

Contents

01. Introduction
02. Ghosts for lists
03. Conclusion & Future work

01

Introduction

The Vessedia project

EU H2020 Vessedia project

- aims at making formal methods more usable in the context of the IoT
- comprises use-cases to evaluate the efficiency of the developed tools and methods
- <https://vessedia.eu>

Contiki-OS

- one of the use-cases targeted in Vessedia
- a lightweight OS for IoT

A lightweight OS for IoT

Contiki is a lightweight operating system for IoT

It provides a lot of features (for a micro-kernel):

- (rudimentary) memory and process management
- networking stack and cryptographic functions
- ...

Typical hardware platform:

- 8, 16, or 32-bit MCU (little or big-endian),
- low-power radio, some sensors and actuators, ...



Note for security: **there is *no* memory protection unit.**

Contiki and Formal Verification

- When started in 2003, **no particular attention to security**
- Later, **communication security** was added at different layers, via standard protocols such as IPsec or DTLS
- **Security of the software** itself did not receive much attention
- Continuous integration system does not include **formal verification**
 - > and unit tests are under-represented

Today's talk: the list module of Contiki

- a critical component of the core part of Contiki
- many client modules in the whole OS
- verification performed with Frama-C

Frama-C at a glance



Software Analyzers

- A **F**ramework for **M**odular **A**nalysis of **C** code
- Developed at CEA List
- Released under **LGPL** license
- **ACSL** annotation language
- **Extensible plugin oriented platform**
 - > **Collaboration of analyses** over same code
 - > **Inter plugin communication** through ACSL formulas
 - > **Adding specialized plugins** is easy
- <http://frama-c.com/> [Kirchner et al. FAC 2015]

ACSL: ANSI/ISO C Specification Language

Presentation

- Based on the notion of **contract** like in Eiffel, JML
- Allows users to specify **functional properties** of programs
 - > Correctness of the specification is crucial
 - > Attacks can exploit every single flaw \Rightarrow Complete proof is required!
- <http://frama-c.com/acsl>

Basic Components

- First-order logic
- Pure C expressions
- C types + \mathbb{Z} (integer) and \mathbb{R} (real)
- Built-in predicates and logic functions particularly over pointers: `\valid(p)`, `\valid(p+0..2)`, `\separated(p+0..2,q+0..5)`, `\block_length(p)`

Plugin Frama-C/WP

WP: A plugin for deductive verification

- Based on **Weakest Precondition** calculus [Dijkstra, 1976]
- **Goal: Prove** that a given program respects its specification
- Requires **formal specification**
- **Capable to formally prove** that
 - > each program function **always** respects its contract
 - > each function call **always** respects the expected conditions on its inputs
 - > each function call **always** provides sufficient guarantees to ensure the caller's contract
 - > common security related errors (e.g. buffer overflows) can **never** occur

02

Ghosts for lists

The LIST module - Overview

Provides a generic list API for linked lists

- about 176 LOC (excl. MACROS)
- required by 32 modules of Contiki
- more than 250 calls in the core part of Contiki

Some special features

- no dynamic allocation
- does not allow cycles
- maintain item unicity

The LIST module - A rich API

```
struct list {
    struct list *next;
};
typedef struct list ** list_t;

void list_init(list_t pLst);
int list_length(list_t pLst);
void * list_head(list_t pLst);
void * list_tail(list_t pLst);
void * list_item_next(void *item);
void * list_pop (list_t pLst);
void list_push(list_t pLst, void *item);
void * list_chop(list_t pLst);
void list_add(list_t pLst, void *item);
void list_remove(list_t pLst, void *item);
void list_insert(list_t pLst, void *previtem, void *newitem);
void list_copy(list_t dest, list_t src);
```

The LIST module - A rich API

```
struct list {  
    struct list *next;  
};  
typedef struct list ** list_t;
```

Observers

```
void list_init(list_t pLst);  
int list_length(list_t pLst);  
void * list_head(list_t pLst);  
void * list_tail(list_t pLst);  
void * list_item_next(void *item);  
void * list_pop (list_t pLst);  
void list_push(list_t pLst, void *item);  
void * list_chop(list_t pLst);  
void list_add(list_t pLst, void *item);  
void list_remove(list_t pLst, void *item);  
void list_insert(list_t pLst, void *previtem, void *newitem);  
void list_copy(list_t dest, list_t src);
```

The LIST module - A rich API

```
struct list {  
    struct list *next;  
};  
typedef struct list ** list_t;
```

```
void list_init(list_t pLst);  
int list_length(list_t pLst);  
void * list_head(list_t pLst);  
void * list_tail(list_t pLst);  
void * list_item_next(void *item);  
void * list_pop (list_t pLst);  
void list_push(list_t pLst, void *item);  
void * list_chop(list_t pLst);  
void list_add(list_t pLst, void *item);  
void list_remove(list_t pLst, void *item);  
void list_insert(list_t pLst, void *previtem, void *newitem);  
void list_copy(list_t dest, list_t src);
```

Observers

Update list beginning

The LIST module - A rich API

```
struct list {  
    struct list *next;  
};  
typedef struct list ** list_t;
```

```
void list_init(list_t pLst);  
int list_length(list_t pLst);  
void * list_head(list_t pLst);  
void * list_tail(list_t pLst);  
void * list_item_next(void *item);  
void * list_pop (list_t pLst);  
void list_push(list_t pLst, void *item);  
void * list_chop(list_t pLst);  
void list_add(list_t pLst, void *item);  
void list_remove(list_t pLst, void *item);  
void list_insert(list_t pLst, void *previtem, void *newitem);  
void list_copy(list_t dest, list_t src);
```

Observers

Update list beginning

Update list end

The LIST module - A rich API

```
struct list {  
    struct list *next;  
};  
typedef struct list ** list_t;
```

```
void list_init(list_t pLst);  
int list_length(list_t pLst);  
void * list_head(list_t pLst);  
void * list_tail(list_t pLst);  
void * list_item_next(void *item);  
void * list_pop (list_t pLst);  
void list_push(list_t pLst, void *item);  
void * list_chop(list_t pLst);  
void list_add(list_t pLst, void *item);  
void list_remove(list_t pLst, void *item);  
void list_insert(list_t pLst, void *previtem, void *newitem);  
void list_copy(list_t dest, list_t src);
```

Observers

Update list beginning

Update list end

Update list anywhere

The LIST Module - User code

```
LIST(route_table);

struct uaadv_rt_entry {
    struct uaadv_rt_entry *next;
    uip_ipaddr_t dest;
    uip_ipaddr_t nexthop;
    uint32_t hseqno;
    uint8_t hop_count;
    uint8_t is_bad;
};

//...

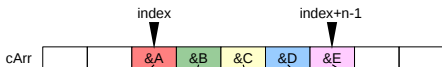
struct uaadv_rt_entry *e;
e = uaadv_rt_lookup_any(dest);
list_push(route_table, e);
```

- based on type erasure
- violates the strict aliasing

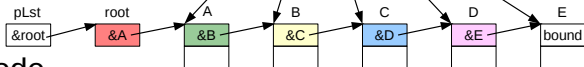
Formalization approach - Overview

We maintain a ghost array that stores the addresses of the different list elements.

Ghost code

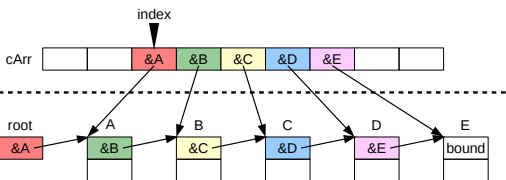


Actual code



Formalization approach - Induction

Ghost code



Actual code

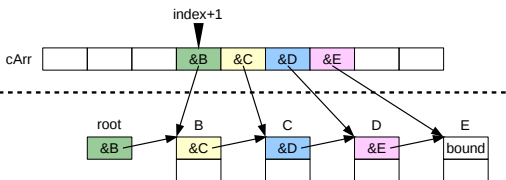
```

inductive linked_n{L}(struct list *root, struct list **cArr,
                      integer index, integer n, struct list *bound) {
// ...
case linked_n_cons{L}:
  \forall struct list *root, **cArr, *bound, integer index, n;
  /*indexes properties*/ ==> \valid(root) ==> root == cArr[index] ==>
  linked_n(root->next, cArr, index + 1, n - 1, bound) ==>
  linked_n(root, cArr, index, n, bound);
}

```

Formalization approach - Induction

Ghost code



Actual code

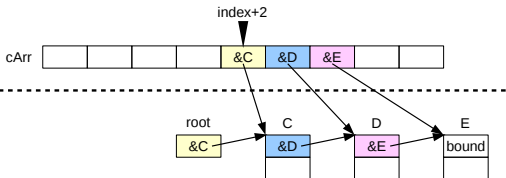
```

inductive linked_n{L}(struct list *root, struct list **cArr,
                      integer index, integer n, struct list *bound) {
// ...
case linked_n_cons{L}:
  \forall struct list *root, **cArr, *bound, integer index, n;
  /*indexes properties*/ ==> \valid(root) ==> root == cArr[index] ==>
  linked_n(root->next, cArr, index + 1, n - 1, bound) ==>
  linked_n(root, cArr, index, n, bound);
}

```

Formalization approach - Induction

Ghost code



Actual code

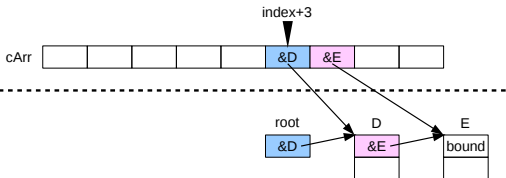
```

inductive linked_n{L}(struct list *root, struct list **cArr,
                    integer index, integer n, struct list *bound) {
// ...
case linked_n_cons{L}:
  \forall struct list *root, **cArr, *bound, integer index, n;
  /*indexes properties*/ ==> \valid(root) ==> root == cArr[index] ==>
  linked_n(root->next, cArr, index + 1, n - 1, bound) ==>
  linked_n(root, cArr, index, n, bound);
}

```

Formalization approach - Induction

Ghost code



Actual code

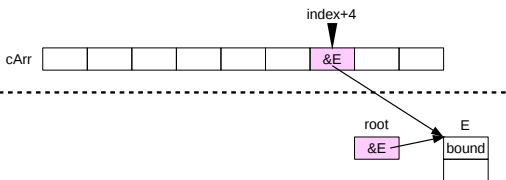
```

inductive linked_n{L}(struct list *root, struct list **cArr,
                      integer index, integer n, struct list *bound) {
// ...
case linked_n_cons{L}:
  \forall struct list *root, **cArr, *bound, integer index, n;
  /*indexes properties*/ ==> \valid(root) ==> root == cArr[index] ==>
  linked_n(root->next, cArr, index + 1, n - 1, bound) ==>
  linked_n(root, cArr, index, n, bound);
}

```

Formalization approach - Induction

Ghost code



Actual code

```

inductive linked_n{L}(struct list *root, struct list **cArr,
                      integer index, integer n, struct list *bound) {
// ...
case linked_n_cons{L}:
  \forall struct list *root, **cArr, *bound, integer index, n;
  /*indexes properties*/ ==> \valid(root) ==> root == cArr[index] ==>
  linked_n(root->next, cArr, index + 1, n - 1, bound) ==>
  linked_n(root, cArr, index, n, bound);
}

```

Formalization approach - Base case

Ghost code

cArr

--	--	--	--	--	--	--	--	--

root

bound

Actual code

```
inductive linked_n{L}(struct list *root, struct list **cArr,
                    integer index, integer n, struct list *bound) {
  case linked_n_bound{L}:
    \forall struct list **cArr, *bound, integer index;
      0 <= index <= MAX_SIZE ==> linked_n(bound, cArr, index, 0, bound);
  // ...
}
```


Formalization approach - Advantages

As long as we maintain the `linked_n` invariant,
we can easily reason about the content of the list:

```
predicate unchanged{L1, L2}(struct list **array, int index, int size) =  
  \forall integer i ; index <= i < index+size ==>  
    \at(array[i]->next, L1) == \at(array[i]->next, L2);
```

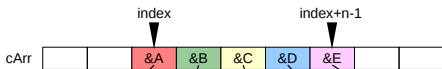
While we have to update the array accordingly when the list is modified.

Example with list_pop

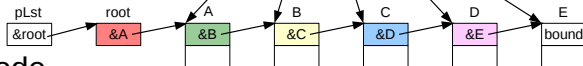
```
void simpl_list_pop(list_t pLst, struct list **cArr, int index, int n){
    if(*pLst != NULL) {
        //@ ghost array_pop(cArr, index, n, pLst, NULL);
        *pLst = (*pLst)->next;
    }
}
```

Current property:

Ghost code



Actual code



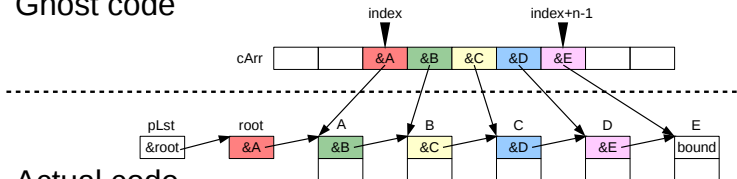
Example with list_pop

```
void simpl_list_pop(list_t pLst, struct list **cArr, int index, int n){
    if(*pLst != NULL) {
        //@ ghost array_pop(cArr, index, n, pLst, NULL);
        *pLst = (*pLst)->next;
    }
}
```

Current property:

```
linked_n(*pLst, cArr, index, n, NULL);
```

Ghost code



Actual code

Example with list_pop

```
void simpl_list_pop(list_t pLst, struct list **cArr, int index, int n){
    if(*pLst != NULL) {
        //@ ghost array_pop(cArr, index, n, pLst, NULL);
        *pLst = (*pLst)->next;
    }
}
```

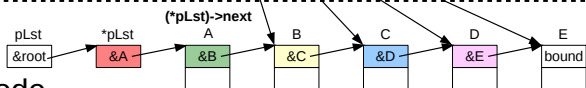
Current property:

```
linked_n((*pLst)->next, cArr, index, n-1, NULL);
```

Ghost code



Actual code



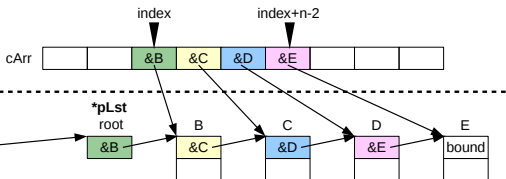
Example with list_pop

```
void simpl_list_pop(list_t pLst, struct list **cArr, int index, int n){
    if(*pLst != NULL) {
        //@ ghost array_pop(cArr, index, n, pLst, NULL);
        *pLst = (*pLst)->next;
    }
}
```

Current property:

```
linked_n(*pLst, cArr, index, n-1, NULL);
```

Ghost code



Actual code

Example of required lemma

```
/*@  
lemma linked_split_segment:  
  \forall struct list *root, **cArr, *bound, *AddrC, integer i, n, k;  
    n > 0 ==> k >= 0 ==>  
      AddrC == cArr[i + n - 1]->next ==>  
        linked_n(root, cArr, i, n + k, bound)  
          (linked_n(root, cArr, i, n, AddrC) &&  
            linked_n(AddrC, cArr, i + n, k, bound));  
*/
```

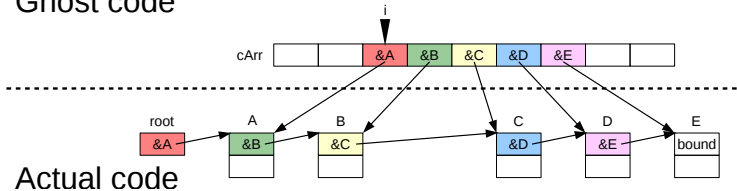
Example of required lemma

```

/*@
lemma linked_split_segment:
  \forall struct list *root, **cArr, *bound, *AddrC, integer i, n, k;
  n > 0 ==> k >= 0 ==>
  AddrC == cArr[i + n - 1]->next ==>
  linked_n(root, cArr, i, n + k, bound)
  (linked_n(root, cArr, i, n, AddrC) &&
   linked_n(AddrC, cArr, i + n, k, bound));
*/

```

Ghost code



Actual code

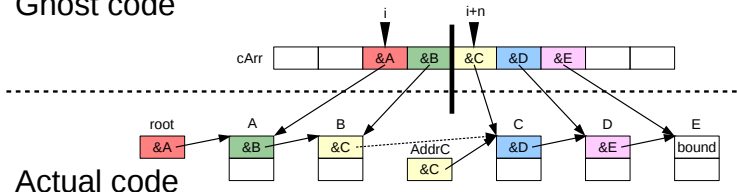
Example of required lemma

```

/*@
lemma linked_split_segment:
  \forall struct list *root, **cArr, *bound, *AddrC, integer i, n, k;
  n > 0 ==> k >= 0 ==>
  AddrC == cArr[i + n - 1]->next ==>
  linked_n(root, cArr, i, n + k, bound)
  (linked_n(root, cArr, i, n, AddrC) &&
   linked_n(AddrC, cArr, i + n, k, bound));
*/

```

Ghost code



Actual code

Results

- Written specification and ghost code
 - > 46 lines for ghost functions
 - > 500 lines for contracts
 - > 240 lines for logic definitions and lemmas
 - > 650 lines of other annotations
- It generates 798 proof obligations
 - > 772 (96.7%) are automatically discharged by SMT solvers
 - > 24 are lemmas proved with Coq
 - > 2 assertions proved with Coq
 - > 2 assertions proved using TIP
- Discharging all PO requires about an hour of computation.
- We have built some tests to check the specification
 - > some bugs in the specification were discovered and fixed
 - > the handling of assigns could be improved

Results

- Written specification and ghost code
 - > 46 lines for ghost functions
 - > 500 lines for contracts
 - > 240 lines for logic definitions and lemmas
 - > 650 lines of other annotations
- It generates 798 proof obligations
 - > 772 (96.7%) are automatically discharged by SMT solvers
 - > 24 are lemmas proved with Coq
 - > 2 assertions proved with Coq
 - > 2 assertions proved using TIP
- Discharging all PO requires about an hour of computation.
- We have built some tests to check the specification
 - > some bugs in the specification were discovered and fixed
 - > the handling of assigns could be improved

Results

- Written specification and ghost code
 - > 46 lines for ghost functions
 - > 500 lines for contracts
 - > 240 lines for logic definitions and lemmas
 - > 650 lines of other annotations
- It generates 798 proof obligations
 - > 772 (96.7%) are automatically discharged by SMT solvers
 - > 24 are lemmas proved with Coq
 - > 2 assertions proved with Coq
 - > 2 assertions proved using TIP
- Discharging all PO requires about an hour of computation.
- We have built some tests to check the specification
 - > some bugs in the specification were discovered and fixed
 - > the handling of assigns could be improved

Results

- Written specification and ghost code
 - > 46 lines for ghost functions
 - > 500 lines for contracts
 - > 240 lines for logic definitions and lemmas
 - > 650 lines of other annotations
- It generates 798 proof obligations
 - > 772 (96.7%) are automatically discharged by SMT solvers
 - > 24 are lemmas proved with Coq
 - > 2 assertions proved with Coq
 - > 2 assertions proved using TIP
- Discharging all PO requires about an hour of computation.
- We have built some tests to check the specification
 - > some bugs in the specification were discovered and fixed
 - > the handling of assigns could be improved

Bug found in `list_insert`

List: `list_insert` bug #254



simonduq opened this issue on 15 Dec 2017 · 4 comments



simonduq commented on 15 Dec 2017 · edited ▾

Owner



The function `list_insert` in `list.c` is buggy: when `previtem` is null, it pushes the new element (which (1) removes any old instance and then (2) inserts the new element). But when `previtem` is non-null, it just adds the new item without removing any old instance. Could in duplicate elements in the latter case.

Only reporting as `bug/low` because the function is currently not used in the codebase.

(report by Nikolai Kosmatov)

Bug found in `list_insert`

List: `list_insert` bug #254



simonduq opened this issue on 15 Dec 2017 · 4 comments



simonduq commented on 15 Dec 2017 · edited ▾

Owner



The function `list_insert` in `list.c` is buggy: when `previtem` is null, it pushes the new element (which (1) removes any old instance and then (2) inserts the new element). But when `previtem` is non-null, it just adds the new item without removing any old instance. Could in duplicate elements in the latter case.

Only reporting as `bug/low` because the function is currently not used in the codebase.

(report by Nikolai Kosmatov)



g-oikonomou commented on 16 Dec 2017 · edited ▾

Owner



For the record, things are actually worse than having the same element in the list twice: This bug will corrupt the list.

03

Conclusion & Future work

Let's sum up!

We presented the verification of the list module of Contiki

Verification

- based on a companion ghost array that tracks the status of the list
- comprises a set of lemmas that reduces the need for interactive proof
- still needs quite a lot of auxiliary annotations
- allowed us to find and fix a bug in the module

Ongoing & Future work

The verification of the list module can be improved

- assigns clauses are not precise enough
 - > there are things to do inside the WP plugin
- separation was really hard to handle
 - > (Ongoing) we should not have to deal with ghost separation
- one would prefer a more abstract specification
 - > (Ongoing) using ACSL logic lists
 - > (Future) using contiguous partial functions
- while based on arrays, the specification is not executable
 - > Provably equivalent executable proof-of-concept version (accepted to TAP 2018)

Ongoing & Future work

The verification of the list module can be improved

- assigns clauses are not precise enough
 - > there are things to do inside the WP plugin
- separation was really hard to handle
 - > (Ongoing) we should not have to deal with ghost separation
- one would prefer a more abstract specification
 - > (Ongoing) using ACSL logic lists
 - > (Future) using contiguous partial functions
- while based on arrays, the specification is not executable
 - > Provably equivalent executable proof-of-concept version (accepted to TAP 2018)

Ongoing & Future work

The verification of the list module can be improved

- assigns clauses are not precise enough
 - > there are things to do inside the WP plugin
- separation was really hard to handle
 - > (Ongoing) we should not have to deal with ghost separation
- one would prefer a more abstract specification
 - > (Ongoing) using ACSL logic lists
 - > (Future) using contiguous partial functions
- while based on arrays, the specification is not executable
 - > Provably equivalent executable proof-of-concept version (accepted to TAP 2018)

Ongoing & Future work

The verification of the list module can be improved

- assigns clauses are not precise enough
 - > there are things to do inside the WP plugin
- separation was really hard to handle
 - > (Ongoing) we should not have to deal with ghost separation
- one would prefer a more abstract specification
 - > (Ongoing) using ACSL logic lists
 - > (Future) using contiguous partial functions
- while based on arrays, the specification is not executable
 - > Provably equivalent executable proof-of-concept version (accepted to TAP 2018)

Ongoing & Future work

The verification of the list module can be improved

- assigns clauses are not precise enough
 - > there are things to do inside the WP plugin
- separation was really hard to handle
 - > (Ongoing) we should not have to deal with ghost separation
- one would prefer a more abstract specification
 - > (Ongoing) using ACSL logic lists
 - > (Future) using contiguous partial functions
- while based on arrays, the specification is not executable
 - > Provably equivalent executable proof-of-concept version (accepted to TAP 2018)

Ongoing & Future work

The verification of the list module can be improved

- assigns clauses are not precise enough
 - > there are things to do inside the WP plugin
- separation was really hard to handle
 - > (Ongoing) we should not have to deal with ghost separation
- one would prefer a more abstract specification
 - > (Ongoing) using ACSL logic lists
 - > (Future) using contiguous partial functions
- while based on arrays, the specification is not executable
 - > Provably equivalent executable proof-of-concept version (accepted to TAP 2018)

Thank you!

Thank you!