Towards Formal Verification of Contiki OS with Frama-C

Allan Blanchard Nikolai Kosmatov

joint work with Simon Duquennoy, Frédéric Loulergue, Frédéric Mangano, Alexandre Peyrard, Shahid Raza, . . .

CEA List, March 6th, 2018

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
- ...

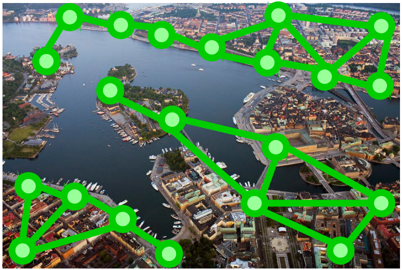
Typical hardware platform:

- 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.

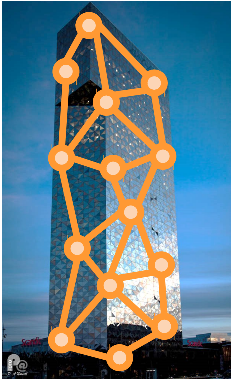
Contiki: Typical Applications

- IoT scenarios: smart cities, building automation, ...
- Multiple hops to cover large areas
- Low-power for battery-powered scenarios
- Nodes are interoperable and addressable (IP)



Traffic lights
Parking spots
Public transport
Street lights
Smart metering
...

Light bulbs
Thermostat
Power sockets
CO2 sensors
Door locks
Smoke detectors
...



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

Verification goals

For low-level library/system code: ideally functional verification

- highly critical code
- frequently used (memory, lists, ...)

For the netstack: absence of runtime errors

- using Eva
- using minimal contracts and WP if Eva cannot prove
- using runtime verification if WP cannot prove either

Verification goals

For low-level library/system code: ideally functional verification

- highly critical code
- frequently used (memory, lists, ...)

For the netstack: absence of runtime errors

- using Eva
- using minimal contracts and WP if Eva cannot prove
- using runtime verification if WP cannot prove either

Frama-C at a glance

Software Analyzers

- A Framework for Modular Analysis 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

- 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 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

Overview of the `memb` Module

- No dynamic allocation in Contiki
  - to avoid fragmentation of memory in long-lasting systems
- Memory is pre-allocated (in arrays of blocks) and attributed on demand
- The management of such blocks is realized by the `memb` module

The `memb` module API allows the user to

- initialize a `memb` store (i.e. pre-allocate an array of blocks),
- allocate or free a block,
- check if a pointer refers to a block inside the store
- count the number of allocated blocks

memb is critical!

- Contiki's main memory allocation module
- about 100 lines of critical code
- kernel and many modules rely on `memb`
  - used for HTTP, CoAP (lightweight HTTP), IPv6 routes, CSMA, the MAC protocol TSCH, packet queues, network neighbors, the file system Coffee or the DBMS Antelope

- `memb` is one of the most critical elements of Contiki

A flaw in `memb` could result in attackers reading or writing arbitrary memory regions, crashing the device, or triggering code execution

The memb Store

- An array of blocks with a given block size and number of blocks
- Defined by an instance of `struct memb`
- Created by a macro for a given block type and number of blocks
  - since there is no polymorphism in C
  - blocks are manipulated as `void*` pointers
- Refers to global definitions added by preprocessing

```c
/* file memb.h */
struct memb {
  unsigned short size; // block size
  unsigned short num;  // number of blocks
  char *count;         // block statuses
  void *mem;           // array of blocks
};
#define MEMB(name, btype, num)...
// macro used to decrare a memb store for
// allocation of num blocks of type btype

void  memb_init(struct memb *m);
void *memb_alloc(struct memb *m);
char  memb_free(struct memb *m, void *p);
...
```

```c
/* file demo.c */
#include "memb.h"
struct point {int x; int y};

// before preprocessing,
// there was the following macro:
// MEMB(pblock, struct point, 2);

// after preprocessing, it becomes:
static char pblock_count[2];
static struct point pblock_mem[2];
struct struct memb pblock = {
  sizeof(struct point), 2,
  pblock_count, pblock_mem };
...
```

Contract of the Allocation Function memb_alloc

```
/*@
  requires valid_memb(m);
  ensures valid_memb(m);
  assigns m→count[0 .. (m→num - 1)];
  behavior free_found:
    assumes ∃ ℤ i; 0 ≤ i < m→num ∧ m→count[i] == 0;
    ensures ∃ ℤ i; 0 ≤ i < m→num ∧ \old(m→count[i]) == 0 ∧ m→count[i] == 1 ∧
      \result == (char*) m→mem + (i * m→size) ∧
      ∀ ℤ j; (0 ≤ j < i ∨ i < j < m→num) ⟹ m→count[j] == \old(m→count[j]);
    ensures \valid((char*) \result + (0 .. (m→size - 1)));
    ensures _memb_numfree(m) == \old(_memb_numfree(m)) - 1;
  behavior full:
    assumes ∀ ℤ i; 0 ≤ i < m→num ⟹ m→count[i] ≠ 0;
    ensures ∀ ℤ i; 0 ≤ i < m→num ⟹ m→count[i] == \old(m→count[i]);
    ensures \result == NULL;
  complete behaviors;
  disjoint behaviors;
*/
void *memb_alloc(struct memb *m);
```

Proven
in Frama-C/WP

Specification in ACSL

We specify the contract of each function and prove it in Frama-C

For instance, the contract of `memb_alloc` has two behaviors

1. If the store is full, then leave it intact and return NULL (lines 12-15)
2. If the store has a free block, then return a free block *b* such that:
   - *b* is properly aligned in the block array (line 8)
   - *b* was marked as free, and is now marked as allocated (line 7)
   - *b* is valid, i.e. points to a valid memory space of a block size that can be safely read or written to (line 10)
   - the states of the other blocks have not changed (line 9)
   - the number of free blocks is decremented (line 11)

These behaviors are disjoint and complete.

Summary

- The `memb` module specified and formally verified with Frama-C/WP
  - 115 lines of annotations
  - 32 additional assertions
  - 126 verification conditions (i.e. proven properties)
- A few client functions proven as expected
  - Proof fails for out-of-bounds access attempts
- A potentially harmful situation detected
  - `count--;` used instead of `count=0;`

Reference: F.Mangano, S.Duquennoy and N.Kosmatov.
A Memory Allocation Module of Contiki Formally Verified with Frama-C. A Case Study. In CRiSIS 2016, LNCS, vol.10158, 114–120. Springer.

Overview of the `aes-ccm` Modules

- Critical! – Used for communication security
  - end-to-end confidentiality and integrity (e.g. Link-layer security or DTLS)
- Advanced Encryption Standard (AES) is a symmetric encryption algorithm
  - AES replaced in 2002 Data Encryption Standard (DES), which became obsolete in 2005
- Modular API – independent from the OS
- Two modules:
  - AES-128
  - AES-CCM* block cypher mode
  - A few hundreds of LoC
- High complexity crypto code
  - Intensive integer arithmetics
  - Intricate indexing
  - based on multiplication over finite field $\mathrm{GF}(2^8)$

Example: Function set_key

```
/*@ requires \valid_read(key+ (0 .. (AES_128_KEY_LENGTH - 1)));
    assigns round_keys[0][0 .. (AES_128_KEY_LENGTH - 1)], round_keys[1][0 .. (AES_128_KEY_LENGTH - 1)],
    round_keys[2][0 .. (AES_128_KEY_LENGTH - 1)], round_keys[3][0 .. (AES_128_KEY_LENGTH - 1)],
    round_keys[4][0 .. (AES_128_KEY_LENGTH - 1)], round_keys[5][0 .. (AES_128_KEY_LENGTH - 1)],
    round_keys[6][0 .. (AES_128_KEY_LENGTH - 1)], round_keys[7][0 .. (AES_128_KEY_LENGTH - 1)],
    round_keys[8][0 .. (AES_128_KEY_LENGTH - 1)], round_keys[9][0 .. (AES_128_KEY_LENGTH - 1)],
    round_keys[10][0 .. (AES_128_KEY_LENGTH - 1)];
*/
static void
set_key(const uint8_t *key)
{
    uint8_t i;
    uint8_t j;
    uint8_t rcon;
    rcon = 0x01;
    /*@ loop invariant 0 <= i <= AES_128_KEY_LENGTH;
        loop assigns i, round_keys[0][0 .. (AES_128_KEY_LENGTH - 1)];
        loop variant AES_128_KEY_LENGTH - i;
    */
    for(i = 0; i < AES_128_KEY_LENGTH; i++) {
        round_keys[0][i] = key[i];
    }
    /*@ loop invariant 1 <= i <= 11;
        loop assigns i, rcon, j, round_keys[1][0 .. (AES_128_KEY_LENGTH - 1)],
        round_keys[2][0 .. (AES_128_KEY_LENGTH - 1)], round_keys[3][0 .. (AES_128_KEY_LENGTH - 1)],
        round_keys[4][0 .. (AES_128_KEY_LENGTH - 1)], round_keys[5][0 .. (AES_128_KEY_LENGTH - 1)],
        round_keys[6][0 .. (AES_128_KEY_LENGTH - 1)], round_keys[7][0 .. (AES_128_KEY_LENGTH - 1)],
        round_keys[8][0 .. (AES_128_KEY_LENGTH - 1)], round_keys[9][0 .. (AES_128_KEY_LENGTH - 1)],
        round_keys[10][0 .. (AES_128_KEY_LENGTH - 1)];
        loop variant 11 - i;
    */
    for(i = 1; i <= 10; i++) {
        round_keys[i][0] = sbox[round_keys[i - 1][13]] ^ round_keys[i - 1][0] ^ rcon;
        round_keys[i][1] = sbox[round_keys[i - 1][14]] ^ round_keys[i - 1][1];
        round_keys[i][2] = sbox[round_keys[i - 1][15]] ^ round_keys[i - 1][2];
        round_keys[i][3] = sbox[round_keys[i - 1][12]] ^ round_keys[i - 1][3];
        /*@ loop invariant 4 <= j <=
        AES_128_BLOCK_SIZE;
            loop assigns j, round_keys[i][4 ..
        (AES_128_KEY_LENGTH - 1)];
            loop variant 16 - j;
        */
        for(j = 4; j < AES_128_BLOCK_SIZE; j++) {
            round_keys[i][j] = round_keys[i - 1][j] ^ round_keys[i][j - 4];
        }
        rcon = galois_mul2(rcon);
    }
}
```

Specification and Verification with Frama-C/WP

- Specification of minimal contracts of each function
  - ∼300 lines of C code
  - ∼100 lines of ACSL spec
  - 467 proof obligations (proved within ∼50 sec.)
- Proof of absence of RTE with Frama-C/WP
- Validation of contracts of a test file
  - to get confidence that the contracts are OK

Reference: A.Peyrard, N.Kosmatov, S.Duquennoy, S.Raza. *Towards Formal Verification of Contiki OS: Analysis of the AES-CCM* Modules with Frama-C.* In RED-IoT 2018, part of EWSN 2018, ACM. To appear.

Introduction

The MEMB Module

The AES-CCM Module

The LIST module

- Overview of the `list` module
- Formalization approach
- Examples
- Results

Conclusion

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;

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

The LIST module - A rich API

Observers

Update list beginning

```
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;

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

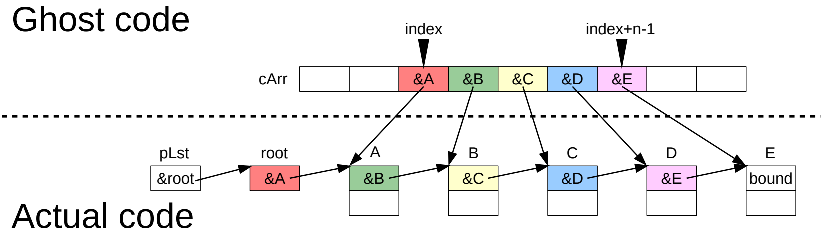
The LIST Module - User code

```
LIST(route_table);

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

//...

struct uaodv_rt_entry *e;
e = uaodv_rt_lookup_any(dest);
list_push(route_table, e);
```

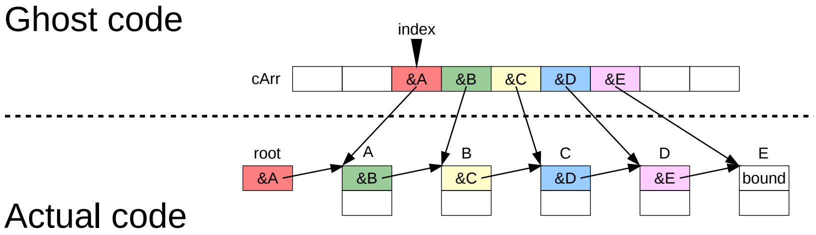
- based on type erasure
- violates the strict aliasing

Formalization approach - Overview

We maintain a ghost array that stores the addresses of the cells.

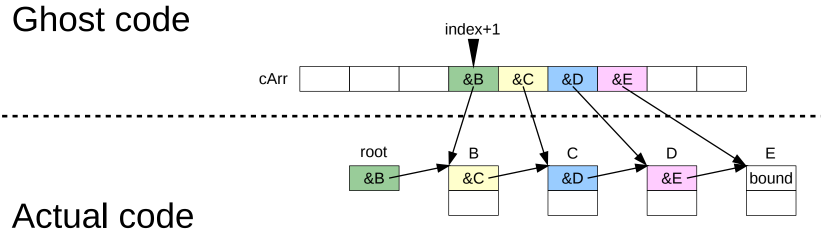


Formalization approach - Induction



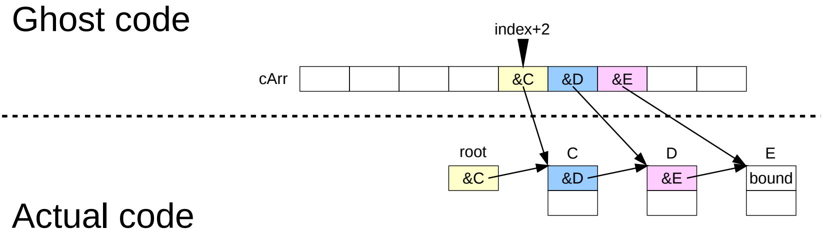

```
inductive linked_n{L}(struct list *root, struct list **cArr,
                      integer index, integer n, struct list *bound) {
// ...
case linked_n_cons{L}:
  \forall struct list *root, **cArr, *bound, integer index, n;
    /*indexes properties*/ ==> \valid(root) ==> root == cArr[index] ==>
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

Formalization approach - Induction



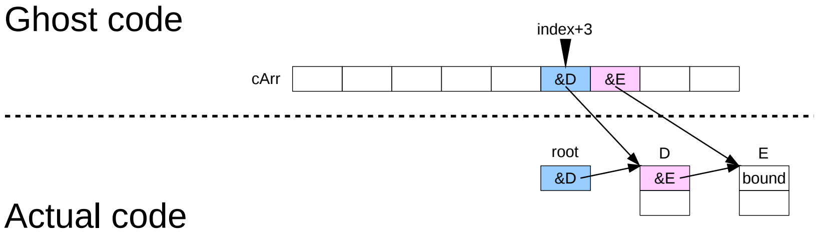

```
inductive linked_n{L}(struct list *root, struct list **cArr,
                      integer index, integer n, struct list *bound) {
// ...
case linked_n_cons{L}:
  \forall struct list *root, **cArr, *bound, integer index, n;
    /*indexes properties*/ ==> \valid(root) ==> root == cArr[index] ==>
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

Formalization approach - Induction



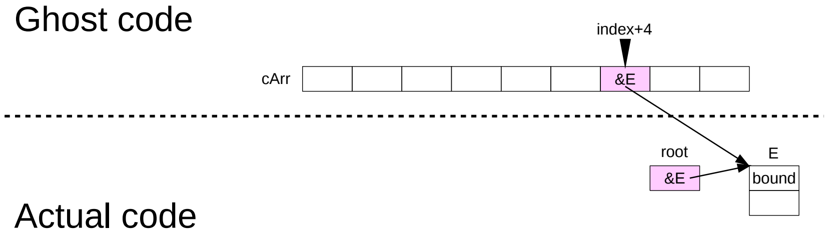

```
inductive linked_n{L}(struct list *root, struct list **cArr,
                      integer index, integer n, struct list *bound) {
// ...
case linked_n_cons{L}:
  \forall struct list *root, **cArr, *bound, integer index, n;
    /*indexes properties*/ ==> \valid(root) ==> root == cArr[index] ==>
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

Formalization approach - Induction




```
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);
}
```

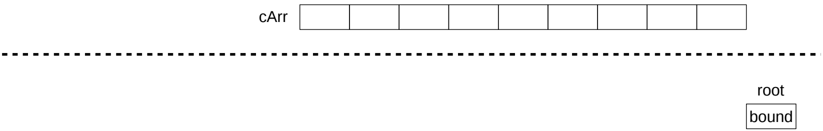
Formalization approach - Induction




```
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 base case

Ghost code



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:

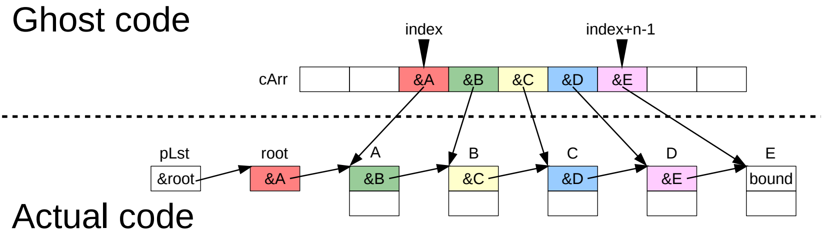
```
predicate unchanged{L1, L2}(struct list **array, int index, int size) =
  \forall integer i ; index <= i < index+size ==>
    \at(array[i]->next, L1) == \at(array[i]->next, L2);
```

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

Example with list_pop

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

Current property:

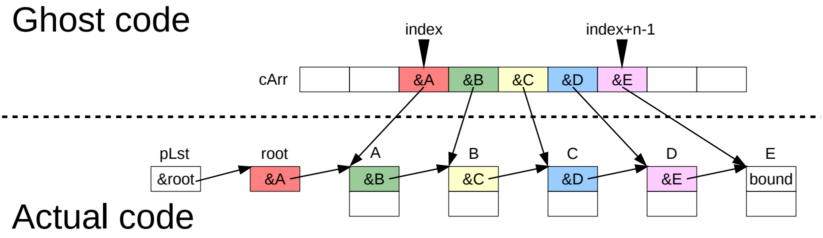


Example with list_pop

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

Current property:

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

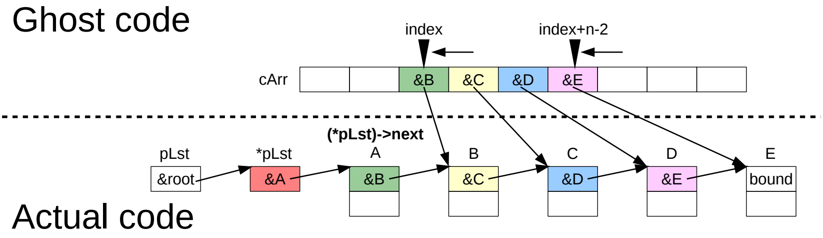


Example with list_pop

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

Current property:

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

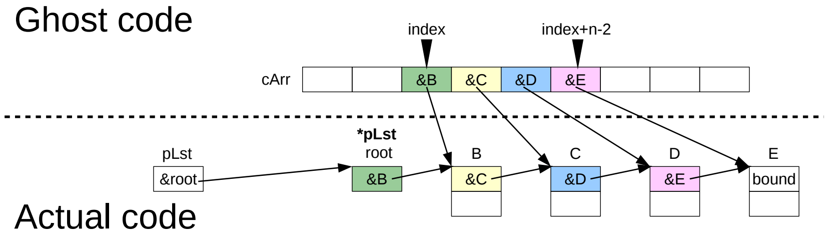


Example with list_pop

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

Current property:

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

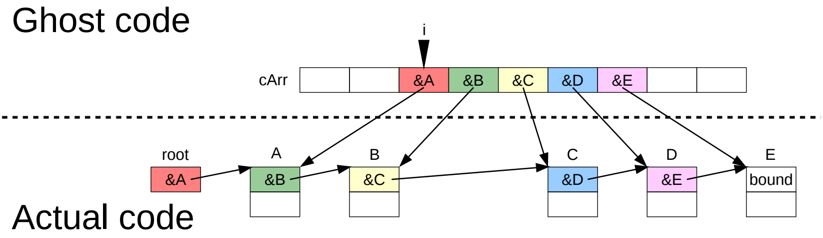


Example of required lemma

```
/*@
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));
*/
```

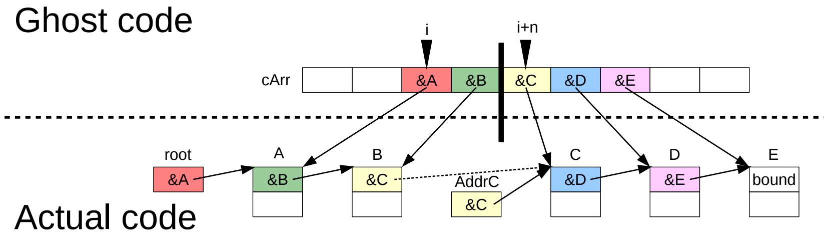
Example of required lemma

```
/*@
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));
*/
```



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));
*/
```



Verification

- Code written and specification
  - 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 are automatically discharged by SMT solvers
  - 24 are lemmas proved with Coq
  - 2 assertions proved with Coq
  - 2 assertions proved using TIP

Results

- 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

Reference: A.Blanchard, N.Kosmatov and F.Loulergue.
Ghosts for Lists: A Critical Module of Contiki Verified in Frama-C. In NFM 2018, LNCS. Springer (to appear).

Bug found in `list_insert`

List: list_insert bug #254

Closed **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

Closed 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.

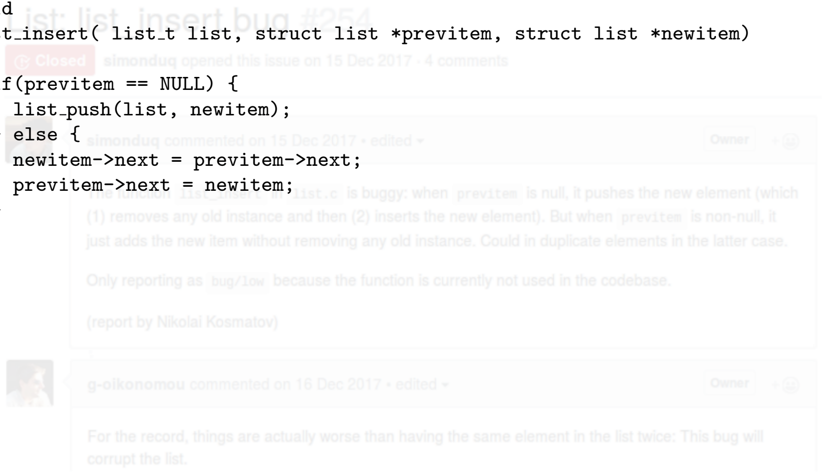
(report by Nikolai Kosmatov)

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.

Bug found in list_insert

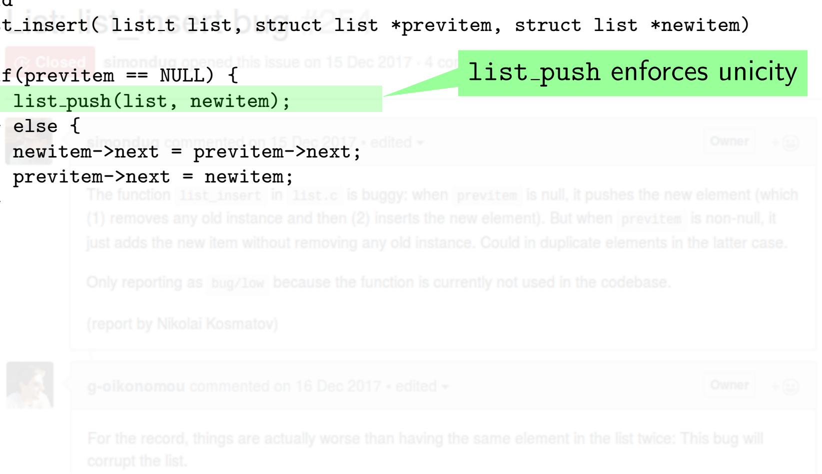
```
void
list_insert( list_t list, struct list *previtem, struct list *newitem)
{
  if(previtem == NULL) {
    list_push(list, newitem);
  } else {
    newitem->next = previtem->next;
    previtem->next = newitem;
  }
}
```



Bug found in list_insert

```
void
list_insert( list_t list, struct list *previtem, struct list *newitem)
{
  if(previtem == NULL) {
    list_push(list, newitem);
  } else {
    newitem->next = previtem->next;
    previtem->next = newitem;
  }
}
```

list_push enforces unicity



Bug found in `list_insert`

```
void
list_insert(list_t list, struct list *previtem, struct list *newitem)
{
  if(previtem == NULL) {
    list_push(list, newitem);
  } else {
    newitem->next = previtem->next;
    previtem->next = newitem;
  }
}
```

`list_push` enforces unicity

This code does not care about it

Bug found in `list_insert`

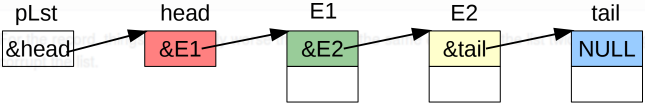
```
void
list_insert(list_t list, struct list *previtem, struct list *newitem)
{
  if(previtem == NULL) {
    list_push(list, newitem);
  } else {
    newitem->next = previtem->next;
    previtem->next = newitem;
  }
}
```

`list_push` enforces unicity

This code does not care about it

Consequence:

```
list_insert(list, &tail, &e1);
```



Bug found in `list_insert`

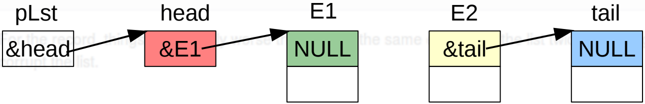
```
void
list_insert( list_t list, struct list *previtem, struct list *newitem)
{
  if(previtem == NULL) {
    list_push(list, newitem);
  } else {
    newitem->next = previtem->next;
    previtem->next = newitem;
  }
}
```

list_push enforces unicity

This code does not care about it

Consequence:

```
list_insert(list, &tail, &e1);
```



Bug found in `list_insert`

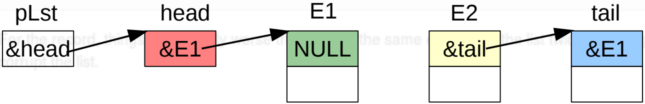
```
void
list_insert( list_t list, struct list *previtem, struct list *newitem)
{
  if(previtem == NULL) {
    list_push(list, newitem);
  } else {
    newitem->next = previtem->next;
    previtem->next = newitem;
  }
}
```

`list_push` enforces unicity

This code does not care about it

Consequence:

```
list_insert(list, &tail, &e1);
```



Bug found in `list_insert`

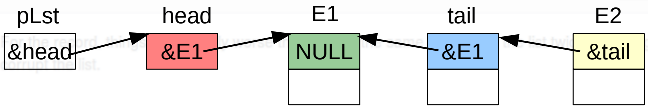
```
void
list_insert(list_t list, struct list *previtem, struct list *newitem)
{
  if(previtem == NULL) {
    list_push(list, newitem);
  } else {
    newitem->next = previtem->next;
    previtem->next = newitem;
  }
}
```

`list_push` enforces unicity

This code does not care about it

Consequence:

```
list_insert(list, &tail, &e1);
```



Bug found in `list_insert`

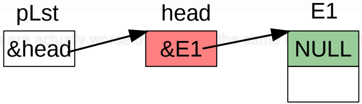
```
void
list_insert(list_t list, struct list *previtem, struct list *newitem)
{
  if(previtem == NULL) {
    list_push(list, newitem);
  } else {
    newitem->next = previtem->next;
    previtem->next = newitem;
  }
}
```

`list_push` enforces unicity

This code does not care about it

Consequence:

```
list_insert(list, &tail, &e1);
```



Let's sum up!

Being an OS for vulnerable connected devices, Contiki is target of choice for verification.

Verified modules presented today:

- MEMB module
- AES-CCM and CCM* functions
- LIST module

Other known verification effort using WP:

- sicslowpan module (Quentin Molle)
- Absence of RtE in some core components
  (Jens Gerlach and Jochen Burghardt from Fraunhofer)

Ongoing work - with WP

- The verification of the list module can be improved
  - assigns are not precise enough (there are things to do in WP),
  - separation was really hard to handle
  - one would prefer a more abstract specification
- Ongoing experimentations
  - using ACSL logic lists
  - using contiguous partial functions

Future work - with EVA

- We want to use EVA for a verification of absence of RtE:
  - with a focus on the portable code
  - in order to find the right parameters to set for EVA+Contiki
- Provide an infrastructure to ease futur verification on Contiki:
  - an abstract "*hardware*" platform for Contiki
  - a way to ease configuration generation for verification
  - integrate Frama-C to Travis ?

Future work - with EVA

- We want to use EVA for a verification of absence of RtE:
  - with a focus on the portable code
  - in order to find the right parameters to set for EVA+Contiki
- Provide an infrastructure to ease futur verification on Contiki:
  - an abstract "*hardware*" platform for Contiki
  - a way to ease configuration generation for verification
  - integrate Frama-C to Travis ?

Thank you for your attention! Questions?

Contiki instance

Main problems for verification:

- an instance of Contiki is a fraction of its source code
- no unit tests independent from a full OS instance

Building an instance:

- **include** basic features
- **define** parameters for some features and/or the platform
- define tasks as new processes (eventually including libs)
- choose a platform (whose source code contains the `main`)

Integrated Frama-C scripts

Verification scripts and Makefile configurations

Integrated to the Contiki `Makefile.include`:

- $+$ **1** OS configuration for verification
- $+$ some configuration for Frama-C and Eva

Some configuration are directly made in the source code

First results with Eva

We can run the Frama-C Eva analysis on examples from Contiki

- with very few more configuration to add to an example

Good news:

- we consider a realistic platform
- the analysis is fast ...

Bad news:

- ... because of a degeneration caused by a recursive function
- there is a lot of alarms

Scheduling of tasks

Processes are handled through a macro library called protothread

- it looks like co-routines

The scheduling algorithm

- maintains a list of “processes”
- propagates events to other processes when needed
- ... by recursive calls

Protothread I

Protothreads uses Duff's device to handle yielding

Example

```
static
PT_THREAD(example(struct pt *pt))
{
  PT_BEGIN(pt);

  while(1) {
    PT_WAIT_UNTIL(pt,
      counter == 1000);
    printf("Threshold reached\n");
    counter = 0;
  }

  PT_END(pt);
}
```

Protothread II

... which generates a code like this

```
static
char example(struct pt *pt)
{
  switch(pt->lc) { case 0:

  while(1) {
    pt->lc = 12; case 12:
    if(!(counter == 1000)) return 0;
    printf("Threshold reached\n");
    counter = 0;
  }

  } pt->lc = 0; return 2;
}
```

Event management

Most events are immediately handled by the scheduler

- by a recursively call to `call_process`
- that can again generates new events

This recursive calls cause the Eva analysis to degenerate

Options:

- change the scheduling algorithm
- improve Eva with recursive function inlining