Towards Formal Verification of IoT Operating Systems with Frama-C

Nikolai Kosmatov

joint work with Allan Blanchard, Simon Duquennoy, Frédéric Loulergue, Frédéric Mangano, Alexandre Peyrard, Shahid Raza, ...
Internet of Things

- connect devices and services
- 22 billion IoT devices by 2025
- transport huge amounts of data

(c) Internet Security Buzz
And Security?
And Security?

Mirai botnet, a DDoS nightmare turning Internet of Things into Botnet of things
And Security?

Mirai botnet, a DDoS nightmare turning Internet of Things into Botnet of things

Hackers remotely kill a Jeep on the highway—with me in it
And Security?

Mirai botnet, a DDoS nightmare turning Internet of Things into Botnet of things

Hackers remotely kill a Jeep on the highway—with me in it

Pacemaker hacking fears rise with critical research report
And Security?

Mirai botnet, a DDoS nightmare turning Internet of Things into Botnet of things

Hacking a computer-aided sniper rifle

PACEMAKER HACKING FEARS RISE WITH CRITICAL RESEARCH REPORT
### Formal Methods Today

- **Improves software quality** in 92% of projects  
  Source: Formal Methods Practice and Experiments, ACM Comp.Surveys

- **More efficient** in practice: faster hardware, more memory, more mature verification tools...

- Finding a proof can require **significant effort and higher expertise**
Formal Verification and the Internet of Things

Formal verification

- can eliminate many **exploitable vulnerabilities** today
  - exploit kits leverage software errors e.g. buffer overflow, missing bounds checks, integer overflow, invalid array access, memory corruption, . . .
- **traditionally** applied to embedded software in many critical domains
  - avionics, energy, rail, . . .
- **rarely** applied to IoT software

This talk

- promotes the usage of formal verification for IoT software
- illustrates it for an IoT operating system Contiki
Contiki: A lightweight OS 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, ...

Any invalid memory access can be dangerous: 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)
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 value analysis (Frama-C/Eva)
- using minimal contracts and deductive verification (Frama-C/WP) if Eva cannot prove
- using runtime verification if WP cannot prove either
Overview of Frama-C

Outline

Introduction

Overview of Frama-C

Cryptography Module AES-CCM

Memory Allocation Module MEMB

Linked List Module LIST

Conclusion
Frama-C Open-Source Distribution

Framework for Analysis of source code written in C

- analysis of C code extended with ACSL annotations
- ACSL Specification Language
  - *langua franca* of Frama-C analyzers
- http://frama-c.com
- targets both academic and industrial usage
Overview of Frama-C

Frama-C, a Collection of Tools

Several tools inside a single platform

- plugin architecture like in Eclipse
- tools provided as plugins
  - over 20 plugins in the open-source distribution
  - close-source plugins, either at CEA (about 20) or outside
- a common kernel
  - provides a uniform setting
  - provides general services
  - synthesizes useful information
Overview of Frama-C

Frama-C Plugin Gallery

Abstract Interpretation
- Eva

Deductive Verification
- Jessie
- Wp

Formal Methods
- Slicing
- Sparecode
- Clang
- Semantic constant folding

Code Transformation

Plugins

Specification Generation
- Aorai
- RTE

Dynamic Analysis
- PathCrawler
- E-ACSL
- StaDy
- Ltest
- Sante

Browsing of unfamiliar code
- Metrics
- Impact
- Occurrence
- Callgraph
- Scope & Data-flow browsing

Browsing of unfamiliar code

Impact

Occurrence

Scope & Data-flow browsing

Metrics

Nikolai Kosmatov Towards Formal Verification of IoT OS with Frama-C December 12, 2019 13 / 43
Plugin Frama-C/Eva for Value Analysis

Compute possible values of variables at each program point

- an automatic analysis
- based on abstract interpretation
- reports alarms for potentially invalid operations
- can prove the absence of runtime errors
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**
Example: a C Program Annotated in ACSL

```c
/*@ requires n>=0 && $valid(t+(0..n-1));
   assigns $nothing;
   ensures $result !=$ 0 <=>
      ($forall integer j; 0 <= j < n ==> t[j] == 0);
*/
int all_zeros(int t[], int n) {
    int k;
    /*@ loop invariant 0 <= k <= n;
       loop invariant $forall integer j; 0<=j<k ==> t[j]==0;
       loop assigns k;
       loop variant n-k;
    */
    for (k = 0; k < n; k++)
        if (t[k] != 0)
            return 0;
    return 1;
}
```

Can be proven with Frama-C/WP
Introduction

Overview of Frama-C

Cryptography Module AES-CCM

Overview of the aes-ccm Modules

Verification of aes-ccm with Frama-C/Eva

Verification of aes-ccm with Frama-C/WP

Memory Allocation Module MEMB

Linked List Module LIST

Conclusion
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 cipher mode
  - A few hundreds of LoC
- **High complexity crypto code**
  - Intensive integer arithmetics
  - Intricate indexing
  - based on multiplication over finite field $\text{GF}(2^8)$
Example: Function `set_key`

```c
static void set_key(const uint8_t *key)
{
    uint8_t i;
    uint8_t j;
    uint8_t rcon;

    rcon = 0x01;
    memcpy(round_keys[0], key, AES_128_KEY_LENGTH);
    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];
        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);
    }
}
```
Verification of aes-ccm with Frama-C/Eva

- **Proof of absence of runtime errors** (and security vulnerabilities) for all possible cases

- Example for AES: we run Eva on the most general context built for AES:

```c
int main () {
    uint8_t key[16];
    uint8_t data[16];
    int i;
    for (i = 0; i < 16; i++) {
        key[i] = Frama_C_interval(0, 255);
        data[i] = Frama_C_interval(0, 255);
    }
    aes_128_set_key(key);
    aes_128_encrypt(data);
}
```

- If Eva does not prove, one can use WP with minimal contracts
Example: Function set_key

```c
static void set_key(const uint8_t *key)
{
    uint8_t i;
    uint8_t j;
    uint8_t rcon;
    rcon = 0x01;
    for(i = 0; i < AES_128_KEY_LENGTH; i++) {
        round_keys[0][i] = key[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];
        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

Introduction

Overview of Frama-C

Cryptography Module AES-CCM

Memory Allocation Module MEMB
  Overview of the memb Module
  Pre-Allocation of a Store in memb
  Verification of memb with Frama-C/WP

Linked List Module LIST

Conclusion
Overview of the \texttt{memb} Module

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

The \texttt{memb} module API allows the user to

- initialize a \texttt{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
    ... an initialized memb store.
}

 Lines 10–14 of Fig. 1b show the result of line 7 after the preprocessing.
```
Contract of the Allocation Function memb_alloc

```c
/*@ 
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 reported (and fixed)
  - `count--;` used instead of `count=0;`


Introduction

Overview of Frama-C

Cryptography Module AES-CCM

Memory Allocation Module MEMB

Linked List Module LIST

Overview of the list module
Formalization approach
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
  - maintains item unicity
The LIST module - A rich API

```c
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

```c
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

```c
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);
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);
```
Formalization approach

Maintain a companion ghost array that stores the addresses of list cells

Ghost code

Actual code

Define an inductive predicate linking the list and the array
Formalization approach - Base case

Ghost code

Actual code

```c
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 - 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 - Advantages

The companion array allows us to easily reason about the list contents:

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

We have to update the array (in ghost code) when the list is modified.
Example of required lemma: split a list into two segments

/*@ 
lemma linked_split_segment: 
\forall \text{struct list } \ast root, \ast\ast cArr, \ast bound, \ast\ast\ast AddrC, \text{integer } i, n, k; 
\quad n > 0 \implies k >= 0 \implies 
\quad AddrC == \text{cArr}[i + n - 1]->next \implies 
\quad \text{linked_n}(root, cArr, i, n + k, bound) \implies 
\quad (\text{linked_n}(root, cArr, i, n, AddrC) \&\& 
\quad \text{linked_n}(AddrC, cArr, i + n, k, bound)); 
*/
Example of required lemma: split a list into two segments

/*@  
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) \Rightarrow  
(linked_n(root, cArr, i, n, AddrC) &&  
linked_n(AddrC, cArr, i + n, k, bound));  
*/

Ghost code

Actual code
Verification Results

- 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

- Discharging all PO requires about an hour of computation.

Bug found in list_insert (now fixed)

List: list_insert bug #254

Closed simonduq opened this issue on 15 Dec 2017 • 4 comments

simonduq commented on 15 Dec 2017 • edited

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` (now fixed)

List: `list_insert` bug #254

Closed simonduq opened this issue on 15 Dec 2017 · 4 comments

simonduq commented on 15 Dec 2017 · edited

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

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

Introduction

Overview of Frama-C

Cryptography Module AES-CCM

Memory Allocation Module MEMB

Linked List Module LIST

Conclusion
Conclusion

Frama-C successfully used to formally verify several critical modules

▶ functional verification of memory allocation (MEMB)
▶ absence of security flaws in cryptography (AES-CCM and CCM*)
▶ functional verification of a key kernel module (LIST)
▶ other studies in progress

Absence of security related errors verified in all cases

End-to-end confidentiality and integrity (via AES-CCM)

Basic module for memory separation of various tasks

Several errors or incoherencies detected

New: Continuous integration framework for formal verification of Contiki is online [submitted]
Further reading

Tutorial papers:

- A. Blanchard, N. Kosmatov, and F. Loulergue. A Lesson on Verification of IoT Software with Frama-C (HPCS 2018)
- on deductive verification:
  N. Kosmatov, V. Prevosto, and J. Signoles. A lesson on proof of programs with Frama-C (TAP 2013)
- on runtime verification:
  - N. Kosmatov and J. Signoles. A lesson on runtime assertion checking with Frama-C (RV 2013)
  - N. Kosmatov and J. Signoles. Runtime assertion checking and its combinations with static and dynamic analyses (TAP 2014)
- on test generation:
  N. Kosmatov, N. Williams, B. Botella, M. Roger, and O. Chebaro. A lesson on structural testing with PathCrawler-online.com (TAP 2012)
- on analysis combinations:
Further reading

More details on the verification of Contiki:

▶ on the MEMB module:
  F. Mangano, S. Duquennoy, and N. Kosmatov. A memory allocation module of Contiki formally verified with Frama-C. A case study (CRiSIS 2016)

▶ on the AES-CCM* module:

▶ on the LIST module:
  ▶ F. Loulergue, A. Blanchard, and N. Kosmatov. Ghosts for lists: from axiomatic to executable specifications (TAP 2018)

*Best ACM Software Development paper award.*