Combining Deductive Verification with Shape Analysis

Téo Bernier[1][0009−0003−4834−7126], Yani Ziani[1,2][0009−0000−8540−1273], Nikolai Kosmatov[1][0000−0003−1557−2813], and Frédéric Loulergue[2][0000−0001−9301−7829]

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

Abstract. Deductive verification tools can prove a large range of program properties, but often face issues on recursive data structures. Abstract interpretation tools based on separation logic and shape analysis can efficiently reason about such structures but cannot deal with so large classes of properties. This short paper presents an ongoing work on combining both techniques. We show how a deductive verifier for C programs, Frama-C/Wp, can benefit from a shape analysis tool, MemCAD, where structural and separation properties proved in the latter become assumptions for the former. A case study on selected functions of the tpm2-tss library using linked lists confirms the interest of the approach.

Keywords: deductive verification, shape analysis, abstract interpretation, linked lists, Frama-C, MemCAD

1 Introduction

Context and Motivation. Deductive verification tools were successfully used in many case studies [4] to prove a large range of safety, security and functional properties. Such tools often have issues to conduct automatic proof on code with *recursive data structures* (e.g. linked lists, trees, etc.), in particular, due to complex memory models they need. The user has to guide the proof by interactively proved lemmas, assertions, etc. Abstract interpretation tools based on separation logic and shape analysis [3] can efficiently reason about such structures but typically cannot deal with so large classes of properties. This short paper presents new ideas and emerging results on combining both techniques trying to take the best of both worlds.

Approach and Results. We present a verification approach combining a popular deductive verifier for C programs, Frama-C/Wp [6], with a shape analysis tool, MemCAD [10]. 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 the latter and overcome some of its limitations. We apply it on a real-life case study using linked lists: a few (slightly simplified) functions of tpm2-tss[3], a popular library for communication with a Trusted Platform Module (TPM). Recent work [11] demonstrated that deductive verification of the library functions manipulating linked lists was relatively hard, and required many additional lemmas and assertions.

The contributions of this paper include the presentation of a combined verification technique using deductive verification and shape analysis, its illustration with Frama-C/Wp and MemCAD on a function manipulating linked lists, as well as a successful case study on a set of functions of the tpm2-tss library.

2 Background

2.1 Deductive Verification with Frama-C/Wp

Frama-C [6] is an integrated toolbox built around a kernel offering core services and plugins dedicated to specific analysis or verification tasks for C code, e.g. value analysis, runtime assertion checking and deductive verification. Acsl (ANSI C Specification Language) [6] is the common specification language of the plugins. The Wp plugin performs *modular* deductive verification: each function is verified independently. It generates verification conditions (VCs) from the C code with Acsl annotations and requests their proof by the QED simplifier or by external provers.

We illustrate the main Acsl features on the running example[4] of Fig. 1, 3, 4, 5, presented as we go, where Acsl notation (e.g. `\forall`, `integer`, `==>`, `<=`, `&&`) is pretty-printed (resp., as $\forall$, $\mathbb{Z}$, $\Rightarrow$, $\leq$, $\wedge$). Lines 69–85 of Fig. 4 show a *contract* for function `list_push` (detailed below) that adds a new value into a linked list (cf. Lines 1–2 of Fig. 1), allocating a new cell. The contract includes pre-conditions (`requires` clauses) and post-conditions (`ensures` clauses). The `assigns` clause is a special kind of post-condition that indicates the memory locations the function is allowed to modify. Acsl formulas are mostly multi-sorted first-order logic where types are either C types or logic types (such as $\mathbb{Z}$, the type of mathematical integers). Acsl provides built-in constructs such as `\result` (the value returned by the function) and predicates such as `\valid(p)` (stating that pointer `p` refers to an allocated memory location, so that `*p` can be safely read and written) and `\separated(p1,p2,...)` (stating that the memory locations referred to by given pointers do not intersect). Notice that the considered memory locations are here indicated by pointers. Users can define predicates such as those in Fig. 1, adapted here from a previous work [1] on verifying linked lists in Wp.

The main predicate is the inductively defined predicate `linked_ll` (Lines 10–19) stating that a linked list (segment) of `int` values (defined on Lines 1–2) from pointer `bl` to pointer `el` (excluded) is a well-formed list represented by an Acsl logical list `ll`. In other words, `ll` contains the pointers to the cells of that list segment (or the whole list if `el` is `NULL`). Acsl lists are similar to

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

[4] Available in a companion artifact on http://doi.org/10.5281/zenodo.10458675

```
1  typedef struct cell_s {struct cell_s* next; int data;} cell;
2  typedef cell* list;
3  /*@
4    predicate ptr_sep_from_list{L}(cell* c, \list<cell*> ll) =
5      ∀ ℤ n; 0 ≤ n < \length(ll) ⇒ \separated(c, \nth(ll, n));
6    predicate dptr_sep_from_list{L}(cell** c, \list<cell*> ll) =
7      ∀ ℤ n; 0 ≤ n < \length(ll) ⇒ \separated(c, \nth(ll, n));
8    predicate in_list{L}(cell* c, \list<cell*> ll) =
9      ∃ ℤ n; 0 ≤ n < \length(ll) ∧ \nth(ll, n) == c;
10   inductive linked_ll{L}(cell *bl,cell *el,\list<cell*> ll) {
11     case linked_ll_nil{L}:
12       ∀ cell *el; linked_ll{L}(el, el, \Nil);
13     case linked_ll_cons{L}:
14       ∀ cell *bl, *el, \list<cell*> tail;
15         (\separated(bl, el) ∧ \valid(bl) ∧
16         linked_ll{L}(bl->next, el, tail) ∧
17         ptr_sep_from_list(bl, tail)) ⇒
18           linked_ll{L}(bl, el, \Cons(bl, tail));
19   }
20   predicate unchanged_ll{L1, L2}(\list<cell*> ll) =
21     ∀ ℤ n; 0 ≤ n < \length(ll) ⇒
22       \valid{L1}(\nth(ll,n)) ∧ \valid{L2}(\nth(ll,n)) ∧
23       \at((\nth(ll,n))->next, L1) == \at((\nth(ll,n))->next, L2) ∧
24       \at((\nth(ll,n))->data, L1) == \at((\nth(ll,n))->data, L2);
25   axiomatic cell_to_ll {
26     logic \list<cell*> to_ll{L}(cell* beg, cell* end)
27       reads {node->next | cell* node;
28               \valid(node) ∧ in_list(node, to_ll(beg, end))};
29     axiom to_ll_nil{L}: ∀ cell *node;
30       to_ll{L}(node, node) == \Nil;
31     axiom to_ll_cons{L}: ∀ cell *beg, *end;
32       (\separated(beg, end) ∧ \valid{L}(beg) ∧
33       ptr_sep_from_list{L}(beg, to_ll{L}(beg->next, end))) ⇒
34         to_ll{L}(beg, end) ==
35         \Cons(beg, to_ll{L}(beg->next, end));
36   }
37 */
38 #include "lemmas_min.h"
```

Fig. 1. Types and ACSL predicates for linked lists.

lists in functional programming. In the inductive case (`linked_ll_cons`) overlapping list cells (or cyclic lists) are avoided by requiring that the first cell `bl` is separated from all the other cells in the list including `el`, so the list is well-formed. The predicates on Lines 4–9 use predefined functions: `\length` and `\nth` that returns the n^{th} element of a logic list. Predicates can take one or several program points (C labels plus some ACSL labels: `Pre` and `Post`). The built-in `\at(e, L)` specifies the value of an expression `e` at a label `L`. Using these features, `unchanged_ll` states that a logic list does not change between two program points (Lines 20–24). Finally, Lines 25–36 define an axiomatic function `to_ll` that constructs a logic list from a C linked list. While it would be possible to write `requires ∃\list<cell>ll; linked_ll(*pl, NULL, ll);` instead of Line 72 of Figure 4, the scope of the existential quantifier is just this line. Therefore, `ll` cannot be used in the post-conditions, hence the need for `to_ll`.

Let us now detail the contract of `list_push` (its code is detailed below). The pre-conditions state that `pl` is a valid pointer to a list (Line 70), separated from every element in the list (Line 71), and refers to a linked list verifying the

```
a ll_cell<0,0> :=
b | [0]
c   - emp
d   - this = 0
e | [2 addr int]
f   - this->0 |-> $0 * $0.ll_cell() *
g     this->4 |-> $1
h   - alloc(this, 8) & this ≠ 0.
i
j plist<0, 0> :=
k | [1 addr]
l   - this->0 |-> $0 * $0.ll_cell()
m   - alloc(this, 4) & this ≠ 0.
```

```
n cell<0,0> :=
o | [0]
p   - emp
q   - this = 0
r | [2 addr int]
s   - this->0 |-> $0 * this->4 |-> $1
t   - alloc(this, 8) & this ≠ 0.
u
v cell_plist<0,0> :=
w | [2 addr addr]
x   - this->0 |-> $0 * $0.cell() *
y     this->4 |-> $1 * $1.plist()
z   - alloc(this, 8) & this ≠ 0.
```

Fig. 2. Inductive predicates for MemCAD.

inductive predicate `linked_ll` (Line 72). Line 73 specifies that the only locations the function is allowed to modify are `*pl`, the head pointer of the list, and `\at(**pl, Post)`, the first element of the list at the exit point, i.e. the freshly allocated cell. We cannot reference the new list cell at the entry point because it is not allocated yet. In post-conditions, the returned value indicates whether or not the allocation is successful (Line 76). Regardless of the success, we expect the list invariants to hold (Lines 74–75). In case the allocation fails, we expect the pointer `*pl` and the list contents to be unchanged (Lines 77–79). If it succeeds, we expect the list to be composed of the new cell followed by the old list (Lines 80–81), the old list being unchanged (Lines 82–83), and the fields of the new cell, `next` and `data`, resp., to point to the old list (Line 84) and to contain the expected value (Line 85).

2.2 Shape Analysis with MemCAD

The purpose of MemCAD [10] is to automatically infer precise invariants about programs manipulating complex data structures. It is based on shape analysis [3], a static code analysis technique that discovers and verifies properties of recursive, dynamically allocated data structures. It relies on separation logic and abstract interpretation. Unlike in Wp, the analysis is global.

To use MemCAD on linked lists defined on Lines 1–2 of Fig. 1, the user first defines an inductive predicate expressing a structural invariant of a well-formed linked list, such as predicate `ll_cell` on Lines a–h of Fig. 2. A list, i.e. a pointer to a list cell, satisfies the predicate in two cases. Each case defines a memory separation formula and additional constraints. In the first case, the pointer is null (Line d) and no specific memory separation is required (Line c). This case has no additional arguments (cf. `[0]` on Line b). The second case has two (existentially quantified) arguments: an address and an integer (Line e), denoted, resp., by `$0` and `$1` in the rest of the case. The pointer is non null and refers to a valid memory block of 8 bytes (Line h), assuming a 32-bit system. Lines f–g define the values of the fields `next` and `data` (at offsets 0 and 4) as `$0` and `$1`, and require separation between those fields and the rest of the list. The separation is expressed by the separating conjunction “`*`” [10]. Notice

```
//@ assigns \nothing;
void mc_chk_plist(list* pl) {
  _memcad("check_inductive(pl,plist)");
}

typedef struct {cell* c; list* pl;} cell_plist;

//@ assigns \nothing;
void mc_chk_sep_cell_plist(cell* c, list* pl) {
  cell_plist tmp;
  tmp.c = c; tmp.pl = pl;
  cell_plist* ptmp = &tmp;
  _memcad("check_inductive(ptmp,cell_plist)");
}
```

Fig. 3. Auxiliary MemCAD checks for linked lists.

that "`...*$0.ll_cell()*...`" on Line f specifies separation recursively, for all list cells reached by the predicate via the inductive case. The user can insert the instruction `_memcad("add_inductive(l,ll_cell)");` to *assume* that list `l` respects predicate `ll_cell`, or `_memcad("check_inductive(l,ll_cell)");` to *check* the same property in MemCAD.

Predicate `cell` on Lines n–t is very close to predicate `ll_cell` except that it only defines one list cell without recursion. Predicate `plist` on Lines j–m expresses that a double pointer to a list cell (i.e. of type `list*`) is valid, refers to a well-formed list and is separated from its cells. Predicate `cell_plist` is explained below.

3 Combined Approach

3.1 Shape Analysis Assisted Verification

To prove complex memory-related annotations with Wp on real-life code [11], the user typically has to manually annotate the code with many additional carefully chosen assertions establishing structural invariants and separation properties at several intermediate program points, and to add numerous lemmas to facilitate reasoning about them (whose proof must usually be done manually in Coq, an interactive proof assistant). Our approach proposes to let MemCAD deal with the structural invariants of recursive data structures and separation properties, and to admit them in Wp at some key points.

In order to use both tools simultaneously in this way, we first need to show the equivalence between MemCAD and Wp inductive predicates. For MemCAD, predicate `ll_cell` (Lines a–h of Fig. 2) specifies that each element of the list is a valid cell, is separated from every other cell of the list and the list is null-terminated. This is equivalent to the `linked_ll` predicate for Wp (Lines 10–19 of Fig. 1) when we consider the whole list. Indeed, when `el` is `NULL`, this predicate also means that every list cell is valid and separated from any other list cell, and the list is null-terminated. Explicit separation conditions in the Acsl predicate for Wp are expressed by the separating conjunction in the MemCAD

```
 59 /*@
 60   assigns \nothing;
 61   ensures \result ≠ NULL ⇒ (\valid(\result) ∧
 62     \result->next == NULL ∧ \result->data == 0); */
 63 cell* calloc_cell() {
 64   cell* c = malloc(sizeof(cell));
 65   if (c) { c->next = NULL; c->data = 0; }
 66   return c;
 67 }
 68
 69 /*@
 70   requires \valid(pl);
 71   requires dptr_sep_from_list(pl,to_ll(*pl, NULL));
 72   requires linked_ll(*pl, NULL, to_ll(*pl, NULL));
 73   assigns *pl, \at(**pl, Post);
 74   ensures dptr_sep_from_list(pl, to_ll(*pl, NULL));
 75   ensures linked_ll(*pl, NULL, to_ll(*pl, NULL));
 76   ensures \result \in {0, 1};
 77   ensures \result == 0 ⇒
 78     unchanged_ll{Pre, Post}(to_ll(*pl, NULL));
 79   ensures \result == 0 ⇒ *pl == \old(*pl);
 80   ensures \result == 1 ⇒
 81     to_ll(*pl, NULL) == ([|*pl|] ^ to_ll(\old(*pl), NULL));
 82   ensures \result == 1 ⇒
 83     unchanged_ll{Pre, Post}(to_ll(\old(*pl), NULL));
 84   ensures \result == 1 ⇒ (*pl)->next == \old(*pl);
 85   ensures \result == 1 ⇒ (*pl)->data == data; */
 86 int list_push(list* pl, int data) {
 87   cell* c = calloc_cell();
 88   if (!c) return 0;
 89   mc_chk_sep_cell_plist(c, pl);
 90   //@ admit ptr_sep_from_list(c,to_ll(*pl,NULL));
 91   //@ admit \separated(pl, c);
 92   //@ ghost Alloc:;
 93   c->next = *pl;
 94   //@ assert unchanged_ll{Alloc,Here}(to_ll{Alloc}(*pl,NULL));
 95   c->data = data;
 96   //@ ghost Link:;
 97   *pl = c;
 98   /*@ assert unchanged_ll{Link,Here}(
 99     to_ll{Link}(\at(*pl,Pre),NULL)); */
100   mc_chk_plist(pl);
101   //@ admit dptr_sep_from_list(pl,to_ll(*pl,NULL));
102   //@ admit linked_ll(*pl,NULL,to_ll(*pl,NULL));
103   return 1;
104 }
```

Fig. 4. Functions `calloc_cell` and `list_push` with contracts.

counterpart. (Notice that separation of `bl` with `NULL` on Line 15 is trivial.) The sequence of list elements, expressed by a logic list in ACSL and used to prove functional properties about the contents of the list (cf. Lines 80–81) in WP, does not need to be specified for MEMCAD, which we only use to reason about structural properties.

To check if invariants hold in MEMCAD, we define check functions shown in Fig. 3. These functions are specified to be side-effect-free (cf. Lines 40, 47) to prevent interference with the proof in WP.

The first function, `mc_chk_plist` (Lines 41–43), checks that `pl` respects the `plist` predicate, i.e. is a valid pointer to a well-formed list from which it is separated (Line 42, see also Lines j–m of Fig. 2).

The goal of the second function, `mc_chk_sep_cell_plist`, is to check that `c` refers to a list cell, `pl` respects the `plist` predicate, and the corresponding pointer and the list cells are separated from the cell referred to by `c`. To do that in MemCAD, we introduce an ad-hoc structure `cell_plist` with both pointers (Line 45). The function initializes a local structure (Lines 49–50) and takes its address (Line 51) in order to express the required check (Line 52). This check relies on the predicate `cell_plist` (Lines v–z of Fig. 2) stating that the given pointer is non-null and refers to a structure with two pointers at offsets 0 and 4, denoted `$0` and `$1`, referring, resp., to a cell and to a double pointer to a well-formed list, which are separated (between them and from the list cells). Notice that "`...*$1.plist()`" on Line y specifies separation recursively, that is, from all locations considered in separation constraints reached via `plist` (and hence via `ll_cell`).

An important benefit of using MemCAD is its capacity to automatically handle dynamic memory allocation, which is not yet supported in Wp. Thus, we define a custom allocator that simulates the behavior of `calloc` for list cells on Lines 59–67 of Fig. 4. Wp uses its contract, which is simple but currently unprovable by Wp since dynamic allocation is not supported (it should become provable when this support is added into Wp).

3.2 Proof of Function `list_push`

We illustrate our approach on function `list_push` of Fig. 4. It tries to allocate a new cell (Lines 87–88), and, in case of success, puts it on top of the list with the given data (Lines 93, 95, 97, 103). Lines 92, 96 define *ghost* labels (that is, labels used only in annotations).

Lines 89–91 show how we use MemCAD to verify that the new cell (referred to by `c`) is separated both from the list cells and the pointer referred to by `pl` (Line 89), and introduce these properties as assumptions for Wp (`admit` clauses on Lines 90–91). They help Wp to prove in an `assert` clause on Line 94 that the list remains unchanged since label `Alloc` (i.e. Line 92) despite writing into the new cell on Line 93, and a similar assertion for the old list on Lines 98–99 despite the assignment on Line 97.

Instead of reasoning about the modified list directly in Wp—which often presents another difficulty for deductive verification—we let MemCAD check the list invariants on Line 100 and admit them on Lines 101–102 for Wp to prove the post-conditions. Thanks to those assumptions, Wp successfully proves this function. Notice that the check instruction for MemCAD and the admit instructions for Wp are placed (for the moment, manually) at the same program location to ensure the soundness of the global verification.

In order to have a full proof, we also need to run MemCAD to verify all the checks in `list_push`. For this purpose, we define a wrapper in Fig. 5 to analyze

```
106 int mc_verify_list_push(void) {
107   list* pl; int i; _memcad("add_inductive(pl,plist)");
108   list_push(pl, i);
109 }
```

Fig. 5. Wrapper to verify `list_push` in MemCAD.

the call to `list_push` on Line 108 with an arbitrary list respecting the given preconditions (which correspond in MemCAD, as we explained above, to assuming predicate `plist` for `pl`, cf. Lines 70–72, 107). MemCAD also succeeds in its analysis, hence, we can conclude that our function respects its Acsl contract.

While the annotation step is done manually in the current work, it can be better automated in the future. A coordinated generation of checks and assumptions for a given recursive data structure for both tools will facilitate the verification and the justification of soundness of the combined approach. An early idea consists in defining a domain-specific language for the description of the target recursive data structure that is then used for the generation of necessary predicates for MemCAD and for Wp as well as necessary assumptions and checks. The investigation of this research direction is left for future work.

4 Case study on the tpm2-tss library

We tested our approach on a few (slightly simplified) functions of the tpm2-tss library, a widely used open-source implementation of the TPM Software Stack (TSS)[5] designed to access the Trusted Platform Module (TPM). The library uses a linked list to store and use TPM resources, such as objects sent to and received from the TPM. List cells are dynamically allocated. Simplifications were applied to data structures used for list cells (and their treatment).

We consider two functions, to add an object and to look for an object in a list, with one called function, and apply MemCAD to verify separation properties for a newly allocated cell that Wp is currently not able to deduce. A recent study [11] demonstrated that deductive verification with Wp of these functions required many additional lemmas and assertions, as well as the replacement of the dynamic memory allocation by a static allocator. Interestingly, the difficulty to verify real-life code was not caused by complex operations on lists—these operations are in reality quite simple in the target code—but by the difficulty to reason about the recursive data structure itself.

The proposed approach combining deductive verification with shape analysis allows us to perform a complete proof with less effort and without replacing dynamic allocator by a static allocator. On the considered functions, the proof with Wp alone [11] required 14 lemmas, leading to the generation of 241 proof obligations, one of which required a manually created Wp script, and took 4m50s. Thanks to combining Wp and MemCAD in our work, we could remove ∼45

[5] https://trustedcomputinggroup.org/work-groups/software-stack/

auxiliary ACSL annotations and 5 lemmas, so the proof required only 9 lemmas, leading to 194 proof obligations using no scripts, and took 1min47s in total for WP and MEMCAD (the latter taking less than 1 sec.).

5 Related Work and Conclusion

Related Work. Various tools based on separation logic were proposed, such as VeriFast [8], Viper [7], VerCors [2]. He et al. [5] extract functional specification from imperative programs using a memory-safe type system and insert dynamic checks into the specification. GRASShopper [9] combines separation logic with an SMT-based verifier. Unlike in our work, GRASShopper does not integrate abstract interpretation based shape analysis (which allows us to infer structural invariants with MEMCAD without having to provide loop invariants for this tool). Issues reported in a recent study [11] motivate such combinations for complex real-life code with recursive data structures. Our work continues previous efforts by proposing a combination of weakest-precondition based deductive verification with abstract interpretation based shape analysis on the source-code level, which, to the best of our knowledge, was not studied and evaluated before.

Conclusion and Future Work. This short paper has presented an approach combining deductive verification with FRAMA-C/WP and shape analysis with MEMCAD. Separation properties and structural invariants for linked data structures can be more easily proved by the latter, and then used as assumptions in the former, thus allowing it to focus on other properties. This work is still ongoing 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.

Data-Availability Statement. Code examples used in this paper are available online as a companion artifact on http://doi.org/10.5281/zenodo.10458675. The artifact includes a Virtual Machine containing the installed tools and code examples used, and can be used to reproduce the results of this paper.

Acknowledgment. Part of this work was supported by ANR (grants ANR-22-CE39-0014, ANR-22-CE25-0018) and French Ministry of Defense via a PhD grant of Yani Ziani. We thank Allan Blanchard, Laurent Corbin, Loïc Correnson, Daniel Gracia Pérez and Xavier Rival for fruitful discussions, and the anonymous referees for helpful comments.

References

1. Blanchard, A., Kosmatov, N., Loulergue, F.: Logic against ghosts: Comparison of two proof approaches for a list module. In: 34th Symp. on Applied Comput-

ing, Software Verification and Testing Track (SAC-SVT'19). pp. 2186–2195. ACM (2019)

2. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: Verification of parallel and concurrent software. In: 13th Int. Conf. on Integrated Formal Methods (iFM'17). LNCS, vol. 10510, pp. 102–110. Springer (2017)
3. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: 12th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06). LNCS, vol. 3920, pp. 287–302. Springer (2006)
4. Hähnle, R., Huisman, M.: Deductive software verification: From pen-and-paper proofs to industrial tools. In: Computing and Software Science - State of the Art and Perspectives, LNCS, vol. 10000, pp. 345–373. Springer (2019)
5. He, P., Westbrook, E., Carmer, B., Phifer, C., Robert, V., Smeltzer, K., Stefanescu, A., Tomb, A., Wick, A., Yacavone, M., Zdancewic, S.: A type system for extracting functional specifications from memory-safe imperative programs. Proc. ACM Program. Lang. **5**, 1–29 (2021)
6. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. **27**(3), 573–609 (2015)
7. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: 17th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'16). LNCS, vol. 9583, pp. 41–62. Springer (2016)
8. Philippaerts, P., Mühlberg, J.T., Penninckx, W., Smans, J., Jacobs, B., Piessens, F.: Software verification with VeriFast: Industrial case studies. Science of Computer Programming **82**, 77–97 (2014)
9. Piskac, R., Wies, T., Zufferey, D.: GRASShopper - complete heap verification with mixed specifications. In: 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14). LNCS, vol. 8413, pp. 124–139. Springer (2014)
10. Sotin, P., Rival, X.: Hierarchical shape abstraction of dynamic structures in static blocks. In: 10th Asian Symposium on Programming Languages and Systems (APLAS'12). LNCS, vol. 7705, pp. 131–147. Springer (2012)
11. Ziani, Y., Kosmatov, N., Loulergue, F., Gracia Pérez, D., Bernier, T.: Towards formal verification of a TPM software stack (2023), http://arxiv.org/abs/2307.16821