Ghosts for Lists: A Critical Module of Contiki Verified in Frama-C

Allan Blanchard[1], Nikolai Kosmatov[2], and Frédéric Loulergue[3]

[1] Inria Lille — Nord Europe, Villeneuve d'Ascq, France
allan.blanchard@inria.fr
[2] CEA, List, Software Reliability and Security Lab, PC 174, Gif-sur-Yvette, France
nikolai.kosmatov@cea.fr
[3] School of Informatics Computing and Cyber Systems, Northern Arizona University, USA
frederic.loulergue@nau.edu

Abstract. Internet of Things (IoT) applications are becoming increasingly critical and require rigorous formal verification. In this paper we target Contiki, a widely used open-source OS for IoT, and present a verification case study of one of its most critical modules: that of linked lists. Its API and list representation differ from the classical linked list implementations, and are particularly challenging for deductive verification. The proposed verification technique relies on a parallel view of a list through a companion ghost array. This approach makes it possible to perform most proofs automatically using the Frama-C/WP tool, only a small number of auxiliary lemmas being proved interactively in the Coq proof assistant. We present an elegant segment-based reasoning over the companion array developed for the proof. Finally, we validate the proposed specification by proving a few functions manipulating lists.

Keywords: linked lists, deductive verification, operating system, internet of things, Frama-C.

1 Introduction

Connected devices and services, also referred to as Internet of Things (IoT), are gaining wider and wider adoption in many security critical domains. This raises important security challenges, which can be addressed using formal verification.

This paper focuses on Contiki [10], a popular open-source operating system for IoT devices providing full low-power IPv6 connectivity, including 6TiSCH, 6LoWPAN, RPL, or CoAP standards. It is implemented in C with an emphasis on memory and power optimization, and contains a kernel linked to platform-specific drivers at compile-time. When Contiki was created in 2002, no particular attention was paid to security. Later, communication security was integrated, but formal verification of code was not performed until very recent case studies [19, 21].

The goal of this work is to perform deductive verification of the linked list module, one of the most critical modules of Contiki. It is performed using the ACSL specification language [4] and the deductive verification plugin WP of FRAMA-C [16]. Our approach is based on a parallel view of a linked list via a companion ghost array. Such a "flattened" view of the list avoids complex inductive predicates and allows for automatic

```
1  struct list {
2    struct list *next;
3    int k; // a data field
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 *previtem, void *newitem);
29 //Get the element following item
30 void * list_item_next(void *item);
```

Fig. 1: API of the `list` module of Contiki (for lists with one integer data field)

proof of most properties, but requires to maintain a strong link between the companion array and the list it models. All proofs have been checked for several list structures. A few auxiliary lemmas have been proved in the Coq proof assistant [24], where a significant effort has been done to make their proofs robust and independent of the specific list structure. Finally, we have validated the proposed specification by proving a few functions manipulating lists.

The contributions of this work include

- formal specification of the `list` module[4] of Contiki in the ACSL specification language and its deductive verification using the WP plugin of FRAMA-C;
- a presentation of the underlying approach based on a companion ghost array;
- formal statement and proof of several lemmas useful for reasoning about this representation;
- pointing out an (unintended?) inconsistency in precondition of one function;
- a preliminary validation of the proposed specification of the module via a successful verification of a few annotated test functions dealing with lists.

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

2 The List Module of Contiki

Required by 32 modules and invoked more than 250 times in the core of the OS, the linked list module (`list`) is a crucial library in Contiki. It is used, for instance, to implement the scheduler, where lists are used to manage timers and processes. Its verification is thus a key step for proving many other modules of the OS.

The API of the module is given in Figure 1. Technically, it differs from many common linked list implementations in several regards. First, in Contiki an existing list

[4] Complete annotated code available at http://allan-blanchard.fr/code/contiki-list-verified.zip

(illustrated by the lower part of Figure 3) is identified or modified through a dedicated list handler – a variable supposed to refer to the first list element – called `root` in Figure 3. Copying one linked list into another is just copying such a list handler (cf. lines 22–23 in Figure 1) without duplicating list elements. In a function call, an existing list is thus 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`) to make it possible to modify the handler in the function.

Second, being implemented in C (that does not offer templates), Contiki uses a generic mechanism to create a linked list for specific field datatypes using dedicated macros. The preprocessor transforms such a macro into a new list datatype definition. Lines 1–4 in Figure 1 show the resulting definition for a list with one integer field. To be applicable for various types, the common list API treats list elements via either **`void`**$\star$ pointers or pointers to a trivial linked list structure (having only lines 1,2,4), and relies on (explicit and implicit) pointer casts. To make possible such a "blind" list manipulation using casts, the first field in any list element structure must be the pointer to the next list element (cf. line 2). That means that, for a user-defined type **`struct`** `some_list`, when a cell of this type is transmitted to the list API, the implementation first erases the type to **`void`**$\star$ and then casts it to **`struct`** `list`$\star$ to perform list manipulations and modifications. Note that according to the C standard, this implementation violates the strict-aliasing rule, since we modify a value of type **`struct`** `some_list` through a pointer to a type **`struct`** `list`. So the compilation of Contiki is configured to deactivate the assumption of strict-aliasing compliance.

Third, Contiki does not provide dynamic memory allocation, which is replaced by attributing (or releasing) a block in a pre-allocated array [19]. In particular, the size of a list is always bounded by the number of such blocks, and their manipulation does not invoke dynamic memory allocation functions.

Fourth, adding an element at the start or the end of a list is allowed even if this element is already in the list: in this case, it will first be removed from its previous position. Finally, the API is very rich: it can handle a list as a FIFO or a stack (lines 14–21), and supports arbitrary removal/insertion and enumeration (lines 24–30).

For all these reasons, the list module of Contiki appears to be a necessary but challenging target for verification with FRAMA-C/WP.

3 The Verification Approach

This section presents our verification approach and results. Since the generic mechanism of type manipulation lies beyond the border of undefined behavior in C (cf. Section 2), formally speaking, a separate verification is necessary for any new list structure datatype as for a different API. That is why in this verification case study we use a precise definition of list structure. In the presentation of this paper, we assume that the list structure is defined by lines 1–4 of Figure 1 and use pointers to **`struct`** `list` in the verified functions instead of generic **`void`**$\star$ pointers. To ensure that this choice of list structure is not a limitation of the approach, we additionally check that our specification and proof remain valid for other common list structures (where data fields are a pointer,

```
1  #define MAX_SIZE INT_MAX-1
2  /*@
3    requires \valid(pLst) ∧ \valid(item) ∧ \valid(cArr + (0 .. MAX_SIZE-1));
4    requires Linked: linked_n (*pLst, cArr, index, n, NULL);
5    requires 0 ≤ index ∧ 0 ≤ n < MAX_SIZE ;
6    requires EnoughSpace: index + n + 1 ≤ MAX_SIZE;
7    requires Unique: ∀ ℤ y, z;
8      index ≤ y < index + n ∧ index ≤ z < index + n ∧ y ≠ z ⇒ cArr[y] ≠ cArr[z];
9    requires item_index == index_of(item, cArr, index, index+n) ;
10
11   assigns cArr[index .. index + n], cArr[item_index - 1]->next, item->next, *pLst;
12
13   behavior contains_item:
14     assumes ∃ ℤ i ; index ≤ i < index+n ∧ cArr[i] == item ;
15     ensures linked_n(*pLst, cArr, index, n, NULL);
16     ensures unchanged{Pre,Post}(cArr, item_index + 1, index+n);
17     ensures array_swipe_right{Pre, Post}(cArr, index + 1, item_index + 1);
18     ensures cArr[index] == item;
19     ensures Unique: ...; // no repetitions in the list (see lines 7-8)
20
21   behavior does_not_contain_item:
22     assumes ∀ ℤ i ; index ≤ i < index+n ⇒ cArr[i] ≠ item ;
23     ensures linked_n(*pLst, cArr, index, n+1, NULL);
24     ensures array_swipe_right{Pre, Post}(cArr, index + 1, index + n + 1);
25     ensures cArr[index] == item;
26     ensures Unique: ...; // no repetitions in the list (similar to lines 7-8)
27
28   complete behaviors;
29   disjoint behaviors;
30 */
31 void list_push(list_t pLst, struct list *item,
32   /* ghost: */ struct list **cArr, int index, int n, int item_index)
33 {
34   struct list *l;
35   //@ ghost int rem_n = item_index == index+n ? n : n-1 ;
36
37   list_remove(pLst, item, /* ghost: */ cArr, index, n, item_index);
38   //@ assert array_swipe_left{Pre,Here}(cArr, item_index, index + rem_n);
39   //@ assert unchanged{Pre,Here}(cArr, index, item_index);
40
41   //@ ghost array_push(cArr, index, rem_n, list, *list, NULL);
42   //@ assert array_swipe_right{Pre,Here}(cArr, index + 1, item_index + 1);
43   //@ assert unchanged{Pre,Here}(cArr, item_index + 1, index + rem_n + 1);
44
45   //@ assert linked(*pLst, cArr, index + 1, rem_n, NULL);
46
47   item->next = *pLst;
48   //@ ghost cArr[index] = item ;
49
50   //@ assert linked_n(item, cArr, index, rem_n+1, NULL);
51   *pLst = item;
52 }
```

Fig. 2: Function `list_push` adds given element `item` to the start of the list

or a structure containing three coordinates of a point). All proofs remain successful for the tested list structures.

3.1 Running Example

We will use the function `list_push` (see Figure 2) to illustrate the specification and verification of the list module in the rest of this article. This function adds a given list

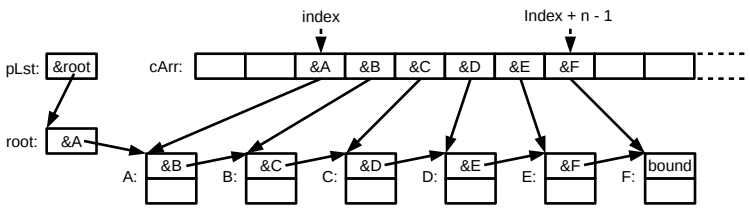

Fig. 3: Parallel view of a list prefix using a companion array formally defined by the `linked_n` predicate in Figure 4

element `item` at the beginning of the list whose list handler is referred to by `pLst`. As the API must ensure that each list element appears at most once in the list, this function first tries to remove `item` from the list (see line 37 of Figure 2). After this operation, it is guaranteed that `item` is not in the list. Then the `next` field of `item` is set to point to the (previous) first element of the list, that is, `*pLst` (line 47). Finally, the handler `*pLst` is updated to point to the new first element `item` (line 51).

The annotated version of the function also takes some ghost parameters (given on line 32) supposed to be ignored during compilation and execution. Note that we include here ghost parameters as regular parameters "considered-as-ghost" by preceding them by a comment `/* ghost: */` rather than syntactically writing them as ghost parameters inside an annotation `/*@ ghost: <type><param>; */`. While being part of the ACSL specification language [4], ghost parameters are currently not yet supported by the public releases of FRAMA-C/WP. It means that in order to verify other modules of the OS using lists, we currently have to modify all list function calls to add such "considered-as-ghost" parameters. To obtain the executable code from the annotated code, only a slight modification of code is required for the moment: to remove these "considered-as-ghost" parameters clearly marked on a separate line (since all other annotations are already seen as comments by a compiler). The support of ghost parameters is in progress in the development version and should be available in the near future. With this support, it will only be necessary to add ghost parameters in the modules under verification using lists without modifying the C API.

3.2 List Representation by a Companion Ghost Array

Our verification approach relies on a parallel view of the list via a companion ghost array whose cells refer to the elements of the list (see Figure 3). Basically, it allows us to transform most of the inductive properties required during the verification into simple universally quantified formulas. Such formulas are easier to handle by SMT solvers, and generally easier to write. The predicate `linked_n` (cf. Figure 4) inductively builds the link between a list and a segment of the companion array that models it. We define a few auxiliary lemmas that help to deal with this property without having to reason by induction (this induction being hidden in the proofs of the lemmas, as we explain in Section 3.5).

```
1  /*@
2    inductive linked_n{L}(struct list *root, struct list **cArr,
3                          ℤ index, ℤ n, struct list *bound) {
4      case linked_n_bound{L}:
5        ∀ struct list **cArr, *bound, ℤ index;
6          0 ≤ index ≤ MAX_SIZE ⇒ linked_n(bound, cArr, index, 0, bound);
7      case linked_n_cons{L}:
8        ∀ struct list *root, **cArr, *bound, ℤ index, n;
9          0 < n ⇒ 0 ≤ index ⇒ 0 ≤ index + n ≤ MAX_SIZE ⇒
10         \valid(root) ⇒ root == cArr[index] ⇒
11         linked_n(root->next, cArr, index + 1, n - 1, bound) ⇒
12           linked_n(root, cArr, index, n, bound);
13   }
14 */
```

Fig. 4: Inductive predicate `linked_n` creating a link between the list prefix of size `n` of the linked list `root` and the segment of indices `index..index+n-1` in the companion array `cArr`, where the sublist boundary `bound` refers to the list element immediately following this prefix. If the list is of size `n`, `bound` is `NULL`.

More precisely, the inductive predicate `linked_n` establishes a relation between the prefix of length `n` of the list starting at `root` and the segment of size `n` in the companion array `cArr` starting from `index`. The relation is established at label `L`, which can be omitted being by default the current program point. The pointer `bound` refers to the list element immediately following the represented sublist, that is, to the (1+`n`)-th element in the list. We refer to it as the sublist *boundary*. For an empty list (cf. lines 4–6), `root` and `bound` must be equal, and the list is considered to be linked with the empty segment of the companion array starting at any `index`. If the segment size (and the list length) `n` is greater than 0 (cf. lines 7–12), the first element `root` must point to a valid list element and this element must be registered in `cArr` at `index`. Moreover, the remaining part of the list prefix (starting at `root->next`) must in turn be linked to the array segment of size `n-1` starting at `index+1` with the same boundary element `bound`. To verify that the list representation relation holds, we will have to update the ghost array each time we modify the linked list. It is done through ghost functions and instructions presented below.

Notice that the ACSL annotations in this paper are slightly pretty-printed: $\mathbb{Z}$ denotes the type **`integer`** of mathematical integers in ACSL, while universal and existential quantifiers are replaced by $\forall$ and $\exists$, respectively.

3.3 Formal Specification

To perform deductive verification of the list module in FRAMA-C/WP, we first provide a formal specification of the `list` API in the ACSL specification language [4]. Let us illustrate the formal specification for the function `list_push`, whose (simplified) contract is given in Figure 2. The parameters on line 32 are considered to be ghost parameters.

The precondition is given by **`requires`** clauses on lines 3–9 in Figure 2. First (cf. line 3), both the pointer `pLst` to the list handler and the list element `item` must be pointers to valid memory locations. The companion array is assumed to be valid for a

```
1  /*@
2    //Range down..up-1 in cArr and the contents of the corresponding list elements
3    //are the same at labels L1 and L2
4    predicate unchanged{L1,L2}(struct list **cArr, ℤ down, ℤ up) =
5      ∀ ℤ i; down ≤ i < up ⇒
6        \at(cArr[i], L1) == \at(cArr[i], L2) ∧
7        (\valid{L1}(\at(cArr[i],K)) ⇒ \valid{L2}(\at(cArr[i],L2))) ∧
8        \at(*(cArr[i]), L1) == \at(*(cArr[i]), L2);
9  */
10 /*@
11   //Range down..up-1 at label L2 is a right shift of down-1..up-2 at label L1
12   predicate array_swipe_right{L1, L2}(struct list **cArr, ℤ down, ℤ up) =
13     ∀ ℤ i; down ≤ i < up  ⇒
14       \at(cArr[i], L2) == \at(cArr[i-1], L1);
15 */
```

Fig. 5: Predicates relating the contents of a list between two program points `L1` and `L2`, expressed in terms of the companion array `cArr`

large range of indices as well. The representation of list `pLst` by the companion array `cArr` is assumed on line 4. The segment of size `n` starting at `index` represents here the whole list (since the list boundary is equal to `NULL`). Lines 5–6 assume necessary domain constraints for `n` and `index`, in particular, the possibility to add one more element at the end of the segment in `cArr`. Lines 7–8 assume that any list element appears in the list at most once. Line 9 assumes that the ghost parameter `item_index` records the position of `item` in `cArr` (and thus in the list). It is computed by the logic function `index_of` whose definition is straightforward and not presented here. This function returns the position of `item` in `cArr` if it can be found in the segment of indices `index..(index+n-1)`, or one past the last segment index (`index+n`) otherwise. Notice that the last two properties of the list are conveniently expressed in terms of the companion array.

The **`assigns`** clause on line 11 specifies the variables that the function is allowed to modify. For other postconditions, we distinguish two cases (defined in ACSL by behaviors, cf. lines 13 and 22): either `item` is already present in the list or not. An **`assumes`** clause defines the domain of application of each behavior. In both cases, the resulting list must be represented by the companion array. However, in the first case (behavior `contains_item`, lines 13–19), it must have the same size as before (cf. line 15), since we first remove the element and then add it again at the start of the list, whereas in the second case (behavior `does_not_contain_item`, lines 21–26), the resulting list will grow (cf. line 23). In both cases, `item` is added at the start of the list (cf. lines 18, 25). To express the conditions on other elements, we need two additional predicates given in Figure 5.

The predicate `unchanged` in Figure 5 states that, between two programs points at labels `L1` and `L2`, in the range `down..up-1` both the elements of the companion array and the contents of the corresponding list elements are the same, and the validity of the list elements has been preserved between `L1` and `L2`. The predicate `array_swipe_right` states that the cells in the range `down-1..up-2` at label `L1` are shifted to the right to become the range `down..up-1` at label `L2`.

Thanks to these predicates, we can specify the remaining parts of the contract. In the behavior `does_not_contain_item`, where `item` is originally not in the list, `item` is simply placed at the beginning of the list. In terms of the companion array, it is expressed as a shift of the corresponding segment to the right (cf. line 24). In the behavior `contains_item`, `item` is removed at its previous position `item_index` and added at the start of the list. In terms of the companion array, the segment of indices `item_index+1..index+n-1` remains unchanged between the **`Pre`** state (before the call) and the **`Post`** state (after the call, cf. line 16). However, the segment of indices `index..item_index-1` at **`Pre`** is shifted to the right to the range of indices `index+1..item_index` at **`Post`** (cf. line 17). This precisely describes the desired postconditions for the list.

The postconditions on lines 19 and 26 state that the uniqueness of list elements (lines 7–8) is preserved by the function in both cases for the new companion array segment representing the list (in the second case, the new segment is one element longer).

Lines 28–29 indicate that the given behaviors are complete and disjoint: they cover all possible cases and cannot apply at the same time. Thanks to these annotations, the completeness and disjointness of behaviors are checked by FRAMA-C/WP.

A quite important part of the contract that we have removed in this simplified version is separation conditions. The ACSL language and the FRAMA-C/WP tool support such separation properties. Each element of the list must be spatially separated from any other to guarantee that the list is well formed. In ACSL, this property can be expressed as follows:

```
1 ∀ ℤ y, z; index ≤ y < index + n ∧ index ≤ z < index + n ∧ y ≠ z ⇒
2   \separated( *(array + y), *(array + z) );
```

Furthermore, the list elements must be separated from the companion ghost array to guarantee that any operation on the companion array does not impact the linked list itself and *vice versa*. It can be expressed as follows:

```
1 ∀ ℤ y ; index ≤ y < index + n ⇒
2   \separated( *(array + y), array + (0 .. MAX_SIZE -1 ) );
```

These properties[5] are systematically included as preconditions and postconditions in each function contract of the API. In the precondition of `list_push`, we also need to state that list elements and companion array cells are separated from the list handler referred to by `pLst` and the element `item` to be added. The complete version of the contracts is available online.

3.4 Ghost Functions

To maintain the list representation by the companion array, we have to update the array each time the linked list is modified. It is done through ghost functions and instructions. Let us illustrate it for the `list_push` function. First, `item` is removed if needed (line 37 in Figure 2). Line 35 defines the length `rem_n` of the resulting list (and companion array segment) after this operation: `n` if `item` was not present, and `n-1` otherwise.

[5] some of which will be no longer necessary when ghost parameters will be fully supported in FRAMA-C.

```
1  /*@
2    requires \valid(pLst) ∧ \valid(cArr + (0 .. MAX_SIZE - 1));
3    requires EnoughSpace: index + n + 1 ≤ MAX_SIZE;
4    requires Linked: linked_n (root, cArr, index, n, bound);
5    requires Unique: ...; // no repetitions in the list
6
7    assigns cArr[index .. index + n];
8
9    ensures Unique: ...; // no repetitions in the list
10   ensures SwipeR: array_swipe_right{Pre, Post}(cArr, index + 1, index + n + 1);
11   ensures Linked: linked_n(root, cArr, index + 1, n, bound);
12 */
13 void array_push(struct list **cArr, int index, int n,
14                 list_t pLst, struct list *root, struct list *bound)
15 {
16   int i = index + n;       // next index to assign in cArr
17   struct list *le = bound; // current boundary between left and right segments
18   /*@
19     loop invariant IInBounds: index ≤ i ≤ index + n ;
20     loop invariant UnchangedLeft: unchanged{Pre, Here}(cArr, index, i);
21     loop invariant ISwipeR: array_swipe_right{Pre, Here}(cArr, i + 1, index+n+1);
22     loop invariant LeftLinked: linked_n(root, cArr, index, i - index, le);
23     loop invariant RightLinked: linked_n(le, cArr, i + 1, n - (i - index), bound);
24     loop assigns i, le, cArr[index+1 .. index + n];
25     loop variant i - index;
26   */
27   while (i > index){
28     cArr[i] = cArr[i-1]; // shift next cArr element
29     i--;
30
31     // compute new boundary
32     if( i == index ) le = root;
33     else             le = cArr[i];
34   }
35 }
```

Fig. 6: Ghost function `array_push` shifts ghost array `cArr` to the right

Lines 38–39 provide two assertions on the two segments around the deleted element: the left segment up to the initial position `item_index` of `item` is unchanged (line 39), while the right segment – non empty only if `item` was present in the list – is obtained as a left shift of a segment at state **`Pre`**. (The definition of a left shift is similar to the right shift and is omitted here.) These properties come from the contract of `list_remove`.

Next, `item` has to be added at the beginning of the list. We make the choice to keep the same starting position `index` for the segments that model the list in the precondition and in the postcondition (cf. lines 4, 15 and 23). To create some place for a new first element in the beginning of the segment, we use the ghost function `array_push` (cf. line 41 in Figure 2).

The `array_push` function (see Figure 6) shifts the segment of the companion array one cell to the right. Starting from the end of the segment (at position `i=index+n` in the array), it moves the element at position `i-1` to position `i` (cf. lines 27–34). The boundary between the shifted and not-yet-shifted segments is maintained in variable `le`. The function contact specifies the shift (cf. line 10) and preserves the link of the list with the shifted segment (cf. lines 4, 11). Line 7 indicates the segment of the companion array that is modified. Uniqueness properties (lines 5, 9) and separation properties (not

```
1  /*@
2    lemma stay_linked{L1, L2}:
3      ∀ struct list *root, **cArr, *bound, ℤ i, n ;
4        linked_n{L1} (root, cArr, i, n, bound) ⇒
5        unchanged{L1, L2}(cArr, i, i+n) ⇒
6          linked_n{L2} (root, cArr, i, n, bound);
7  */
8  /*@
9    lemma linked_split_segment:
10     ∀ struct list *root, **cArr, *bound, *b0, ℤ i, n, k;
11       n > 0 ⇒ k ≥ 0 ⇒
12         b0 == cArr[i + n - 1]->next ⇒
13         linked_n(root, cArr, i, n + k, bound) ⇒
14           (linked_n(root, cArr, i, n, b0) ∧ linked_n(b0, cArr, i + n, k, bound));
15 */
16 /*@
17   lemma linked_merge_segment:
18     ∀ struct list *root, **cArr, *bound, *b0, ℤ i, n, k;
19       n ≥ 0 ⇒ k ≥ 0 ⇒
20       (linked_n(root, cArr, i, n, b0) ∧ linked_n(b0, cArr, i + n, k, bound)) ⇒
21         linked_n(root, cArr, i, n + k, bound);
22 */
```

Fig. 7: Examples of lemmas about the `linked_n` predicate

presented in the simplified version given in Figure 6) should also be included in the contract (as explained in Section 3.3). The loop contract (lines 18–26) is necessary to reason about segment manipulation as we will explain in Section 3.5.

After the call of `array_push` in the ghost code on line 41 in Figure 2, the cell at position `index` in the companion array is not used. Combining the partial left shift due to a list element removal by the call to `list_remove` and the complete right shift by the call to `array_push`, we obtain the properties of the assertions on lines 42–43 in Figure 2. The interested reader will easily check them by considering separately the case of each behavior. Moreover, we have the representation property on line 45 for the shifted segment.

Thanks to these properties, after connecting the list element `item` to the original list (line 47) and recording `item` in the companion array at position `index` in a ghost assignment (line 48), we obtained the representation of the list started with `item` by the segment of the companion array from position `index` with `rem_n+1` elements and with a boundary `NULL`. After the assignment of the list handler `*pLst` on line 51, the required representation of the resulting list by the companion array is reconstructed (cf. lines 15, 23).

These examples illustrate a benefit of the companion array for this specification – and more precisely for expressing predicates like `unchanged`, `array_swipe_left` and `array_swipe_right`, and separation properties. All these properties can be directly expressed using the companion array. It means that we do not need induction any more to reason about them.

3.5 Auxiliary Lemmas and Proofs

Inductive properties are generally not well handled by SMT solvers since most of the time, a proof that involves inductive properties requires reasoning by induction. On

the contrary, SMT solvers are efficient when they just have to instantiate lemmas that directly state implications between known properties. Thus, providing lemmas about inductive properties can significantly improve the treatment of inductive properties in a proof by requiring reasoning by induction only in the proofs of the lemmas. We already successfully experimented a similar approach to count values within a range of indices in an array [6].

For example, a very simple property (illustrated by Figure 7, lines 2–6) currently not handled by SMT solvers in our verification is the fact that if a list `root` is linked to a companion array `cArr` at a given program point `L1` (line 4), and if the list representation (i.e. the corresponding companion array segment and pointed list elements) has not changed between `L1` and another program point `L2` (line 5), then the list `root` is still linked to `cArr` at program point `L2` (line 6). For example, when we modify some element in the array, this lemma is useful to ensure that a `linked_n` relation still holds for other, unmodified segments before and after this element.

Two more subtle properties are the facts that we can *split* the `linked_n` property at a given valid index into two segments, or conversely *merge* two consecutive segments into a longer one. The `linked_split_segment` lemma (cf. Figure 7) states that if a list starting from `root` is linked to the segment in `cArr` at index `i` with size `n+k` and reaches a given boundary `bound`, we can split it into two relations: the first one linking `root` to the segment at position `i` with `n` elements and reaching boundary `b0`, and the second linking `b0` to the segment at position `i+n` with `k` elements and reaching boundary `bound`, where the intermediary boundary `b0` is defined by line 12. Conversely, if we have two consecutive linked segments in the same companion array, where the boundary of the first segment refers to the first list element of the second list, we can deduce the `linked_n` relation for a longer segment that combines them (cf. lemma `linked_merge_segment` in Figure 7).

This kind of property is, for example, useful for the verification of the `array_push` function. Indeed, in the loop, when the assignment `cArr[i] = cArr[i-1]` at line 28 (of Figure 6) overwrites the cell at position `i`, we have to maintain the representation of the list prefix before this position (line 22 of the invariant), and the *split* lemma allows this because we can detach the end using it. At the same time, this assignment puts a new element just before the beginning of the segment specified by the invariant of line 23, that gives a new `linked_n` predicate, progressively built by the function. As we know that this list element is linked to the first cell of this range, the lemma *merge* allows to combine them into a longer segment.

In the current formalization, these lemmas have been proved using the Coq proof assistant [5,24]. Most of the lemmas related to the predicate `linked_n` have been proved by induction on the predicate itself or on the size `n` of the list. The induction principle used is an induction principle similar to the basic induction principle on Peano natural numbers but applied on the positive subset of relative numbers. The axiomatic function `index_of` is defined on a lower and an upper bound. It was therefore necessary to reason by induction on their difference. One challenge for these proof scripts was to make them very robust so they can be valid for various versions of the list structure. In particular the `unchanged` predicate does not take the same number of arguments for different versions of the list structure.

In addition to these lemmas, four assertions are not proved directly by the SMT solvers. Two are proved in Coq: they are basically applications of lemmas, under some conditions. Two are proved by the recently introduced Interactive Proof Editor (TIP) of the WP plugin. It offers a new panel that focuses on one goal generated by WP, and allows the user to visualize the goal to be proved. The user can then interactively decompose a complex proof into smaller pieces by applying tactics, and the pieces are proved by SMT solvers. In our case two assertions were proved inside each branch of a conditional but were not automatically proved just after the conditional. The proof using TIP was straightforward.

3.6 Results of the Verification

In this work, we have annotated and verified all functions of the list module, except `list_insert` (as detailed below). In total, for about 176 lines of C code in the module (excluding macros), we wrote 46 lines for ghost functions, and about 1400 lines of annotations, including about 500 lines for contracts and 240 lines for logic definitions and lemmas. 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. For this annotated version of the module, the verification using FRAMA-C/WP generates 798 goals. This number includes 108 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 24 auxiliary lemmas (that is, in total only about 3.3% of properties). The 24 lemmas are proved using Coq v.8.6.1. Out of the 774 remaining goals, almost all are automatically discharged by SMT solvers, except for 4 goals that are proved interactively (as mentioned in Section 3.5). In this work, we used FRAMA-C v.16 Sulfur and the solvers Alt-Ergo v.1.30 (with direct translation from WP and via Why3), as well as Z3 v.4.5 and CVC3 v.2.4.1 (via Why3).

The verification helped to identify an inconsistency for the remaining function, `list_insert` (cf. lines 26–28 in Figure 1), with respect to the assumptions of other functions (and, in a sense, to itself). This function adds an element `new_item` into a list just after a given element `prev_item`. If the element `prev_item` is `NULL`, the function directly calls `list_push`, meaning that if the element is already there, it is removed and then added to the start. However, if `prev_item` is present in the list, the function directly adds `new_item` after `prev_item` without removing a previous instance of `new_item` from the list (if any). It shows that the uniqueness property is in general not preserved by the function, but in some cases it is. Thus this function does not respect a contract consistent with the other functions. We decided to ask the authors of Contiki to clarify this potentially dangerous behavior, that could for example allow to break the integrity of the list[6]. Moreover, in the entire code of Contiki, we have found only one call to `list_insert`, and not a single one in the core part of the system.

Unit tests could have permitted to identify such a bug. However, one difficulty with tests (that is also the cause of many security bugs) is the fact that we tend to test valid scenarios rather than invalid ones.

[6] the issue can be found at: https://github.com/contiki-ng/contiki-ng/issues/254

3.7 Validation of Specification

To get confidence in the proposed specification, we have implemented 15 simple valid test functions[7] manipulating lists, and tried to prove simple properties on lists in them using the proposed contracts of the list module functions. The results show that in all tests, the correct properties were successfully proved by WP.

We have also implemented 15 invalid tests. Each invalid function is a variant of a valid one where we have altered the contract (including such dangerous cases as violations of separation, or validity, or uniqueness), the ghost values or the code itself. For those functions, the verification leads, as expected, to proof failure.

This gives us further confidence that the proposed contracts can be successfully used for a larger deductive verification of Contiki. This step has also allowed us to detect and fix some minor deficiencies in the contracts at the latest phases of the work and can be recommended to be systematically performed for all similar verification projects.

4 Related Work

To our knowledge, there is no other specification of a linked list API using ghost arrays. This approach has two main advantages. First such specifications may be more readable to developers not used to formal specifications written in a more functional style using inductive type definitions. Secondly, it makes these specifications close to be usable in a context of dynamic verification [8], in particular using the E-ACSL plugin of FRAMA-C [17]. One drawback is that the support of ghost function parameters is not yet available, and another more important one, is that the specifications should contain assertions stating the separation of the actual memory cells and the ghost array.

Dross et al [9] present the proof of an implementation of red-black trees in SPARK that involves underlying arrays. They also rely on ghost code for the proof, the main difference in our work is that the array we use for representation is only part of the specification and thus the cells can be allocated independently of this array using another policy, while they use arrays as the actual implementation of the trees. On the proof side, the main difference is that we rely on Coq to prove simple lemmas that can be used automatically by SMT solvers, while they use the so-called auto-active proof [18] to avoid the use of an interactive proof assistant.

In this context, separation logic [23] is more suitable than the Hoare logic on which the WP plugin of FRAMA-C is based. Tools based on such a kind of logic may therefore be more suitable for the verification of a linked list API, for example VeriFast [15] or the Verified Software Toolchain [1, 3]. The former has been used in several industrial case studies [22] while the latter has been mainly used to verify crytography related software [2,26] but also a message passing system [20]. We are not aware of efforts dedicated to the verification of linked list functions, but the example gallery of VeriFast [8] and the VST case studies do include linked list function specifications and verification. They are based on a logic list data structure and an inductive predicate relating the memory and such a logical data structure, as initially done by Reynolds [23]. Reynolds

[7] included in the online archive with the annotated code.

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

reasoned on sequences of instructions rather than functions. He thus mostly expressed loop invariants. This has an impact on the style of the specifications. In this case, an existential quantification of logical data structures is convenient. VeriFast has the concept of patterns, that are kind of free variables in preconditions, that are bound to values during symbolic execution, and that can be used in postconditions. Reynolds' style of specification is therefore possible in VeriFast. In the case of ACSL and JML, when specifying functions or methods, an existential quantification in the precondition only binds a variable in the precondition: the scope does not extend to the postcondition. Using Reynolds' style of specification is therefore not possible. In the case of JML, to specify Java methods on a linked list data structure, Gladisch and Tyszberowicz [12] used a pure observer method that takes a list object and an index, and returns the object at that index in the list. The methods they consider are simpler than the list API of Contiki, but essentially our ghost arrays can be seen as observations of the linked lists.

VeriFast and VST are based on *concurrent* separation logic [7, 14]. It is difficult to compare the specifications because most VeriFast case studies take into account concurrency. For some of the examples related to linked lists, being based on separation logic does not seem to have a significant impact on the size of specifications with respect to the specifications we have.

For the verification, FRAMA-C is the most automated. Moreover, in addition to the automated and interactive provers it natively supports, it can output verification conditions to many different provers using Why3ML [11] as an intermediate verification language. This eases the verification, and makes it more trustworthy. VeriFast is based on its embedded SMT solver Redux and can also use Z3. VST is a framework for Coq. The specifications can be written in the Gallina language of Coq, making them very expressive. However, even with the dedicated tactics, proofs are less automated in VST than in FRAMA-C or VeriFast.

One strong point of VST is that its logic is fully formalized in Coq and its soundness has been proved in Coq. It makes it the safer framework. Both for the WP Plugin and VeriFast, there exist some results about the correctness of subsets of the tools [13, 25].

5 Conclusion and Future Work

The expansion of devices connected to the Internet raises many security risks. One promising way to tackle this challenge is to use formal methods, which is one of the goals of the EU H2020 VESSEDIA project. This paper reports on verification of the list module of Contiki, which is one of the most critical modules of the operating system. It requires, in the context of C language, to deal with linked data structures and memory separation that are still hard to handle automatically.

Our verification approach relies on a companion ghost array and some ghost code. Although the idea of using ghost code is not new, it appears to be highly beneficial in this context, allowing for an elegant reasoning over the companion array and for an automatic proof of the great majority of goals: more than 96% of goals in this case study have been proved automatically. Imposing a limit on the size of the companion array (and therefore, the lists it models) is not a limitation since list sizes in Contiki are always bounded. The verification of the list module, intensively used by other modules of

Contiki, opens the way to formal verification of higher-level modules depending on it. All proofs of this case study have been checked for several list structures. Moreover, we have implemented several test functions working with lists, and validated the proposed specification by obtaining a successful proof for correct properties, and a proof failure in the erroneous cases.

In future work, we plan to start the verification of higher-level modules of Contiki. This verification is planned to be done using both the deductive verification tool FRAMA-C/WP and the value analysis tool FRAMA-C/EVA. The latter can handle linked data-structures slightly better than the former, but again, it is more suitable to reason about arrays. While it would have been impossible to prove the equivalence between our array representation and the lists using EVA (that is not really meant to treat functional properties), it can still benefit of this verification since we can now directly reason about arrays. Note however that it still requires to instrument user code, and to verify that lists are only modified through the API.

Acknowledgment. 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. Appel, A.W.: Verified software toolchain. In: Barthe, G. (ed.) Programming Languages and Systems (ESOP). LNCS, vol. 6602, pp. 1–17. Springer (2011)
2. Appel, A.W.: Verification of a cryptographic primitive: SHA-256. ACM Trans. Program. Lang. Syst. 37(2), 7:1–7:31 (Apr 2015)
3. Appel, A.W., Dockins, R., Hobor, A., Beringer, L., Dodds, J., Stewart, G., Blazy, S., Leroy, X.: Program Logics for Certified Compilers. Cambridge University Press (2014)
4. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, http://frama-c.com/acsl.html
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Springer (2004)
6. Blanchard, A., Kosmatov, N., Lemerre, M., Loulergue, F.: A case study on formal verification of the anaxagoros hypervisor paging system with Frama-C. In: Proc. of the 20th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2015). LNCS, vol. 9128, pp. 15–30. Springer (Jun 2015)
7. Brookes, S., O'Hearn, P.W.: Concurrent separation logic. ACM SIGLOG News 3(3), 47–65 (Aug 2016)
8. Clarke, L.A., Rosenblum, D.S.: A historical perspective on runtime assertion checking in software development. SIGSOFT Softw. Eng. Notes 31(3), 25–37 (May 2006)
9. Dross, C., Moy, Y.: 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. pp. 68–83 (2017)
10. Dunkels, A., Gronvall, B., Voigt, T.: Contiki – a lightweight and flexible operating system for tiny networked sensors. In: LCN 2014. IEEE (2004)

11. Filliâtre, J.C., Paskevich, A.: Why3 - where programs meet provers. In: ESOP 2013
12. Gladisch, C., Tyszberowicz, S.: Specifying linked data structures in JML for combining formal verification and testing. Science of Computer Programming 107-108, 19 – 40 (2015)
13. Herms, P., Marché, C., Monate, B.: A certified multi-prover verification condition generator. In: Verified Software: Theories, Tools, Experiments (VSTTE). LNCS, vol. 7152, pp. 2–17. Springer (2012)
14. Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle semantics for concurrent separation logic. In: Drossopoulou, S. (ed.) ESOP. pp. 353–367. No. 4960 in LNCS, Springer (2008)
15. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Nasa Formal Methods (NFM). pp. 41–55. No. 6617 in LNCS, Springer-Verlag, Berlin Heidelberg (2011)
16. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. 27(3), 573–609 (2015), http://frama-c.com
17. Kosmatov, N., Signoles, J.: A lesson on runtime assertion checking with Frama-C. In: Runtime Verification (RV). LNCS, vol. 8174, pp. 386–399. Springer (2013)
18. Leino, K.R.M., Moskal, M.: Usable auto-active verification (2010), http://fm.csl.sri.com/UV10/
19. Mangano, F., Duquennoy, S., Kosmatov, N.: A memory allocation module of Contiki formally verified with Frama-C. A case study. In: CRiSIS 2016. LNCS, vol. 10158. Springer (2016)
20. Mansky, W., Appel, A.W., Nogin, A.: A verified messaging system. Proc. ACM Program. Lang. 1(OOPSLA), 87:1–87:28 (Oct 2017)
21. Peyrard, A., Duquennoy, S., Kosmatov, N., Raza, S.: Towards formal verification of Contiki: Analysis of the AES–CCM* modules with Frama-C. In: RED-IoT 2018, co-located with EWSN 2018. ACM (2018), to appear
22. 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)
23. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS). pp. 55–74. IEEE Computer Society (2002)
24. The Coq Development Team: The Coq proof assistant. http://coq.inria.fr,
25. Vogels, F., Jacobs, B., Piessens, F.: Featherweight VeriFast. Logical Methods in Computer Science 11(3) (2015)
26. Ye, K.Q., Green, M., Sanguansin, N., Beringer, L., Petcher, A., Appel, A.W.: Verified correctness and security of mbedTLS HMAC-DRBG. In: ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 2007–2020. ACM, New York, NY, USA (2017)