Towards Formal Verification of a TPM Software Stack: Achievements and Opportunities

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

To cite this version:

Yani Ziani, Téo Bernier, Nikolai Kosmatov, Frédéric Loulergue, Daniel Gracia Perez. Towards Formal Verification of a TPM Software Stack: Achievements and Opportunities. Formal Aspects of Computing, 2025, 37 (4), ⟨10.1145/3743153⟩. ⟨hal-05095685⟩

HAL Id: hal-05095685
https://hal.science/hal-05095685v1

Submitted on 1 Nov 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

Towards Formal Verification of a TPM Software Stack: Achievements and Opportunities

YANI ZIANI, Thales Research and Technology, Palaiseau, France and LIFO EA 4022, Univ. Orléans, INSA Centre Val de Loire, Orleans, France
TÉO BERNIER, Thales Research and Technology, Palaiseau, France
NIKOLAI KOSMATOV, Thales Research and Technology, Palaiseau, France
FRÉDÉRIC LOULERGUE, LIFO EA 4022, Univ. Orléans, INSA Centre Val de Loire, Orleans, France
DANIEL GRACIA PÉREZ, Thales Research and Technology, Palaiseau, France

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). The open-source library *tpm2-tss* is a popular implementation of the TSS. Vulnerabilities in its code could allow attackers to recover sensitive information and take control of the system. This article presents 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 difficulties and tool limitations we faced, illustrate them with examples and describe solutions that allowed us to verify functional properties and the absence of runtime errors for a representative subset of functions. In particular, their verification required several lemmas proved in the interactive proof assistant Coq. We describe our verification results and desired tool improvements necessary to achieve a full formal verification of the target code.

CCS Concepts: • **Software and its engineering → Formal software verification;**

Additional Key Words and Phrases: Deductive verification, Frama-C, TPM Software Stack, linked data structures, Coq

ACM Reference Format:
Yani Ziani, Téo Bernier, Nikolai Kosmatov, Frédéric Loulergue, and Daniel Gracia Pérez. 2025. Towards Formal Verification of a TPM Software Stack: Achievements and Opportunities. *Form. Asp. Comput.* 37, 4, Article 33 (October 2025), 29 pages. https://doi.org/10.1145/3743153

Part of this work was supported by ANR (grants ANR-22-CE39-0014, ANR-22-CE25-0018, ANR-24-ASM2-0001) and French Ministry of Defense via a PhD grant of Yani Ziani.
Authors' Contact Information: Yani Ziani, Thales Research and Technology, Palaiseau, France and LIFO EA 4022, Univ. Orléans, INSA Centre Val de Loire, Orleans, Centre-Val de Loire, France; e-mail: yani.ziani@thalesgroup.com; Téo Bernier, Thales Research and Technology, Palaiseau, France; e-mail: teo.bernier@thalesgroup.com; Nikolai Kosmatov, Thales Research and Technology, Palaiseau, France; e-mail: nikolai.kosmatov@thalesgroup.com; Frédéric Loulergue, LIFO EA 4022, Univ. Orléans, INSA Centre Val de Loire, Orleans, France; email: frederic.loulergue@univ-orleans.fr; Daniel Gracia Pérez, Thales Research and Technology, Palaiseau, France; e-mail: daniel.graciaperez@thalesgroup.com.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).
ACM 1433-299X/2025/10-ART33
https://doi.org/10.1145/3743153

1 Introduction

The ***Trusted Platform Module (TPM)*** [54] 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.*[1] 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 article, we present a first case study on formal verification of tpm2-tss using the Frama-C verification platform [34]. 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. This subset is relatively small (with 1k lines of executable code and 10k lines of interface) but handles important internal operations of the library. We verify both functional properties and the absence of runtime errors, such as invalid pointers and buffer overflows, often leading to security vulnerabalities. The functions are annotated in the acsl specification language [5]. 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 [8] to our case study, add new lemmas and prove them in the Coq proof assistant [53]. We identify some deficiencies in the new Frama-C–Coq extraction for lists (modified since [8]), 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.[2] Finally, we identify desired extensions and improvements of the verification tool.

Contributions. The main contributions of this work 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) and assertions 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.

This journal article has been extended with respect to the earlier conference paper [57] by additional explanations, illustrations and examples, as detailed in the *Publication History* paragraph below.

[1] https://github.com/tpm2-software/tpm2-tss

[2] Available (with the illustrative example, all necessary lemmas and their proofs) on https://doi.org/10.5281/zenodo.13693011

Outline. The article 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, respectively, to memory allocation and memory management. Interactive proofs of necessary lemmas and assertions are discussed in Section 6. Section 7 describes our verification results. Finally, Sections 8 and 9 present related work and a conclusion with necessary tool improvements.

Publication History. This article is an extended version of the earlier conference paper [57] presented at iFM 2023. Major extensions include the following. Section 4 has been extended by a more detailed presentation of the linked list representation with logic lists using Figure 2, by a discussion about handling separation using frame conditions and by Figure 5. Section 5 has been extended by a detailed presentation of proof-guiding annotations for separation propagation and by adding Figure 9. Section 6 has been enriched by a deeper discussion of necessary interactive proofs for lemmas and assertions, illustrated by additional Figures 12, 13, and 14. Additional files illustrating these aspects have also been added into the companion artifact whose new version has been published online. Finally, related work in Section 8 has been extended as well.

2 Frama-C Verification Platform

Frama-C [34] 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) [5], 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, respectively, by **`requires`** and **`ensures`** clauses) expressing properties that must hold, respectively, 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 [28], by SMT solvers or an interactive proof assistant like Coq [53]. To ensure the absence of **runtime errors** (**RTE**) (such as invalid pointer accesses, arithmetic or buffer overflows), 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 [22] 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 [23].

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 [54] and reference books as [3] for more detail.

TPM Software Stack. The TPM is a standard conceived by the **Trusted Computing Group** (**TCG**)[3] for a passive secure cryptoprocessor designed to protect secure hardware from

[3] https://trustedcomputinggroup.org/

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*[4] and *dm-crypt*.[5] However, it can be (and actually is) 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 and their implementations, as any software component or the TPM itself, can have vulnerabilities[6] that attackers can exploit to recover sensitive data communicated with the TPM or 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 corresponds to — and calls — the `TPM2_Create` command of the TPM), and the back-end containing the core of that layer's functionalities. Each API function calls 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. The target implementation uses complex data structures (such as structures with several nested levels of unions and sub-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 Figures 1–10 that will be explained below step-by-step. Its full code being available in the companion artifact, we omit in this article some less significant definitions and assertions which are not mandatory to understand the article (but we preserve line numbering of the full example for convenience of the reader). This example is heavily simplified to fit the article, yet it is representative for most issues we faced (except the complexity of data structures). It contains a main list manipulation function,

[4]https://learn.microsoft.com/en-us/windows/security/operating-system-security/data-protection/bitlocker/
[5]https://docs.kernel.org/admin-guide/device-mapper/dm-crypt.html
[6]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.

`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 the `createNode` function (`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 Figure 3. Figure 4 defines a (simplified) context and additional logic definitions to handle pointer separation and memory freshness. The node creation function is defined in Figure 5. The search function is shown in Figures 6, 7, and 8. As it is often done, some ACSL notation (e.g., `\forall`, `integer`, `==>`, `<=`, `!=`)

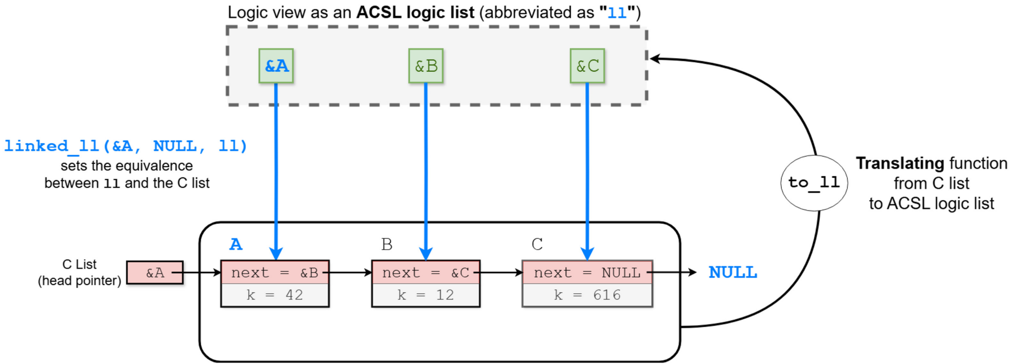

Fig. 2. Graphic representation of the translation of a C linked list into an ACSL logic list, and a link between a C linked list and its ACSL representation, using the logic definitions of Figure 1.

is pretty-printed (respectively, as ∀, ℤ, ⇒, ≤, ≠). In this section, we detail Figures 1–4 and some parts of Figure 5. Some parts of the specification will be presented in later sections.

Lists of Resources. Lines 11–15 of Figure 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.[7] In our example, a resource structure (omitted in Figure 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 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, as Frama-C does not provide any form of automated reasoning over linked lists, we use and adapt the logic definitions from [8], which rely on a companion ACSL logic list to represent a C linked lists. These definitions are given on lines 26–44, 48–57 of Figure 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 ends just before that node. For instance, the lower part of Figure 2 shows a well-formed linked list of head pointer `&A` (referring to head node `A`). That list contains 3 nodes, `A`, `B`, and `C`, with the latter having its `next` field being the `NULL` pointer. Thus, to translate that list into its companion ACSL logic list, we simply need to use `to_ll(A, NULL)`. The resulting logic list contains three pointers (to the three nodes of the linked list) as represented in the upper part of Figure 2. For convenience, we denote that logic list as `ll` in this example.

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

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

\separated(p,q) states that the referred memory locations *p and *q do not overlap (i.e., the bytes representing these memory locations are disjoint, see the Wp Memory Model paragraph in Section 6 for more detail). For instance, for the linked list and the logic list considered in Figure 2, linked_ll (&A, NULL, ll) establishes the equivalence between the C linked list of head pointer &A (or head node A) and the ACSL logic list ll. As mentioned above, this equivalence includes the memory separation between nodes (that is to say, A, B and C do not overlap) and validity of access for each node (that is to say, **\valid**(&A), **\valid**(&B), and **\valid**(&C) are true).

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, respectively, the nth element of logic list l and the length of l. For instance, if we consider again the example of Figure 2, then \nth(ll,0), \nth(ll,1), and \nth(ll,2) return, respectively, &A &B, and &C, and \length(ll) is equal to 3.

The predicate unchanged_ll on 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 Section 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[8] 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 Figure 3 a bank-based static allocator calloc_NODE_T that replaces calls to calloc used in the real-life code. It attributes pre-allocated cells, following some existing implementations (like the memb module of Contiki [41]). 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 on lines 67–73 specify separation between a logic list of nodes (respectively, 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

[8] 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[_alloc_idx];
79   ensures valid_rsrc_mem_bank;
  ...
81   behavior not_allocable:
82     assumes _alloc_idx == _alloc_max;
83
84     ensures _alloc_idx == _alloc_max;
85     ensures \result == NULL;
  ...
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. 3. Allocation bank and static allocator.

```
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. 4. Context and predicates to handle separation from a list and memory freshness.

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 this validity clause as well (so that the specification reflects more accurately the behavior of any allocator, rather than simply our custom static one). In the same way, lines 96–97 specify that the nodes of the bank other than the newly allocated block have not been modified.[9]

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 Figure 4 define a simplified context structure, comprising 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

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

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 Figure 4 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 Figure 3. 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 of the allocable nodes — was not used in the list before the allocation (cf. line 310 in Figure 6). 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 Figure 1.

Handling Separation: Frame Conditions. In acsl, frame conditions are expressed at the level of function contracts, using the **`assigns`** clause to define which non-local memory locations *can* be modified by the function. When such a clause is declared, Wp tries to prove it at the scale of the function. More specifically, it tries to prove that all non-local memory locations that may be modified by the function are included in the set of memory locations listed in the **`assigns`** clause. The set of memory locations that may be modified is composed of all the locations that may be written to in the code of the function, as well as all the locations provided by the **`assigns`** clauses of its callees, to provide Wp information about the callees' side effects.

For example, line 78 of Figure 3 specifies that only the allocation index `_alloc_idx` and the next allocable node `_rsrc_bank[_alloc_idx]` can be modified by our custom allocator. Because the node creation function `createNode` in Figure 5 calls the custom allocator on line 186 (in replacement to the original call to `calloc` commented on line 185), the **`assigns`** clause we have to write for the function contract of `createNode` (cf. line 145) must include the locations specified in the **`assigns`** clause of `calloc_NODE_T` (that is to say, `_alloc_idx` and `_rsrc_bank[_alloc_idx]`). It must also include `ctx->rsrc_list`, as it is modified on lines 197 and 216, as well as `*out_node`, which is modified on line 232. We do not need to (nor can we) add `new_head` to the clause, as it is defined only locally. We do not need to add `*new_head` either, while the corresponding memory location is not defined locally, because Wp knows from the function contract of `calloc_NODE_T`, with lines 85 and 93, that within the scope of the function, `new_head` is either `NULL`, or `*new_head` is the freshly allocated node (on line 186) from `rsrc_bank` — that is, `_rsrc_bank[_alloc_idx]`), — which is already present in the list. In the same way, callers of the node creation function need to include its modified memory locations in their respective **`assigns`** clauses.

```
135 /*@
136   requires valid_rsrc_mem_bank ∧ freshness(ctx, out_node);
    ...
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);
    ...
161   behavior allocated:
162     assumes 0 ≤ _alloc_idx < _alloc_max;
    ...
167     ensures ctx->rsrc_list == &_rsrc_bank[_alloc_idx - 1];
    ...
175   disjoint behaviors; complete behaviors;
176 */
177 int createNode(CONTEXT * ctx, uint32_t esys_handle, NODE_T ** out_node){
178 //@ ghost pre_calloc:;
    ...
185   // NODE_T *new_head = calloc(1, sizeof(NODE_T));  /*library version*/
186   NODE_T *new_head = calloc_NODE_T();
187   /*@ assert unchanged_ll{pre_calloc, Here}(
188               to_ll{pre_calloc}(ctx->rsrc_list, NULL)); */
    ...
190   if (new_head == NULL){return 833;}
    ...
195   if (ctx->rsrc_list == NULL) {
196     /* The first object of the list will be added */
197     ctx->rsrc_list = new_head;
    ...
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 */
    ...
208     new_head->next = ctx->rsrc_list;
    ...
216     ctx->rsrc_list = new_head;
    ...
223   }
    ...
232   *out_node = new_head;
    ...
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. 5. The node creation function, where some annotations are removed.

Specifying frame conditions is in practice necessary in order to ensure validity and separation properties. In the general case, if no **assigns** clause is given for a specific function, it means that the function potentially modifies every known memory location, making it practically impossible to prove anything in its callers (where Wp would not have any precise information on the callee's side effects).

```
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 */
```

Fig. 6. The search function contract, where some annotations are removed.

Thus, in order to ensure separation properties (as well as functional properties such as the well-formedness of the list), precise **assigns** clauses are required. In our case, in order to propagate to callers separation of a newly allocated node with the list, we need to preserve throughout function contracts the `freshness` predicate, as well as information to associate an allocated node with the allocation bank. In particular, the property on line 93 ensuring that the pointer returned by the allocator (in case of a successful allocation) points to the `_alloc_idx`-th element of array `_rsrc_bank` is required. This is necessary in order to propagate to the node creation function that, given the freshness, the returned allocated node on line 186 of `createNode` is separated from the list. The callee's **assigns** clause helps prove the assertion line 187-188, which indicates that the structure of the list was not changed (as the location potentially written by the call on line 186 was separated from the list). Similarly, we need to ensure the same equality in the postcondition of the node creation function with line 167, in order to provide the necessary information to specify and prove **assigns** clauses in callers, and to help propagate to the callers (in this case, the search function of Figures 6 and 7) that the newly allocated node is separated from the previous list.

5 Memory Management

This section presents how we use the definitions introduced in Section 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 present some parts of Figure 5 and Figures 6–10.

```
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. 7. The (slightly rewritten) search function body.

The Search Function. Figures 6 (contract) and 7 (body) provide the search operation getNode with a partial contract illustrating functional and memory safety properties we aim at verifying and judging 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.

In Figure 7, 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. Figure 6 provides 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.

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 that 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 Unprovable Property. Consider the assertion on line 377 of Figure 7. Despite the presence of the same property as a precondition of the function (line 313 in Figure 6), 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[10] from Figure 7 by removing the curly braces on lines 380, 416, 422, 452. This issue is due to a limitation of the WP memory model.

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

```
a int getNode(..., NODE_T ** node){
b   // list properties unprovable
c   int r;
d
e   NODE_T *tmp_node;
f   ... // iterate over the list
g
h
i
j   r = createNode(..., &tmp_node);
k   ...
l   *node = tmp_node;
m
n   return 1611;
o }
```

```
a int getNode(..., NODE_T ** node){
b   // list properties proved
c   int r;
d   {
e     NODE_T *tmp_node;
f     ... // iterate over the list
g   }
h   {
i     NODE_T *tmp_node_2 = NULL;
j     r = createNode(..., &tmp_node_2);
k     ...
l     *node = tmp_node_2;
m   }
n   return 1611;
o }
```

Fig. 8. Comparison of the real-life code of `getNode` (on the left) and its rewriting with additional blocks (on the right) for proving list properties.

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 [28]. 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 Figure 7, 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` (e.g., on line 136 in Figure 5). 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 Figure 7) 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 8 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`, 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 Figure 7) 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, the assertions in lines 450–451 and 453–454 help propagate information over the structure of

```
422   { /* Block added 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:;
  ...
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;
```

Fig. 9. Examples of supplementary annotations needed to propagate properties with additional blocks.

the linked list (by its logic list representation) outside each block, and finally to postconditions. 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.

Additional intermediate annotations can be required in order to help propagate separations from the list, and the well-formedness of the list through the introduced anonymous blocks, and from the scope of the function to its postconditions. For instance, the assertions in lines 455–461 of Figure 9 are used to deduce and prove — as part of postconditions, outside the anonymous blocks—functional properties on the list.

More specifically, the assertions on lines 460 and 461 (which serve to preserve the freshness and the separation of the context pointer to any reachable node of the list) are proved in this instance using information provided by assertions on lines 453–459. Lines 456–458 in particular help prove lines 460–461 by describing the list at this program point as the list provided at the entry of the function, to which a new node was added (as the new head of the list) using `createNode`. In the same vein, lines 453–454 that establish that the structure of the previously known list has not changed between the start of the function and the current program point help prove properties of lines 456–458. This is however only achievable at this scale, but not when the property is in the postcondition of the function. Indeed, given a specific postcondition (e.g., line 371), the corresponding proof obligation generated by Wp will "only" contain information on the program memory (be

```
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];
    ...
279 */
280 void memcpy_custom(uint8_t *dest, uint32_t *src) {
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);
    ...
306 }
```

Fig. 10. Definition for `memcpy` replacement in marshal.

it from the ACSL annotations or the C code itself) after return clauses but "before postconditions", hence after any locally defined variable has been freed.

In this instance, the proof obligation does not contain any information pertaining to other **ensures** clauses, but without anonymous blocks, would still contain the variable `allocations` (relatively to the memory model) such as that of `tmp_node_2` that would modify the modification status of the list as a whole in the scope of the function. The list would be considered to be "modified" again after any **return** clause, as `tmp_node_2` would be freed (relatively to the memory model). From there, most postconditions would be very difficult to prove. Anonymous blocks and the described assertions allow for the proof of all postconditions, and thus a complete functional proof of the function.

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 Figure 10, 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. The **assigns** clause on line 275 implies that

```
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. 11. New lemmas proved in our verification work (in addition to those in [8]).

the function may only modify **sizeof**(*src) bytes, that is to say, **sizeof**(uint32_t)= 4 bytes, at address dest, 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 Interactively Proved Lemmas and Assertions

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 [8] defined and proved several lemmas using the Coq proof assistant. We have added two new useful lemmas (defined in Figure 11) 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 [8] and the one we use, we could not reuse the proofs of these lemmas. While older Frama-C versions directly generated Coq specifications, the Frama-C version used in this work lets Why3 generate them. Even if the new translation is close to the previous one, the way logic lists are handled was modified significantly.[11]

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 the Frama-C version used in this work, ACSL logic lists are axiomatized as follows: two functions nil and cons are declared, as well as a few other functions on logic 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 for logic lists, and because nil and cons are not constructors, it is not possible to reason by case on logic lists in the Coq 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.

In this case study, there are typically two inductive definitions that can allow proof by induction of a property in which they are involved: the inductive definition of the list type and the definition of the inductive predicate linked_ll. Interestingly, even if the translation of logic lists into Coq does not allow proof by induction over a logic list, when the hypotheses of our lemmas include a fact expressed using linked_ll (defined in lines 34-40 of Figure 1), it is still possible to reason by case, because this inductive predicate is translated into Coq as an inductive predicate.

[11]Thanks to the identification of these limitations in this work, this issue was fixed in the following versions of Frama-C.

```
772 Inductive P_linked_ll: (Numbers.BinNums.Z → Numbers.BinNums.Z) →
773   (addr → addr) → addr → addr → list addr → Prop :=
774   | Q_linked_ll_nil :
775       ∀ (Malloc:Numbers.BinNums.Z → Numbers.BinNums.Z)
776         (Mptr:addr → addr) (el:addr),
777       P_linked_ll Malloc Mptr el el (nil : list addr)
778   | Q_linked_ll_cons :
779       ∀ (Malloc:Numbers.BinNums.Z → Numbers.BinNums.Z)
780         (Mptr:addr → addr) (bl:addr) (el:addr) (tail:list addr),
781       P_ptr_sep_from_list bl tail → valid_rw Malloc bl 73%Z →
782       separated bl 73%Z el 73%Z →
783       P_linked_ll Malloc Mptr (Mptr (shift bl 72%Z)) el tail →
784       P_linked_ll Malloc Mptr bl el (cons bl tail).
```

Fig. 12. Coq definition of `linked_ll` generated by Why3.

Consequently, there are only two possible cases for the logic list: either it is empty, or it is built with `cons`. To allow proof by induction 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 [8] — 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. Thanks to our approach, we expect that the required changes in our proofs of lemmas will remain minimal for later versions of Frama-C/Wp in which the translation of logic lists into Coq will be improved: we will only have to prove the statements of the axioms we introduced on `tail` and our decomposition principle.

When SMT solvers fail on apparently simple goals, it can be necessary to dive into the Coq translation, including the formalization of the Wp memory model, in order to better understand the underlying difficulties. The remainder of this section will explore the difficulties to verify the initial `getNode` function, i.e., before adding the anonymous block in Section 4. We will present the application of the Coq interactive prover with reference to Figures 12–14. These figures are extracted from the Coq proof obligation of the first assertion of `getNode` (line 377 of Figure 7) without adding anonymous blocks as a workaround. The Coq file has initially been generated by Why3, and then the Coq proof script has been crafted by hand. It is available in file

`artifact/proofs/lemmas/wp_coq/getNode_assert_target.v`

in the companion artifact. The full illustrative example, from which it was generated, is contained in file `artifact/proofs/example/linked_ll_self_contained_no_block.c` of the companion artifact.

Coq Representation. Lines 772–784 of Figure 12 illustrate the direct translation generated by Why3 of the `linked_ll` predicate as defined in Figure 1. We observe that both definitions are inductive and that their constructors have matching parameters including `bl`, `el` and `tail`. Moreover, both definitions also exhibit matching built-in predicates such as `separated` and `valid` (named `valid_rw` in the Coq representation) or user-defined predicates like `ptr_sep_from_list` and

```
156 Definition separated (p:addr) (p_size:Numbers.BinNums.Z) (q:addr)
157     (q_size:Numbers.BinNums.Z) : Prop :=
158   (p_size <= 0%Z)%Z ∨
159   (q_size <= 0%Z)%Z ∨
160   ~ ((base p) = (base q)) ∨
161   (((offset q) + q_size)%Z <= (offset p))%Z ∨
162   (((offset p) + p_size)%Z <= (offset q))%Z.
    ...
174 Definition valid_rw (Malloc:Numbers.BinNums.Z → Numbers.BinNums.Z) (p:addr)
175     (size:Numbers.BinNums.Z) : Prop :=
176   (0%Z < size)%Z →
177   (0%Z < (base p))%Z ∧
178   (0%Z <= (offset p))%Z ∧ (((offset p) + size)%Z <= (Malloc (base p)))%Z.
    ...
227 Definition framed (Mptr:addr → addr) : Prop :=
228   ∀ (p:addr), ((region (base p)) <= 0%Z)%Z →
229   ((region (base (Mptr p))) <= 0%Z)%Z.
    ...
794 Definition P_unchanged_ll (Malloc:Numbers.BinNums.Z → Numbers.BinNums.Z)
795     (Mptr:addr → addr) (Malloc1:Numbers.BinNums.Z → Numbers.BinNums.Z)
796     (Mptr1:addr → addr) (ll:list addr) : Prop :=
797   ∀ (i:Numbers.BinNums.Z),
798   let a := nth ll i in
799   let a1 := shift a 72%Z in
800   (0%Z <= i)%Z → (i < (length ll))%Z →
801   (((Mptr a1) = (Mptr1 a1)) ∧ valid_rw Malloc a 73%Z) ∧
802   valid_rw Malloc1 a 73%Z.
```

Fig. 13. Examples of built-in and user-defined predicates in Coq.

`linked_ll` (which are prefixed by `P_` in the Coq representation). However, the representation also reveals new parameters, namely `Malloc` and `Mptr`, which constitute elements of the memory model employed by Frama-C/Wp.

Wp Memory Model. Firstly, the Wp memory model handles pointer validity as we can observe in the Coq representation. Indeed, this is the purpose of the allocation table `Malloc`, which is formalized as a function taking and returning an integer (represented here by the `Numbers.BinNums.Z` Coq type). This allocation table can be conceptualized as a map-based collection of memory blocks, wherein each block's unique ID is associated with its corresponding size. In accordance with this rationale, an address (designated by the `addr` Coq type) is defined as a tuple comprising a base (the ID of a memory block) and an offset (the offset with respect to the beginning of the block). The allocation table is employed through the `valid` built-in predicate, defined on lines 174–178 of Figure 13, to guarantee the validity of memory accesses. This is achieved by verifying that, given a starting address and a size of a memory segment, the whole memory segment belongs to an existing memory block (and, in particular, does not exceed its allocated size).

Secondly, Wp memory model handles memory values. It is important to note that the memory is abstracted as a set of unsized cells containing any primitive type. Hence, a valid pointer has to refer to one of these cells, and any type can be accessed through it. This is formalized in the Wp memory model through the use of type-specific value tables for each utilized primitive type. To illustrate this, the `linked_ll` predicate only refers to pointers when reasoning about the `next`

```
546 Axiom Q_L_tmp_node_1142_region : ((region 1143%Z) = 2%Z).
    ...
1005 Theorem simpl_wp_goal :
1006     ∀ (Malloc: Numbers.BinNums.Z → Numbers.BinNums.Z) (Mptr: addr → addr) (ctx:addr),
1007     let bl := Mptr (shift ctx 1%Z) in
1008     let ll := L_to_ll Malloc Mptr bl null in
1009     let m := map.Map.set Malloc 1143%Z 1%Z in
1010     framed Mptr →
1011     ((region (base ctx)) <= 0%Z)%Z →
1012     P_linked_ll Malloc Mptr bl null (L_to_ll Malloc Mptr bl null) →
1013     P_linked_ll m Mptr bl null (L_to_ll m Mptr bl null).
```

Fig. 14. Slightly simplified proof obligation in Coq of the first assertion of `getNode` without anonymous blocks.

field of the cell, so it only uses the pointer table, referenced here as `Mptr`. It would also include the integer table `Mint` if we had to reason about pointers of integers. These tables are represented as functions that accept Wp addresses and return values of the type of the table in question. While this modeling approach potentially permits the mapping of the same address into multiple tables, in practice, each address must be mapped into only one value table to ensure soundness of the memory model.

Wp finally introduces the concept of memory regions, which are used to discriminate existentially quantified memory blocks (usually introduced through `valid` clauses in the function's precondition) and new blocks from the function's local memory. The mapping of blocks into their regions is formalized as a function taking a block ID and returning the corresponding region ID. Usually, regions are used to categorize memory locations based on their provenance. The rationale for that is that global variables, addresses to the heap and to callers' local variables are stored in regions with a negative or null ID, and addresses referring to function local memory, such as variables, are stored in regions with a strictly positive ID.

Wp Predicates. The Coq representation in Figure 13 also shows the formalization of each predicate used in the lemma definition. Along with the `valid` predicate already described above, there are several other built-in predicates. The `separated` predicate, defined on lines 156–162, specifies that two addresses point to different memory blocks, or to non-overlapping parts of the same block. The `framed` predicate, defined on lines 227–229, specifies that pointers to non-local memory locations, assigned prior to the current function, actually contain addresses that also point to non-local memory locations (that is, with a negative or null ID).

We observe that the user-defined predicates translated into Coq are very similar to their ACSL counterparts, but accept more arguments, such as an allocation table or value tables. For example, the `unchanged_ll` predicate, defined on lines 794–802 of Figure 13, accepts four new arguments: `Malloc`, `Mptr`, `Malloc1`, and `Mptr1`. These new arguments represent the memory states at the labels (i.e., program points, previously mentioned in Section 4), between which the structure of the linked list must remain unchanged.

Modifying the Allocation Table. Now, thanks to the introduction of the memory model used by Frama-C/Wp, the issue outlined in Section 5 can be elucidated with more clarity. We observe that the allocation table is an argument to both the `linked_ll` predicate and the logic function `to_ll`, for example on lines 1012–1013 of Figure 14. Consequently, given that both of them are defined recursively, any modification to the allocation table requires a proof of the preservation of `linked_ll` and the absence of change of `to_ll` through induction.

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

This happens in the `getNode` function presented in Figure 7 on the first assertion (line 377). Without the added anonymous blocks in Figure 8, the allocation table is modified at the function entry and thus, all assertions about the inductive predicate will now fail.

The proof can still be done inductively in Coq and a slightly modified proof obligation in Coq is presented on lines 1005–1013 in Figure 14 (where all unused preconditions are filtered out). The concept underlying the proof is relatively simple, as only the allocation table has been altered. So, it is necessary to demonstrate that `to_ll` remains unchanged and `linked_ll` is preserved via induction. Both inductions are similar, let us briefly describe the induction used to prove the preservation of `linked_ll` (an interested reader will find the full proof script in the companion artifact). The induction step is about proving that an arbitrary list cell, previously head of the `linked_ll` predicate (the `bl` parameter of the inductive case, lines 778–784 of Figure 12), can still be head of a newly constructed `linked_ll` predicate with the modified allocation table. To achieve this, it is first needed to reconstruct the proofs of all the previously known properties affected by the allocation table, which only correspond to the valid clauses here. To prove that the valid clauses still hold, we have to prove that the modified block (line 1,096 of Figure 14) is always different from the existentially quantified block containing the list cell. To that purpose, the axiom provided by Why3 on line 546 of Figure 14 is mandatory, as it allows reasoning on the memory region of the modified block. The second requisite property is the `framed` built-in predicate (defined on lines 227–229 of Figure 13), which similarly permits reasoning about the memory region of the list cells. It should be noted, however, that to use the `framed` predicate, it is needed to remember that the region ID of the list head is always less than or equal to zero. Therefore, this property must be carried over in the induction step. Once this has been done, it is evident that since both blocks must correspond to different regions, they must be different. From this point onward, the proof of the preservation of `linked_ll` is straightforward.

7 Verification Results

Proof results. Proof results, presented in Figure 15, 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 16 GB 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 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, and so on.

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. It was the case for example for the assertion on line 429 (see Figure 8), presumably, because properties related to memory separation in our case study are more complicated to prove for SMT solvers.

Solutions to memory manipulation problems presented in this article 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.

Development time. The verification work took about 8 person-months and was mainly conducted by a junior verification engineer with no knowledge of the target code and verification tools. This effort includes the time required to study and understand the target code, to write and refine the specification over time, to understand and analyze proof issues, as well as to find, understand and report tool limitations, and to determine, test and implement workarounds.

Thanks to our findings and proposed workarounds, we expect the average verification time for similar code to be considerably shorter now, reduced to 1-2 person-months.

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 [56] provides a security model to capture TPM protections on keys and protocols. The authors of [55] 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 [48]. 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 [8] to formalize and manipulate C linked lists as ACSL logic lists in our effort, while another approach [9] 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 [27], 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 [8]. 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 red-black trees [26] is related to the verification of linked lists in Frama-C using ghost arrays including

the auto-verification aspects [10]. However, the trees themselves were implemented using arrays, since pointers have only been recently introduced in SPARK [25]. 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.

In the verification of C programs, specifying data-structures in a high-level way requires the possibility to express **abstract data types** (**ADT**) in the specification language. That is what the logic types of ACSL basically provide. In high-level languages, in particular object-oriented languages such as Eiffel or Java, the language itself is expressive enough to write side-effect free immutable ADTs that can be leveraged in specifications [46]. Such data-structures are often called models and are related to implementation classes in a fashion similar to how we relate C linked lists with ACSL logic lists. An advantage of such model classes over logic lists is that they are valid classes of the programming language and as such can be executed for dynamic verification tasks. On the deductive verification side, most of the time these classes represent concepts (e.g., sets) that can be translated into elements of theories of provers (and it is possible to verify the faithfulness of the translation [17]). In **Eiffel Verification Environment** (**EVE**), model classes together with auto-active verification techniques [29] were used to verify a container library [47]. Gladisch and Tyszberowicz [31] specify Java methods on linked data structures, including lists, in JML. However, unlike the previously described approaches, they do not relate Java lists to logical lists or model classes. Instead, they use a pure observer method (hence that can be used in specifications) that returns the object held by the list at a given index.

In the verification of Rust programs, the strong non-aliasing rules of Rust ownership policy remove major parts of the complexity when reasoning about memory accesses. As Rust is a system programming language, it can be used in embedded and critical systems. Combined with the compiler invariants, it is a target of choice for formal verification. As the verification ecosystem for Rust is still growing, several tools are available. The authors of [7] present the landscape of Rust verification tools. Creusot [19], relying on Why3, was used to prove iterators' behavior in [20], as well as a SAT solver in [49]. Prusti [4] is implemented on top of the Viper [43], which performs deductive verification using separation logic. A significant case study using Prusti is a verified non-trivial key-value store data structure [30]. Verus [37] aims at the verification of low-level system programs. Unlike Creusot, it supports the verification of concurrent code. Both Creusot and Verus trust the results of Rust's borrow checker. Verus allows the deductive verification of safe Rust code but also goes beyond and allows reasoning on raw pointers using ghost code associated with the pointers, specifying the protocol for their safe usage. This feature is particularly important to reason about linked data structures. Verus has been applied in particular to several system case studies [36].

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 [32]. Such studies include [24], providing feedback on the authors' experience of using ACSL and Frama-C on a real-world example. The authors of [22] managed a large scale formal verification of global security properties on the C code of the JavaCard Virtual Machine. SPARK was used in [15] to translate and formally verify the TCP layer of an embedded TCP/IP library. The authors of [38] 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. The authors of [13] were able to prove thread safety of the interprocess communication mechanism of FreeRTOS using VeriFast [33]. The authors of [51] propose a deductive verification approach for embedded systems that are modeled with SystemC using VerCors [11], a deductive verification tool for concurrent programs. In [45], Oortwijn and Huisman found undesired behavior using VerCors

in an industrial safety-critical traffic tunnel control system. In [18], de Gouw et. al. investigate the Java standard library and its main sorting method using KeY [1], a semi-automatic interactive theorem prover for Java that relying on symbolic execution: they found that Java 8's implementation of TimSort performed incomplete checks of the algorithm's invariants, which could lead to an array-out-of-bound exception. The authors of [52] used KeY to formally specify OpenJDK's `BitSet` class and identified several bugs. Deductive verification has also been applied to ensure the correctness of smart contracts. For instance, the authors of [44] have shown that the Why3 platform [28] can be used to ensure functional properties and the absence of runtime errors, and to write provably correct contracts that can be compiled on the runtime environment for smart contracts on the Ethereum blockchain. Another recent work relies on Dafny [39], a powerful verification-friendly programming language. In particular, Cassez, Fuller, and Quiles [12] propose a methodology using Dafny to model and reason about Ethereum smart contracts, allowing the authors to specify and prove the main properties of a smart contract. Dafny was also used in [14] to implement, specify and verify the QOI image format.

While all the cited tools have been successful in verifying real-life code, their maturity in terms of usability by engineers differ. Some tools have a scarce documentation and are no longer maintained (for e.g., VCC) while others are very actively maintained and extended (Frama-C, KeY, Dafny, Why3, VeriFast, SPARK) with good documentation and even books [2, 35, 40, 42]. In addition, Frama-C is a recognized verification tool by certification authorities, and a JavaCard virtual machine verified with Frama-C was awarded the highest **Evaluation Assurance Level (EAL7)** of the Common Criteria security evaluation [23].

Combining deductive verification with shape analysis. In a recent NIER (New Ideas and Emerging Results) article [6], we sketched a possible combination of deductive verification with shape analysis. That work proposed a verification approach combining Frama-C/Wp with a shape analysis tool, MemCAD [50]. The purpose of MemCAD is to automatically infer precise invariants about programs manipulating complex data structures. It is based on shape analysis [21], a static code analysis technique that verifies properties of recursive, dynamically allocated data structures. It is based on separation logic and abstract interpretation. Unlike in Wp, the analysis in MemCAD is global. In the combined approach, the main idea is to prove structural and separation properties in MemCAD and then to assume them in Frama-C/Wp in order to increase the level of automation of deductive verification and overcome some of its limitations. Thus, this approach proposes to let MemCAD deal with the structural invariants of recursive data structures and separation properties, and to admit them in Frama-C/Wp at some key points. Explicit separation conditions in the acsl predicate for Wp are expressed by the separating conjunction in the MemCAD counterpart. A practical application of this approach was illustrated on a few (slightly simplified) functions of tpm2-tss. This work is still preliminary and opens interesting research questions and perspectives: automation of the proposed verification technique including a coordinated generation of checks and assumptions, proof of its soundness, design of a common (higher-level) specification mechanism for recursive data structures with automatic translation into suitable definitions for MemCAD and Frama-C, as well as evaluation on other relevant case studies.

9 Conclusion and Future Work

This article 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. In addition, it was not designed with the goal of formal verification in mind.

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. The verified properties include both functional properties and the absence of runtime errors such as invalid pointers and buffer overflows, which often lead to security vulnerabilities. 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 [8]) 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.[12]

This work opens the way toward 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 is also an interesting research direction, in particular, by automating the generation of proof scripts as recently proposed in [16]. Combinations of deductive verification with shape analysis, mentioned in Section 8, should be further investigated. An ongoing project on Frama-C, called CoMeMoV, targets the development of collaborative memory models (with different levels of detail for memory representation for different parts of code), which are expected to improve the global capacity of proof for real-life code. 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

We thank Allan Blanchard, Laurent Corbin and Loïc Correnson for useful discussions, and the anonymous referees for helpful comments.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. 2014. The key platform for verification and analysis of java programs. In *Proceedings of the 6th International Conference on Verified Software: Theories, Tools and Experiments.* LNCS, Vol. *8471*, Springer, 55–71. DOI: https://doi.org/10.1007/978-3-319-12154-3_4

[2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification – The KeY Book: From Theory to Practice.* Springer. DOI: https://doi.org/10.1007/978-3-319-49812-6

[3] Will Arthur and David Challener. 2015. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security* (1st. ed.). Apress, USA.

[4] Vytautas Astrauskas, Aurel Bílý, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. 2022. The prusti project: Formal verification for rust. In *Proceedings of the 14th International NASA Formal Methods Symposium.* LNCS, Vol. *13260,* Springer, 88–108. DOI: https://doi.org/10.1007/978-3-031-06773-0_5

[5] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. [n. d.]. *ACSL: ANSI/ISO C Specification Language.* Retrieved from http://frama-c.com/acsl.html

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

[6] Téo Bernier, Yani Ziani, Nikolai Kosmatov, and Frédéric Loulergue. 2024. Combining deductive verification with shape analysis. In *Proceedings of the 27th International Conference on Fundamental Approaches to Software Engineering, Held as Part of the European Joint Conferences on Theory and Practice of Software*. LNCS, Vol. *14573,* Springer, 280–289. DOI: https://doi.org/10.1007/978-3-031-57259-3_14

[7] Alex Le Blanc and Patrick Lam. 2024. Surveying the rust verification landscape. https://doi.org/10.48550/arXiv.2410.01981

[8] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. 2019. Logic against ghosts: Comparison of two proof approaches for a list module. In *Proceedings of the 34th Annual ACM/SIGAPP Symposium on Applied Computing, Software Verification and Testing Track*. ACM, 2186–2195. DOI: https://doi.org/10.1145/3297280.3297495

[9] Allan Blanchard, Frédéric Loulergue, and Nikolai Kosmatov. 2018. Ghosts for lists: A critical module of contiki verified in frama-C. In *Proceedings of the 10th International NASA Formal Methods Symposium*. LNCS, Vol. *10811,* Springer, 37–53. DOI: https://doi.org/10.1007/978-3-319-77935-5

[10] Allan Blanchard, Frédéric Loulergue, and Nikolai Kosmatov. 2019. Towards full proof automation in frama-c using auto-active verification. In *Proceedings of the 11th International NASA Formal Methods Symposium*. LNCS, Vol. *11460,* Springer, 88–105. DOI: https://doi.org/10.1007/978-3-030-20652-9_6

[11] Stefan Blom and Marieke Huisman. 2014. The VerCors tool for verification of concurrent programs. In *Proceedings of the 19th International Symposium on Formal Methods*. LNCS, Vol. 8442, Springer, 127–131. DOI: https://doi.org/10.1007/978-3-319-06410-9_9

[12] Franck Cassez, Joanne Fuller, and Horacio Mijail Anton Quiles. 2022. Deductive verification of smart contracts with dafny. In *Proceedings of the 27th International Conference on Formal Methods for Industrial Critical Systems*. LNCS, Vol. 13487, Springer, 50–66. DOI: https://doi.org/10.1007/978-3-031-15008-1_5

[13] Nathan Chong and Bart Jacobs. 2021. Formally verifying FreeRTOS' interprocess communication mechanism. (2021). Retrieved June 16, 2025 from https://www.amazon.science/publications/formally-verifying-freertos-interprocess-communication-mechanism

[14] Ştefan Ciobâcă and Diana-Elena Gratie. 2024. Implementing, specifying, and verifying the QOI format in dafny: A case study. In *Proceedings of the 19th International Conference on Integrated Formal Methods*. LNCS, Vol. *15234,* Springer, 35–52. DOI: https://doi.org/10.1007/978-3-031-76554-4_3

[15] Guillaume Cluzel, Kyriakos Georgiou, Yannick Moy, and Clément Zeller. 2021. Layered formal verification of a TCP stack. In *Proceedings of the IEEE Secure Development Conference*. IEEE, 86–93. DOI: https://doi.org/10.1109/SecDev51306.2021.00028

[16] Loïc Correnson, Allan Blanchard, Adel Djoudi, and Nikolai Kosmatov. 2024. Automate where automation fails: Proof strategies for frama-C/WP. In *Proceedings of the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Held as Part of the European Joint Conferences on Theory and Practice of Software*. LNCS, Vol. *14570,* Springer, 331–339. DOI: https://doi.org/10.1007/978-3-031-57246-3_18

[17] Ádám Darvas and Peter Müller. 2010. Proving consistency and completeness of model classes using theory interpretation. In *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering*. LNCS, Vol. 6013, Springer, 218–232. DOI: https://doi.org/10.1007/978-3-642-12029-9_16

[18] Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hähnle. 2015. OpenJDK's Java.utils.Collection.sort() is broken: The good, the bad and the worst case. In *Proceedings of the 27th International Conference on Computer Aided Verification*. LNCS, Vol. 9206, Springer, 273–289. DOI: https://doi.org/10.1007/978-3-319-21690-4_16

[19] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A foundry for the deductive verification of rust programs. In *Proceedings of the 23rd International Conference on Formal Methods and Software Engineering*. LNCS, Vol. *13478,* Springer, 90–105. DOI: https://doi.org/10.1007/978-3-031-17244-1_6

[20] Xavier Denis and Jacques-Henri Jourdan. 2023. Specifying and verifying higher-order rust iterators. In *Proceedings of the 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Held as Part of the European Joint Conferences on Theory and Practice of Software*. LNCS, Vol. 13994, Springer, 93–110. DOI: https://doi.org/10.1007/978-3-031-30820-8_9

[21] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2006. A local shape analysis based on separation logic. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. LNCS, Vol. 3920, Springer, 287–302. DOI: https://doi.org/10.1007/11691372_19

[22] Adel Djoudi, Martin Hána, and Nikolai Kosmatov. 2021. Formal verification of a javacard virtual machine with frama-c. In *Proceedings of the 24th International Symposium on Formal Methods*. LNCS, Vol. *13047,* Springer, 427–444. DOI: https://doi.org/10.1007/978-3-030-90870-6_23

[23] Adel Djoudi, Martin Hána, Nikolai Kosmatov, Milan Kříženecký, Franck Ohayon, Patricia Mouy, Arnaud Fontaine, and David Féliot. 2022. A bottom-up formal verification approach for common criteria certification: Application to JavaCard virtual machine. In *Proceedings of the 11th European Congress on Embedded Real-Time Systems*. Retrieved from https://hal.science/hal-03704287

[24] Frank Dordowsky. 2015. 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 (2015), 28–41. DOI: https://doi.org/10.4204/EPTCS.187.3

[25] Claire Dross and Johannes Kanig. 2020. Recursive data structures in SPARK. In *Proceedings of the 32nd International Conference on Computer Aided Verification.* LNCS, Vol. *12225,* Springer, 178–189. DOI: https://doi.org/10.1007/978-3-030-53291-8_11

[26] Claire Dross and Yannick Moy. 2017. Auto-active proof of red-black trees in SPARK. In *Proceedings of the 9th International NASA Formal Methods Symposium.* LNCS, Vol. *10227,* 68–83. DOI: https://doi.org/10.1007/978-3-319-57288-8_5

[27] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. 2004. Contiki - A lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE Conference on Local Computer Networks.* IEEE Computer Society, 455–462. DOI: https://doi.org/10.1109/LCN.2004.38

[28] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - where programs meet provers. In *Proceedings of the 22nd European Symposium on Programming.* LNCS, Vol. 7792, Springer, 125–128. DOI: https://doi.org/10.1007/978-3-642-37036-6_8

[29] Carlo A. Furia, Martin Nordio, Nadia Polikarpova, and Julian Tschannen. 2017. AutoProof: Auto-active functional verification of object-oriented programs. *STTT* 19, 6 (2017), 697–716. DOI: https://doi.org/10.1007/s10009-016-0419-0

[30] Stef Gijsberts. 2023. *Prusti in Practice.* Master's thesis. Radboud University.

[31] Christoph Gladisch and Shmuel S. Tyszberowicz. 2015. Specifying linked data structures in JML for combining formal verification and testing. *Science of Computer Programming* 107-108 (2015), 19–40. DOI: https://doi.org/10.1016/j.scico.2015.02.005

[32] Reiner Hähnle and Marieke Huisman. 2019. Deductive software verification: From pen-and-paper proofs to industrial tools. In *Proceedings of the Computing and Software Science – State of the Art and Perspectives.* LNCS, Vol. 10000, Springer, 345–373. DOI: https://doi.org/10.1007/978-3-319-91908-9_18

[33] Bart Jacobs and Frank Piessens. 2008. *The Verifast Program Verifier.* Technical Report CW-520. KU Leuven.

[34] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Aspects of Computing* 27, 3 (2015), 573–609. DOI: https://doi.org/10.1007/s00165-014-0326-7

[35] Nikolai Kosmatov, Virgile Prevosto, and Julien Signoles (Eds.). 2024. *Guide to Software Verification with Frama-C: Core Components, Usages, and Applications.* Springer. DOI: https://doi.org/10.1007/978-3-031-55608-1

[36] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A practical foundation for systems verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles.* ACM, 438–454. DOI: https://doi.org/10.1145/3694715.3695952

[37] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages* 7, OOPSLA (2023), 806–809. DOI: https://doi.org/10.1145/3586037

[38] Dirk Leinenbach and Thomas Santen. 2009. Verifying the microsoft Hyper-V hypervisor with VCC. In *Proceedings of the 2nd World Congres on Formal Methods.* LNCS, Vol. 5850, Springer, 806–809. DOI: https://doi.org/10.1007/978-3-642-05089-3_51

[39] K. Rustan M. Leino and Valentin Wüstholz. 2014. The dafny integrated development environment. In *Proceedings of the F-IDE 2014 (Electronic Proceedings in Theoretical Computer Science, Vol. 149).* 3–15. DOI: https://doi.org/10.4204/EPTCS.149.2

[40] Rustan K. Leino. 2024. *Program Proofs.* MIT Press.

[41] Frédéric Mangano, Simon Duquennoy, and Nikolai Kosmatov. 2016. A memory allocation module of contiki formally verified with frama-c. a case study. In *Proceedings of the 11th International Conference on Risks and Security of Internet and Systems.* LNCS, Vol. 10158, Springer, 114–120. DOI: https://doi.org/10.1007/978-3-319-54876-0_9

[42] John W. McCormick and Peter C. Chapin. 2015. *Building High Integrity Applications with Spark.* Cambridge University Press.

[43] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A verification infrastructure for permission-based reasoning. In *Proceedings of the 17th 17th International Conference on Verification, Model Checking, and Abstract Interpretation.* LNCS, Vol. 9583, Springer, 41–62.

[44] Zeinab Nehaï and François Bobot. 2019. Deductive proof of industrial smart contracts using why3. In *Proceedings of the FM 2019 International Workshops, Revised Selected Papers.* LNCS, Vol. *12232,* Springer, 299–311. DOI: https://doi.org/10.1007/978-3-030-54994-7_22

[45] Wytse Oortwijn and Marieke Huisman. 2019. Formal verification of an industrial safety-critical traffic tunnel control system. In *Proceedings of the 15th International Conference on Integrated Formal Methods.* LNCS, Vol. 11918, Springer, 418–436. DOI: https://doi.org/10.1007/978-3-030-34968-4_23

[46] Nadia Polikarpova, Carlo A. Furia, and Bertrand Meyer. 2010. Specifying reusable components. In *Proceedings of the 3rd International Conference on Verified Software: Theories, Tools, Experiments.* LNCS, Vol. *6217,* Springer, 127–141. DOI: https://doi.org/10.1007/978-3-642-15057-9_9
[47] Nadia Polikarpova, Julian Tschannen, and Carlo A. Furia. 2018. A fully verified container library. *Formal Aspects of Computing* 30, 5 (2018), 495–523. DOI: https://doi.org/10.1007/s00165-017-0435-1
[48] Jianxiong Shao, Yu Qin, and Dengguo Feng. 2018. Formal analysis of HMAC authorisation in the TPM2.0 specification. *IET Information Security* 12, 2 (2018), 133–140. DOI: https://doi.org/10.1049/iet-ifs.2016.0005
[49] Sarek Høverstad Skotåm. 2022. *CreuSAT-Using Rust and Creusot to Create the World's Fastest Deductively Verified SAT Solver.* Master's thesis.
[50] Pascal Sotin and Xavier Rival. 2012. Hierarchical shape abstraction of dynamic structures in static blocks. In *Proceedings of the 10th Asian Symposium on Programming Languages and Systems.* LNCS, Vol. 7705, Springer, 131–147. DOI: https://doi.org/10.1007/978-3-642-35182-2_10
[51] Philip Tasche, Raúl E. Monti, Stefanie Eva Drerup, Pauline Blohm, Paula Herber, and Marieke Huisman. 2024. Deductive verification of parameterized embedded systems modeled in systemc. In *Verification, Model Checking, and Abstract Interpretation - 25th International Conference, VMCAI 2024, London, United Kingdom, January 15-16, 2024, Proceedings, Part II.* Rayna Dimitrova, Ori Lahav, and Sebastian Wolff (Eds.), LNCS, Vol. 14500, Springer, 187–209. DOI: https://doi.org/10.1007/978-3-031-50521-8_9
[52] Andy S. Tatman, Hans-Dieter A. Hiep, and Stijn de Gouw. 2023. Analysis and formal specification of openjdk's bitset. In *Proceedings of the 18th International Conference on Intergrated Formal Methods.* LNCS, Vol. 14300, Springer, 134–152. DOI: https://doi.org/10.1007/978-3-031-47705-8_8
[53] The Coq Development Team. [n. d.]. The Coq Proof Assistant. Retrieved June 16, 2025 from http://coq.inria.fr
[54] Trusted Computing Group. 2019. Trusted Platform Module Library Specification, Family "2.0", Level 00, Revision 01.59 – November. Retrieved from https://trustedcomputinggroup.org/work-groups/trusted-platform-module/ last accessed: May 2023.
[55] Weijin Wang, Yu Qin, Bo Yang, Yingjun Zhang, and Dengguo Feng. 2016. Automated security proof of cryptographic support commands in TPM 2.0. In *Proceedings of the 18th International Conference on Information and Communications Security.* LNCS, Vol. 9977, Springer, 431–441. DOI: https://doi.org/10.1007/978-3-319-50011-9_33
[56] Qianying Zhang and Shijun Zhao. 2020. A comprehensive formal security analysis and revision of the two-phase key exchange primitive of TPM 2.0. *Computer Networks* 179 (2020), 107369. DOI: https://doi.org/https://doi.org/10.1016/j
[57] Yani Ziani, Nikolai Kosmatov, Frédéric Loulergue, Daniel Gracia Pérez, and Téo Bernier. 2023. Towards formal verification of a TPM software stack. In *Proceedings of the 18th International Conference on Integrated Formal Methods.* LNCS, Vol. *14300,* Springer, 93–112. DOI: https://doi.org/10.1007/978-3-031-47705-8_6

Received 7 September 2024; revised 4 March 2025; accepted 3 June 2025