Logic against Ghosts: Comparison of Two Proof Approaches for a List Module

Allan Blanchard
Inria Lille – Nord Europe
Villeneuve d'Ascq, France
allan.blanchard@inria.fr

Nikolai Kosmatov
CEA, List, Software Reliability Lab
Gif-sur-Yvette, France
nikolai.kosmatov@cea.fr

Frédéric Loulergue
Northern Arizona University
Flagstaff, USA
frederic.loulergue@nau.edu

ABSTRACT

Modern verification projects continue to offer new challenges for formal verification. One of them is the linked list module of Contiki, a popular open-source operating system for the Internet of Things. It has a rich API and uses a particular list representation that make it different from the classical linked list implementations. Being widely used in the OS, the list module is critical for reliability and security. A recent work verified the list module using ghost arrays.

This article reports on a new verification effort for this module. Realized in the Frama-C/Wp tool, the new approach relies on logic lists. A logic list provides a convenient high-level view of the linked list. The specifications of all functions are now proved faster and almost all automatically, only a small number of auxiliary lemmas and a couple of assertions being proved interactively in Coq. The proposed specifications are validated by proving a few client functions manipulating lists. During the verification, a more efficient implementation for one function was found and verified. We compare the new approach with the previous effort based on ghost arrays, and discuss the benefits and drawbacks of both techniques.

CCS CONCEPTS

• **Security and privacy** → **Logic and verification**; *Operating systems security*; • **Software and its engineering** → **Software verification**; • **Computer systems organization** → Sensor networks;

KEYWORDS

deductive verification, linked lists, formal specification, operating system, internet of things, Frama-C

ACM Reference Format:
Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. 2018. Logic against Ghosts: Comparison of Two Proof Approaches for a List Module. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Nowadays, the Internet of Things (IoT) software has become a new area of application for formal verification. Billions of connected devices and services are in use, and their number is continuously growing. This many devices connected to the internet raises many security risks, which can be addressed using formal methods.

Context and Motivation. Formal verification of IoT software brings new verification targets and challenges. In this paper, we consider the linked list module of Contiki, a widely used open-source operating system for IoT devices. Contiki [10] provides basic features needed from any OS as well as some others more specific to IoT devices, such as sensor drivers and full low-power IPv6 connectivity, including 6TiSCH, 6LoWPAN, RPL, or CoAP standards. Contiki is written in C with a focus on memory optimization and power consumption, as targeted devices are generally small battery-operated chips. The kernel is linked to platform-specific drivers at compile-time. At the beginning of its development in 2002, the security of Contiki was not the main concern. Security of communication was introduced later, while the source code has become a target of formal verification only very recently [6, 19, 21].

A particularly interesting verification target discovered in these projects is the linked list module, one of the most critical modules of Contiki. This module offers a very rich API that is usable in many other contexts, and relies on a specific representation of lists. That makes it different from other classical implementations of linked lists and justifies the need for a dedicated verification effort. In a previous work [6], we performed formal specification and deductive verification of the list module using the Frama-C verification platform [15]. The approach is based on a companion ghost array, reflecting the contents of the actual linked list and used to specify the list manipulations more easily. That approach allowed for a proof of almost all functions of the module (except one function), and helped to detect and fix a dangerous bug (in the remaining unproven function, called `list_insert`). Most properties were proved automatically, while a couple of annotations, as well as some lemmas, were proved interactively.

However, the ghost array approach has its drawbacks. It requires a significant annotation effort from the verification engineer and prevents provers from being efficient because of a big number of necessary annotations (in particular, various intermediate assertions and separation properties). Given that the proof of simpler functions already required a lot of assertions to deduce some — often relatively straightforward — properties, and that automatic proof for some of them failed, the verification of the remaining unproven function `list_insert` appears even more challenging since it is more complex and involves a bigger number of possible behaviors. We presume that automatic deductive verification is approaching its limits — at least as for today — for that kind of specification technique and that kind of code. Moreover, from a methodological

Conference'17, July 2017, Washington, DC, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
1  struct list {
2    struct list *next; // must be the first field
3    //int k;           // possible data fields
4  };
5  typedef struct list ** list_t;
6  //Initialize a list
7  void list_init(list_t pLst);
8  //Get the length of a list
9  int  list_length(list_t pLst);
10 //Get the first element of a list
11 void * list_head(list_t pLst);
12 //Get the last element of a list
13 void * list_tail(list_t pLst);
14 //Remove the first element of a list
15 void * list_pop (list_t pLst);
16 //Add an item to the start of a list.
17 void list_push(list_t pLst, void *item);
18 //Remove the last element of a list.
19 void * list_chop(list_t pLst);
20 //Add an item at the end of a list.
21 void list_add(list_t pLst, void *item);
22 //Duplicate a list (copy head pointer)
23 void list_copy(list_t dest, list_t src);
24 //Remove element item from a list
25 void list_remove(list_t pLst, void *item);
26 //Insert newitem after previtem in a list
27 void list_insert(list_t pLst,
28   void *prev, void *new);
29 //Get the element following item
30 void * list_next(void *item);
```

Figure 1: API of the `list` module of Contiki (for trivial lists and lists with one integer data field)

point of view, it can be desirable to have a specification approach with a more abstract view of the list than a ghost array.

Approach and Results. The present paper reports on a new verification effort for the list module based on a completely different specification approach. All functions of the module (including the function not proved correct previously) have been specified and formally verified. As in the previous work [6], the verification has been performed in the Frama-C/Wp tool [15] and relies on a formal specification in Acsl [4], the specification language offered by Frama-C. The proposed specifications have also been validated by proving a few client functions manipulating lists.

The new specification approach is based on Acsl logic lists whose support in the tool has been recently improved. The main principle of the verification is to show how the linked list data structure can be related to a logic view of the list. This relation is defined inductively. Thus, to allow automatic reasoning with SMT solvers in the rest of the proofs, we state a few lemmas about these inductive properties. All those lemmas have been proved using the Coq proof assistant [25]. Thanks to them, the specifications of all functions are proved automatically (except for a couple of assertions in the most complex function, `list_insert`, which were proved in Coq) and relatively fast, which makes us believe that the new approach is more suitable for automatic deductive verification than the previous approach [6] using ghost arrays.

Finally, we compare both approaches and point out some advantages and drawbacks of each technique. This comparison will help to choose the most suitable approach for similar case studies in the future.

The contributions of this work include

- formal specification of the `list` module[1] of Contiki in the Acsl specification language and its *complete* deductive verification using the Wp plugin of Frama-C;
- a presentation of the underlying approach based on the Acsl logic list type;
- formal statement and proof of several lemmas useful for reasoning about this representation;
- a preliminary validation of the proposed specification of the module via a successful verification of a few annotated test functions dealing with lists;

[1]Complete annotated code available at https://allan-blanchard.fr/code/contiki-list-verified-acsl-list.zip.

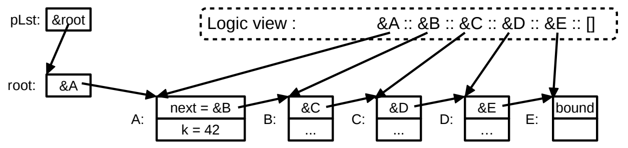

Figure 2: Parallel view of a list prefix using a logic list, formally defined by the `linked_ll` predicate in Figure 4

- a comparison with our previous approach based on ghost arrays, underlining strong and weak points of each technique.

Outline. The paper is organized as follows. Section 2 presents the specifics of the linked list module. Section 3 describes our verification approach. Section 4 presents the results of the verification, and Section 5 compares this work with a previous verification approach. Section 6 provides some related work, while Section 7 gives the conclusion and future work.

2 THE LIST MODULE OF CONTIKI

The linked list module `list` is a critical library in Contiki. It is required by 32 modules in the core of the OS and invoked more than 250 times. Lists are for example used to manage timers in the scheduler. Formal specification and verification of this module are thus required to prove many other modules of Contiki.

Figure 1 gives the API of the module. This API differs from common linked list implementations in multiple aspects. First, an existing list (illustrated by the lower part of Figure 2) is identified thanks to a *list handler* — a variable supposed to refer to the first element of the list — named `root` in Figure 2. Creating a copy (cf. lines 22–23 in Figure 1) of the list does not consist in duplicating each element of the list but only in copying the list handler. In a function call, to make it possible to modify the handler in the function, an existing list is passed as a function parameter via a pointer referring to the handler (denoted in this paper by `pLst` and having a double pointer type `list_t`), rather than just the address of the first list element (i.e. a single pointer, contained in `root`).

Second, since C does not offer templates, in order to provide a generic mechanism to declare a linked list for specific field datatypes, Contiki uses dedicated macros. When needed, the user creates a user-defined type **struct** some_list using such a macro. The preprocessor transforms such macros into new datatype definitions. The definition produced by such a macro for a list with one integer field would look like the definition on lines 1–4 of Figure 1 with line 3 uncommented. The generic behavior of the API is enabled through the use of **void*** pointers with explicit and implicit casts from (and to) pointers to a trivial linked list structure **struct** list (with lines 1,2,4 in Figure 1) having only one field: the pointer to the next element. To ensure that this "blind" manipulation using casts is possible, the first field in any list element structure must be a pointer to the next list element (cf. line 2). When a cell of type **struct** some_list is transmitted to the list API, the type is first erased to **void***. Then, the code of the function casts it to **struct** list* to perform the desired manipulations and to be able to refer to the next element. Note that according to the C standard, it violates the strict aliasing rule since we modify a value typed **struct** some_list through a pointer to a type **struct** list. The compilation of Contiki is configured to prevent the compiler to make code transformation following the assumption of strict-aliasing compliance.

Third, in Contiki, memory is provided to (and released by) processes through blocks stored in a pre-allocated array (see [19] for more detail). Thus, the list module does not perform any dynamic allocation, and the size of the list is bounded to the number of originally available memory blocks. All elements in a list should be distinct memory regions. In particular, duplicates (leading to a circular list) are not allowed.

Fourth, calling a function to add an element into a list is allowed even if this element is already in the list. In such a case, the element is first removed from the list, and then added (at the beginning, the end, or elsewhere, depending on the function called).

Finally, the API is very rich: it can handle a list as a queue or a stack (lines 14–21) and allows arbitrary removal or insertion, as well as enumeration (lines 24–30).

3 VERIFICATION APPROACH

This section presents our verification approach. The generic mechanism of type manipulation being an undefined behavior in C (cf. Section 2), the verification needs to be done separately on each "instantiated" list datatype. In the rest of this paper, we assume that the list structure **struct** list is defined as illustrated in lines 1–4 of Figure 1 (with line 3 uncommented) and use a pointer to a **struct** list instead of a generic **void*** pointer in the verified functions. To ensure that this choice of structure is not a limitation of the approach, we also check that the proofs remain valid for other common list structures (with a pointer as a data field, or with multiple numeric fields).

3.1 Modeling Linked Lists using Logic Lists

For specification and verification of the list module, it is convenient to have a more abstract view of the elements of a linked list. In a previous work [6], we used a companion ghost array for this purpose. Our new approach relies on a companion logic list and uses the logic list datatype available in ACSL. This datatype, denoted by \list<**type**>, is parameterized by **type**, the type of a list element. It has two usual constructors: \Nil (also written []), the empty list, and \Cons(**type**, \list<**type**>), that builds a new list \Cons(e, l), also

```
1  /*@
2    predicate in_list{L}
3              (struct list* e, \list<struct list*> l) =
4    ∃ ℤ n; 0 ≤ n < \length(l) ∧ \nth(l, n) == e;
5
6    predicate separated_from_list{L}
7              (struct list* e, \list<struct list*> l) =
8    ∀ ℤ n; 0 ≤ n < \length(l) ⇒
9      \separated(\nth(l, n), e);
10
11   predicate unchanged{L1, L2}(\list<struct list*> l) =
12   ∀ ℤ n; 0 ≤ n < \length(l) ⇒
13     (\valid{L1}(\nth(l,n)) ∧ \valid{L2}(\nth(l,n)) ∧
14     \at(\nth(l,n)->next,L1) == \at(\nth(l,n)->next,L2));
15 */
```

Figure 3: Predicates about the contents of a logic list

```
/*@
  inductive linked_ll{L}(struct list *bl,
                         struct list *el,
                         \list<struct list*> ll) {
  case linked_ll_nil{L}:
    ∀ struct list *el; linked_ll{L}(el, el, \Nil);
  case linked_ll_cons{L}:
    ∀ struct list *bl, *el, \list<struct list*> tail;
      \separated(bl, el) ⇒ \valid(bl) ⇒
      linked_ll{L}(bl->next, el, tail) ⇒
      separated_from_list(bl, tail) ⇒
        linked_ll{L}(bl, el, \Cons(bl, tail));
  }

  axiomatic to_logic_list {
    logic \list<struct list*>
    to_ll{L}(struct list* bl, struct list* el)
      reads { e->next | struct list* e;
              \valid(e) ∧ in_list(e, to_ll(bl, el)) };

    axiom to_ll_nil{L}: ∀ struct list *el;
      to_ll{L}(el, el) == \Nil;

    axiom to_ll_cons{L}: ∀ struct list *bl, *el;
      \let tail = to_ll{L}(bl->next, el);
      \separated(bl, el) ⇒ \valid(bl) ⇒
      separated_from_list(bl, tail) ⇒
        to_ll{L}(bl, el) == (\Cons(bl, tail));
  }
*/
```

Figure 4: Linking predicate linked_ll and translating function to_ll, used to build a logic view of a C linked list

denoted by e::l, from a given list l and an element e added at the beginning. Instead of \Cons(item,\Nil), a singleton list can be written [|item|]. It also provides some other features, for example concatenation (denoted by l1 ^ l2), length, nth element, etc. In our case, we use logic lists of type \list<**struct** list*>, that will contain the addresses of elements of the linked list.

For example, lines 2–4 of Figure 3 show a predicate stating that list element e is present in logic list l, while lines 6–9 illustrate a predicate stating that list element e is *separated* from (that is, has no overlap with) the elements in logic list l. In an ACSL annotation, each of the predicates can specify the program point (or *label*) L where it should hold. Examples of predefined labels in ACSL are **Pre** and **Post** (respectively, to refer to the states before and after the function execution), as well as **Here** (for the current program point). If the label is omitted, it is set by default to the current point.

A logic view of a C linked list is built using two different constructs (see Figure 4): an inductive predicate `linked_ll` establishing the equivalence between a C linked list and a logic list, and an axiomatically defined logic function `to_ll` ("to logic list") that translates a C list into the corresponding logic list. We also call them *linking predicate* and *translating function*. If the linking predicate holds then the corresponding logic list does not contain duplicate elements, and if the translating function application is defined then the resulting logic list does not contain duplicates either.

The linking predicate and the translating function are defined for a sublist (i.e. list prefix) of a C list starting at the element `bl` and ending at the (excluded) element `el`. If `el` is `NULL`, the whole C list until the end is linked with (or translated into) a logic list. For example, for the C list in Figure 2, `linked_ll(&A, &D, &A::&B::&C::[])` holds, and we have `to_ll(&A, &D)=&A::&B::&C::[]`. If `el` is equal to `NULL`, we also have `linked_ll(&A, NULL, &A::&B::&C::&D::&E::[])` and `to_ll(&A, NULL)=&A::&B::&C::&D::&E::[]`. These definitions allow us to easily merge and split sublists as shown in Section 3.4.

More precisely, predicate `linked_ll{L}(bl,el,ll)` establishes the relation between a logic list `ll` and a C sublist starting with element `bl` and ending just before element `el` (or going until the end of the C list if `el` is `NULL`). This relation is established at label `L`. The first case (lines 5–6 in Figure 4) in the inductive definition states that the C sublist starting and finishing at the same element (being an excluded end) is related to an empty logic list `\Nil`. The second case (lines 7–12) is used to deduce the predicate for a nonempty logic list of the form `\Cons(bl,tail)` (cf. line 12). It contains several conditions. First, `bl` and `el` are not only expected to be different, but also separated (cf. line 9). Second, `bl` should be *valid*, that is, the pointed memory location `*bl` can be read and written, to ensure that `bl->next` makes sense. Third, the sublist that starts with `bl->next` and ends at `el` must be related to logic list `tail` (cf. line 10), and the element `bl` we are adding must be separated from all elements in `tail` (cf. line 11). In particular, this ensures the absence of duplicates in the list. If all these conditions hold, the C sublist that starts with `bl` and ends at `el` is related to logic list `\Cons(bl,tail)`.

The translating function `to_ll` is defined in a similar way using an axiomatic predicate. The main difference is that we have to indicate the set of memory locations read by the function (cf. lines 18–19). Thanks to this, the tool can deduce that if a memory location in this set is modified between two program points, the value returned by `to_ll` might have changed as well. Here, the structure of a C list is modified whenever the next fields are, so the specified set contains the next fields of the elements that are **`\valid`** and belong to the list[2].

We can show that `linked_ll{L}(bl,el,ll)` implies `ll=to_ll{L}(bl,el)`, but the converse is not true. Indeed, the axiomatic predicate does not provide a suitable inductive hypothesis necessary to prove the other implication. A similar situation occurs in some lemmas on the translating function (as we will illustrate in Section 3.4) where we also need a stronger property in terms of the linking predicate to provide such an inductive hypothesis. Another reason why we need both definitions is related to a limitation of ACSL and will be explained below in Section 3.3.

[2]A rigorous reader may be surprised to find the function result used on line 19. The **reads** section being used to reason about changing values rather than to compute the function itself, it is allowed to refer to the function result.

3.2 Running Example

We will use the function `list_remove` (see Figure 5) to illustrate the specification and verification of the list module. This function removes an element `item` from a given list `pLst`. If `item` is not in the list, that is, when the list is empty (line 24) or when the loop (lines 38–41) reaches the end of the list without finding `item`, the list remains unchanged. If `item` is the first element (line 25), it is just popped from the list (line 26). Finally, if the loop (lines 38–41) finds `item` in the list, the cell that precedes it is connected to the cell that follows it, ensuring that it is removed from the list (lines 42,46). ACSL annotations (written in special comments `/*@...*/` or `//@...`) are presented in the next section.

3.3 Formal Specification

Deductive verification in FRAMA-C/WP requires a formal specification of the code in ACSL [4]. Let us describe the formal specification of the `list` module using the (simplified) contract of `list_remove`.

The **`requires`** clauses (lines 2–5 in Figure 5) give the precondition of the function. Here, `pLst` and `item` are expected to point to valid memory locations (line 2). The C list must be linked to its logic representation (line 3). Finally, `item` must be either in the list (line 4) or separated from all elements of the list (line 5).

The **`ensures`** clauses (lines 7–17) give the postcondition of the function. There are two postconditions that must apply in any case: the C list must still be linked to its logic representation (line 7), and `item` must not be in the list. In addition, we have two different *behaviors* (or cases): either `item` was not in the list (line 11) and in this case the new list remains equal to the old one (line 12), or `item` was in the list (line 14) and in this case the new list is obtained by concatenating the sublist that goes from the beginning of the list to `item` with the sublist that starts from the element that follows `item` and goes until the end of the list (line 15–16). The postconditions of both behaviors are conveniently expressed using logic lists. The **`\old`**`(e)` construct (that is a shortcut for **`\at`**`(e,`**`Pre`**`)`) expresses the value of expression `e` before the function execution, that is, at label **Pre**. It is required if the value `e` can change (as for `*pLst`) and can be omitted if this value does not change (here, for `item->next`, as explained for the **`assigns`** clause below). Moreover, these two behaviors are *complete* and *disjoint:* they cover all possible situations, and never apply simultaneously (line 17).

The **`assigns`** clause indicates the memory locations that can be modified by the function. Here, they contain the list handler `*pLst` (line 19) and the next fields of the elements `l` that were in the initial list and pointed to `item` (line 20–21). The set of such elements `l` is either empty, if `item` was not in the list or was its first element, or contains exactly one cell: the one that preceded `item`.

Finally, the contract of the loop on lines 38–41 can also be conveniently specified using logic lists. We give in Figure 5 only a partial version of the contract allowing us to ensure that `l` belongs to the list. Variable `l` is used in the loop to iterate over list elements until the next element is `item` (or until the end of the list is reached). The

```
1  /*@
2    requires \valid(pLst) ∧ \valid(item);
3    requires linked_ll(*pLst, NULL, to_ll(*pLst, NULL));
4    requires in_list(item, to_ll(*pLst, NULL)) ∨
5      separated_from_list(item, to_ll(*pLst, NULL));
6
7    ensures linked_ll(*pLst, NULL, to_ll(*pLst, NULL));
8    ensures separated_from_list(item, to_ll(*pLst, NULL));
9
10   behavior does_not_contain:
11     assumes ! in_list(item, to_ll(*pLst, NULL));
12     ensures to_ll(*pLst, NULL) == to_ll{Pre}(\old(*pLst), NULL);
13   behavior contains:
14     assumes in_list(item, to_ll(*pLst, NULL));
15     ensures to_ll(*pLst, NULL) ==
16       to_ll{Pre}(\old(*pLst), item)^to_ll{Pre}(item->next, NULL);
17   complete behaviors; disjoint behaviors;
18
19   assigns *pLst,
20         { l->next | struct list* l; \at(l->next, Pre) == item ∧
21           in_list(l, to_ll{Pre}(\at(*pLst, Pre), NULL)) };
22 */
23 void list_remove(list_t pLst, struct list *item){
24   if( *pLst == NULL ) return;
25   if( *pLst == item ){
26     *pLst = (*pLst)->next;
27     //@ assert unchanged{Pre, Here}(to_ll((*pLst)->next, NULL));
28     return;
29   }
30
31   struct pLst *l = *pLst;
32   //@ ghost int n = 0;
33
34   /*@ loop invariant \nth(to_ll(*pLst, NULL), n) == l;
35       loop invariant 0 ≤ n < \length(to_ll(*pLst, NULL));
36       loop assigns l, n;
37     @*/
38   while(l->next ≠ item ∧ l->next ≠ NULL){
39     l = l->next;
40     //@ ghost n++;
41   }
42   if( l->next == item ){
43     /*@ assert to_ll{Pre}(\old(*pLst), NULL) ==
44                to_ll{Pre}(\old(*pLst), item) ^ [|item|] ^
45                to_ll{Pre}(item->next, NULL); */
46     l->next = item->next;
47     /*@ assert to_ll{Pre}(\at(*pLst,Pre), item) ==
48                to_ll(*pLst, l->next); */
49     /*@ assert to_ll{Pre}(item->next, NULL) ==
50                to_ll(l->next, NULL); */
51     /*@ assert to_ll(*pLst, NULL) ==
52                to_ll(*pLst, l->next) ^ to_ll(l->next, NULL); */
53   }
54 }
```

Figure 5: Function list_remove and simplified specification

loop invariant specifies that l belongs to the list by saying that l is the n-th element of the list (line 34). It uses for that purpose an additional ghost variable n (line 32) that should remain a valid element number (line 35). The variables modified by the loop are given by the clause on line 36.

The assertions in the function body will be presented in Section 3.4. The complete version of the specification can be found in the annotated code available online.

Linking Predicate vs. Translating Function. We are now ready to explain another reason for using both the linking predicate and the translating function. We mentioned in the end of Section 3.1 that a link of a C sublist with a logic list expressed with the linking predicate is stronger than with the translating function. Can we

```
1  /*@
2    lemma linked_ll_unchanged{L1, L2}:
3    ∀ struct list *bl, *el, \list<struct list*> ll;
4      linked_ll{L1}(bl, el, ll) ⇒
5      unchanged{L1, L2}(ll) ⇒
6        linked_ll{L2}(bl, el, ll);
7
8    lemma to_ll_split{L}:
9    ∀ struct list *bl, *el, *sep, \list<struct list*> ll;
10     linked_ll(bl, el, ll) ⇒ //implies to_ll(bl, el)==ll
11     in_list(sep, ll) ⇒
12       ll == to_ll(bl, sep) ^ to_ll(sep, el);
13
14   lemma to_ll_merge{L}:
15   ∀ struct list *bl,*sep,*el, \list<struct list*> l1,l2;
16     linked_ll(bl, sep, l1) ⇒ //implies to_ll(bl, sep)==l1
17     to_ll(sep, el) == l2 ⇒
18     \separated(bl, el) ⇒ all_separated_in_list(l1 ^ l2) ⇒
19     separated_from_list(el, l1) ⇒
20       to_ll(bl, el) == l1 ^ l2;
21 */
```

Figure 6: Examples of (simplified) auxiliary lemmas

express all properties using the linking predicate only? The main reason why we need both definitions is the fact that Acsl does not allow to declare a quantified variable for a complete contract, for example, an existentially quantified variable that can be used in both pre- and postconditions. For instance, it can be desirable to replace to_ll{**Pre**}(**\old**(pLst),NULL) in lines 3, 4, 7, 8, 12, etc. by a logic variable but there is no practical way to introduce such an additional contract-level variable in Acsl so that it can be used in various pre- and postcondition.

3.4 Auxiliary Lemmas

Reasoning by induction is still not well handled by SMT solvers and, consequently, inductive properties are generally hard to prove automatically. SMT solvers are more efficient when they just have to instantiate lemmas stating implications between known properties. A good way to ease the verification process when it involves inductive properties is thus to provide lemmas about these properties. This kind of approach has already been successfully applied, e.g. for properties about ranges of values [7].

Unchanged Sublists. SMT solvers cannot deduce the preservation of some list properties from the fact that a list remains *unchanged* between two program points (meaning that its list structure is unchanged). The predicate unchanged (lines 11–14 of Figure 3) states that for two program points L1 and L2, all list elements in logic list l have the same value of the next field, and refer to valid memory locations at both labels L1 and L2.

The lemma linked_ll_unchanged (illustrated by Figure 6, lines 2–6) states that if some C sublist starting with bl and ending at el is linked, at a program point L1, to logic list ll, and ll remains unchanged between program points L1 and L2, then the same C sublist is still linked to ll at program point L2. A similar lemma is stated for the translating function to_ll.

Lemma linked_ll_unchanged helps to show that when we modify or remove some list element while the others remain unchanged, the properties for the unchanged sublists remain valid. For example,

```c
void
list_insert(list_t pLst, struct list *prev, struct list *new){
  if(prev == NULL) {
    list_push(pLst, new);
  } else {
    // remove & insert new even if it is at the right place
    list_remove(pLst, new);
    new->next = prev->next;
    prev->next = new;
  }
}
```

```c
void
list_insert(list_t pLst, struct list *prev, struct list *new){
  if(prev == NULL) {
    list_push(pLst, new);
  } else {
    if(prev->next ≠ new){ // remove & insert new only if necessary
      list_remove(pLst, new);
      list_force_insert(pLst, prev, new);
    }
  }
}
```

Figure 7: Rewriting of the `list_insert` function: (a) initial code (left) and (b) new optimized code (right)

this is useful in `list_remove`, when `item` is the first element of the list that is removed (lines 25–29 in Figure 5). To help the prover, the assertion on line 27 indicates that the sublist starting from the second element of the initial list is not modified by the assignment on line 26. Lemma `linked_ll_unchanged` (along with a similar lemma for the translating function `to_ll`) allows the prover to deduce that the linking relation still holds for this sublist, and thus, to deduce line 7 from line 3 in Figure 5.

Split and Merge. Two more technical properties, useful for reasoning about the contents of sublists, are the facts that we can *split* a sublist into two sublists, and conversely *merge* two consecutive sublists into a longer one. Lemma `to_ll_split` (lines 8–12 in Figure 6) states that if some C sublist starting with `bl` and ending at `el`, contains an element `sep`, then the corresponding logic list can be split into two sublists, the first one starting with `bl` and ending at `sep` and the second one starting with `sep` and ending at `el`. (Recall that the ending node is excluded.) Conversely, lemma `to_ll_merge` (lines 14–20 in Figure 6) states that two sublists can be merged, and deduces line 20 from lines 16, 17. It requires to add some missing separation conditions for elements of both sublists (lines 18–19, some of which are not detailed here) in order to deduce the separation conditions included in the property on line 20 (cf. Section 3.1). Notice that, as mentioned in the end of Section 3.1, we sometimes need a stronger assumption (cf. lines 10, 16) using the linking predicate rather than the translating function, in order to provide a suitable induction hypothesis during the proof of these lemmas with the Coq proof assistant. Similar split and merge lemmas are stated for the linking predicate.

Again, we can illustrate the use of these lemmas for `list_remove`. Consider a more complex case when the element to remove is not at the beginning of the list (lines 42–53 in Figure 5). In this case, the previous loop has identified a list element `l` such that `l->next==item`. Let us sketch how the postcondition on lines 15–16 can be deduced in this particular case. For simplicity, we show the reasoning only in terms of logic lists returned by the translating function, the properties in terms of the linking predicate being obtained similarly. Assertion on lines 43–45 follows from a double application of lemma `to_ll_split` and performs a split of the initial list into three sublists, the sublist ending at element `item` (excluded), element `item` itself, and the rest of the list starting with `item->next`. The split is performed in the initial state at label `Pre` (that is explicitly indicated, except for variables that do not change). After the removal of `item` (line 46), we have the assertions on lines 47–48 and 49–50. They are obtained using the lemmas about unchanged sublists[3], similarly to the example presented earlier in this section. Finally, thanks to lemma `to_ll_merge`, assertion on lines 51–52 follows from lemma `to_ll_merge` and states that the whole list at this program point is a merge of its two sublists before and after `l->next`. The postcondition on lines 15–16 now immediately follows from these assertions.

Proof of Lemmas. We have proved all lemmas using the Coq proof assistant [5, 25]. Most of the lemmas related to the linking predicate `linked_ll` have been proved by induction on the predicate itself. For lemmas about the translating function `to_ll`, an additional premise on `linked_ll` is required to enable the same kind of reasoning.

3.5 The list_insert Function

The present work is the first complete verification of the linked list module of Contiki, including the `list_insert` function. The `list_insert` function inserts a given list element `new` into a given linked list `pLst` after another given element `prev`, or at the beginning of the list if `prev` is `NULL` (cf. Figure 1). Our previous verification study [6] of the list module detected and reported an error in this function, which thus remained unverified. The error was fixed since that time, so we address the verification of this function in this study.

The verification of the `list_insert` function was particularly challenging, mainly because it has several quite different cases to handle, depending on where the element `new` should be inserted and whether it is already present somewhere in the list or not. Indeed, as mentioned in Section 2, if the element is already present, it should be first removed before being inserted again (cf. Figure 7a). While most of the functions of the module have two or three behaviors, `list_insert` has six. This section presents some additional points related to the verification for this function.

Additional Functions. To facilitate the proof, we wrote two additional functions: `list_split` and `list_force_insert`, in order to limit the need for assertions in the proof of the `list_insert` function itself. Indeed, a big number of assertions leads to proof failures due to the size of the resulting proof context the SMT solvers have to deal with. Separating these proofs in other functions makes the proof more modular and reduces the size of the proof context. These additional functions were specified and proved in Frama-C/Wp as well.

[3]A rigorous reader may notice that, strictly speaking, the first sublist is not unchanged since the last element's `next` field, `l->next`, was modified, so the proof of the assertion on lines 47–48 requires some additional assertions to perform another split-merge step in order to separately analyze the sublist containing the last element `l` and, basically, check necessary separation properties.

The `list_split` function is inspired by the so-called autoactive verification approach [17]. We basically use a specified C function (sometimes also called *lemma-function*) that does not compute anything but allows us to deduce properties relating the list with its sublists by visiting it. Indeed, adding assertions to deduce such properties can be practical when there are not so many sublists to split and merge (as it was the case for all other functions of the module), but it becomes more complex in the case of `list_insert` that involves a lot of sublists. In this case the assertions required to help the prover can be numerous and pollute the proof context. A lemma-function allows to directly deduce the properties of interest.

The `list_force_insert` function realizes a specific case of insertion when we are sure that the element `new` is not in the list. It includes two actions: first modify the `next` field in the element to insert, second modify the `next` field in `prev` (cf. lines 8–9 in Figure 7a). We put these actions into a separate function to simplify the reasoning about the memory in the `list_insert` function. Indeed, by creating such a function and calling it in `list_insert` (cf. lines 7–8 in Figure 7b), we remove an intermediate memory state and directly deduce the effect on the list.

Optimization. Thanks to the analysis of different behaviors during the specification process, we observed that in the case when the element `new` must be inserted at its current position (that is, when `new` is already after `prev`), the list should remain unchanged, but the initial implementation uselessly removes the element and reinserts it again (cf. lines 6–9 in Figure 7a). We have proposed an optimized version, shown in Figure 7b, that avoids useless actions in this case. In particular, it avoids the waste of time to enumerate the list elements (in the loop in Figure 5) for the removal. Moreover, for this optimized version of the function, the automatic proof also works much better, only two assertions being proved using Coq.

4 RESULTS OF THE VERIFICATION

In this work, we have formally specified and verified all functions of the list module of Contiki. In total, for about 176 lines of C code in the module (excluding macros), we wrote about 1700 lines of annotations, including about 410 lines for contracts and 270 lines for logic definitions and lemmas. They constitute a solid basis for the verification of client modules using linked lists, its future extensions and other similar implementations. We did not specifically try to minimize the number of intermediate assertions: they were added to explicitly state the expected local properties and to help the automatic proof, and some of them could probably be removed. On the other hand, these assertions can be useful to illustrate the expected reasoning of the provers and to make the logic of this reasoning clearer for the verification engineer.

For this annotated version of the module, the verification using Frama-C/Wp generates 757 goals (also called *proof obligations* or *verification conditions*). This number includes 68 goals for the verification of absence of runtime errors that are often responsible for security vulnerabilities and have also been carefully checked by Frama-C/Wp. It also includes 33 auxiliary lemmas (that is, only about 4.4% of all properties). These lemmas are proved interactively using Coq v.8.7.2. All other goals are automatically discharged by SMT solvers, except two assertions proved using Coq. In this work, we used the latest development version of Frama-C (v.18 Argon), as well as the SMT solvers Alt-Ergo v.2.20 and CVC4 v.1.6 (via Why3).

	ens.	ass.	req.	total pre/post	guiding annot.	RTE	total
Logic	83	53	16	152	264	54	470
Ghost	134	71	69	274	399	108	781

Figure 8: Generated goals per category for the versions with logic lists and with ghost arays (for the subset of functions verified in both studies, excluding lemmas)

Note that the verification of the `list_insert` function represents, in terms of annotations and verification conditions, the third of the verification effort. In total, for the three functions (`list_insert`, `list_split` and `list_force_insert`) discussed in Section 3.5, we wrote 626 lines of annotations, that is more than a third of all annotations. For them, Wp generates 259 proofs obligations (again, a third of all generated proof obligations). Finally, this function is the only one that forced us to use Coq to prove assertions. Interestingly, these assertions were not hard to verify with Coq, so we assume that the failure of the SMT solvers to prove these goals is due to the size of the proof context.

In order to get confidence in our specification of the list module functions, we wrote 15 valid test functions manipulating lists, and proved simple properties about them using Frama-C/Wp. We have also implemented 15 invalid tests. Each invalid function is an altered version of a valid one where either the contract or the function contains an error. As expected for those functions, the verification leads to proof failure. This gives us further confidence that the proposed contracts can be used to verify client functions.

5 COMPARISON WITH THE APPROACH USING GHOST ARRAYS

This section compares the proposed approach based on logic lists with the previous verification effort based on ghost arrays [6] and underlines some weaknesses and advantages of each approach. For a fair comparison, we consider the subset of functions verified in both studies, and use exactly the same versions of the tools.

Compared Versions. Regarding the new specified code relying on logic lists, we exclude the `list_insert` function, as well as `list_split` and `list_force_insert` used for its verification, since they were not verified in the previous work [6]. Without these functions, the code contains 1000 lines of annotations, including 290 lines for function contracts, and 250 for logic definitions and lemmas.

Regarding the version specified using ghost arrays (provided in [6]), we slightly modified some assertions in order to prove it with exactly the same versions of Frama-C, Alt-Ergo and CVC4 as used in the present work. The new version contains about 1420 lines of annotations, including about 500 lines for function contracts and 240 for logic definitions and lemmas.

Numerical Results. Figure 8 sums up the numbers of proof obligations per category. The "ensures" and "assigns" columns represent the numbers of the corresponding postconditions to prove. The "requires" column indicates the number of preconditions to prove (for function calls in the callers' code, for example when `list_remove` is called from `list_push`). The column "guiding annotations" represents the annotations needed to guide the proof itself, such as loop contracts and assertions. We also show the numbers of goals for the absence of runtime errors (RTE), as well as the total numbers for function contracts and for all annotations. Experiments are conducted on a Core i7 6700HQ with 16GB DDR4.

For the new specified code relying on logic lists, Wp generates 33 goals for lemmas and 470 for other annotations. Except lemmas, all goals are automatically discharged in about 5 min 30 s, that is, 0.702 s per goal on average.

For the version based on ghost arrays, Wp generates 24 goals for lemmas and 781 goals for other annotations. Using more recent versions allowed to decrease the time needed for the proof using ghost arrays from about 50 min (using the same versions of the code and the tools as in [6] and the script provided to launch the verification) to 21 min 20 s (that is, 1.639 s per goal on average). This speedup can be explained by improvements in the recent versions of Frama-C and the provers, and the fact that we use less SMT solvers, thus allowing to verify more proof obligations in parallel (since for each obligation, we start two SMT solvers instead of four solvers used in [6]). All lemmas and one assertion for the modified version are proved using the Coq proof assistant.

Analysis of Results. First, we can notice that the function contracts in the new version based on logic lists are 210 lines (or 42%) shorter than in the ghost array based version. Hence, our version generates much less proof obligations. The difference between the two formalizations on this aspect is mainly due to the fact that in the ghost array formalization, a lot of *separation properties* needed to be explicitly stated before and after a function call, since no support in Frama-C is currently available to consider that the ghost variables are implicitly separated from the regular program memory. This need disappears with logic lists since they exclusively belong to the specification world: their representation is external to the C memory representation. While the inductive and axiomatic predicates definitions are extremely similar to what we wrote in the ghost array based version, our new version is more convenient to use when it comes to the specification of function behavior. For example, the specification of the ensures clause of the behavior `contains` on lines 15–16 in Fig. 5 was basically written as (see details in [6]):

```
ensures unchanged{Pre,Post}(cArr, 0, item_idx - 1);
ensures array_shift_left{Pre,Post}(cArr, item_idx, len - 1);
```

which is not so convenient to read and requires additional preconditions to specify that `item_idx` is the index of the item to remove in the companion ghost array `cArr`. In these regards, our formalization brings the benefit of *simplifying the specifications and the proof.*

Second, with the logic list version, we need to provide 37% less annotations to guide the provers. That is a very important benefit of this approach, which makes the verification *significantly easier for the verification engineer.*

Third, the automatic proof becomes faster for the new version, both for the total proof (~4× faster) and even per proof obligation (2.3× faster). This brings several benefits. Of course, once a proof is complete, the time to replay the proof is not so important, but during the verification process, often requiring many proof attempts, a faster proof makes *the verification engineer's work more efficient.* A faster proof per goal indicates that the new version is more *suitable for automatic verification with SMT solvers*, and thus has a greater potential of being *usable in a larger context.*

On the other hand, *the manipulation of logic lists requires more complex lemmas*, that have to deal with two versions of properties related to unchanged sublists, splitting and merging: one for the linking predicate and one for the translating function. Their proofs are longer than for the lemmas in the ghost array based formalization [6]. Thus, we expect the logic list based formalizations to be *harder to handle by non-specialists.* For experienced engineers, this should not be a major obstacle though: once the lemmas are proved and understood, they can be applied again and again.

Another advantage of the previous version is its *compatibility with runtime verification.* A previous work [18] provides a proof of concept that the ghost array based formalization can be made executable, and thus can be used to check by runtime verification that the preconditions stated for the functions of the list module are respected by some user code. Currently, is not clear whether the new approach can be made executable. The runtime verification plugin E-Acsl [16] of Frama-C does not support logic lists, and adding this support would require to implement this datatype in E-Acsl.

Finally, *ghost arrays can be used to model other linked data structures.* For example, Dross et al. [9] prove an implementation of red-black trees in SPARK that involves underlying arrays. They also rely on ghost code for the proof. In another work [7], Frama-C was used to prove a heap data structure implemented as a tree flattened into an array. So we expect that tree data structures could be proved in Frama-C using an approach based on ghost arrays. In comparison, *using another pure logic type would require an extension of the Acsl language and its tool support.*

6 RELATED WORK

For the verification of linked data structures, that involves important separation properties, separation logic [24] can be more suitable than Hoare logic, on which the Wp plugin of Frama-C is based. Tools based on separation logic can be more efficient to verify a linked list API. This is for example the case for VeriFast [14], that has been used in several industrial case studies [22], or the Verified Software Toolchain (VST) [1, 3], mainly used for the verification of crytography related software [2, 27] but also a message passing system [20]. We are currently not aware of efforts specifically dedicated to linked list modules, even if the example gallery of VeriFast[4] and the VST case studies do include linked list function specifications and verification. These specifications follow the idea proposed by Reynolds in [24] and are based on a logic list data structure and an inductive predicate relating the memory and such a logical data structure. In our work, we essentially followed the same idea, by expressing separation hypotheses in another way. We used Frama-C since it enables the use of various verification techniques for different parts of code, that is an important advantage in such an ambitious project as the verification of Contiki.

The logic types of Acsl basically provide a way of defining abstract data types. For programming languages that provide high-level ways of describing abstract data types (ADT), such as Eiffel or

[4]https://people.cs.kuleuven.be/~bart.jacobs/verifast/examples/

Java, the language itself can define side-effect free immutable ADTs. Such ADTs can then be used in specifications [23]. Such ADTs are often called models and are related to implementation classes in a way similar to the way we related a C linked list with an Acsl logic list. As these model classes are valid classes, they can be used in dynamic verification tasks. Very often these classes represent concepts (for e.g. sets or sequences) that can be translated into elements of theories of provers: they can therefore also be used in deductive verification tasks. It is possible to verify the faithfulness of the translation [8].

One distinctive advantage of Frama-C is that in addition to the automated and interactive provers it supports directly, it can output verification conditions to many different provers by relying on Why3 [11]. It both eases and makes more trustworthy the verification: some provers may be more efficient on some conditions than others, and having a condition proved by several provers increases the confidence in the result. In contrast, VeriFast relies on its embedded SMT solver Redux and possibly Z3 while VST is a framework for Coq, thus benefits from much less automation. Being a framework for Coq also means that the specifications can be written in the very rich Gallina language of Coq. Additionally, the soundness of the underlying logic of VST has been proved in Coq. The same applies to the Wp plugin and VeriFast, even though only subsets of the tools have been considered so far [13, 26].

Based on our experience of teaching both Frama-C and Coq, the former is much more accessible to students and junior engineers who know how to program in C. With basic knowledge on first order logic, Acsl specifications are more easily understood than Coq statements, and therefore VST specifications.

7 CONCLUSION

The linked list module of Contiki is both one of the most critical and widely used modules in the OS. The present work performed a first complete formal specification and deductive verification of this module. It proposed a specification approach using logic lists in order to represent and manipulate the elements of a C list. Logic lists provide a practical high-level view of C lists for the specification of the functions. The proof was tested for several definitions of list data structure.

We have stated several useful lemmas to reason about logic lists, and proved them in the interactive proof assistant Coq. These lemmas will be useful in future verification case studies for client functions and similar modules. The specification of the `list_insert` function allowed us to propose an optimized implementation, which also made the verification easier. This work also allowed to identify and implement new optimizations related to the support of logic lists in the simplifier of Wp, called Qed. These optimizations will be available in the upcoming release of Frama-C.

We compared the proposed approach with a previous partial verification of the list module using ghost arrays, and analyzed the results. The results essentially indicate that the new approach is more suitable for automatic verification, leads to simpler specifications and a faster proof, while the technique based on ghost arrays brings the benefits to be compatible with runtime verification, easier to use for an unexperienced verification engineer and more suitable for verification of other linked data structures such as trees. This comparison will help verification engineers to choose the most appropriate technique in future projects.

Ghost arrays and logic lists are two ways for verifying a linked list API. In the future, we plan to experiment with a third way in the spirit of the work of Gladisch and Tyszberowicz [12]. They verified linked data structures implemented in Java using a technique that does not involve any ghost structure or model. The main idea is to define a pure observer method. In the case of linked lists, such a method takes a list object and an index, and returns the object at that index in the list. Such an observer method can then help to specify Java methods on linked lists. We plan to design such an observer directly written as a logical function in Acsl, with an efficient equivalent C implementation for dynamic verification tasks. C pointers are however not Java references, so we expect the formalization in Acsl to diverge significantly from the JML specification of Gladisch and Tyszberowicz. Future work also includes verification of a larger set of modules of Contiki, in particular, using linked lists, as well as investigating if the reported results on the logic-against-ghosts comparison also apply to other verification tools and case studies.

ACKNOWLEDGMENTS

This work was partially supported by a grant from CPER DATA and the project VESSEDIA, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731453. The authors thank the Frama-C team for providing the tools and support, as well as Patrick Baudin, François Bobot and Loïc Correnson for many fruitful discussions and advice. Many thanks to the anonymous referees for their helpful comments.

REFERENCES

[1] Andrew W. Appel. 2011. Verified Software Toolchain. In *Programming Languages and Systems (ESOP)* (March 26-April 3) *(LNCS)*, Gilles Barthe (Ed.), Vol. 6602. Springer, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

[2] Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.* 37, 2, Article 7 (April 2015), 7:1–7:31 pages. https://doi.org/10.1145/2701415

[3] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers.* Cambridge University Press.

[4] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. 2018. *ACSL: ANSI/ISO C Specification Language.* http://frama-c.com/acsl.html

[5] Y. Bertot and P. Castéran. 2004. *Interactive Theorem Proving and Program Development.* Springer. https://doi.org/10.1007/978-3-662-07964-5

[6] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. 2018. Ghosts for Lists: A Critical Module of Contiki Verified in Frama-C. In *Proc. of the 10th NASA Formal Methods Symposium (NFM 2018) (LNCS)*, Vol. 10811. Springer, 37–53.

[7] Jochen Burghardt and Jens Gerlach. 2018. ACSL by Example. https://github.com/fraunhoferfokus/acsl-by-example

[8] Ádám Darvas and Peter Müller. 2010. Proving Consistency and Completeness of Model Classes Using Theory Interpretation. In *Proc. of the 13th International Conference on Fundamental Approaches to Software Engineering (FASE 2010) (LNCS)*, Vol. 6013. Springer, 218–232.

[9] Claire Dross and Yannick Moy. 2017. Auto-Active Proof of Red-Black Trees in SPARK. In *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings.* 68–83. https://doi.org/10.1007/978-3-319-57288-8_5

[10] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. 2004. Contiki – A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *LCN 2014.* IEEE.

[11] Jean-Christophe Filliâtre and Andrei Paskevich. [n. d.]. Why3 - Where Programs Meet Provers. In *ESOP 2013.*

[12] Christoph Gladisch and Shmuel Tyszberowicz. 2015. Specifying linked data structures in JML for combining formal verification and testing. *Science of*

Computer Programming 107-108 (2015), 19 – 40. https://doi.org/10.1016/j.scico.2015.02.005

[13] Paolo Herms, Claude Marché, and Benjamin Monate. 2012. A Certified Multi-prover Verification Condition Generator. In *Verified Software: Theories, Tools, Experiments (VSTTE)* (January 28-29) *(LNCS)*, Vol. 7152. Springer, 2–17. https://doi.org/10.1007/978-3-642-27705-4

[14] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Franck Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *Nasa Formal Methods (NFM) (LNCS)*. Springer-Verlag, Berlin Heidelberg, 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

[15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Asp. Comput.* 27, 3 (2015), 573–609. http://frama-c.com

[16] Nikolai Kosmatov and Julien Signoles. 2013. A Lesson on Runtime Assertion Checking with Frama-C. In *Runtime Verification (RV)* (September 24-27) *(LNCS)*, Vol. 8174. Springer, 386–399. https://doi.org/10.1007/978-3-642-40787-1_29

[17] K. Rustan M. Leino and Michał Moskal. 2010. Usable Auto-Active Verification. http://fm.csl.sri.com/UV10/

[18] Frédéric Loulergue, Allan Blanchard, and Nikolai Kosmatov. 2018. Ghosts for Lists: from Axiomatic to Executable Specifications. In *Proc. of the 12th International Conference on Tests and Proofs (TAP 2018) (LNCS)*, Vol. 10889. Springer, 177–184. https://doi.org/10.1007/978-3-319-92994-1_11

[19] Frédéric Mangano, Simon Duquennoy, and Nikolai Kosmatov. 2016. A Memory Allocation Module of Contiki Formally Verified with Frama-C. A Case Study. In *CRiSIS 2016 (LNCS)*, Vol. 10158. Springer.

[20] William Mansky, Andrew W. Appel, and Aleksey Nogin. 2017. A Verified Messaging System. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 87 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133911

[21] Alexandre Peyrard, Nikolai Kosmatov, Simon Duquennoy, and Shahid Raza. 2018. Towards Formal Verification of Contiki OS: Analysis of the AES-CCM* Modules with Frama-C. In *RED-IoT 2018, co-located with EWSN 2018*. ACM.

[22] Pieter Philippaerts, Jan Tobias Mühlberg, Willem Penninckx, Jan Smans, Bart Jacobs, and Frank Piessens. 2014. Software verification with VeriFast: Industrial case studies. *Science of Computer Programming* 82 (2014), 77–97. https://doi.org/10.1016/j.scico.2013.01.006

[23] Nadia Polikarpova, Carlo A. Furia, and Bertrand Meyer. 2010. Specifying Reusable Components. In *Proc. of the 3rd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2010) (LNCS)*, Vol. 6217. Springer, 127–141.

[24] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 55–74.

[25] The Coq Development Team. [n. d.]. The Coq Proof Assistant. http://coq.inria.fr,.

[26] Frédéric Vogels, Bart Jacobs, and Frank Piessens. 2015. Featherweight VeriFast. *Logical Methods in Computer Science* 11, 3 (2015). https://doi.org/10.2168/LMCS-11(3:19)2015

[27] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. 2017. Verified Correctness and Security of mbedTLS HMAC-DRBG. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, New York, NY, USA, 2007–2020. https://doi.org/10.1145/3133956.3133974