter behavior, lines 369–370 ensure that if a new node was successfully allocated and added to the list, the old head becomes the second element of the list, while line 372 ensures the separation of known pointers from the new list. We specify that the complete list of provided behaviors must be complete and disjoint (line 374).

As global preconditions, we notably require for the list to be well-formed (through the use of the linking predicate, cf. line 313), and the validity of our bank and freshness of allocable nodes with respect to function arguments and global variables (cf. line 310). Line 317 requires memory separation of known pointers from the list of resources using the `sep_from_list` predicate, and separation among known pointers using the `\separated` predicate.

```
309 /*@
310   requires valid_rsrc_mem_bank{Pre} ∧ freshness(ctx, node);
313   requires linked_ll(ctx->rsrc_list, NULL, to_ll(ctx->rsrc_list, NULL));
317   requires sep_from_list(ctx, node) ∧ \separated(node, ctx);
...
321   ensures valid_rsrc_mem_bank ∧ freshness(ctx, node);
...
325   behavior handle_in_list:
326     assumes in_list_handle(rsrc_handle, to_ll(ctx->rsrc_list, NULL));
...
332     ensures unchanged_ll{Pre, Post}(to_ll(ctx->rsrc_list, NULL));
333     ensures \result == 616;
...
355   behavior handle_not_in_list_and_node_allocated:
356     assumes !(in_list_handle(rsrc_handle, to_ll(ctx->rsrc_list, NULL)));
357     assumes rsrc_handle ≤ 31U ∨ (rsrc_handle \in {0x10AU, 0x10BU})
358             ∨ (0x120U ≤ rsrc_handle ≤ 0x12FU);
359     assumes 0 ≤ _alloc_idx < _alloc_max;
...
369     ensures \old(ctx->rsrc_list) ≠ NULL ⇒
370             \nth(to_ll(ctx->rsrc_list, NULL), 1) == \old(ctx->rsrc_list);
371     ensures linked_ll(ctx->rsrc_list, NULL, to_ll(ctx->rsrc_list, NULL));
372     ensures sep_from_list(ctx, node);
373     ensures \result == 1611;
374   disjoint behaviors; complete behaviors;
375 */
376 int getNode(PSEUDO_CONTEXT *ctx, uint32_t rsrc_handle, NODE_T ** node) {
377   /*@ assert linked_ll(ctx->rsrc_list, NULL, to_ll(ctx->rsrc_list, NULL));*/
378   int r;
379   uint32_t tpm_handle;
380   { /* Block added to circumvent issues with the WP memory model */
381     NODE_T *tmp_node;
401     for (tmp_node = ctx->rsrc_list; tmp_node ≠ NULL;
402         tmp_node = tmp_node->next) {
405       if (tmp_node->handle == rsrc_handle){*node = tmp_node; return 616;}
415     }
416   }
420   r = iesys_handle_to_tpm_handle(rsrc_handle, &tpm_handle);
422   { /* Block added to circumvent issues with the WP memory model */
423     NODE_T *tmp_node_2 = NULL;
428     r = createNode(ctx, rsrc_handle, &tmp_node_2);
429     /*@ assert sep_from_list(ctx, node);*/
430     if (r == 833) {return r;};
435     tmp_node_2->rsrc.handle = tpm_handle;
436     tmp_node_2->rsrc.rsrcType = 0;
437     size_t offset = 0;
440     r = uint32_Marshal(tpm_handle, &tmp_node_2->rsrc.name.name[0],
441                        sizeof(tmp_node_2->rsrc.name.name),&offset);
443     if (r ≠0) {return r;};
444     tmp_node_2->rsrc.name.size = offset;
449     *node = tmp_node_2;
450     /*@ assert unchanged_ll{Pre, Here}(
451                to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
452   }
453   /*@ assert unchanged_ll{Pre, Here}(
454              to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
461   /*@ assert sep_from_list(ctx, node);*/
462   return 1611;
463 }
```

**Fig. 4.** The (slightly rewritten) search function, where some annotations are removed.

As a global postcondition, we require that our bank stays valid, and that freshness of the (remaining) allocable nodes relatively to function arguments and global variables is maintained (cf. line 321). However, properties regarding the list itself — such as the preservation of the list when it is not modified (line 332), or ensuring it remains well-formed after being modified (line 371) — have to be issued to ACSL behaviors to be proved, due to the way how local variables are handled in the memory model of WP. The logic list properties are much more difficult for solvers to manipulate in global behaviors.

*Memory Model Limitation: an Uprovable Property.* Consider the assertion on line 377 of Fig. 4. Despite the presence of the same property as a precondition of the function (line 313), currently this assertion cannot be proved by WP at the entry point for the real-life version of the function. Basically, the real-life version can be obtained[9] from Fig. 4 by removing the curly braces on lines 380, 416, 422, 452. This issue is due to a limitation of the WP memory model.

Indeed, for such an assertion (as in general for any annotation to be proved), WP generates a proof obligation, to be proved by either WP itself or by external provers via the WHY3 platform [13]. Such an obligation includes a representation of the current state of the program memory. In particular, pointers such as the resource list `ctx->rsrc_list` (and by extension, any reachable node of the list) will be considered part of the heap. To handle the existence of a variable in memory — should it be the heap, the stack or the static segments — WP uses an allocation table to express when memory blocks are used or freed, which is where the issue lies. For instance, on line 428 of Fig. 4, the `temp_node_2` pointer has its address taken, and is considered as used locally due to `requires` involving it in our function contract for `createNode`. It is consequently transferred to the memory model, where it has to be allocated.

Currently, the memory model of WP does not provide separated allocation tables for the heap, stack and static segments. Using `temp_node_2` the way it is used on line 428 changes the modification status of the allocation table, which is then considered as modified as a whole. This affects the status of other "allocated" (relatively to the memory model) variables as well, including (but not limited to) any reachable node of the list.

Therefore, the call to `createNode` line 428 of Fig. 4 in the real-life code that uses the address of a local pointer as a third argument is sufficient to affect the status of the resource list on the scale of the entire function. As a result, the assertion on line 377 is not proved.

*A Workaround.* As a workaround (found thanks to an indication of the WP team) to the aforementioned issue, we use additional blocks and variable declarations. Figure 5 presents those minor rewrites (with line numbers in alphabetical style to avoid confusion with the illustrative example). The left side illustrates the structure of the original C code, where the address of `temp_node` is taken and used in the `createNode` call on line j, and the same pointer is used to iterate on the list. On the right, we add additional blocks and a new pointer `temp_node_2`,

[9] another difference — removing variable `tmp_node2` declared on line 423 and using `tmp_node` instead — can be ignored in this context.

```
int getNode(..., NODE_T ** node){
  // list properties unprovable
  int r;

  NODE_T *tmp_node;
  ... // iterate over the list



  r = createNode(..., &tmp_node);
  ...
  *node = tmp_node;

  return 1611;
}
```

```
int getNode(..., NODE_T ** node){
  // list properties proved
  int r;
  {
    NODE_T *tmp_node;
    ... // iterate over the list
  }
  {
    NODE_T *tmp_node_2 = NULL;
    r = createNode(..., &tmp_node_2);
    ...
    *node = tmp_node_2;
  }
  return 1611;
}
```

**Fig. 5.** Comparison of the real-life code of `getNode` (on the left) and its rewriting with additional blocks (on the right) for proving list properties.

```
/*@
  requires \valid(src) ∧ \valid(dest + (0 .. sizeof(*src)-1));
...
*/
void memcpy_custom(uint8_t *dest, uint32_t *src) {
  dest[3] = (uint8_t)(*src & 0xFF);
  dest[2] = (uint8_t)((*src >> 8) & 0xFF);
  dest[1] = (uint8_t)((*src >> 16) & 0xFF);
  dest[0] = (uint8_t)((*src >> 24) & 0xFF);
}
...
int uint32_Marshal(uint32_t in,uint8_t buff[],size_t buff_size,size_t *offset){
  size_t  local_offset = 0;
...
  // memcpy(&buff[local_offset], &in, sizeof (in));
  memcpy_custom(&buff[local_offset], &in);
...
}
```

**Fig. 6.** Definition for `memcpy` replacement in marshal.

initialized to `NULL` to match the previous iteration over the list. Each block defines a new scope, outside of which the pointer used by `createNode` will not exist and side-effect-prone allocations will not happen. It solves the issue.

*Additional Proof-Guiding Annotations.* Additional annotations (mostly omitted in Fig. 4) include, as usual, loop contracts and a few assertions. Assertions can help the tool to establish necessary intermediate properties or activate the application of relevant lemmas. For instance, assertions of lines 450–451 and 453–454 help propagate information over the structure of the linked list (by its logic list representation) outside of each block, and finally to postconditions. Assertions on lines 429 and 461 help propagate separations from the list through the function and its anonymous blocks. Some other intermediate assertions are needed to prove the unchanged nature of the list. Such additional assertions can be tricky to find in some cases and need some experience.

*Handling Pointer Casts.* Another memory manipulation issue we have encountered comes from the function call on line 440 in `getNode`: after having been added to the resource list, the newly allocated node must have its name (or more precisely, the name of its resource) set from its TPM handle `tpm_handle` (derived from the handle of the node by the function call on line 420). This is done through marshaling using the `uint32_Marshal` function, partially shown on lines 298–306 of Fig. 6, whose role is to store a 4-byte unsigned int (in this case, our TPM handle) in a flexible array of bytes (the name of the resource). The function calls `memcpy` on (commented) line 302, which is the source of our issue (a correct endianness being ensured by a previous byte swap in `in`).

For most functions of the standard libraries, Frama-C provides basic acsl contracts to handle their use. However, for memory manipulation functions like `memcpy`, such contracts rely on pointer casts, whose support in Wp is currently limited. To circumvent this issue, we define our own memory copy function on lines 280–285: instead of directly copying the 4-byte unsigned int pointed by `src` byte per byte using pointer casts using `memcpy`, we extract one-byte chunks using byte shifts and bitmasks (cf. lines 281–284, 303) without casts. Line 272 requires that both source and destination locations are valid, also without casts. This version is fully handled by Wp. Current contracts are sufficient for the currently considered functional properties and the absence of runtime errors (and we expect they will be easy to extend for more precise properties if needed).

## 6 Lemmas

When SMT solvers become inefficient (e.g. for inductive definitions), it can be necessary to add lemmas to facilitate the proof. These lemmas can then be directly instantiated by solvers, but proving them often requires to reason by induction, with an interactive proof assistant.

The previous work using logic lists [4] defined and proved several lemmas using the Coq proof assistant. We have added two new useful lemmas (defined in Fig. 7) and used twelve of the previous ones to verify both the illustrative example and the subset of real-life functions. However, because the formalization of the memory models and various aspects of acsl changed between the version of Frama-C used in the previous work and the one we use, we could not reuse the proofs of these lemmas. While older Frama-C versions directly generated Coq specifications, more recent Frama-C versions let Why3 generate them. Even if the new translation is close to the previous one, the way logic lists are handled was modified significantly.

In the past, Frama-C logic lists were translated into the lists Coq offers in its standard library: an inductively defined type as usually found in functional programming languages such as OCaml and Haskell. Such types come with an induction principle that allows to reason by induction. Without reasoning inductively, it also offers the possibility to reason by case on lists: a list is defined either as empty, or as built with the `cons` constructor. In recent versions of Frama-C, acsl logic lists are axiomatized as follows: two functions `nil` and `cons` are

```
lemma in_next_not_bound_in{L}: ∀ NODE_T *bl, *el, *item, \list<NODE_T*> ll;
  linked_ll(bl, el, ll) ⇒ in_list(item, ll) ⇒ item->next ≠ el ⇒
    in_list(item->next, ll);
lemma linked_ll_split_variant{L}:
  ∀ NODE_T *bl, *bound, *el, \list<NODE_T*> l1, l2;
  linked_ll(bl, el, l1 ^ l2) ⇒ l2 ≠ \Nil ⇒
  bound == \nth(l1 ^ l2, \length(l1 ^ l2) - \length(l2)) ⇒
    linked_ll(bl, bound, l1) ∧ linked_ll(bound, el, l2);
```

**Fig. 7.** New lemmas proved in our verification work (in addition to those in [4]).

declared, as well as a few other functions on lists, including the length of a list (`length`), the concatenation of two lists (`concat`), and getting an element from a list given its position (`nth`). However, there is no induction principle to reason by induction on lists, and because `nil` and `cons` are not constructors, it is not possible to reason by case on lists in this formalization. It is possible to test if a list is empty, but if not, we do not know that it is built with `cons`. Writing new recursive functions on such lists is also very difficult. Indeed, we only have `nth` to observe a list, while the usual way to program functions on lists uses the head and the tail of a list for writing the recursive case.

Interestingly, when the hypotheses of our lemmas include a fact expressed using `linked_ll`, it is still possible to reason by case, because this inductive predicate is translated into Coq as an inductive predicate. Consequently, there are only two possible cases for the logic list: either it is empty, or it is built with `cons`. When such a hypothesis is missing, we axiomatized a `tail` function, and a decomposition principle stating that a list is either `nil` or `cons`. These axioms are quite classic and can be implemented using a list type defined by induction. We did not need an inductive principle on logic lists as either the lemmas did not require a proof by induction, or we reasoned inductively on the inductive predicate `linked_ll`. However, we proved such an induction principle using only the axioms we added. It is thus available to prove some other lemmas provided in [4] — not needed yet in our current work — that were proved by induction on lists.

Because of these changes, to prove all lemmas we need, we had to adapt all previous proof scripts, and in a few cases significantly. The largest proof scripts are about 100 lines long excluding our axioms, and the shortest takes a dozen lines. We suggest that the next versions of Frama-C come back to a concrete representation of lists. Thanks to our approach, we expect that the required changes in our proofs of lemmas will remain minimal: we will only have to prove the axioms introduced on `tail` and our decomposition principle.

## 7 Verification Results

Proof results, presented in Fig. 8, were obtained by running Frama-C 26.1 (Iron) on a desktop computer running Ubuntu 20.04.4 LTS, with an Intel(R) Core(TM) i5-6600 CPU @ 3.30 GHz, featuring 4 cores and 4 threads, with 16GB

| | | User-provided ACSL | RTE | Total | |
|---|---|---|---|---|---|
| Code subset | Prover | #Goals | #Goals | #Goals | Time |
| Illustrative example | Qed | 105 | 18 | 123 (43.62%) | |
| | Script | 1 | 0 | 1 (0.35%) | |
| | SMT | 137 | 21 | 158 (56.03%) | |
| | All | 243 (86.17%) | 39 (13.83%) | **282** | **5m13s** |
| Library code subset | Qed | 274 | 38 | 312 (47.34%) | |
| | Script | 5 | 0 | 5 (0.76%) | |
| | SMT | 311 | 31 | 342 (51.90%) | |
| | All | 590 (89.53%) | 69 (10.47%) | **659** | **18m07s** |

**Fig. 8.** Proof results for the illustrative example and the real-life code.

RAM. We ran Frama-C with options `-wp-par 3` and `-wp-timeout 30`. We used the Alt-Ergo v2.4.3 and CVC4 v1.8 solvers, via Why3 v1.5.1. Both functional properties and the absence of runtime errors (RTE) were proved. Assertions to ensure the absence of runtime errors are automatically generated by the Rte plugin of Frama-C (using the `-wp-rte` option). Functional properties include usual properties such as the fact that the well-formedness of the list is preserved, that a new resource has been successfully added to the resource list, that the searched element is correctly found if present, etc.

In our illustrative example, 282 goals were proved in a total time of 5min13s with 56% proved by SMT solvers, and the rest by the internal simplifier engine Qed of Wp and one Wp script. The maximum time to prove a goal was 20s.

Solutions to memory manipulation problems presented in this paper were used on a larger verification study over 10 different functions of the target library (excluding macro functions, and interfaces without code whose behaviors needed to be modeled in acsl), related to linked-list manipulations and some internal ESAPI feasibility checks and operations (cryptographic operations excluded). Over 659 goals proved in a total of 18m07s, 52% were proved by SMT solvers and 47% by Qed. Only 5 Wp proof scripts were used, when automatic proof either failed or was too slow. This shows a high level of automation achieved in our project, in particular, thanks to carefully chosen predicates and lemmas (which are usually tricky to find for the first time and can be useful in other similar projects). The maximum time to prove a goal was 1min50s.

We also used smoke-tests to detect unexpected dead code or possible inconsistencies in the specification, and manually checked that no unexpected cases of those were detected.

As for the 14 lemmas we used, 11 are proved by Coq using our scripts, and the remaining 3 directly by Alt-Ergo. Their proof takes 6 seconds in our configuration, with the maximum time to prove a goal being 650ms.

## 8 Related Work

*TPM related safety and security.* Various case studies centered around TPM uses have emerged over the last decade, often focusing on use cases relying on functionalities of the TPM itself. A recent formal analysis of the key exchange primitive of TPM 2.0 [22] provides a security model to capture TPM protections on keys and protocols. Authors of [21] 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 [18]. Such works focus on the TPM itself, but to the best of our knowledge, none of the previously published works aim at verifying the tpm2-tss library or any implementation of the TSS.

*Linked lists and recursive data structures.* We use logical definitions from [4] to formalize and manipulate C linked lists as ACSL logic lists in our effort, while another approach [3] relies on a parallel view of a linked list via a companion ghost array. Both approaches were tested on the linked list module of the Contiki OS [12], which relies on static allocations and simple structures. In this work we used a logic list based approach rather than a ghost code based approach following the conclusions in [4]. Realized in SPARK, a deductive verification tool for a subset of the Ada language and also the name of this subset, the approach to the verification of black-red trees [11] is related to the verification of linked lists in Frama-C using ghost arrays including the auto-verification aspects [5]. However, the trees themselves were implemented using arrays as pointers have only been recently introduced in SPARK [10]. Programs with pointers in SPARK are based on an ownership policy enforcing non-aliasing which makes their verification closer to Rust programs than C programs.

*Formal verification for real-life code.* Deductive verification on real-life code has been spreading in the last decades, with various verification case studies where bugs were often found by annotating and verifying the code [14]. Such studies include [9], providing feedback on the authors' experience of using ACSL and Frama-C on a real-world example. Authors of [7] managed a large scale formal verification of global security properties on the C code of the JavaCard Virtual Machine. SPARK was used in the verification of a TCP Stack [6]. Authors of [16] highlight some issues specific to the verification of the Hyper-V hypervisor, and how they can be solved with VCC, a deductive verification tool for C.

## 9 Conclusion and Future Work

This paper presents a first case study on formal verification of the tpm2-tss library, a popular implementation of the TPM Software Stack. Making the bridge between the TPM and applications, this library is highly critical: to take advantage of security guarantees of the TPM, its deductive verification is highly desired. The library code is very complex and challenging for verification tools.

We have presented our verification results for a subset of 10 functions of the ESAPI layer of the library that we verified with Frama-C. We have described current limitations of the verification tool and temporary solutions we used to address them. We have proved all necessary lemmas (extending those of a previous case study for linked lists [4]) in Coq using the most recent version of the

FRAMA-C–COQ translation and identified some necessary improvements in handling logic lists. Finally, we identified desired tool improvements to achieve a full formal verification of the library: support of dynamic allocations and basic ACSL clauses to handle them, a memory model that works at byte level, and clearer separation of modification statuses of variables between the heap, the stack, and static segments. The real-life code was slightly simplified for verification, but the logical behavior was preserved in the verified version. While the current real-life code cannot be verified without adaptations, we expect that it will become provable as soon as those improvements of the tool are implemented[10].

This work opens the way towards a full verification of the tpm2-tss library. Future work includes the verification of a larger subset of functions, including lower-level layers and operations. Specification and verification of specific security properties is another future work direction. Maintaining proofs for changing versions of tools and axiomatizations is also an interesting research direction. Finally, combining formally verified modules with modules which undergo a partial verification (e.g. limited to the absence of runtime errors, or runtime assertion checking of expected specifications on large test suites) can be another promising work direction to increase confidence in the security of the library.

*Acknowledgment.* Part of this work was supported by ANR (grants ANR-22-CE39-0014, ANR-22-CE25-0018) and French Ministry of Defense via a PhD grant of Yani Ziani. We thank Allan Blanchard, Laurent Corbin and Loïc Correnson for useful discussions, and the anonymous referees for 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., 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`
3. Blanchard, A., Kosmatov, N., Loulergue, F.: Ghosts for Lists: A Critical Module of Contiki Verified in Frama-C. In: Proc. of the 10th NASA Formal Methods Symposium (NFM 2018). LNCS, vol. 10811, pp. 37–53. Springer (2018)
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)
5. Blanchard, A., Loulergue, F., Kosmatov, N.: Towards Full Proof Automation in Frama-C using Auto-Active Verification. In: Proc. of the 11th NASA Formal Methods Symposium (NFM 2019). LNCS, vol. 11460, pp. 88–105. Springer (2019)
6. Cluzel, G., Georgiou, K., Moy, Y., Zeller, C.: Layered formal verification of a TCP stack. In: Proc. of the IEEE Secure Development Conference (SecDev 2021). pp. 86–93. IEEE (2021)

[10] Detailed discussions of limitations and ongoing extensions of FRAMA-C can be found at `https://git.frama-c.com/pub/frama-c/`.

7. Djoudi, A., Hána, M., Kosmatov, N.: Formal Verification of a JavaCard Virtual Machine with Frama-C. In: Proc. of the 24th International Symposium on Formal Methods (FM 2021). LNCS, vol. 13047, pp. 427–444. Springer (2021)
8. Djoudi, A., Hána, M., Kosmatov, N., Křížsenecký, M., Ohayon, F., Mouy, P., Fontaine, A., Féliot, D.: A bottom-up formal verification approach for common criteria certification: Application to JavaCard virtual machine. In: Proc. of the 11th European Congress on Embedded Real-Time Systems (ERTS 2022) (Jun 2022)
9. Dordowsky, F.: An experimental study using ACSL and Frama-C to formulate and verify low-level requirements from a DO-178C compliant avionics project. Electronic Proceedings in Theoretical Computer Science 187, 28–41 (2015)
10. Dross, C., Kanig, J.: Recursive data structures in SPARK. In: Proc. of the 32nd International Conference on Computer Aided Verification (CAV 2020). LNCS, vol. 12225, pp. 178–189. Springer (2020)
11. Dross, C., Moy, Y.: Auto-active proof of red-black trees in SPARK. In: Proc. of the 9th NASA Formal Methods Symposium (NFM 2017). LNCS, vol. 10227, pp. 68–83 (2017)
12. Dunkels, A., Grönvall, B., Voigt, T.: Contiki - A lightweight and flexible operating system for tiny networked sensors. In: Proc. of the 29th Annual IEEE Conference on Local Computer Networks (LCN 2004). pp. 455–462. IEEE Computer Society (2004)
13. Filliâtre, J.C., Paskevich, A.: Why3 - where programs meet provers. In: Proc. of the 22nd European Symposium on Programming (ESOP 2013). LNCS, vol. 7792, pp. 125–128. Springer (2013)
14. Hähnle, R., Huisman, M.: Deductive software verification: From pen-and-paper proofs to industrial tools. In: Computing and Software Science – State of the Art and Perspectives, LNCS, vol. 10000, pp. 345–373. Springer (2019)
15. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. 27(3), 573–609 (2015)
16. Leinenbach, D., Santen, T.: Verifying the microsoft Hyper-V hypervisor with VCC. In: Proc. of the Second World Congres on Formal Methods (FM 2009). LNCS, vol. 5850, pp. 806–809. Springer (2009)
17. Mangano, F., Duquennoy, S., Kosmatov, N.: A memory allocation module of Contiki formally verified with Frama-C. A case study. In: Proc. of the 11th International Conference on Risks and Security of Internet and Systems (CRiSIS 2016). LNCS, vol. 10158, pp. 114–120. Springer (2016)
18. Shao, J., Qin, Y., Feng, D.: Formal analysis of HMAC authorisation in the TPM2.0 specification. IET Inf. Secur. 12(2), 133–140 (2018)
19. The Coq Development Team: The Coq proof assistant. `http://coq.inria.fr`,
20. 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
21. Wang, W., Qin, Y., Yang, B., Zhang, Y., Feng, D.: Automated security proof of cryptographic support commands in TPM 2.0. In: Proc. of the 18th International Conference on Information and Communications Security (ICICS 2016). LNCS, vol. 9977, pp. 431–441. Springer (2016)
22. 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)

## A Appendix: Supplementary Material

This appendix presents the complete illustrative example.

### A.1 Complete Illustrative Example

Figures 10, 11, 12, 13, 14, 15, 16, 17 give the complete version of the illustrative example (presented in Fig. 1–6 in the paper), annotated in ACSL. It was proved with Frama-C 26.1, Why3 1.5.1, Alt-Ergo 2.4.3 and CVC4 1.8. The command used to run the proof is given at the end of the file.

Figure 9 provides the definition of the lemmas required to perform the proof. The same lemmas are used for the illustrative example and the proved subset of the real-life code. All necessary lemmas were proved with Coq 8.16.1 (but other recent versions should also work). The Coq proof scripts and the instructions how to run the proof are available in the companion artifact.

```
/********************* lemmas_node_t.h *********************/
/*@
  lemma linked_ll_in_valid{L}: ∀ NODE_T *bl, *el, \list<NODE_T*> ll;
    linked_ll(bl, el, ll) ⇒ ∀ ℤ n ; 0 ≤ n < \length(ll) ⇒
      \valid(\nth(ll, n));
  lemma ptr_sep_from_nil{L}: ∀ NODE_T* l;
    ptr_sep_from_list(l, \Nil);
  lemma ptr_sep_from_cons{L}: ∀ NODE_T *e, *hd, \list<NODE_T*> l;
    ptr_sep_from_list(e, \Cons(hd, l))⟺
      (\separated(hd, e) ∧ ptr_sep_from_list(e, l));
  lemma dptr_sep_from_nil{L}:
    ∀ NODE_T** l ; dptr_sep_from_list(l, \Nil);
  lemma linked_ll_all_separated{L}: ∀ NODE_T *bl, *el, \list<NODE_T*> ll;
    linked_ll(bl, el, ll) ⇒ all_sep_in_list(ll);
  lemma linked_ll_unchanged_ll{L1, L2}: ∀ NODE_T *bl, *el, \list<NODE_T*> ll;
    linked_ll{L1}(bl, el, ll) ⇒
      unchanged_ll{L1, L2}(ll) ⇒ linked_ll{L2}(bl, el, ll);
  lemma linked_ll_to_ll{L}: ∀ NODE_T *bl, *el, \list<NODE_T*> ll;
    linked_ll(bl, el, ll) ⇒ ll == to_ll(bl, el);
  lemma to_ll_split{L}:  ∀ NODE_T *bl, *el, *sep, \list<NODE_T*> ll;
    ll ≠ \Nil ⇒ linked_ll(bl, el, ll) ⇒ ll == to_ll(bl, el) ⇒
      in_list(sep, ll) ⇒ ll == (to_ll(bl, sep) ^ to_ll(sep, el));
  lemma in_list_in_sublist: ∀ NODE_T* e, \list<NODE_T*> rl, ll, l;
    (rl ^ ll) == l ⇒ (in_list(e, l)⟺(in_list(e, rl) ∨ in_list(e, ll)));
  lemma linked_ll_end{L}: ∀ NODE_T *bl, *el, \list<NODE_T*> ll;
    ll ≠ \Nil ⇒ linked_ll(bl, el, ll) ⇒
      \nth(ll, \length(ll)-1)->next == el;
  lemma linked_ll_end_separated{L}: ∀ NODE_T *bl, *el, \list<NODE_T*> ll;
    linked_ll(bl, el, ll) ⇒ ptr_sep_from_list(el, ll);
  lemma linked_ll_end_not_in{L}: ∀ NODE_T *bl, *el, \list<NODE_T*> ll;
    linked_ll(bl, el, ll) ⇒ !in_list(el, ll);
//new lemmas wrt. previous work on linked lists [Blanchard et al., SAC'19]
  lemma in_next_not_bound_in{L}: ∀ NODE_T *bl, *el, *item, \list<NODE_T*> ll;
    linked_ll(bl, el, ll) ⇒ in_list(item, ll) ⇒ item->next ≠ el ⇒
      in_list(item->next, ll);
  lemma linked_ll_split_variant{L}:
    ∀ NODE_T *bl, *bound, *el, \list<NODE_T*> l1, l2;
    linked_ll(bl, el, l1 ^ l2) ⇒ l2 ≠ \Nil ⇒
    bound == \nth(l1 ^ l2, \length(l1 ^ l2) - \length(l2)) ⇒
      linked_ll(bl, bound, l1) ∧ linked_ll(bound, el, l2);
*/
```

**Fig. 9.** Lemmas used to prove the illustrative example and the subset of real-life code.

```
#include <stdint.h>       // for uint types definitions
#include <string.h>       // for size_t definition
#include <byteswap.h>     // used in marshal
#define HOST_TO_BE_32(value) __bswap_32 (value) // swap endianness
typedef struct TPM2B_NAME { uint16_t size; uint8_t name[68];} TPM2B_NAME;
typedef struct {
  uint32_t      handle;   // handle used by TPM
  TPM2B_NAME      name;   // TPM name of the object
  uint32_t      rsrcType; // selector for resource type
} RESOURCE;
typedef struct NODE_T {
  uint32_t      handle;   // the handle used as reference
  RESOURCE      rsrc;     // the metadata for this rsrc
  struct NODE_T *   next; // next node in the list
} NODE_T; // linked list of resource
/*@
  predicate zero_tpm2b_name(TPM2B_NAME tpm2b_name) =
    tpm2b_name.size == 0 ∧ ∀ int i; 0 ≤ i  < 68 ⇒ tpm2b_name.name[i] == 0;
  predicate zero_resource(RESOURCE rsrc) =
    rsrc.handle == 0 ∧ zero_tpm2b_name(rsrc.name) ∧ rsrc.rsrcType == 0;
  predicate zero_rsrc_node_t(NODE_T node) =
    node.handle == 0 ∧ zero_resource(node.rsrc) ∧ node.next == \null;
*/
/************** Logic lists and linked lists definitions *************/
/*@
  predicate ptr_sep_from_list{L}(NODE_T* e, \list<NODE_T*> ll) =
    ∀ ℤ n; 0 ≤ n < \length(ll) ⇒ \separated(e, \nth(ll, n));
  predicate dptr_sep_from_list{L}(NODE_T** e, \list<NODE_T*> ll) =
    ∀ ℤ n; 0 ≤ n < \length(ll) ⇒ \separated(e, \nth(ll, n));
  predicate in_list{L}(NODE_T* e, \list<NODE_T*> ll) =
    ∃ ℤ n; 0 ≤ n < \length(ll) ∧ \nth(ll, n) == e;
  predicate in_list_handle{L}(uint32_t out_handle, \list<NODE_T*> ll) =
    ∃ ℤ n; 0 ≤ n < \length(ll) ∧ \nth(ll, n)->handle == out_handle;
  inductive linked_ll{L}(NODE_T *bl, NODE_T *el, \list<NODE_T*> ll) {
    case linked_ll_nil{L}: ∀ NODE_T *el; linked_ll{L}(el, el, \Nil);
    case linked_ll_cons{L}: ∀ NODE_T *bl, *el, \list<NODE_T*> tail;
      (\separated(bl, el) ∧ \valid(bl) ∧ linked_ll{L}(bl->next, el, tail) ∧
      ptr_sep_from_list(bl, tail)) ⇒
        linked_ll{L}(bl, el, \Cons(bl, tail));
  }
  predicate unchanged_ll{L1, L2}(\list<NODE_T*> ll) =
    ∀ ℤ n; 0 ≤ n < \length(ll) ⇒
      \valid{L1}(\nth(ll,n)) ∧ \valid{L2}(\nth(ll,n)) ∧
      \at((\nth(ll,n))->next, L1) == \at((\nth(ll,n))->next, L2);
  predicate all_sep_in_list(\list<NODE_T*> ll) =
    ∀ ℤ n1, n2; (0 ≤ n1 < \length(ll) ∧ 0 ≤ n2 < \length(ll) ∧ n1 ≠ n2) ⇒
        \separated(\nth(ll, n1), \nth(ll, n2));
  axiomatic Node_To_ll {
    logic \list<NODE_T*> to_ll{L}(NODE_T* beg, NODE_T* end)
      reads {node->next | NODE_T* node; \valid(node) ∧
                        in_list(node, to_ll(beg, end))};
    axiom to_ll_nil{L}: ∀ NODE_T *node; to_ll{L}(node, node) == \Nil;
    axiom to_ll_cons{L}: ∀ NODE_T *beg, *end;
      (\separated(beg, end) ∧ \valid{L}(beg) ∧
      ptr_sep_from_list{L}(beg, to_ll{L}(beg->next, end))) ⇒
        to_ll{L}(beg, end) == \Cons(beg, to_ll{L}(beg->next, end));
  }
*/

#include "lemmas_node_t.h"
```

**Fig. 10.** Illustrative provable example of the adjusted tpm2-tss list manipulation code, part 1/8.

```

#define _alloc_max 100
static NODE_T _rsrc_bank[_alloc_max];  // bank used by the static allocator
static int _alloc_idx = 0;  // index of the next rsrc node to be allocated
/*@
  predicate valid_rsrc_mem_bank{L} = 0 ≤ _alloc_idx ≤ _alloc_max;
  predicate list_sep_from_allocables{L}(\list<NODE_T*> ll) =
    ∀ int i; _alloc_idx ≤ i < _alloc_max ⇒
                        ptr_sep_from_list{L}(&_rsrc_bank[i], ll);
  predicate ptr_sep_from_allocables{L}(NODE_T* node) =
    ∀ int i; _alloc_idx ≤ i < _alloc_max ⇒ \separated(node, &_rsrc_bank[i]);
  predicate dptr_sep_from_allocables{L}(NODE_T** p_node) =
    ∀ int i; _alloc_idx ≤ i < _alloc_max ⇒ \separated(p_node, &_rsrc_bank[i]);
*/
/************************************************************************/
/*@
  requires valid_rsrc_mem_bank;
  assigns _alloc_idx, _rsrc_bank[\old(_alloc_idx)];
  ensures valid_rsrc_mem_bank;

  behavior not_allocable:
    assumes _alloc_idx == _alloc_max;

    ensures _alloc_idx == _alloc_max;
    ensures \result == NULL;
    ensures _rsrc_bank == \old(_rsrc_bank);
    ensures ∀ int i; 0 ≤ i < _alloc_max ⇒
              _rsrc_bank[i] == \old(_rsrc_bank[i]);
  behavior allocable:
    assumes 0 ≤ _alloc_idx < _alloc_max;

    ensures _alloc_idx == \old(_alloc_idx) + 1;
    ensures \result == &(_rsrc_bank[ _alloc_idx - 1]);
    ensures \valid(\result);
    ensures zero_rsrc_node_t( *(\result) );
    ensures ∀ int i; 0 ≤ i < _alloc_max ∧ i ≠ \old(_alloc_idx) ⇒
            _rsrc_bank[i] == \old(_rsrc_bank[i]);
  disjoint behaviors; complete behaviors;
*/
NODE_T *calloc_NODE_T()
{
  static const RESOURCE empty_RESOURCE;
  if(_alloc_idx < _alloc_max) {
    _rsrc_bank[_alloc_idx].handle = (uint32_t) 0;
    _rsrc_bank[_alloc_idx].rsrc = empty_RESOURCE;
    _rsrc_bank[_alloc_idx].next = NULL;
    _alloc_idx += 1;
    return &_rsrc_bank[_alloc_idx - 1];
  }
  return NULL;
}
```

**Fig. 11.** Illustrative provable example of the adjusted tpm2-tss list manipulation code, part 2/8.

```

typedef struct CONTEXT {
  int placeholder_int;
  NODE_T *rsrc_list;
} CONTEXT;
/*@
  predicate ctx_sep_from_list(CONTEXT *ctx, \list<NODE_T*> ll) =
    ∀ ℤ i; 0 ≤ i < \length(ll) ⇒ \separated(\nth(ll, i), ctx);
  predicate ctx_sep_from_allocables(CONTEXT *ctx) =
    ∀ int i; _alloc_idx ≤ i < _alloc_max ⇒ \separated(ctx, &_rsrc_bank[i]);

  predicate freshness(CONTEXT * ctx, NODE_T ** node) =
    ctx_sep_from_allocables(ctx)
    ∧ list_sep_from_allocables(to_ll(ctx->rsrc_list, NULL))
    ∧ ptr_sep_from_allocables(ctx->rsrc_list)
    ∧ ptr_sep_from_allocables(*node)
    ∧ dptr_sep_from_allocables(node);

  predicate sep_from_list{L}(CONTEXT * ctx, NODE_T ** node) =
    ctx_sep_from_list(ctx, to_ll{L}(ctx->rsrc_list, NULL))
    ∧ dptr_sep_from_list(node, to_ll{L}(ctx->rsrc_list, NULL));
*/

/*@
  requires valid_rsrc_mem_bank ∧ freshness(ctx, out_node);
  requires \valid(ctx);
  requires ctx->rsrc_list ≠ NULL ⇒ \valid(ctx->rsrc_list);
  requires linked_ll(ctx->rsrc_list, NULL, to_ll(ctx->rsrc_list, NULL));
  requires sep_from_list(ctx, out_node);
  requires ptr_sep_from_list(*out_node, to_ll(ctx->rsrc_list, NULL));
  requires !(in_list_handle(esys_handle, to_ll(ctx->rsrc_list, NULL)));
  requires \valid(out_node) ∧ \separated(ctx, out_node);
  requires *out_node ≠ NULL ⇒ \valid(*out_node) ∧ (*out_node)->next == NULL;
  assigns _alloc_idx, _rsrc_bank[_alloc_idx], ctx->rsrc_list, *out_node;
  ensures valid_rsrc_mem_bank ∧ freshness(ctx, out_node);
  ensures sep_from_list(ctx, out_node);
  ensures unchanged_ll{Pre, Post}(to_ll{Pre}(\old(ctx->rsrc_list), NULL));
  ensures \result \in {1610, 833};

  behavior not_allocable:
    assumes _alloc_idx == _alloc_max;

    ensures _alloc_idx == _alloc_max;
    ensures \valid(ctx);
    ensures !(in_list_handle(esys_handle, to_ll(ctx->rsrc_list, NULL)));
    ensures ctx->rsrc_list == \old(ctx->rsrc_list);
    ensures *out_node == \old(*out_node);
    ensures unchanged_ll{Pre, Post}(to_ll(ctx->rsrc_list, NULL));
    ensures \result == 833;
  behavior allocated:
    assumes 0 ≤ _alloc_idx < _alloc_max;

    ensures _alloc_idx == \old(_alloc_idx) + 1;
    ensures in_list_handle(esys_handle, to_ll(ctx->rsrc_list, NULL));
    ensures \valid(ctx->rsrc_list) ∧ *out_node == ctx->rsrc_list;
    ensures ctx->rsrc_list == &_rsrc_bank[_alloc_idx - 1];
    ensures ctx->rsrc_list->handle == esys_handle;
    ensures ctx->rsrc_list->next == \old(ctx->rsrc_list);
    ensures linked_ll(ctx->rsrc_list, NULL, to_ll(ctx->rsrc_list, NULL));
    ensures unchanged_ll{Pre, Post}(to_ll{Pre}(\old(ctx->rsrc_list), NULL));
    ensures \old(ctx->rsrc_list) ≠ NULL ⇒
        \nth(to_ll(ctx->rsrc_list, NULL), 1) == \old(ctx->rsrc_list);
    ensures \result == 1610;
  disjoint behaviors; complete behaviors;
*/
```

**Fig. 12.** Illustrative provable example of the adjusted tpm2-tss list manipulation code, part 3/8.

```
int createNode(CONTEXT * ctx, uint32_t esys_handle, NODE_T ** out_node){
//@ ghost pre_calloc:;
  //@ghost int if_id = 0;
  /*@ assert \separated(out_node, &_rsrc_bank[_alloc_idx]);*/
  /*@ assert \separated(ctx->rsrc_list, &_rsrc_bank[_alloc_idx]); */
  // NODE_T *new_head = calloc(1, sizeof(NODE_T));  /*library version*/
  /*@ assert list_sep_from_allocables(to_ll(ctx->rsrc_list, NULL)); */
  /*@ assert ptr_sep_from_list(&_rsrc_bank[_alloc_idx], to_ll{pre_calloc}(ctx->rsrc_list, NULL)); */
  /*@ assert ptr_sep_from_list(&_rsrc_bank[_alloc_idx], to_ll(ctx->rsrc_list, NULL)); */
  NODE_T *new_head = calloc_NODE_T();
  /*@ assert unchanged_ll{pre_calloc, Here}(
             to_ll{pre_calloc}(ctx->rsrc_list, NULL)); */
//@ ghost post_calloc:;
  if (new_head == NULL){return 833;}
  /*@ assert \valid(new_head) ∧ new_head->next == NULL; */
  /*@ assert ptr_sep_from_list(new_head, to_ll(ctx->rsrc_list, NULL)); */
  /*@ assert unchanged_ll{Pre, Here}(to_ll{Here}(ctx->rsrc_list, NULL));*/
//@ ghost pre_if:;
  if (ctx->rsrc_list == NULL) {
    /* The first object of the list will be added */
    ctx->rsrc_list = new_head;
    /*@ assert unchanged_ll{pre_if, Here}(to_ll(new_head, NULL));*/
    /*@ assert to_ll(new_head, NULL) == [|new_head|]; */
    /*@ assert \separated(new_head, new_head->next);*/
    new_head->next = NULL;
    /*@ assert to_ll(new_head, NULL) == [|new_head|]; */
  }
  else {
    /* The new object will become the first element of the list */
    /*@ assert dptr_sep_from_list(&ctx->rsrc_list,
                                  to_ll(ctx->rsrc_list, NULL));*/
    new_head->next = ctx->rsrc_list;
//@ ghost post_assign:;
    /*@ assert unchanged_ll{pre_if, Here}(
                            to_ll{pre_if}(ctx->rsrc_list, NULL));*/
    /*@ assert to_ll(new_head, NULL) ==
        ([|new_head|] ^ to_ll(\at(ctx->rsrc_list, pre_if), NULL));*/
    /*@ assert dptr_sep_from_list(&ctx->rsrc_list,
                                  to_ll(new_head, NULL));*/
    ctx->rsrc_list = new_head;
    /*@ assert unchanged_ll{post_assign, Here}(
                            to_ll{post_assign}(new_head, NULL));*/
    /*@ assert ctx->rsrc_list->next == \at(ctx->rsrc_list, Pre);*/
    /*@ assert to_ll(ctx->rsrc_list, NULL) ==
        ([|new_head|] ^ to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
    /*@ assert ctx->rsrc_list == \nth(to_ll(ctx->rsrc_list, NULL), 0);*/
  }
  //@ ghost post_add:;
  /*@ assert ctx->rsrc_list == \nth(to_ll(ctx->rsrc_list, NULL), 0);*/
  /*@ assert ctx_sep_from_list(ctx, to_ll(ctx->rsrc_list, NULL));*/
  /*@ assert ctx->rsrc_list == new_head;*/
  /*@ assert unchanged_ll{Pre, Here}(to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
  /*@ assert to_ll(new_head, NULL) ==
             ([|new_head|] ^ to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
  /*@ assert dptr_sep_from_list(out_node, to_ll(new_head, NULL));*/
  *out_node = new_head;
  /*@ assert unchanged_ll{post_add, Here}(to_ll{post_add}(new_head, NULL));*/
  /*@ assert ctx->rsrc_list == \nth(to_ll(ctx->rsrc_list, NULL), 0);*/
  new_head->handle = esys_handle;
  /*@ assert \nth(to_ll(ctx->rsrc_list, NULL), 0)->handle == esys_handle;*/
  /*@ assert in_list_handle(esys_handle, to_ll(ctx->rsrc_list, NULL));*/
  /*@ assert list_sep_from_allocables(to_ll(ctx->rsrc_list, NULL)); */
  /*@ assert unchanged_ll{Pre, Here}(to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
  return 1610;
}
```

**Fig. 13.** Illustrative provable example of the adjusted tpm2-tss list manipulation code, part 4/8.

```
/*@
  requires \valid(out_handle);
  assigns *out_handle;
  ensures \result \in {0, 12};
  ensures *out_handle \in {esys_handle, 0x4000000A, 0x4000000B,
                  0x40000110 + (esys_handle - 0x120U), \old(*out_handle)};
  behavior ok_handle:
    assumes esys_handle ≤ 31U ∨ 0x120U ≤ esys_handle ≤ 0x12FU
        ∨ esys_handle \in {0x10AU, 0x10BU};
    ensures \result == 0;
  behavior wrong_handle:
    assumes esys_handle > 31U
        ∧ (esys_handle < 0x120U ∨ esys_handle > 0x12FU);
    assumes !(esys_handle \in {0x10AU, 0x10BU});
    ensures *out_handle == \old(*out_handle);
    ensures \result == 12;
  disjoint behaviors; complete behaviors;
*/
int iesys_handle_to_tpm_handle(uint32_t esys_handle,  uint32_t * out_handle)
{
  if (esys_handle ≤ 31U) {*out_handle = (uint32_t) esys_handle; return 0;}
  if (esys_handle == 0x10AU){*out_handle = 0x4000000A; return 0;}
  if (esys_handle == 0x10BU){*out_handle = 0x4000000B; return 0;}
  if (esys_handle ≥ 0x120U ∧ esys_handle ≤ 0x12FU)
   {*out_handle = 0x40000110 + (esys_handle - 0x120U); return 0;}
  return 12;
}

/*@
  requires \valid(src) ∧ \valid(dest + (0 .. sizeof(*src)-1));
  requires \separated(dest+(0..sizeof(*src)-1),src);

  assigns dest[0 .. sizeof(*src)-1];

  ensures \valid(src);
  ensures \valid(dest + (0 .. sizeof(*src)-1));
*/
void memcpy_custom(uint8_t *dest, uint32_t *src) {
  dest[3] = (uint8_t)(*src & 0xFF);
  dest[2] = (uint8_t)((*src >> 8) & 0xFF);
  dest[1] = (uint8_t)((*src >> 16) & 0xFF);
  dest[0] = (uint8_t)((*src >> 24) & 0xFF);
}

/*@
  requires \valid(offset) ∧ 0 ≤ *offset ≤ UINT8_MAX - sizeof(in);
  requires buff_size > 0 ∧ \valid(&buff[0] + (0 .. buff_size - 1));
  requires *offset ≤ buff_size ∧ sizeof(in) + *offset ≤ buff_size;
  requires \separated(offset, buff);

  assigns *offset, (&buff[*offset])[0..sizeof(in) - 1];

  ensures *offset ==  \old(*offset) + sizeof(in);
  ensures \result == 0;
*/
int uint32_Marshal(uint32_t in,uint8_t buff[],size_t buff_size,size_t *offset){
  size_t  local_offset = 0;
  if (offset ≠ NULL){local_offset = *offset;}
  in = HOST_TO_BE_32(in);
  // memcpy(&buff[local_offset], &in, sizeof (in));
  memcpy_custom(&buff[local_offset], &in);
  if (offset ≠ NULL){*offset = local_offset + sizeof (in);}
  return 0;
}
```

**Fig. 14.** Illustrative provable example of the adjusted tpm2-tss list manipulation code, part 5/8.

```


/*@
  requires valid_rsrc_mem_bank{Pre} ∧ freshness(ctx, node);
  requires \valid(ctx);
  requires ctx->rsrc_list ≠ \null ⇒ \valid(ctx->rsrc_list);
  requires linked_ll(ctx->rsrc_list, NULL, to_ll(ctx->rsrc_list, NULL));
  requires 0 ≤ \length(to_ll(ctx->rsrc_list, NULL)) < INT_MAX;
  requires \valid(node);
  requires *node ≠ \null ⇒( \valid(*node) ∧ (*node)->next == \null);
  requires sep_from_list(ctx, node) ∧ \separated(node, ctx);
  requires ptr_sep_from_list(*node, to_ll(ctx->rsrc_list, NULL));
  assigns _alloc_idx, _rsrc_bank[_alloc_idx], ctx->rsrc_list;
  assigns *node, (&ctx->rsrc_list->rsrc.name.name[0])[0];
  ensures valid_rsrc_mem_bank ∧ freshness(ctx, node);
  ensures \separated(node, ctx);
  ensures \result \in {616, 833, 1611, 12};

  behavior handle_in_list:
    assumes in_list_handle(rsrc_handle, to_ll(ctx->rsrc_list, NULL));

    ensures _alloc_idx == \old(_alloc_idx);
    ensures ctx->rsrc_list == \old(ctx->rsrc_list);
    ensures in_list(*node, to_ll(ctx->rsrc_list, NULL)) ∧ *node ≠ NULL;
    ensures (*node)->handle == rsrc_handle ∧ sep_from_list(ctx, node);
    ensures unchanged_ll{Pre, Post}(to_ll(ctx->rsrc_list, NULL));
    ensures \result == 616;
  behavior handle_not_converted:
    assumes !(in_list_handle(rsrc_handle, to_ll(ctx->rsrc_list, NULL)));
    assumes rsrc_handle > 31U ∧ ! ( rsrc_handle \in {0x10AU, 0x10BU} );
    assumes rsrc_handle < 0x120U ∨ rsrc_handle > 0x12FU;

    ensures unchanged_ll{Pre, Post}(to_ll(ctx->rsrc_list, NULL));
    ensures ptr_sep_from_list(*node, to_ll(ctx->rsrc_list, NULL));
    ensures sep_from_list(ctx, node) ∧ *node == \old(*node);
    ensures \result == 12;
  behavior handle_not_in_list_and_node_not_allocable:
    assumes !(in_list_handle(rsrc_handle, to_ll(ctx->rsrc_list, NULL)));
    assumes rsrc_handle ≤ 31U ∨ (rsrc_handle \in {0x10AU, 0x10BU})
            ∨ (0x120U ≤ rsrc_handle ≤ 0x12FU);
    assumes _alloc_idx == _alloc_max;

    ensures _alloc_idx == _alloc_max;
    ensures unchanged_ll{Pre, Post}(to_ll{Pre}(ctx->rsrc_list, NULL));
    ensures *node == \old(*node) ∧ ctx->rsrc_list == \old(ctx->rsrc_list);
    ensures ptr_sep_from_list(*node, to_ll{Pre}(ctx->rsrc_list, NULL));
    ensures sep_from_list{Pre}(ctx, node); // has to stay in behavior
    ensures \result == 833;
  behavior handle_not_in_list_and_node_allocated:
    assumes !(in_list_handle(rsrc_handle, to_ll(ctx->rsrc_list, NULL)));
    assumes rsrc_handle ≤ 31U ∨ (rsrc_handle \in {0x10AU, 0x10BU})
            ∨ (0x120U ≤ rsrc_handle ≤ 0x12FU);
    assumes 0 ≤ _alloc_idx < _alloc_max;

    ensures in_list_handle(rsrc_handle, to_ll(ctx->rsrc_list, NULL));
    ensures (*ctx->rsrc_list).handle == rsrc_handle;
    ensures _alloc_idx == \old(_alloc_idx) + 1;
    ensures \valid(ctx->rsrc_list) ∧ *node == ctx->rsrc_list;
    ensures ctx->rsrc_list ≠ \old(ctx->rsrc_list);
    ensures ctx->rsrc_list->next == \old(ctx->rsrc_list);
    ensures to_ll(ctx->rsrc_list, NULL)
        == ([|ctx->rsrc_list|] ^ to_ll{Pre}(\old(ctx->rsrc_list), NULL) );
    ensures \old(ctx->rsrc_list) ≠ NULL ⇒
            \nth(to_ll(ctx->rsrc_list, NULL), 1) == \old(ctx->rsrc_list);
    ensures linked_ll(ctx->rsrc_list, NULL, to_ll(ctx->rsrc_list, NULL));
    ensures sep_from_list(ctx, node);
    ensures \result == 1611;
  disjoint behaviors; complete behaviors;
*/
```

**Fig. 15.** Illustrative provable example of the adjusted tpm2-tss list manipulation code, part 6/8.

```
int getNode(CONTEXT *ctx, uint32_t rsrc_handle, NODE_T ** node) {
  /*@ assert linked_ll(ctx->rsrc_list, NULL, to_ll(ctx->rsrc_list, NULL));*/
  int r;
  uint32_t tpm_handle;
  { /* Block added to circumvent issues with the WP memory model */
    NODE_T *tmp_node;
    /*@ ghost int n = 0;*/
    /*@
      loop invariant unchanged_ll{Pre, Here}(to_ll(ctx->rsrc_list, NULL));
      loop invariant linked_ll(ctx->rsrc_list, NULL,
                     to_ll(ctx->rsrc_list, NULL));
      loop invariant linked_ll(ctx->rsrc_list, tmp_node,
                     to_ll(ctx->rsrc_list, tmp_node));
      loop invariant ptr_sep_from_list(tmp_node,
                     to_ll(ctx->rsrc_list, tmp_node));
      loop invariant tmp_node ≠ \null ⇒
                     in_list(tmp_node, to_ll(ctx->rsrc_list, NULL));
      loop invariant !in_list_handle(rsrc_handle,
                     to_ll(ctx->rsrc_list, tmp_node));
      loop invariant n == \length(to_ll(ctx->rsrc_list, tmp_node));
      for handle_in_list : loop invariant
          in_list_handle(rsrc_handle, to_ll(ctx->rsrc_list, NULL));
      loop assigns n, tmp_node;
      loop variant \length(to_ll(tmp_node, NULL));
    */
    for (tmp_node = ctx->rsrc_list; tmp_node ≠ NULL;
        tmp_node = tmp_node->next) {
      /*@ assert tmp_node == \nth(to_ll(ctx->rsrc_list, NULL), n);*/
      /*@ assert linked_ll(tmp_node, NULL, to_ll(tmp_node, NULL));*/
      if (tmp_node->handle == rsrc_handle){
        /*@ assert dptr_sep_from_list(node, to_ll(ctx->rsrc_list, NULL));*/
        *node = tmp_node;
        /*@ assert unchanged_ll{Pre, Here}(to_ll{Pre}(ctx->rsrc_list, NULL));*/
        /*@ assert ptr_sep_from_allocables(*node);*/
        return 616;
      }
    /*@ assert to_ll(ctx->rsrc_list, tmp_node->next)
          == (to_ll(ctx->rsrc_list, tmp_node) ^ [|tmp_node|]);*/
    /*@ghost n++;*/
    }
  }
```

**Fig. 16.** Illustrative provable example of the adjusted tpm2-tss list manipulation code, part 7/8.

```
//@ ghost post_loop:;
  /*@ assert !(in_list_handle(rsrc_handle, to_ll(ctx->rsrc_list, NULL)));*/
  /*@ assert unchanged_ll{Pre, Here}(to_ll(ctx->rsrc_list, NULL));*/
  r = iesys_handle_to_tpm_handle(rsrc_handle, &tpm_handle);
  if (r == 12) { return r; };
  { /* Block added to circumvent issues with the WP memory model */
    NODE_T *tmp_node_2 = NULL;
    /*@ assert dptr_sep_from_list(&tmp_node_2,
                                  to_ll{post_loop}(ctx->rsrc_list, NULL));*/
    /*@ assert unchanged_ll{Pre, Here}(to_ll{Pre}(ctx->rsrc_list, NULL));*/
    /*@ assert \separated(node, &tmp_node_2);*/
    r = createNode(ctx, rsrc_handle, &tmp_node_2);
    /*@ assert sep_from_list(ctx, node);*/
    if (r == 833) {/*@ assert sep_from_list(ctx, node);*/ return r;};
//@ ghost post_alloc:;
    /*@ assert to_ll(ctx->rsrc_list, NULL)
        ==([|ctx->rsrc_list|] ^ to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
    /*@ assert ctx_sep_from_list(ctx, to_ll(ctx->rsrc_list, NULL));*/
    tmp_node_2->rsrc.handle = tpm_handle;
    tmp_node_2->rsrc.rsrcType = 0;
    size_t offset = 0;
    /*@ assert ptr_sep_from_list(tmp_node_2,
               to_ll(ctx->rsrc_list->next, NULL));*/
    r = uint32_Marshal(tpm_handle, &tmp_node_2->rsrc.name.name[0],
                        sizeof(tmp_node_2->rsrc.name.name),&offset);
    /*@ assert in_list_handle(rsrc_handle, to_ll(ctx->rsrc_list, NULL));*/
    if (r ≠ 0) { return r;};
    tmp_node_2->rsrc.name.size = offset;
    /*@ assert unchanged_ll{post_alloc, Here}(to_ll(ctx->rsrc_list, NULL));*/
    /*@ assert dptr_sep_from_list(node, to_ll(ctx->rsrc_list, NULL));*/
    /*@ assert dptr_sep_from_list(node,
               to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
    *node = tmp_node_2;
    /*@ assert unchanged_ll{Pre, Here}(
               to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
  }
  /*@ assert unchanged_ll{Pre, Here}(
             to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
  /*@ assert in_list_handle(rsrc_handle, to_ll(ctx->rsrc_list, NULL));*/
  /*@ assert ctx->rsrc_list->next == \at(ctx->rsrc_list, Pre);*/
  /*@ assert \at(ctx->rsrc_list, Pre) ≠ \null ⇒
      ctx->rsrc_list->next == \nth(to_ll(ctx->rsrc_list, NULL), 1);*/
  /*@ assert ctx->rsrc_list->handle == rsrc_handle;*/
  /*@ assert freshness(ctx, node);*/
  /*@ assert sep_from_list(ctx, node);*/
  return 1611;
}

/* Command to run the proof with Frama-C:
frama-c-gui -c11 example.c -wp -wp-rte -wp-prover altergo,cvc4,cvc4-ce,script
-wp-timeout 50 -wp-smoke-tests -wp-prop="-@lemma"
*/
```

**Fig. 17.** Illustrative provable example of the adjusted tpm2-tss list manipulation code, part 8/8.