Formal Verification of a Memory Allocation Module of Contiki with Frama-C: a Case Study

Nikolai Kosmatov

joint work with Simon Duquennoy and Frédéric Mangano

C&ESAR 2016, Rennes, November 22, 2016

The Internet of Things Software

The Internet of Things (IoT) devices

- increasingly popular, massively connected to the Internet
- increasingly critical: a compromised IoT device
  - may get access to sensitive or private data
  - may reconfigure an industrial automation process
  - may interfere with alarms or locks in a building
  - may alter a pacemaker or other vital devices
  - ...
- create new opportunities for attackers and new challenges for verification
  - Oct. 2016. Dyn DDoS Attack: Million Hacked IoT devices almost broke Internet

Formal Methods Today

- Improves software quality in 92% of projects

  Source: Formal Methods Practice and Experiments, ACM Comp.Surveys, Oct 2009
- 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 work

- promotes the usage of formal verification for IoT applications
- presents a case study on deductive verification of IoT software
  - for a memory allocation module of an IoT OS, Contiki

Outline

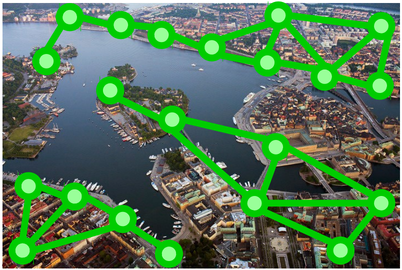
Outline

Contiki at a glance

- An Open Source OS for the Internet of Things, created in 2003
- More and more commercial products
- Open source: BSD
- C-based (+ protothreads)
- Supports many embedded platforms
- Supports standard low-power IPv6
- Includes Cooja simulator
- Web: `http://www.contiki-os.org/`
- Git: `https://github.com/contiki-os/contiki`

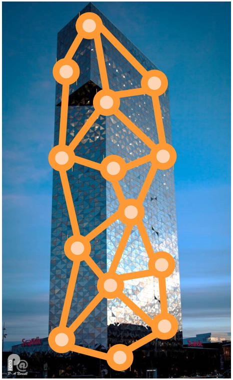
Contiki: Typical Applications

- 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

Outline

Frama-C at a glance

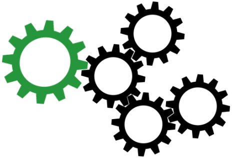
- A Framework for Modular Analysis of C code
- Developed at CEA List
- Released under LGPL license
- ACSL annotation language
- Extensible plugin oriented platform
  - Collaboration of analyses over same code
  - Inter plugin communication through ACSL formulas
  - Adding specialized plugins is easy
- 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 ⇒ Complete proof is required!
- `http://frama-c.com/acsl`

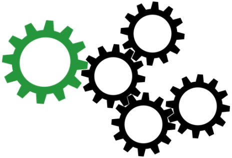
Deductive verification: What is the point?

- Testing seems sufficient for a correct program!

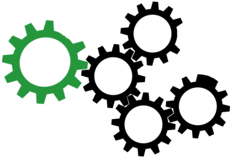


Deductive verification: What is the point?

- Testing seems sufficient for a correct program!

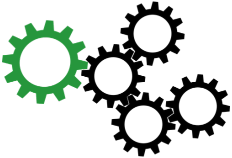


- And for an erroneous one?

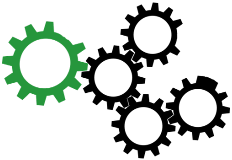


Deductive verification: What is the point?

- Testing seems sufficient for a correct program!



- And for an erroneous one?



- Specification and deductive verification help to find issues undetected by testing!

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

Let us illustrate it on a simple example.

Example: checks if given array t conains only zeros

```
int all_zeros(int t[], int n) {
  int k;




  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

How can we verify it with Frama-C/WP?

Example: Formal Specification in ACSL

```
/*@ requires n>=0 && \valid(t+(0..n-1));




*/
int all_zeros(int t[], int n) {
  int k;






  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

First, specify
a function contract

Example: Formal Specification in ACSL

```
/*@ requires n>=0 && \valid(t+(0..n-1));

    ensures \result != 0 <==>
      (\forall integer j; 0 <= j < n ==> t[j] == 0);
*/
int all_zeros(int t[], int n) {
  int k;






  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

First, specify
a function contract

Example: Formal Specification in ACSL

```
/*@ 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;





  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

First, specify
a function contract

Example: Formal Specification in ACSL

```
/*@ 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;


  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

Then, write loop contracts

Example: Formal Specification in ACSL

```
/*@ 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;

  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

Then, write loop contracts

Example: Formal Specification in ACSL

```
/*@ 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;
}
```

Then, write loop contracts

Example: a complete C program annotated in ACSL

```
/*@ 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;
}
```

Finally, prove it in Frama-C/WP

Outline

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 };
...
```

Outline

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 bahaviors

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.

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

Outline

Conclusion

- The `memb` module specified and formally verified with Frama-C
  - 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;`

Formal verification should be more systematically applied to IoT software to guarantee safety and security.

Future Work

- Continue verification of `memb` with a more precise specification
  - stronger isolation between blocks
- Verification of other modules of Contiki (list, . . .)
  - may require beter support of memory-related features in Frama-C/WP
- Specification and Verification of other IoT software
  - European project VESSEDIA (CEA, INRIA, Dassault Aviation, Search Lab, Fraunhofer FOKUS,...)