

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

Frédéric Mangano, Simon Duquennoy, Nikolai Kosmatov

▶ **To cite this version:**

Frédéric Mangano, Simon Duquennoy, Nikolai Kosmatov. Formal Verification of a Memory Allocation Module of Contiki with Frama-C: a Case Study. CRiSIS 2016 - 11th International Conference on Risks and Security of Internet and Systems, Sep 2016, Roscoff, France. hal-01351142

HAL Id: hal-01351142
https://inria.hal.science/hal-01351142

Submitted on 22 Sep 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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

Frédéric Mangano[1], Simon Duquennoy[2,3], and Nikolai Kosmatov[1]

[1] CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr
[2] Inria Lille - Nord Europe, France
firstname.lastname@inria.fr
[3] SICS Swedish ICT, Sweden

Abstract. Formal verification is still rarely applied to the IoT (Internet of Things) software, whereas IoT applications tend to become increasingly popular and critical. This short paper promotes the usage of formal verification to ensure safety and security of software in this domain. We present a successful case study on deductive verification of a memory allocation module of Contiki, a popular open-source operating system for IoT. We present the target module, describe how the code has been specified and proven using Frama-C, a software analysis platform for C code, and discuss lessons learned.

Keywords: deductive verification, specification, FRAMA-C, Contiki, memory allocation.

1 Introduction

While formal verification is traditionally applied to embedded software in many critical domains (avionics, energy, rail, etc.), its usage for the Internet of Things (IoT) has not yet become common practice, probably because the first IoT applications were not considered as critical. However, with the emergence of the Internet of Things, embedded devices get massively connected to the Internet. In this context, security concerns become of utmost importance as IoT applications today often deal with sensitive data and act on the physical world. The devices are both constrained and network-facing, thus creating new opportunities for attackers and new challenges for verification. One of these challenges is to verify an embedded yet full-fledged low-power IPv6 stack underlying many potentially critical IoT applications. In this paper we focus on the Contiki OS [3], and in particular on its memory allocation module `memb` providing a generic mechanism for allocation of a bounded number of blocks of any given type.

Contributions. This experience report paper advocates the use of formal verification for IoT and presents a case study on verification of the `memb` module performed with FRAMA-C [4], a rich and powerful toolset for analysis of C code. We formally specify `memb` operations and prove it using the deductive verification tool FRAMA-C/WP. We emphasize two specific issues: the generic nature of the module (resulting in heavier pointer arithmetics and casts) and the need to specify the number of available blocks (requiring an axiomatic definition of occurrence counting). Finally, we describe the verification results and discuss lessons learned.

2 Contiki and its Memory Allocation Module

Contiki and Formal Verification. Today's IoT software is highly critical because it runs on hardware that is able to sense or even act on physical things. A compromised IoT device may get access to sensitive or private data. Worse, it might become able to take action on the physical world, potentially with safety consequences. Examples of such include reconfiguring an industrial automation process, interfering with alarms or locks in a building, or in the e-health domain, altering a pacemaker or other vital devices.

Contiki is an Operating System for the Internet of Things. It was among the pioneers in advocating IP in the low-power wireless world. In particular, it features a 6LoWPAN stack [5], that is, a compressed IPv6 stack for IEEE 802.15.4 communication. This enables constrained devices to interoperate and connect directly to the Internet. Sensors, actuators or consumer devices can be brought together and create applications in various areas such as home automation or the smart grid.

Contiki is targeted at constrained devices with a 8, 16 or 32-bit MCU and no MMU. The devices usually feature a low-power radio module, some sensors, a few kB RAM and tens of kB ROM. Contiki has a kernel, written in portable C, that is linked to platform-specific drivers at compile-time. At the time of writing, it supports 36 different hardware platforms.

When Contiki started in 2003, the focus was on enabling communication in the most constrained devices, with no particular attention given to security. As it matured and as commercial applications arose, communication security was added at different layers, via standard protocols such as IPsec or DTLS. The security of the software itself, however, did not receive much attention. Although a continuous integration system is in place, it does not include formal verification. While formal verification has already been applied to microkernels and Cloud hypervisors (see [2, Sec. 5] for related work), we are not aware of similar verification projects for IoT software.

Contiki's `memb` Module. In this case study, we turn our attention to Contiki's main memory management module: `memb`. To avoid fragmentation in long-lasting systems, Contiki does not use dynamic allocation. Memory is pre-allocated in blocks on a per-feature basis, and the `memb` module helps the management of such blocks. For instance, the routing module provisions for N entries, which are stored in a static memory area N times the size of a routing entry. Entries are managed by allocating and freeing blocks at runtime.

The module `memb` offers a simple API, enabling to *initialize* a `memb` store, *allocate* a block, *free* a block, *check* if a pointer refers to a block inside the store and *count* the number of allocated blocks. `memb` consists in about 100 lines of code but is one of the most critical elements of Contiki, as the kernel and many modules rely on it. A flaw in `memb` could result in attackers reading or writing arbitrary memory regions, crashing the device, or triggering code execution.

The Contiki code base involves a total of 56 instances of `memb`. Not all are included in a given Contiki firmware, but a subset is included depending on the application and configuration. `memb` is used for instance for HTTP, CoAP (lightweight HTTP), IPv6 routes, CSMA, the MAC protocol TSCH, packet queues, network neighbors, the file system Coffee or the DBMS Antelope.

```
1  /* file memb.h */
2  struct memb {
3    unsigned short size; // block size
4    unsigned short num;  // number of blocks
5    char *count;         // block statuses
6    void *mem;           // array of blocks
7  };
8  #define MEMB(name, btype, num)...
9  // macro used to decrare a memb store for
10 // allocation of num blocks of type btype
11
12 void  memb_init(struct memb *m);
13 void *memb_alloc(struct memb *m);
14 char  memb_free(struct memb *m, void *p);
15 ...
```

(a)

```
1  /* file demo.c */
2  #include "memb.h"
3  struct point {int x; int y};
4
5  // before preprocessing,
6  // there was the following macro:
7  // MEMB(pblock, struct point, 2);
8
9  // after preprocessing, it becomes:
10 static char pblock_count[2];
11 static struct point pblock_mem[2];
12 struct struct memb pblock = {
13   sizeof(struct point), 2,
14   pblock_count, pblock_mem };
15 ...
```

(b)

Fig. 1: (a) Extract of file `memb.h` defining a template macro `MEMB`, and (b) its usage (in file `demo.c`) to prepare allocation of up to 2 blocks of type **`struct`** `point`

3 Verification of `memb` with FRAMA-C

3.1 FRAMA-C Platform and its Deductive Verification Plugin WP

FRAMA-C [4] is a popular software analysis platform for C programs that offers various static and dynamic analyzers as individual plugins. They include the deductive verification tool WP, abstract interpretation based value analysis, dependency and impact analysis, program slicing, test generation, runtime verification, and many others. FRAMA-C comes with a behavioral specification language ACSL [1]. The user specifies the desired properties of their program by adding ACSL annotations (preconditions, postconditions, loop invariants, assertions, etc.). These annotations are written in special comments `/*@ <annotation>*/` or `//@ <annotation>`. FRAMA-C/WP can be used then to establish a rigorous mathematical proof that the program satisfies its specification. Technically, it relies on automatic provers (SMT solvers) that try to prove the theorems (*verification conditions*, or VCs) automatically generated by WP.

3.2 Declaration of Memory Allocation Data via a Pseudo-Template

A `memb` store is represented by the **`struct`** `memb` structure (see Fig. 1a, lines 2–7). Being written in C, it does not allow polymorphism. Instead, it stores as a field of the structure the size of the block type it is meant to store, which enables the implementation to dynamically compute the addresses of the blocks.

The actual blocks are stored in an array (referred to as field `mem`), statically allocated using a global array definition. This is illustrated in Fig. 1b, lines 10–14, for 2 blocks of type **`struct`** `point`. Since its length varies, the array cannot be declared directly in the structure, that is why the structure contains a pointer that is initialized to point to the global array. The `count` array is allocated in the same fashion.

All the fields of the `memb` structure are initialized at compile-time and shall not change when the program executes. That is conveniently realized using the `MEMB` macro (whose definition is omitted in Fig. 1a), which relies on the preprocessor in order to

- generate the global array definitions for the `count` and `mem` arrays,
- declare an initialized `memb` store.

Lines 10–14 of Fig. 1b show the result of line 7 after the preprocessing.

```
1  /*@
2    requires valid_memb(m);
3    ensures valid_memb(m);
4    assigns m→count[0 .. (m→num - 1)];
5    behavior free_found:
6      assumes ∃ ℤ i; 0 ≤ i < m→num ∧ m→count[i] == 0;
7      ensures ∃ ℤ i; 0 ≤ i < m→num ∧ \old(m→count[i]) == 0 ∧ m→count[i] == 1 ∧
8        \result == (char*) m→mem + (i * m→size) ∧
9        ∀ ℤ j; (0 ≤ j < i ∨ i < j < m→num) ⟹ m→count[j] == \old(m→count[j]);
10     ensures \valid((char*) \result + (0 .. (m→size - 1)));
11     ensures _memb_numfree(m) == \old(_memb_numfree(m)) - 1;
12   behavior full:
13     assumes ∀ ℤ i; 0 ≤ i < m→num ⟹ m→count[i] ≠ 0;
14     ensures ∀ ℤ i; 0 ≤ i < m→num ⟹ m→count[i] == \old(m→count[i]);
15     ensures \result == NULL;
16   complete behaviors;
17   disjoint behaviors;
18 */
19 void *memb_alloc(struct memb *m);
```

Fig. 2: (Simplified) ACSL contract for allocation function `memb_alloc` (file `memb.h`)

3.3 Specifying Operations

First, we specify the functions of `memb` in ACSL. Let us describe here the contract for the allocation function `memb_alloc` shown in Fig. 2. Its ACSL contract contains preconditions (**`requires`** clauses) that are assumed to be ensured by the caller, and postconditions (**`ensures`** clauses) that should be ensured by the function at the end of its execution and thus proved by the verification tool. Lines 2–3 specify that the store respects a global validity property before and after the call. A specific behavior can be expressed in ACSL using a **`behavior`** section, which defines additional post-conditions whenever the behavior guard given in the **`assumes`** clause is satisfied. The contract may stipulate that the given behaviors are disjoint and complete (lines 16-17), the verification tool verifies it as well. The `memb_alloc` function has two behaviors:

1. If the `memb` store is full, then it is left intact, and `NULL` is returned (lines 12–15).
2. If the `memb` store has at least one free block, then it must be guaranteed that:
   - the returned block is properly aligned in the block array (line 8),
   - the returned block was marked as free, and is now marked as allocated (line 7),
   - the returned block 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, see also Sec. 3.4).

3.4 Keeping Track of Free Blocks

When allocating, we may need to ensure the `memb` store is not full, that is, some blocks are available. To this end, we make assumptions on their number that we compute using a logic function named `_memb_numfree`. For instance, requiring that at least n blocks are free ensures that the n subsequent allocations will succeed. The specification states that the number of free blocks is decremented when allocating, and incremented back when a block is freed. Allocation succeeds if and only if the number of free blocks before the allocation is non-zero.

```
15 ...
16 /* file demo.c, continued */
17 /*@
18   requires valid_memb(&pblock) ∧ pblock.num == 2;
19   requires pblock.size == sizeof(struct point);
20 */
21 void main() { // all contracts proven with WP except out-of-bounds pointer line 28
22   memb_init(&pblock);
23   /*@ assert _memb_numfree(&pblock) == 2; */
24   void *obj1 = memb_alloc(&pblock), *obj2 = memb_alloc(&pblock);
25   /*@ assert _memb_numfree(&pblock) == 0 ∧ obj1 ≠ NULL ∧ obj2 ≠ NULL; */
26   /*@ assert \valid((char*) obj1 + (0 .. sizeof(struct point)-1)); */
27   /*@ assert \valid((char*) obj2 + (0 .. sizeof(struct point)-1)); */
28   /*@ assert \valid((char*) obj1 + sizeof(struct point)); */ // UNPROVEN - invalid
29   memb_free(&pblock, obj1);  memb_free(&pblock, obj2);
30   /*@ assert _memb_numfree(&pblock) == 2; */
31 }
```

Fig. 3: Example of ACSL-annotated function using `memb` (file `demo.c`)

A `memb` store uses its `count` array to determine whether its i^{th} block is free ($count[i] = 0$) or allocated ($count[i] = 1$). In the logic world of ACSL, counting the number of free blocks, or more precisely counting the number of occurrences of 0 in the `count` array, requires an inductive definition using sub-arrays:

- If $to \leq from$, then $count[from \ldots to[$ obviously contains no zeros.
- If $from > to$ and $count[to - 1] = 0$ then $count[from \ldots to[$ contains one more zero than $count[from \ldots to - 1[$.
- If $from > to$ and $count[to - 1] \neq 0$ then $count[from \ldots to[$ contains as many zeros as $count[from \ldots to - 1[$.

In our specification, we use a more general inductive definition from [2], as well as a few auxiliary lemmas proven by induction in the proof assistant Coq (v.8.4pl6).

3.5 Deductive Verification Results

The current specifications of the `memb` modules are fully proven automatically using FRAMA-C/WP Magnesium-20151002, Alt-Ergo 0.99.1, CVC4 1.4 and Z3 4.4.2. The ACSL specification of `memb` is 115 lines of code long, for a total of 259 lines in the header file. To prove it, 32 additional lines of annotations were required in the implementation file, summing up to 154 lines. 126 verification conditions are generated.

Fig. 3 shows an annotated function using `memb` that can be automatically proven with FRAMA-C/WP, except for line 28 that contains an out-of-bounds pointer. Thus, out-of-bounds accesses are automatically detected thanks to the provided specification.

This verification case study also allowed to detect a potentially harmful situation. The `memb_free` function used to decrement the `count` associated to the given block, instead of setting it to 0. An awkward consequence of this is that calling `memb_free` on a block with an unusual `count` (e.g. greater than 2) would not actually free it. While this should not happen under normal circumstances, we have decided to replace that decrement operation by a set to 0. This choice makes `memb_free` both simpler and more robust, easing the verification process, and we recommend to integrate it into the production code.

4 Conclusion and Future Work

IoT software is becoming more critical and widely used. We argue that formal verification should be more systematically applied in this domain to guarantee that critical software meets the expected level of safety and security.

This paper reports on a case study where deductive verification with FRAMA-C/WP has been applied on the memory allocation module `memb`, one of the most critical and largely used components of the Contiki OS. We have described the verification methodology and results. In particular, the presented verification *formally guarantees* the absence of out-of-bounds accesses to the block array in the `memb` module.

We have emphasized two technical aspects. One is related to pointer arithmetics and casts due to the generic implementation of the module for all possible block types. The second one concerns inductive definitions and proofs necessary to count elements in the block status array and to state some properties on the corresponding counting functions. While these aspects could constitute an obstacle for formal verification of real-life C software a few years ago, they can be successfully treated today by modern verification tools like FRAMA-C/WP. This experience report also shows that automatic theorem provers have made significant progress, and that interactive proof, e.g. with Coq proof assistant, can be used in complement on remaining properties that are too complex to be proven automatically.

One future work direction is the verification of `memb` with a slightly more precise specification, including for example stronger isolation properties between blocks of the same store. This would require a better support of ACSL allocation primitives in WP (such as the `frees` clause) in order to better trace validity of individual blocks. Secondly, the results of this case study should facilitate the verification of other components of Contiki relying on `memb`. For some of them (such as `list`, defining chained lists), this could require further extensions of FRAMA-C and ACSL. Finally, specification and proof of other IoT software modules is another future work direction.

Acknowledgment. Part of the research work leading to these results has received funding for DEWI project (www.dewi-project.eu) from the ARTEMIS Joint Undertaking under grant agreement No. 621353. The second author has also been partially supported by a grant from CPER Nord-Pas-de-Calais/FEDER DATA and the distributed environment Ecare@Home funded by the Swedish Knowledge Foundation 2015-2019. Special thanks to Allan Blanchard, François Bobot and Loïc Correnson for advice, and to the anonymous referees for their helpful comments.

References

1. Dunkels, A., Gronvall, B., Voigt, T.: Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In: LCN 2014
2. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. **27**(3) (2015) 573–609
3. Montenegro, G., Kushalnagar, N., Hui, J., Culler, D.: Transmission of IPv6 packets over IEEE 802.15.4 networks. RFC 4944 (September 2007) http://www.rfc-editor.org/rfc/rfc4944.txt.
4. Blanchard, A., Kosmatov, N., Lemerre, M., Loulergue, F.: A case study on formal verification of the Anaxagoros hypervisor paging system with Frama-C. In: FMICS 2015
5. 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.