Towards Formal Verification of a TPM Software Stack

Yani Ziani, Nikolai Kosmatov, Frédéric Loulergue, Daniel Gracia Pérez, Téo Bernier

PREPRINT

Towards Formal Verification of a TPM Software Stack

Yani Ziani[1,2], Nikolai Kosmatov[1], Frédéric Loulergue[2], Daniel Gracia Pérez[1], and Téo Bernier[1]

[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) [15] 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. 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 complex, heavily based on complex data structures (with multiple levels of imbricated structures and unions), low-level code, 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 [10]. We focus on a subset of functions related to storing an encryption key in the TPM, one of the most critical features

of the TSS. 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 [14]. 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[3], while the (slightly adapted) real-life functions annotated in ACSL and fully proved in FRAMA-C are available online as a companion artifact[4]. 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 memory allocation and ((NK: memory management?)). Additional 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 [10] 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 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

[3] For convenience of the reviewers, its full code is also given in Appendix.

[4] Available (with the illustrative example, all necessary lemmas and their proof) in https://nikolai-kosmatov.eu/iFM2023.zip.

(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 that can be sent to the WHY3 platform [8] and discharged either by SMT solvers or an interactive proof assistant like COQ [14].

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 [15] and reference books as [1] for more detail.

TPM Software Stack. The TPM is a standard conceived by the Trusted Computing Group (TCG)[5] 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 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 themselves, can have vulnerabilities[6] 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 mainly split into two parts: the API part containing

[5] https://trustedcomputinggroup.org/

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

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.

((NK: mv this paragr later? to results or conclusion?))Each API function of the ESAPI layer is similar in the type and order of function calls to internal operations of the layer (with some variations depending on the command), so knowing how to formally specify and verify one of them should allow us to apply the same method to all API functions.

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 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 ((NK: mv Fig5 to appendix?))and 5. As it is often done, some ACSL notation (e.g. `\forall`, `integer`, `==>`, `<=`, `!=`) is pretty-printed (resp., as $\forall$, $\mathbb{Z}$, $\Rightarrow$, $\leq$, $\neq$). 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. In our example, a resource structure (omitted in Fig. 1) is assumed to contain only a few fields

```
...
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 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);
...
  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. 1. Linked list and logic definitions.

of relatively simple types. The real code uses a more extensive and complex definition (with several levels of imbricated structures and unions), covering all possible types of TPM resources. While it does add some complexity to prove certain properties (as some of them may require to completely unfold all resource substructures ((NK: maybe say this later, in results?))), 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 contains no overlap between nodes — and stays that way throughout the layer. To accomplish that, we use and adapt the logic definitions from [4], given in 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 until the end, 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 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.*

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 [12]). Line 63 defines 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 contains several cases (*behaviors*) that cover all

```
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.

situations and 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.

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).

```
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.

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 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.

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 450–461 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.

In particular, anonymous blocks designated by the curly brackets lines 380, 416, 422 and 452, as well as the multiple declarations of `NODE_T` objects, are part of minor rewrites we will explain later with Fig. 5.

Lines 309–375 provide a partial non exhaustive function contract, covering the two main behaviors of `getNode`: if the element was found by its handle in

```
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   {
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   {/** Anonymous used 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.

the list, corresponding to the `handle_in_list` behavior in the ACSL contracts, and if the element was not found at first, but then successfully allocated and added, corresponding to `handle_not_in_list_and_node_allocated`.

As preconditions, we notably require with line 313 for the list to be well-formed (through the use of the linking predicate), and with line 310 the validity of

our bank and freshness relative to function arguments and global variables. Line 317 establishes memory separation of known pointers from the list of resources using the `sep_from_list` predicate, and separation among known pointers using the `\separated` clause.

As global postcondition, we require with line 321 that our bank stays valid, and that freshness relative to function arguments and global variables is maintained. However, properties regarding the list itself – such as the preservation of the list when it is not modified, or ensuring it remains well-formed after being modified – have to be issued to ACSL behaviors to be proved, due to an issue with how local variables are handled in the memory model of WP.

Let us take the assertion line 377 of Fig. 4 as an example. For such an assertion (and in general for any property to be proved), WP generates a proof obligation, to be proved by either WP itself or external provers via the WHY3 platform. Such an obligation includes a representation of the state of the memory of the program, at the program point where the assertion is located. In particular, for the assertion line 376, pointers such as our list pointer `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 – be it 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 our issue lies. For instance, in 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 in line 428 changes the status of the allocation table 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.

Originally, the call to `createNode` line 428 of Fig. 4 would use the address of `temp_node`, declared line 381, as a return pointer instead. This was sufficient to affect the status of the resource list on the scale of the entire function. As such, properties such as the assertion line 377 could not be proven, despite it being the same property as the one expressed in our `requires` clause line 312, and being positioned before any C instruction.

As a workaround, we use additional blocks and variable declarations to these side-effects in memory representation. Figure 5 provides an illustrative example for those minor rewrites. The left side of the figure shows a code snippet illustrating the structure of the original C code, where the `temp_node` pointer having its address taken and used in the `createNode` call line 10 is the same as the one used to iterate on the list. On the right, we define with line 8–13 an anonymous block, as well as a new `temp_node_2` list pointer, initialized to `NULL` to match the previous iteration over the list. The block defines a new scope, outside of which the pointer used by `createNode` will not exist and side-effect allocations

```
1  int getNode(..., NODE_T ** node){
2    // linking assertion unprovable
3    // properties on lists unprovable
4    NODE_T *tmp_node;
5    int r;
6    // iterate over the list
7
8
9    r = createNode(... , &tmp_node);
10   ...
11   *node = tmp_node;
12
13   return 1610;
14 }
```

```
1  int getNode(..., NODE_T ** node){
2    // linking assertion proved
3    // properties on lists provable
4    NODE_T *tmp_node;
5    int r;
6    // iterate over the list
7    {
8    NODE_T *tmp_node_2 = NULL;
9    r = createNode(... , &tmp_node_2);
10   ...
11   *node = tmp_node_2;
12   }
13   return 1610;
14 }
```

Fig. 5. A code snippet of `getNode` with a local pointer (on the left) and its rewrite with an anonymous block (on the right) for proving list properties.

will not happen. The block used here is equivalent to the one defined in lines 422–452 in Fig. 4.

To maintain as much as possible the behavior of the original code, we encapsulate the first part of the function in a block as well, in lines 380–416, so that there is never more than one additional `NODE_T` pointer defined and used in memory. We use assertions like those of lines 450–451 and 453–454 to propagate information over the structure of the C list (by its logic list representation) outside of each blocks, and from there to post-conditions.

We use assertions such as the ones lines 429 and 461 to help propagate separations from the list through the function and its anonymous blocks, and lines 450–451 and 453–454 to preserve structural integrity of the list. Some more intermediary assertions are also needed to prove the unchanged nature of the list. This is not something we could have done directly in post-conditions in the function contract, as it is currently not possible to prove post-conditions using other post-conditions, and their proof relies solely on what is known inside the scope of the function.

Incidentally, properties that rely on the structure and content of the list will often need to be proved as post-conditions in behaviors, rather than in global post-conditions. In the latter, the logic list representation of our resource list is much more difficult for WP and solvers to evaluate. Such local post-conditions include the `ensures` clause line 332, ensuring the list should stay unchanged if the handle sought was found while iterating over the list. 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. Line 371 establishes the link between the C list after modifications, and its logic representation evaluated after the last `return` clause line 462. Line 372 ensures the separation of known pointers from the new list.

Another memory manipulation issue we have encountered during our verification work comes from the function call line 440 in `getNode`: after having been added to the resource list, the newly allocated node must have its name (or more accurately, the name of its resource) set from its TPM handle `tpm_handle` (derived from the handle of the node by the function call in line 420). This is done

```
271 /*@
272   requires \valid(src) ∧ \valid(dest + (0 .. sizeof(*src)-1));
  ...
279 */
280 void memcpy_custom(uint8_t *dest, uint32_t * src, size_t n) {
281   dest[3] = (uint8_t)(*src & 0xFF);
282   dest[2] = (uint8_t)((*src >> 8) & 0xFF);
283   dest[1] = (uint8_t)((*src >> 16) & 0xFF);
284   dest[0] = (uint8_t)((*src >> 24) & 0xFF);
285 }
  ...
298 int uint32_Marshal(uint32_t in, uint8_t buff[], size_t buff_size, size_t *offset) {
299   size_t  local_offset = 0;
  ...
302   // memcpy(&buff[local_offset], &in, sizeof (in));
303   memcpy_custom(&buff[local_offset], &in, sizeof(in));
  ...
306 }
```

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

through marshaling using the `uint32_Marshal`, partially defined in lines 298–307 of Fig. 6, whose role is to store a 4-byte unsigned int (in this case, our TPM handle) in a flexible array of 1-byte unsigned int (the name of the resource). Modulo endianness, the function relies on a call to `memcpy` commented line 303, which is the source of our issue.

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 casting — whose support is limited — in order to reason at byte level — which the current memory model is unable to handle — over read and written objects. To circumvent both issues, we define our own memory copy function in lines 280–285: instead of directly copying the 4-byte unsigned int pointed by `src` byte per byte, we extract chunks of 1 byte (that is to say, the block size for the destination array pointed by `dest`) from it with mask `0xFF`, shifting it from 1 byte three times to cover it entirely. As a consequence, we require with line 272 that both our pointers are valid, and that memory accesses in the written part of the array are valid.

6 Lemmas

In some situations where SMT solvers can become inefficient — for example on properties, axioms or functions defined recursively — it can be necessary to state lemmas to conduct proofs. These lemmas can then be directly instantiated and easily used by solvers to verify specifications, but proving them usually requires to reason by induction, and thus to prove them interactively. As such, mixing automatic and interactive approaches offers a good trade-off for a complete proof.

The previous work using logic lists [4] that we adapted to fit our use case defined and proved lemmas. Twelve of these lemmas were necessary for the proof of both our illustrative example of real-life functions and these functions themselves, and we added two new lemmas (defined in Fig. 7). These lemmas were previously proved using the Coq proof assistant. However, because the formalization of the memory models and various aspects of ACSL offered by Frama-C changed between the version which the previous work relied on and the one we used, we could not reuse the proofs of these lemmas. While older

```
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. Addtionnal lemmas used in our verification work

Frama-C versions directly generated Coq specifications, more recent Frama-C versions instead let Why3 generate them. While the new is very 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, 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 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 not any principle to reason by induction on such 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 when it is not, we do not know 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 specify or program functions on lists use the head (easily replaced by `nth`) and tail of a list for writing the recursive case.

Even so, when the hypotheses of our lemmas include a fact expressed using `linked_ll`, it is possible to reason by cases, 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 using a list type defined by induction. We did not need an inductive principle on lists as either the lemmas did not require a proof by induction, or we reasoned inductively on inductive predicates. However, we proved such an induction principle using only the axioms we added. Some of the lemmas provided by the previous work on lists — but that we do not need yet in our work — are proved by induction on lists.

Because of theses changes, all previous proof scripts needed to be modified, 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. It is likely the next version of Frama-C will come back to a concrete representation of lists. The required changes in this case should however be minimal: we will only have to prove the axioms we introduced on `tail` and our decomposition principle, and the proofs themselves should remain unchanged.

		User-provided ACSL	Smoke tests	RTE	Total	
Code subset	Prover	#Goals	#Goals	#Goals	#Goals	Time
Illustrative example	Qed	105	3	18	126 (38.65%)	
	Script	1	0	0	1 (0.31%)	
	SMT	137	41	21	199 (61.04%)	
	All	243 (74.54%)	44 (13.50%)	39 (11.96%)	**326**	**5m13s**
Library code subset	Qed	275	3	38	316 (41.20%)	
	Script	5	0	0	5 (0.65%)	
	SMT	314	101	31	446 (58.15%)	
	All	594 (77.44%)	104 (13.56%)	69 (9.00%)	**767**	**21m44s**

Table 1. Proof results for the illustrative example and the real-life code.

7 Verification Results

Proof results for our example 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 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, through the Why3 v1.5.1 platform used by Frama-C.

In our illustrative example, 276 goals were proved in a total of 3min13s with the small majority of them by SMT solvers, and the rest by the internal simplifier engine of WP. The maximum time to prove a goal was 20s.

Solutions to memory manipulation problems presented in this paper (and solutions derived from them) are used on a larger verification study over 11 different functions of the main 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). 767 goals were proved in a total of 21m44s, the majority of which corresponding to user-provided notations to prove RTEs and functional properties. The maximum time to prove a goal was 1min50s.

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 [17] provides a security model to capture TPM protections on keys and protocols. Authors of [16] 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 formalise the session-based HMAC authorisation and encryption mechanisms [13]. Such works focus on the TPM itself, but to our knowledge, none of the previously published works were directed the *tpm2-tss* library or any implementation of the TSS.

Linked lists. We use logical definitions from [4] to formalize and manipulate C linked lists as ACSL logic lists in our effort, but it is worth noting previous works in [3] rely 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 [7], which relies on static allocations and simple structures.

Formal verification for other real-life codes. 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 codes [9]. Such studies include [6], providing a feedback on the authors' experience of using ACSL and FRAMA-C on a real-world example. Authors of [5] managed a large scale formal verification of global security properties on the C code of the JavaCard Virtual Machine.

Dynamic allocations in C programming Authors of [11] propose a memory model for low-level imperative languages such as C, including mechanics for allocations of fresh memory blocks, which they formally verified using the COQ proof assistant.

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 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. 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 status of variables between the heap, the stack, and static segments. We expect that the real-life code will become provable (without adaptations and simplifications used in this work) as soon as those improvements are implemented.

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. 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.

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. 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)
6. 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)
7. 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)
8. Filliâtre, J.C., Paskevich, A.: Why3 - where programs meet provers. In: Proc. of the 22nd European Symposium on Programming (ESOP 2013). pp. 125–128. LNCS, Springer (2013)
9. 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)
10. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. 27(3), 573–609 (2015)
11. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. Journal of Automated Reasoning 41(1), 1–31 (2008), https://inria.hal.science/inria-00289542
12. 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)
13. Shao, J., Qin, Y., Feng, D.: Formal analysis of HMAC authorisation in the TPM2.0 specification. IET Inf. Secur. 12(2), 133–140 (2018)
14. The Coq Development Team: The Coq proof assistant. http://coq.inria.fr,
15. 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
16. 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)
17. 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 is added for convenience of the reviewers, not for publication.

The complete illustrative example present in this appendix will be available in the long online version of the paper and in the companion artifact published online.

The full code of the subset of (slightly adapted) real-life functions of the library annotated in ACSL and fully verified in FRAMA-C will be available in the companion artifact published online.

The companion artifact is (or will be soon) available on

https://nikolai-kosmatov.eu/iFM2023.zip

We also plan to submit this artifact for evaluation of the iFM 2023 artifact evaluation committee. For convenience of the reviewers, a VM under Ubuntu with the companion artifact files and all necessary tools installed is (or will be soon) available for reviewers on

https://nikolai-kosmatov.eu/VM_iFM2023.ova

in case they want to run the proof before the official artifact evaluation. It was tested with VirtualBox ((NK: give version)). Please contact the authors via the PC chairs in case of any temporary access issue.

A.1 Complete Illustrative Example

Figures 9, 10, 11, 12, 13, 14, 15, 16 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 8 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. 8. 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. 9. Illustrative provable example of the adjusted *tpm2-tss* list manipulation code, part 1/8.

```
61
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 */
75 /*****************************************************************************/
76 /*@
77   requires valid_rsrc_mem_bank;
78   assigns _alloc_idx, _rsrc_bank[\old(_alloc_idx)];
79   ensures valid_rsrc_mem_bank;
80
81   behavior not_allocable:
82     assumes _alloc_idx == _alloc_max;
83
84     ensures _alloc_idx == _alloc_max;
85     ensures \result == NULL;
86     ensures _rsrc_bank == \old(_rsrc_bank);
87     ensures ∀ int i; 0 ≤ i < _alloc_max ⇒
88                 _rsrc_bank[i] == \old(_rsrc_bank[i]);
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. 10. Illustrative provable example of the adjusted *tpm2-tss* list manipulation code, part 2/8.

```
112
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 */
134
135 /*@
136   requires valid_rsrc_mem_bank ∧ freshness(ctx, out_node);
137   requires \valid(ctx);
138   requires ctx->rsrc_list ≠ NULL ⇒ \valid(ctx->rsrc_list);
139   requires linked_ll(ctx->rsrc_list, NULL, to_ll(ctx->rsrc_list, NULL));
140   requires sep_from_list(ctx, out_node);
141   requires ptr_sep_from_list(*out_node, to_ll(ctx->rsrc_list, NULL));
142   requires !(in_list_handle(esys_handle, to_ll(ctx->rsrc_list, NULL)));
143   requires \valid(out_node) ∧ \separated(ctx, out_node);
144   requires *out_node ≠ NULL ⇒ \valid(*out_node) ∧ (*out_node)->next == NULL;
145   assigns _alloc_idx, _rsrc_bank[_alloc_idx], ctx->rsrc_list, *out_node;
146   ensures valid_rsrc_mem_bank ∧ freshness(ctx, out_node);
147   ensures sep_from_list(ctx, out_node);
148   ensures unchanged_ll{Pre, Post}(to_ll{Pre}(\old(ctx->rsrc_list), NULL));
149   ensures \result \in {1610, 833};
150
151   behavior not_allocable:
152     assumes _alloc_idx == _alloc_max;
153
154     ensures _alloc_idx == _alloc_max;
155     ensures \valid(ctx);
156     ensures !(in_list_handle(esys_handle, to_ll(ctx->rsrc_list, NULL)));
157     ensures ctx->rsrc_list == \old(ctx->rsrc_list);
158     ensures *out_node == \old(*out_node);
159     ensures unchanged_ll{Pre, Post}(to_ll(ctx->rsrc_list, NULL));
160     ensures \result == 833;
161   behavior allocated:
162     assumes 0 ≤ _alloc_idx < _alloc_max;
163
164     ensures _alloc_idx == \old(_alloc_idx) + 1;
165     ensures in_list_handle(esys_handle, to_ll(ctx->rsrc_list, NULL));
166     ensures \valid(ctx->rsrc_list) ∧ *out_node == ctx->rsrc_list;
167     ensures ctx->rsrc_list == &_rsrc_bank[_alloc_idx - 1];
168     ensures ctx->rsrc_list->handle == esys_handle;
169     ensures ctx->rsrc_list->next == \old(ctx->rsrc_list);
170     ensures linked_ll(ctx->rsrc_list, NULL, to_ll(ctx->rsrc_list, NULL));
171     ensures unchanged_ll{Pre, Post}(to_ll{Pre}(\old(ctx->rsrc_list), NULL));
172     ensures \old(ctx->rsrc_list) ≠ NULL ⇒
173         \nth(to_ll(ctx->rsrc_list, NULL), 1) == \old(ctx->rsrc_list);
174     ensures \result == 1610;
175   disjoint behaviors; complete behaviors;
176 */
```

Fig. 11. Illustrative provable example of the adjusted *tpm2-tss* list manipulation code, part 3/8.

```
177 int createNode(CONTEXT * ctx, uint32_t esys_handle, NODE_T ** out_node){
178 //@ ghost pre_calloc:;
179   //@ghost int if_id = 0;
180   /*@ assert \separated(out_node, &_rsrc_bank[_alloc_idx]);*/
181   /*@ assert \separated(ctx->rsrc_list, &_rsrc_bank[_alloc_idx]); */
182   // NODE_T *new_head = calloc(1, sizeof(NODE_T));  /*library version*/
183   /*@ assert list_sep_from_allocables(to_ll(ctx->rsrc_list, NULL)); */
184   /*@ assert ptr_sep_from_list(&_rsrc_bank[_alloc_idx], to_ll{pre_calloc}(ctx->rsrc_list, NULL)); */
185   /*@ assert ptr_sep_from_list(&_rsrc_bank[_alloc_idx], to_ll(ctx->rsrc_list, NULL)); */
186   NODE_T *new_head = calloc_NODE_T();
187   /*@ assert unchanged_ll{pre_calloc, Here}(
188               to_ll{pre_calloc}(ctx->rsrc_list, NULL)); */
189 //@ ghost post_calloc:;
190   if (new_head == NULL){return 833;}
191   /*@ assert \valid(new_head) ∧ new_head->next == NULL; */
192   /*@ assert ptr_sep_from_list(new_head, to_ll(ctx->rsrc_list, NULL)); */
193   /*@ assert unchanged_ll{Pre, Here}(to_ll{Here}(ctx->rsrc_list, NULL));*/
194 //@ ghost pre_if:;
195   if (ctx->rsrc_list == NULL) {
196     /* The first object of the list will be added */
197     ctx->rsrc_list = new_head;
198     /*@ assert unchanged_ll{pre_if, Here}(to_ll(new_head, NULL));*/
199     /*@ assert to_ll(new_head, NULL) == [|new_head|]; */
200     /*@ assert \separated(new_head, new_head->next);*/
201     new_head->next = NULL;
202     /*@ assert to_ll(new_head, NULL) == [|new_head|]; */
203   }
204   else {
205     /* The new object will become the first element of the list */
206     /*@ assert dptr_sep_from_list(&ctx->rsrc_list,
207                                   to_ll(ctx->rsrc_list, NULL));*/
208     new_head->next = ctx->rsrc_list;
209 //@ ghost post_assign:;
210     /*@ assert unchanged_ll{pre_if, Here}(
211                             to_ll{pre_if}(ctx->rsrc_list, NULL));*/
212     /*@ assert to_ll(new_head, NULL) ==
213         ([|new_head|] ^ to_ll(\at(ctx->rsrc_list, pre_if), NULL));*/
214     /*@ assert dptr_sep_from_list(&ctx->rsrc_list,
215                                   to_ll(new_head, NULL));*/
216     ctx->rsrc_list = new_head;
217     /*@ assert unchanged_ll{post_assign, Here}(
218                             to_ll{post_assign}(new_head, NULL));*/
219     /*@ assert ctx->rsrc_list->next == \at(ctx->rsrc_list, Pre);*/
220     /*@ assert to_ll(ctx->rsrc_list, NULL) ==
221         ([|new_head|] ^ to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
222     /*@ assert ctx->rsrc_list == \nth(to_ll(ctx->rsrc_list, NULL), 0);*/
223   }
224   //@ ghost post_add:;
225   /*@ assert ctx->rsrc_list == \nth(to_ll(ctx->rsrc_list, NULL), 0);*/
226   /*@ assert ctx_sep_from_list(ctx, to_ll(ctx->rsrc_list, NULL));*/
227   /*@ assert ctx->rsrc_list == new_head;*/
228   /*@ assert unchanged_ll{Pre, Here}(to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
229   /*@ assert to_ll(new_head, NULL) ==
230               ([|new_head|] ^ to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
231   /*@ assert dptr_sep_from_list(out_node, to_ll(new_head, NULL));*/
232   *out_node = new_head;
233   /*@ assert unchanged_ll{post_add, Here}(to_ll{post_add}(new_head, NULL));*/
234   /*@ assert ctx->rsrc_list == \nth(to_ll(ctx->rsrc_list, NULL), 0);*/
235   new_head->handle = esys_handle;
236   /*@ assert \nth(to_ll(ctx->rsrc_list, NULL), 0)->handle == esys_handle;*/
237   /*@ assert in_list_handle(esys_handle, to_ll(ctx->rsrc_list, NULL));*/
238   /*@ assert list_sep_from_allocables(to_ll(ctx->rsrc_list, NULL)); */
239   /*@ assert unchanged_ll{Pre, Here}(to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
240   return 1610;
241 }
```

Fig. 12. Illustrative provable example of the adjusted *tpm2-tss* list manipulation code, part 4/8.

```
243
244 /*@
245   requires \valid(out_handle);
246   assigns *out_handle;
247   ensures \result \in {0, 12};
248   ensures *out_handle \in {esys_handle, 0x4000000A, 0x4000000B,
249                     0x40000110 + (esys_handle - 0x120U), \old(*out_handle)};
250   behavior ok_handle:
251     assumes esys_handle ≤ 31U ∨ 0x120U ≤ esys_handle ≤ 0x12FU
252          ∨ esys_handle \in {0x10AU, 0x10BU};
253     ensures \result == 0;
254   behavior wrong_handle:
255     assumes esys_handle > 31U ∧ (esys_handle < 0x120U ∨ esys_handle >
    0x12FU);
256     assumes !(esys_handle \in {0x10AU, 0x10BU});
257     ensures *out_handle == \old(*out_handle);
258     ensures \result == 12;
259   disjoint behaviors; complete behaviors;
260 */
261 int iesys_handle_to_tpm_handle(uint32_t esys_handle,  uint32_t * out_handle)
262 {
263   if (esys_handle ≤ 31U) {*out_handle = (uint32_t) esys_handle; return 0;}
264   if (esys_handle == 0x10AU){*out_handle = 0x4000000A; return 0;}
265   if (esys_handle == 0x10BU){*out_handle = 0x4000000B; return 0;}
266   if (esys_handle ≥ 0x120U ∧ esys_handle ≤ 0x12FU)
267    {*out_handle = 0x40000110 + (esys_handle - 0x120U); return 0;}
268   return 12;
269 }
270
271 /*@
272   requires \valid(src) ∧ \valid(dest + (0 .. sizeof(*src)-1));
273   requires \separated(dest+(0..sizeof(*src)-1),src);
274
275   assigns dest[0 .. sizeof(*src)-1];
276
277   ensures \valid(src);
278   ensures \valid(dest + (0 .. sizeof(*src)-1));
279 */
280 void memcpy_custom(uint8_t *dest, uint32_t * src, size_t n) {
281   dest[3] = (uint8_t)(*src & 0xFF);
282   dest[2] = (uint8_t)((*src >> 8) & 0xFF);
283   dest[1] = (uint8_t)((*src >> 16) & 0xFF);
284   dest[0] = (uint8_t)((*src >> 24) & 0xFF);
285 }
286
287 /*@
288   requires \valid(offset) ∧ 0 ≤ *offset ≤ UINT8_MAX - sizeof(in);
289   requires buff_size > 0 ∧ \valid(&buff[0] + (0 .. buff_size - 1));
290   requires *offset ≤ buff_size ∧ sizeof(in) + *offset ≤ buff_size;
291   requires \separated(offset, buff);
292
293   assigns *offset, (&buff[*offset])[0..sizeof(in) - 1];
294
295   ensures *offset ==  \old(*offset) + sizeof(in);
296   ensures \result == 0;
297 */
298 int uint32_Marshal(uint32_t in, uint8_t buff[], size_t buff_size, size_t *offset) {
299   size_t  local_offset = 0;
300   if (offset ≠ NULL){local_offset = *offset;}
301   in = HOST_TO_BE_32(in);
302   // memcpy(&buff[local_offset], &in, sizeof (in));
303   memcpy_custom(&buff[local_offset], &in, sizeof(in));
304   if (offset ≠ NULL){*offset = local_offset + sizeof (in);}
305   return 0;
306 }
```

Fig. 13. 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. 14. Illustrative provable example of the adjusted *tpm2-tss* list manipulation code, part 6/8.

```
376 int getNode(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   {
381   NODE_T *tmp_node;
382   /*@ ghost int n = 0;*/
383   /*@
384     loop invariant unchanged_ll{Pre, Here}(to_ll(ctx->rsrc_list, NULL));
385     loop invariant linked_ll(ctx->rsrc_list, NULL,
386                     to_ll(ctx->rsrc_list, NULL));
387     loop invariant linked_ll(ctx->rsrc_list, tmp_node,
388                     to_ll(ctx->rsrc_list, tmp_node));
389     loop invariant ptr_sep_from_list(tmp_node,
390                     to_ll(ctx->rsrc_list, tmp_node));
391     loop invariant tmp_node ≠ \null ⇒
392                     in_list(tmp_node, to_ll(ctx->rsrc_list, NULL));
393     loop invariant !in_list_handle(rsrc_handle,
394                     to_ll(ctx->rsrc_list, tmp_node));
395     loop invariant n == \length(to_ll(ctx->rsrc_list, tmp_node));
396     for handle_in_list : loop invariant
397         in_list_handle(rsrc_handle, to_ll(ctx->rsrc_list, NULL));
398     loop assigns n, tmp_node;
399     loop variant \length(to_ll(tmp_node, NULL));
400   */
401   for (tmp_node = ctx->rsrc_list; tmp_node ≠ NULL;
402       tmp_node = tmp_node->next) {
403     /*@ assert tmp_node == \nth(to_ll(ctx->rsrc_list, NULL), n);*/
404     /*@ assert linked_ll(tmp_node, NULL, to_ll(tmp_node, NULL));*/
405     if (tmp_node->handle == rsrc_handle){
406       /*@ assert dptr_sep_from_list(node, to_ll(ctx->rsrc_list, NULL));*/
407       *node = tmp_node;
408       /*@ assert unchanged_ll{Pre, Here}(to_ll{Pre}(ctx->rsrc_list, NULL));*/
409       /*@ assert ptr_sep_from_allocables(*node);*/
410       return 616;
411     }
412   /*@ assert to_ll(ctx->rsrc_list, tmp_node->next)
413         == (to_ll(ctx->rsrc_list, tmp_node) ^ [|tmp_node|]);*/
414   /*@ghost n++;*/
415   }
416   }
```

Fig. 15. Illustrative provable example of the adjusted *tpm2-tss* list manipulation code, part 7/8.

```
417 //@ ghost post_loop:;
418   /*@ assert !(in_list_handle(rsrc_handle, to_ll(ctx->rsrc_list, NULL)));*/
419   /*@ assert unchanged_ll{Pre, Here}(to_ll(ctx->rsrc_list, NULL));*/
420   r = iesys_handle_to_tpm_handle(rsrc_handle, &tpm_handle);
421   if (r == 12) { return r; };
422   {/** Anonymous used to circumvent issues with the WP memory model*/
423   NODE_T *tmp_node_2 = NULL;
424   /*@ assert dptr_sep_from_list(&tmp_node_2,
425                                 to_ll{post_loop}(ctx->rsrc_list, NULL));*/
426   /*@ assert unchanged_ll{Pre, Here}(to_ll{Pre}(ctx->rsrc_list, NULL));*/
427   /*@ assert \separated(node, &tmp_node_2);*/
428   r = createNode(ctx, rsrc_handle, &tmp_node_2);
429   /*@ assert sep_from_list(ctx, node);*/
430   if (r == 833) {/*@ assert sep_from_list(ctx, node);*/ return r;};
431 //@ ghost post_alloc:;
432   /*@ assert to_ll(ctx->rsrc_list, NULL)
433       ==([|ctx->rsrc_list|] ^ to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
434   /*@ assert ctx_sep_from_list(ctx, to_ll(ctx->rsrc_list, NULL));*/
435   tmp_node_2->rsrc.handle = tpm_handle;
436   tmp_node_2->rsrc.rsrcType = 0;
437   size_t offset = 0;
438   /*@ assert ptr_sep_from_list(tmp_node_2,
439               to_ll(ctx->rsrc_list->next, NULL));*/
440   r = uint32_Marshal(tpm_handle, &tmp_node_2->rsrc.name.name[0],
441                      sizeof(tmp_node_2->rsrc.name.name),&offset);
442   /*@ assert in_list_handle(rsrc_handle, to_ll(ctx->rsrc_list, NULL));*/
443   if (r ≠ 0) { return r;};
444   tmp_node_2->rsrc.name.size = offset;
445   /*@ assert unchanged_ll{post_alloc, Here}(to_ll(ctx->rsrc_list, NULL));*/
446   /*@ assert dptr_sep_from_list(node, to_ll(ctx->rsrc_list, NULL));*/
447   /*@ assert dptr_sep_from_list(node,
448               to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
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));*/
455   /*@ assert in_list_handle(rsrc_handle, to_ll(ctx->rsrc_list, NULL));*/
456   /*@ assert ctx->rsrc_list->next == \at(ctx->rsrc_list, Pre);*/
457   /*@ assert \at(ctx->rsrc_list, Pre) ≠ \null ⇒
458       ctx->rsrc_list->next == \nth(to_ll(ctx->rsrc_list, NULL), 1);*/
459   /*@ assert ctx->rsrc_list->handle == rsrc_handle;*/
460   /*@ assert freshness(ctx, node);*/
461   /*@ assert sep_from_list(ctx, node);*/
462   return 1611;
463 }
464
465 /* Command to run the proof with Frama-C:
466 frama-c-gui -c11 example.c -wp -wp-rte -wp-prover altergo,cvc4,cvc4-ce
467 -wp-timeout 240 -wp-smoke-tests -wp-model Typed+cast -wp-prop="-@lemma"
468 */
```

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