D5.2

Inria's use case final report

Project number:	731453
Project acronym:	VESSEDIA
Project title:	Verification engineering of safety and security critical dynamic industrial applications
Start date of the project:	1st January, 2017
Duration:	36 months
Programme:	H2020-DS-2016-2017

Deliverable type:	Report
Deliverable reference number:	DS-01-731453 / D5.2/ 1.0
Work package contributing to the deliverable:	WP 5
Due date:	December 2019 – M36
Actual submission date:	15th January 2020

Responsible organisation:	INRIA
Editor:	Allan BLANCHARD
Dissemination level:	PU
Revision:	1.0

Abstract:	Inria's use-case final report at M36. Applying the VESSEDIA tools to the verification of the Contiki Operating System.
Keywords:	Contiki OS, static & dynamic analysis, Frama-C

The project VESSEDIA has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731453.

Editor

Allan BLANCHARD (INRIA)

Contributors (ordered according to beneficiary numbers)

Balázs Berkes (SLAB)

Oscar Llorente (FD)

Disclaimer

Executive Summary

IoT (Internet of Things), which denotes connected devices and services, is on a rapid increase, leading to a constantly growing number of interconnected computing devices and as they are gaining wider and wider adoption in the security critical fields, it becomes more and more urgent to ensure the security of these devices. The VESSEDIA project aims to enhance the security of IoT devices by improving already existing software analysis tools to help the manufacturers to develop more secure devices.

In order to evaluate the ability of the VESSEDIA tools to allow efficient security analysis of IoT software, the VESSEDIA project comprises several use-cases. One of these use-cases consists in verifying the Contiki operating system, a lightweight operating system for IoT.

The goal of this document is to present the verification effort performed during the project on the Contiki operating system. This verification is mostly conducted using the Frama-C platform and the FlowGuard tool.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Goal of the document

This report describes the work done in Task 5.1 of the VESSEDIA project, about the verification of the Contiki operating system. This task assesses the mechanisms developed in VESSEDIA on Contiki's low-power IPv6 stack and OS primitives, mainly using static analysis. The level of these analyses being chosen depends on the criticality and the complexity of the modules to verify. This allows the evaluation of the usability and the effectiveness of VESSEDIA tools for analysis of actual IoT systems. In addition, parts of Contiki are annotated with "minimal contracts" as a fast alternative to value analysis. In this way, we gathered for Task 3.3 valuable experience on the trade-off between automated, but monolithic, abstract interpretation, and writing minimal contracts to be verified in parallel.

During the period M7-M18, we have studied different parts of Contiki. Most of the work has been done in the core libraries of Contiki using deductive verification, even though some first experiments have also been conducted on the networking stack with abstract interpretation. During the period M18-M36, we continued the work on the core libraries, using new methods to make verification easier. Among this, we also conducted some experiments in order to enable runtime verification in Contiki.

1.2 Structure of the document

Chapter 2 recalls the target of evaluation and the related security objectives. Chapter 3 gives an overview of the different tools and methods used during the analysis of this use case. 3.2 presents the realization of the use case, comprising for each subpart how it has been prepared, the process of verification and the results we obtained. 4.6 provides some discussion, in particular the lessons learnt during the verification and Chapter 6 concludes this document.

1.3 Related deliverables

The deliverable D5.1 is a preliminary version of this deliverable and presented the work done until M18. Both these deliverables are related to the deliverable D1.2 about the security requirements of the WP5 use cases. During the verification of this use case, we used a verification methodology called *minimal contracts* which is described in deliverable D1.1. The work performed on this use-case also helped to improve the VESSEDIA tools for the final release available in the deliverable D2.3. Deliverable D4.6 also describes Security Evaluation carried out on the Use Cases, including Contiki OS.

Chapter 2 Target of evaluation

Contiki is an open-source operating system (OS) for the Internet of Things. It provides basic OS features on an event-based kernel, including the scheduler, a timer system, the networking stack, and a file system. Contiki focuses generally on low-power IPv6 connectivity, that is, it enables constrained devices to connect to the Internet with standard protocols. This guarantees interoperability between devices that come from different vendors, since Contiki runs on sensor and actuator devices covering a wide spectrum of hardware architectures. The target device typically has an 8, 16, or 32-bit MCU (little or big-endian), a low-power radio, some sensors and actuators, and is battery-operated.

Some recent platforms might have a Memory Protection Unit (MPU), but Contiki is not yet able to make use of it. Consequently, the devices targeted by Contiki are generally platforms without Memory Management Unit (MMU), so there is no protection between different applications nor between applications and kernel. They have some access to the physical world, via sensors and actuators, while being connected to the Internet. Thus, Contiki mostly evolves at *perception* and *network layers* of an IoT architecture, as presented in D1.1. This makes such devices a target of choice for attackers. A misbehaving or compromised network node could be used as the source of DDoS attacks, leak sensitive data, or worse, trigger potentially life-threatening actuations.

2.1 Description

Contiki is released under a BSD license and is hosted on GitHub. The project started in 2003, and has now over 150 contributors and 2000 followers on GitHub. Contributors and users are from both industry and academia. It is written in portable C, in order to enable portability across the platforms. In the beginning of the project, Contiki 3 was the most recent version of Contiki, it has been forked to Contiki-NG[1] recently. Contiki-NG being actively maintained and updated, we have decided to switch to this version, even if it has required some work to port previous verification effort.

The development workflow is as follows: a firmware image consists in a Contiki configuration along with some application processes. The configuration dictates which modules of Contiki are to be included. When compiling the firmware, the user selects the target hardware platform. The cross-compilation is then carried out, and only the relevant modules are linked in the final image. Note that, as a result, large portions of the kernel are unutilized in any given final image. For instance, an application that does not require the file system or IPv6 stack will be simply compiled without it.

2.1.1 Considered code base

The platform independent directory of Contiki is now "os" and is composed of a number of sub-directories Table 1 summarizes the size of each module and discusses the priority of verifying them in VESSEDIA. The relevance is selected among:

- **NO:** We chose not to verify this module in VESSEDIA, because it is deprecated or only used by a minority of applications.
- **LOW:** The module is generally used, and it is worth verifying it.
- **MED:** Widely-used module, verification is important.
- **HIGH:** Critical component, verification is top-priority.

[1] https://github.com/contiki-ng/contiki-ng

Module	kLOC	Description	Priority
dev	1.3	Platform-independent parts of drivers.	LOW
lib	39.3	Different general purpose librairies	
|- lib/*.[ch]	2.1	Memory management, lists, crypto, etc.	HIGH
|- lib/dgb-io	0.7	Debugging tools using input/output	MED
|- lib/fs	35.8	File-system	NO
|- lib/json	0.7	JSON format handling	NO
net	36	Networking stack	
|- net/app-layer	7	Application layer protocols	NO
|- net/ipv6	11.9	IPv6 stack	MED
|- net/mac	8.2	MAC layers	
|- net/mac/ble	0.5	Bluetooth low energy L2CAP implementation	LOW
|- net/mac/csma	0.5	Standard CSMA MAC	MED
|- net/mac/framer	1.3	Encoding and decoding of MAC frame headers	MED
|- net/mac/tsch	5.7	Standard power-saving MAC	LOW
|- net/mac/*.[ch]	0.2	MAC API	MED
|- net/routing	8	Currently RPL implementations	MED
|- net/*.[ch]	0.9	Neighbour tables, packet buffers etc.	MED
services	9.8	Application layer	NO
storage	6.3	Contiki File System, and related applications.	NO
sys	1.8	Core components: scheduler, timers, etc.	HIGH

Table 1 - The different modules of Contiki with respective sizes and verification priority

To summarize, we identify a total of 7.5 kLOC low-priority codes, 23.5 kLOC medium-priority, 3.9 kLOC high-priority. There is slightly more high and medium priority code than we presented in the deliverable D1.2 about Contiki 3. This is mainly due to the addition of some features. This increase is balanced by the fact that the majority of those features are currently not used intensively in the critical part of the system, this is for example the case for the different new variants of the linked lists.

2.2 Security objectives

The security objectives of the Contiki OS can be enumerated from what is listed in the CC OS Protection Profile BSI-CC-PP-0067 [1]document. While, we have detailed the security objectives in the deliverable D1.2, we provide a reminder in this section. Some of the objectives like auditing are left out from this chapter because they are not applicable for the Contiki OS right now. It is not impossible to implement these objectives on the long run, but in the current status of the project conforming to the CC is not an issue right now. Some parts are nevertheless included, since they can be used as a recommendation in the future.

Cryptographic services: The TSF must allow authorized users to remotely access the TOE using a cryptographically-protected network protocol that ensures integrity and confidentiality of the transported data and is able to authenticate the end points of the communication. Note that the same protocols may also be used in the case where the TSF is physically separated into multiple parts that must communicate securely with each other over untrusted network connections. The TSF must control access of subjects and/or users to named resources based on identity of the object. The TSF must allow authorized users to specify for each access mode which users/subjects are allowed to access a specific named object in that access mode.

Cryptographic functionality is a necessity for the overall security for the OS, but these functionalities are hard to verify with software verification. As we will detail later, we have verified some simple aspects of these modules in Contiki, namely the absence of runtime errors.

Identification and authentication: The TOE must ensure that users have been successfully authenticated before allowing any action the TOE has defined to provide to authenticated users only. The Contiki OS uses Datagram Transport Layer Security (DTLS) for this purpose.

Discretionary access control: The TSF must control access of subjects and/or users to named resources based on identity of the object. The TSF must allow authorized users to specify for each access mode which users/subjects are allowed to access a specific named object in that access mode, and thus prevent unauthorized read access, modification, deletion of the object, creation of new objects, and management of object attributes.

Management of security mechanisms: The TSF must provide all the functions and facilities necessary to support the authorized users that are responsible for the management of TOE security mechanisms and must ensure that only authorized users are able to access such functionality. For Cyber-Physical Systems (CPS), there is a concern regarding the physical security of the product, and it is necessary to prevent the attacker from controlling the TOE. One way to achieve this is to protect the sensor data and the controlling functionalities from the attacker, so the Security Functional Requirements (SFRs) can be broadened with these functionalities.

In the case of Contiki, the notion of user does not appear at the OS level and is most of the time expressed at the application level, thus the questions of identification, authentication and discretionary access control are not considered here. This is also the case for the management of security mechanisms that relies on the identification of some kind of "administrator".

Network information flow control: The TOE shall mediate communication between sets of TOE network interfaces, between a network interface and the TOE itself, and between subjects in the TOE and the TOE itself in accordance with its security policy.

Subject communication: The TOE shall mediate communication between subjects acting with different subject security attributes in accordance with its security policy.

Trusted channel: The TSF must be designed and implemented in a manner that allows for establishing a trusted channel between the TOE and a remote trusted IT system that protects the user data and TSF data transferred over this channel from disclosure and undetected modification, and prevents masquerading of the remote trusted IT system.

At the OS level, we can only partially consider the two latter objectives since the subjects are generally defined at the application layer. Thus, to fulfil the objectives, we must rely on these definitions that are use-case specific.

On the overall, the security objectives that are listed by the OSPP are (purposely) specific. On the opposite on Contiki, we focused on more general aspects without which we cannot achieve these specific objectives. Namely, we verified critical components of the operating system that are used widely by the other modules (thus with a high impact in case of a problem), and we also worked on complete OS instances in order to provide some way to check the absence of runtime errors (that can impact any of the previous objectives).

Chapter 3 Overview of used tools and methods

In this chapter, we present the methods and the tools that we have used to verify different aspects and parts of the system. We first present the tools we have used for the verification of functional properties, and then the tools used for verification of absence of runtime errors.

3.1 Verification of functional properties

The verification of functional properties consists in verifying that a module or a program does the job it is supposed to do. Most of the time this kind of verification is done by testing programs. For example, the Contiki project contains different tests to check that different protocols correctly work.

During the VESSEDIA project, we mainly applied static analysis to check this kind of properties. We used the WP plugin of Frama-C which is a deductive verification plugin based on Dijkstra's weakest precondition calculus. This plugin receives as an input a list of function annotated with their contract (that expresses what they are supposed to do) and other annotations to guide the proof process. From this input, the plugin generated what we call verification conditions, that are formulas that are then verified (or not) by SMT solvers. During our different experiments, we have used different SMT solvers, namely Alt-Ergo, CVC3, CVC4, Z3 and E-prover.

This kind of verification is quite hard to perform. Meanwhile, as we will detail later, we have made some experiments in order to make it easier, and we also provided a way to check that client module respects their contracts at runtime using the E-ACSL plugin of Frama-C. From the annotations that are written in the code (or generated by Frama-C), E-ACSL can generate some C code that will check it at runtime.

Summary of the tools:

- Frama-C
- WP plugin of Frama-C (deductive verification)
- E-ACSL plugin Frama-C (runtime verification)
- SMT solvers:
    - Alt-Ergo,
    - CVC3,
    - CVC4,
    - Z3,
    - E-prover

3.2 Verification of absence of runtime errors

In C, when some features are wrongly used, the ISO norm generally does not specify that some error will happen, most of the time they produce what is called an *undefined behaviour*. In such a case, the error is not guaranteed to be detected at runtime, thus they can for example corrupt silently the memory or open some security breaches. The detection of these problems is generally done at runtime using debuggers (and by compiling the program with some special parameters that makes verification easier), or using specific features of compilers called "sanitizers".

During the VESSEDIA project, we first focused on static analysis. First, we used the previously mentioned WP plugin, indeed, verifying functional properties also requires to verify the absence of runtime errors. Interestingly, it can be used to verify only the later. Basically, the idea is to provide in the contract of the functions only the properties that are needed to verify the absence of runtime errors, leading to simpler annotations (thus less work). This method called "minimal contracts" has been used as an alternative to the second method we have used: abstract interpretation using the EVA plugin of Frama-C. This is the most automated tool we have used during the project, while as we will explain later, preparing the project for this analysis is more complex and requires a good knowledge of the code base. Furthermore, we met some difficulties due to the features used in

Contiki and decided to also conduct some experiment for the verification of absence of runtime errors using runtime verification.

For the verification at runtime, we again used the E-ACSL plugin of Frama-C, in combination with the RTE plugin of Frama-C that is used to insert some annotation for each operation that can cause a runtime error during execution.

We also used the FlowGuard plugin to GCC. This plugin is used to instrument a program during compilation in order to check and enforce data-flow integrity at runtime.

Summary of the tools:

- Frama-C
- WP plugin of Frama-C (with minimal contracts)
- Alt-Ergo (SMT solver)
- EVA plugin of Frama-C (abstract interpretation)
- E-ACSL plugin Frama-C (runtime verification)
- FlowGuard (dataflow integrity)

Chapter 4 Use case realization

This chapter describes the verification done on different parts of the Contiki operating system. For each section, we give a description of the component or the goal of the corresponding verification, and then we explain the preparation of the component, the process of verification and expose the obtained results. Note that when the preparation or the process does not deserve an entire paragraph, it is eluded.

Section 4.1 presents the use of minimal contracts to verify the absence of runtime errors in some low level modules of the OS. Then we present, the verification of absence of runtime errors in the AES-CCM module, first by using minimal contracts (Section 4.2) and then by abstract interpretation (Section 4.3). Then we consider the whole operating system, first with the use of abstract interpretation on a complete Contiki instance (Section 4.4), then using runtime verification with E-ACSL (Section 4.5), and finally using FlowGuard for dataflow integrity (Section 4.6). In Section 4.7, we describe the different techniques we have used for the verification of functional properties on the linked list module of Contiki. Finally, we present the verification of the memory allocation module (Section 4.8).

4.1 Absence of runtime errors in small modules (minimal contracts)

Verifying the absence of runtime errors is a good way to guarantee the absence of some classes of security issues. While providing contracts for a full functional verification often requires an important effort, as well as the proof itself, it is often not necessary to have such complex contracts to guarantee the absence of runtime errors.

Deliverable D1.1 reported on a first experiment of a feasibility for the verification of Contiki, using so-called *minimal contracts* to perform the verification of absence of runtime errors of particular modules of Contiki using Frama-C and the WP plugin. The main advantage of this method is that verification with Frama-C and the WP plugin validates any correct use of the module. Whereas a verification using Frama-C and the EVA plugin on a complete program that uses the module only ensures the absence of runtime errors in the context of this program. The drawback is the fact that it is less automatic, since we have to write contracts and loop invariants in the source code. However, minimal contracts lead to simpler contracts than the ones required for the verification of functional properties.

The modules verified for the Deliverable D1.1 are small modules (a few hundred lines of code) that are part of the core libraries of the operating system and the system part.

Among this, we also conducted a verification of a bigger module in the networking stack, namely the SICSLOWPAN module on which the Task 5.2 relies. This module is an implementation of the 6LoWPAN [2] protocol produced by SICS for Contiki. This protocol provides a way to implement the IPv6 that is suitable for low power equipment, for example for the IoT devices that are targeted by Contiki. This protocol is often use to build Low-power Lossy Networks, like the ones targeted by the 6LoWPAN Management Platform analysed in the CEA's use case. The SICSLOWPAN module relies on the AES-CCM* module, that we will describe in Section 4.2, to ensure security and authentication.

4.1.1 Preparation

As mentioned before, the analysis was done during the first months of the project for the Deliverable D1.1 by FOKUS, this verification was done on the Contiki 3 version of the OS. The corresponding contracts have been integrated to our verification of Contiki-NG.

Here, the goal was to provide some contracts to the considered function, but only the properties that are necessary to guarantee the absence of runtime errors. Most of the time, these contracts only contain assigns specification (the memory locations that are modified by the function), properties about pointers validity, as well as bounds on integers (in order to avoid bad memory accesses).

However, the exact process of specification is related to the process of verification that we detail in next section.

4.1.2 Verification process

In the minimal contracts approach, the verification process strongly influences the work of specification. The process of verification is carried out in a bottom-up approach, we start the verification with the lowest-level functions (that generally do not have any dependencies) and as we verify these functions, we can progressively start the verification of their callers and so on.

The verification process is guided by the use of the RTE plugin of Frama-C (in another tool, it could be guided by any component whose role is to generate the conditions necessary for the absence of runtime errors in the program). The principle, for the verification of a function, is the following:

- Generate the assertions necessary to prove the absence of runtime errors with RTE,
- Write a contract which allows to verify these assertions,
- Start a proof with the WP plugin.

Of course, that also means that a contract written for a function is not necessarily the final contract of the function at some point, as some caller function might need more information from the called function to guarantee its own absence of runtime errors. As an illustration, let us consider the following example:

```
/*@
  requires val > INT_MIN ;
  assigns \nothing ;
  // ensures val > 0 ;
*/
int abs(int val){
  if(val < 0){
    //@ assert RTE: val-1 >= INT_MIN ;
    return -val ;
  }
  return val ;
}

int user_code(int val){
  return sqrt(abs(val)) ;
}
```

Figure 1 - A simple example with minimal contracts

While verifying the absence of runtime errors in the `abs` function only requires to be sure that the parameter of the function is greater than `INT_MIN` (because of the assertion generated by RTE on line 8), the verification of the function `user_code` requires more information from the `abs` function, namely that the output value is positive (as the square root function waits a positive value).

We will not further detail the minimal contracts approach, for this one could consider the Deliverable D1.1 of the VESSEDIA project.

4.1.3 Results

The results of this verification are equivalent to what was done for the D1.1. Some modules have been removed from Contiki 3 to Contiki-NG so we do not list them here.

- For the following files, we built minimal contracts and verified them:
  - `os/lib/crc16.c`

    - os/lib/ifft.c
    - os/lib/ringbuf.c
    - os/lib/ringbufindex.c
- Besides these files we found that some files were trivial to analyse. No manual annotations were needed for them in order to make Frama-C prove the absence of run-time exceptions. These were:
    - os/lib/assert.c
    - os/lib/random.c
    - os/sys/energest.c
- On the other hand, some files were intractable, i.e. couldn't be handled with Frama-C/WP due to implementation restrictions. These were, grouped by intractability reason:
    - *validity of unsized-array not implemented yet*:
        - lib/sensors.c
        - sys/procinit.c
    - ***\valid_function** not yet implemented*:
        - lib/aes-128.c
        - lib/ccm-star.c
        - lib/trickle-timer.c
        - sys/process.c
        - sys/rtimer.c
    - *calculus failed on strategy for XXX behaviour YYY, all properties, both assigns or not because unsupported non-natural loop(s): try [-wp-invariants] option (abort):*
        - sys/ctimer.c
        - sys/etimer.c

The SICSLOWPAN module was partially verified by Quentin Molle during his internship at CEA. However, the verification is not complete. Apparently, this module of the networking stack was remarkably harder to handle than the other modules we had verified so far, even using only minimal contracts. The main reason is that there is a lot of type casting and that some functions are really long. Consequently, performing the verification requires to add a lot of annotations to guide the provers. Further investigation is necessary to understand if it is possible to handle this kind of module using minimal contracts.

4.2 Absence of runtime errors in the AES-CCM module (minimal contracts)

Contiki implements the Advanced Encryption Standard (AES), a symmetric encryption algorithm. AES was designed to be efficient in both hardware and software implementations, and supports a block length of 128 bits and key lengths of 128, 192 and 256 bits. In Contiki, only 128-bit keys are supported. In order to secure arbitrarily long data chunks, the AES-CCM block cipher mode of operation is also implemented in Contiki. In term of security, data encryption (AES) and authentication (CCM) is a very important ingredient of wireless communication in a network. Thus, a flaw in this component would be critical, and we have to ensure its security.

4.2.1 Preparation

As mentioned in Section 4.1.3, the AES-CCM contains a language construct that is currently not handled by Frama-C and the WP plugin. Thus to perform the analysis using minimal contracts we had to slightly adapt the source code.

The reason why the AES and CCM modules need function pointers is that they must give the ability to replace the implementation proposed by the operating system with the hardware implementation proposed by the targeted platform when such a function is available, for performance reasons. So, instead of directly calling the function provided by the implementation of the OS, the module uses

some function pointers that either point to this implementation or to the hardware function provided by the platform.

In this verification, we want to be sure that the implementation provided by the operating system is runtime error free (we do not expect to verify the hardware platforms). Thus we simply replaced the calls that were using functions pointers with the corresponding actual implementation in the operating system.

Furthermore, in order to ease some part of the verification, we replaced some calls to the `memset` C function with an inline code that does the same job.

4.2.2 Verification process

The verification process is the same as the one described in Section 4.1.2. One difference is the fact that we also verified some caller examples built from the examples provided by the IEEE standard corresponding to the AES-CCM protocol IEEE 802.15.4 [3].

4.2.3 Results

This module was verified by Alexandre Peyrard [4]. The absence of runtime errors has been established using the WP plugin of Frama-C. The verification of the 255 lines of code that compose the module required to write 103 lines of ACSL annotations. Some tests that were written previously as unit-tests of the module have also been specified and verified with Frama-C and its WP plugin.

4.3 Absence of runtime errors in the AES-CCM module (abstract interpretation)

While we succeeded in verifying the absence of runtime errors with Frama-C and WP for the AES-CCM module, we also experimented the verification of absence of runtime errors with the EVA plugin of Frama-C.

4.3.1 Preparation

We analysed the module in isolation from the rest of the operating system. For this, we took the source written for the verification of the AES-CCM module with minimal contracts. We then generalized the tests in order to get verification independent of any input data.

For example, for the AES module alone, from the following test:

```
uint8_t key[16] = { 0x00 , 0x01 , 0x02 , 0x03 ,
                    0x04 , 0x05 , 0x06 , 0x07 ,
                    0x08 , 0x09 , 0x0A , 0x0B ,
                    0x0C , 0x0D , 0x0E , 0x0F };
uint8_t data[16] = { 0x00 , 0x11 , 0x22 , 0x33 ,
                     0x44 , 0x55 , 0x66 , 0x77 ,
                     0x88 , 0x99 , 0xAA , 0xBB ,
                     0xCC , 0xDD , 0xEE , 0xFF };
aes_128_set_key(key);
aes_128_encrypt(data);
```

Figure 2 - The AES test proposed in IEEE 802.15.4

We built the following generalized test:

```
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);
```

Figure 3 - Generalized test for AES-CCM

In this test, the content of the key and the data, while being initialized, can have any (valid) value. Thus, when we analyse this program we are sure that if the plugin answers that there is no runtime error in the module, it does not depend on the content of the considered data.

4.3.2 Verification process

The EVA plugin of Frama-C had a push button approach here: we parameterized the analysis with the advised level of precision of the analysis and ran the plugin.

4.3.3 Results

The absence of runtime errors has also been proved using this method. While the minimal contracts where indeed (relatively) easy to write for this module, the EVA analysis remained more efficient here. We will further discuss this aspect in Chapter 5 about lessons learnt.

4.4 Absence of runtime errors in the whole OS (abstract interpretation)

The EVA plugin is meant to scale on large code bases. Thus we wanted to run this plugin on a complete Contiki instance in order to verify the absence of runtime errors on complete OS instances.

An instance of Contiki is basically a configuration of the operating system, on a particular hardware platform, with some applications (defined as processes) on top of it. All these components are linked together during compilation. Thus, to analyse such an instance, we have to collect what is necessary to build it (which is indicated by the Makefile) and to give those files to Frama-C.

4.4.1 Preparation

4.4.1.1 Integration of the Frama-C analysis script

The common way to verify real world software with Frama-C and the EVA plugin is to build a Makefile based on the one available in the corresponding project and to provide some rules for analysis. Here, we decided not to create a new Makefile but to directly integrate the Frama-C analysis scripts and Makefile rules to the original Makefile of Contiki to be more robust against the modification that could happen in Contiki. That happened to be efficient since porting this work from Contiki 3.0 to Contiki-NG took less than a day, and we did not need to adapt our configuration due to the updates that have been done on Contiki since we ported the script.

The Makefile provided in the folder `analysis-scripts` of Frama-C provides well configured rules for parsing, analysis with Frama-C and EVA, and some other plugins of Frama-C, as well as some predefined options that allow good results by default. Except for the `clean` rule that we modified to allow extension, we have not modified the existing rules of the Makefile. We only added new independent rules and pre-processor instruction that are specific to the Frama-C target.

4.4.1.2 Selection of a platform and adaptation of the code

As numerous platforms are available for Contiki, we started with the most popular one. For the analysis, we currently target the CC2538DK[2] platform which is a common platform for the users of Contiki. The configuration of the platform is either the default one provided by Contiki or the one provided (if it is) by the configuration of the specific instance we are working on.

Some special parameters are set to allow the parsing of Contiki by Frama-C. We added the path to the headers of the ARM-EABI distribution to the include path in order to avoid some type definition conflicts. We also forced the version of ARM to ARMv7 (which is the version supported by the Cortex-M3 on which CC2538DK is based).

Second, we had to replace some hardware addresses in the source of Contiki when we analyse it with Frama-C. Indeed, the device drivers define a lot of physical addresses that allow interacting with the different devices of the platform. However, such physical addresses are considered as bad memory locations by default in Frama-C. An option of the kernel allows indicating to Frama-C that those are valid addresses. However, it is currently not precise enough (we can only give a single range of values, while we would like to have multiple ones). While Frama-C could be easily improved on this aspect, we decided, in a first time, to use volatile arrays to "simulate" those devices and to re-define the physical addresses as offsets in the array.

4.4.1.3 Preparation of an instance

Preparing an instance for analysis only requires the user to add a file `analysis.mk` in the folder of their project. Basically, this file is used to specify for each sub part of the project (if any), which files are necessary to compile it. For example, for the RPL-UDP example, two submodules exist: one for the server and one for the client. So the configuration file is as follows:

```
ifeq ($(SUB), client)
  FC_PROJECT_FILES=udp-client.c
endif
ifeq ($(SUB), server)
  FC_PROJECT_FILES=udp-server.c
endif
```

We consider the above simple enough to be usable. Once configured, the analysis is run by using the following commands:

`$ make frama-c.parse TARGET=cc2538dk SUB=client` (parsing)
`$ make frama-c.eva TARGET=cc2538dk SUB=client` (Eva analysis)
`$ make frama-c.eva.gui TARGET=cc2538dk SUB=client` (start Frama-C's GUI)

Note that it is necessary to indicate the platform for now even if it is only one that we support because we plan to provide similar way to configure the Makefile for other platforms.

4.4.2 Results

4.4.2.1 Analysis of alarms

We were able to run Frama-C/EVA on the different examples available in the Contiki repository. However, we have not classified reported alarms as false or actual alarms, as we noticed that most of these alarms appear when the code uses the linked list and MEMB modules. Lists involve a lot of indirections since it is a linked data-structure and while it is better handled by Eva than by WP, it is still a complex analysis to perform. The MEMB module performs upcasts from `void*` to the types expected by the client modules, which tends to be complex to analyse in some situations (in particular in combination with lists).

To get those results, we had to make sure that the debug mode was deactivated in Contiki, because part of the code, which was triggering some fatal warnings have prevented us to really analyse the

[2] http://www.ti.com/tool/CC2538DK

actual features of Contiki. We did not get further on this aspect of the analysis as we detail in Section 4.4.2.3.

4.4.2.2 Dealing with recursion

One blocking problem currently is the fact that the scheduler of Contiki is a recursive algorithm. Basically, the execution of a Contiki process is resumed using a function named `call_process`, if the corresponding process has to stop, `exit_process` is called which in turn can also use `call_process` on the other processes to make them react to the end of the first process. However, recursive calls are not yet supported by Frama-C/EVA, resulting in a degeneration of the analysis. We can force the analysis to continue, but the results are not sound.

We identify two ways to deal with this part of the code. Either to use a new option of Frama-C called inline-call, or to rewrite this part of the code without recursion.

The first option consists in syntactically inlining a call. For example, here, we would replace the calls to `call_process` with its code. That would not totally remove recursion, but since this recursion is finite, we could repeat the operation until we reach the end of the recursion because the number of processes is statically defined. If we cannot reach such an end, it probably means that something bad is happening in this part of the code or that our precision is not sufficient during the analysis.

The second option is to rewrite this part of the code. However, it is a complex task that is in a highly critical part of Contiki. Moreover, we would have to compare the behaviours of the current and new code to ensure that they have the same behaviour, which is hard, due to the lack of unit-tests in the Contiki project.

4.4.2.3 Difficulties with the configuration of the instances

The building process of Contiki (and the different Makefiles that it involves) is more complex that it might appear at first sight. During the second part of the project (M18-M36), we worked on two new aspects, first precisely identifying the software/hardware interface of the operating system and second being sure that we were dealing with possible instances of the operating system.

The hardware platform API is under-documented. In fact, even if in Contiki-NG some functions are now indicated to be necessary to implement in order to provide a particular hardware implementation that can run Contiki, for some specific tasks, it is not the case, mainly for tasks that require the operating system to communicate with drivers. In order to identify these interactions, we modified one of the scripts of the continuous integration of Contiki to catch the symbols that are associated to platform specific code or platform independent code.

These steps helped us to reveal some subtle bugs and code smells in the operating system and to discover that some pre-processor directives were not correctly set in different parts of the operating system and could lead to incoherent instances. Thanks to these different experiments, we were able to correctly parse the operating system with Frama-C. However, due to the size of the code to analyse and maybe to other changes in the new versions of Frama-C and EVA, the analysis on this new instance become extremely long to execute.

At this point, we decided not to invest more time in this analysis in order to experiment runtime verification and the ability of the VESSEDIA tools to instrument and execute instrumented instances of the operating system.

4.5 Absence of runtime errors in the whole OS (runtime verification)

In the Contiki code base, different tests are executed in simulation. The simulation can take two different forms: either the code is compiled to x86 machine code where the different input and output operations are bound to the libraries of the host system (and thus we basically execute Contiki as a "normal" program on top of Linux) or the code can be compiled for the Cooja simulator on which we can build networks of Contiki instances and simulate interactions between the different nodes. The latter is of course the most interesting as it allows executing a bigger part of the operating system.

In order to increase the level of confidence provided by these tests, a possibility is to add, using Frama-C and its RTE plugin, some annotations to check the different operations that can lead to a runtime error and to use the E-ACSL plugin of Frama-C to generate executable code from these annotations in order to check that everything is fine at runtime.

In this section we report on the experimentation we have made in this direction and the limitation we have met in the case of the Cooja simulator.

4.5.1 Preparation

In order to instrument an instance of Contiki, we need to parse the corresponding source code using Frama-C, most of the work was already done for the work we presented in Section 4.4. From this source code, Frama-C can generate a single C file containing all the source code of the instance (as well as the annotations that could have been added by the different analysers).

Once this standalone file is generated, it can be instrumented by the E-ACSL plugin. The configuration of the plugin is somewhat standard. The only little subtle was to use full memory model of E-ACSL, since optimizing the runtime checking in the case of Contiki is complex (Contiki does a lot of tricky operations with memory), thus optimization only costs time without bringing any advantage.

Now, it is necessary to distinguish two cases:

- Instrumentation for an execution on top of Linux,
- Instrumentation for an execution on top of Cooja.

The first case is mostly standard, in fact the only problems we had to face were due to the compilation process of Contiki that was including files during the compilation process that were not necessary. While the linker was able to remove the corresponding symbols, once instrumented it was not possible anymore and some other symbols (dependencies) that were not included were then not found during the link.

The second case is more complex. In Cooja, the execution of the instances is done by loading the operating system as a shared library, the different high level functions of the operating system are then called from Cooja at each tick of the timer. Cooja is written in Java, while Contiki is written in C. Thus, a particular target of Contiki is dedicated to Cooja and includes Java Native Interface (JNI) functions that can be called from the Java code of Cooja. The instrumentation of libraries is not the most common use case of E-ACSL, so we had to explore how to do it correctly.

We proceeded in two steps. First, we took back the work that was done to execute Contiki on top of Linux, and we slightly modified the example in order to build it as a library that we loaded and executed from a simple C code. And once this work was done, we started to adapt the Contiki Cooja target with the same method.

To be able to perform the instrumentation of the memory and in particular to build the shadow memory, E-ACSL needs to know the location of different parts of the memory layout, namely the stack, the thread local storage and the global storage (in the case of Contiki, heap is not a concern since there is no dynamic allocation). Since we cannot directly obtain all information by looking at the variables of the program, we modified E-ACSL in order to load the locations of the different storages from the memory map of the executable as provided by the Linux `/proc/PID/maps`.

Then, we tried to transpose this method to the Cooja simulator. That requires more configuration on the E-ACSL side. Indeed, we want to check the behaviour of the operating system. However, as we previously mentioned, the Cooja target adds some JNI functions that we do not want to instrument since they are not really part of the operating system itself. The E-ACSL plugin has thus been improved so that we can exclude some functions in the instrumentation process. Consequently, we were able to exclude the JNI functions. Finally, we met some blocking problems with this instrumentation that we will further detail it in Section 4.5.3.

4.5.2 Verification process

The verification process is simple. In fact, it basically consists in executing the existing tests of Contiki except that now every runtime error whose detection is supported by E-ACSL is necessarily detected.

4.5.3 Results

4.5.3.1 Runnable Examples

We were able to instrument three different examples from the Contiki code base to run them on top of Linux. These three examples are:

- The Contiki hello-world (that comprises a pingable IPv6 stack),
- The CoAP example server,
- The MQTT client.

Now these examples should be stress tested with the instrumentation to determine their robustness.

4.5.3.2 Detected problems in Contiki and the E-ACSL plugin

The parsing and building phase of the instrumented operating system allowed identifying different problems in both the operating system and the tool.

In Contiki, this is mainly related to the building process of the operating system that compiles more code than needed because of some missing static configuration parameters.

We also met some bugs during the generation of the E-ACSL instrumentation, which have now been fixed by the Frama-C team.

4.5.3.3 Instrumentation of libraries with E-ACSL

Instrumenting and loading libraries with E-ACSL allowed identifying new opportunities to improve the E-ACSL plugin, and more precisely the handling of shared libraries. Loading external libraries, either instrumented or not, makes the tracking of the memory more complex. In the case of Contiki for example, the original way of tracking the different memory storages was not compatible with the loading of the operating system as a library.

Basically, it seems that some kind of introspection is necessary in order to have as precise memory locations as possible.

4.5.3.4 Problems related to the loading of libraries from Cooja

We were not able to execute an instrumented instance of Contiki in Cooja. In fact, as we mentioned before, we need some kind of introspection to be able to determine where the different storages are in memory. The basic version (without Cooja) uses the mapping provided by the host operating system for this purpose. However, such a method was finally inapplicable in the case of a loading from Java.

Indeed, if we collect this information, we get the memory layout related to the Java Virtual Machine (JVM) and not the one related to the executed program. For this, the JVM creates some memory allocation in the heap and then manages the execution stack by itself (and there is probably an equivalent mechanism for the different other storages). The location of this allocation cannot be obtained from the JVM, and that would be considered as a security breach since one could probably attack a Java program thanks to this information. Knowing the location of the different storages of the library should however not be critical (note that it should be carefully checked) at least if this information remain in the memory of the shared library.

Thus, E-ACSL probably needs to create a more integrated introspection that does not rely on the host program/operating system, in order to be as independent as possible from them.

4.5.3.5 General conclusion on runtime verification of the whole OS

While we succeeded in executing the instrumented operating system on top of Linux, it did not allow identifying any critical bug. Indeed, most of these tests are not intensive enough to really stress test the system and visit a large part of the code of the operating system.

It would have been more efficient to be able to run dozens of instances of Contiki to push the OS to visit different parts of the code, but the unexpected difficulties due to the JVM would require quite a lot of work to find a robust solution.

4.6 Dataflow integrity

4.6.1 Preparation

Instead of directly instrumenting the Makefile of Contiki with the rules and options related to the Flowguard tool, we took advantage of what we had already done with Frama-C for the generation of instrumented files. That is to say: we used Frama-C to directly generate a standalone file containing the complete code of the operating system.

Once this file is generated, the Makefile provided with Flowguard can be used directly.

4.6.2 Results

We were able to instrument the same examples mentioned in Section 4.5.3.1, again these examples should be stress-tested with this instrumentation.

4.7 The linked list module (deductive and runtime verification)

In this section, we present four published studies about the verification of the linked list module of Contiki. The first one is the use of a companion ghost array to model the linked list and the corresponding formal verification performed with Frama-C/WP [5], the second is the adaptation of the corresponding ACSL specification to make it executable using the E-ACSL plugin of Frama-C [6]. As this verification was hard to perform we tried to improve it through two different ways: we have made another proof with another approach using logic lists for modelling [7]and we have improved the ghost array version in order to remove the need for interactive proof [8].

4.7.1 Overview of the linked list module

The list module is required by 32 modules and invoked more than 250 times in the core of the OS. The linked list module is a crucial library in Contiki. Its verification is thus a key step for proving many other modules of the OS.

```
1  struct list {
2    int k; // a data field
3  };
4
5  typedef struct list ** list_t;
6  //Initialize a list
7  void list_init(list_t pLst);
8  //Get the length of a list
9  int  list_length(list_t pLst);
10 //Get the first element of a list
11 void * list_head(list_t pLst);
12 //Get the last element of a list
13 void * list_tail(list_t pLst);
14 //Remove the first element of a list
15 void * list_pop (list_t pLst);
16 //Add an item to the start of a list.
17 void list_push(list_t pLst, void *item);
18 //Remove the last element of a list.
19 void * list_chop(list_t pLst);
20 //Add an item at the end of a list.
21 void list_add(list_t pLst, void *item);
22 //Duplicate a list (copy head pointer)
23 void list_copy(list_t dest, list_t src);
24 //Remove element item from a list
25 void list_remove(list_t pLst, void *item);
26 //Insert newitem after previtem in a list
27 void list_insert(list_t pLst,
28   void *previtem, void *newitem);
29 //Get the element following item
30 void * list_item_next(void *item);
```

Figure 4 - The linked-list API of Contiki

The API of the module is given in Figure 4 - The linked-list API of Contiki. Technically, it differs from many common linked list implementations in several regards. On one hand, while in most implementations a function of the API receives a pointer to the first element of the list and returns the modified list, in Contiki the API receives a double pointer, which is a pointer to some handler (which is a pointer to the first element of the list) that identifies an existing linked list. Thus, rather than returning the new list after some modification, the function directly modifies the pointed handler. In Figure 5 - A linked list in Contiki, the pointer that we give to the function is `pLst`, it points to the handler `root` that itself points to the first element of the list `A`.

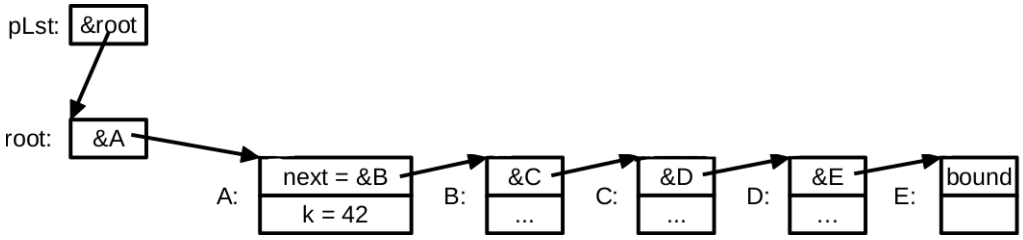

Figure 5 - A linked list in Contiki

On the second hand, being implemented in C (that does not offer templates), Contiki uses a generic mechanism to create a linked list for specific field datatypes using dedicated macros. The pre-processor transforms such a macro into a new list datatype definition. To be applicable for various types, the common list API treats list elements via either `void*` pointers or pointers to a trivial linked list structure, and relies on (explicit and implicit) pointer casts.

Third, Contiki does not provide dynamic memory allocation, which is replaced by attributing (or releasing) a block in a pre-allocated array. In particular, the size of a list is always bounded by the number of such blocks, and their manipulation does not invoke dynamic memory allocation functions.

Fourth, adding an element at the start or at the end of a list is allowed even if this element is already in the list: in this case, it will first be removed from its previous position. Finally, the API is very rich: it can handle a list as a FIFO or a stack, and supports arbitrary removal/insertion and enumeration.

For all these reasons, the linked list module of Contiki appears to be a necessary but challenging target for verification with Frama-C/WP.

4.7.2 Preparation

Here, we are interested in the verification about functional properties of the linked list module, thus we need to provide contracts and invariant written in ACSL to specify the behaviour of each function. However, finding the good representation to reason about the different elements of a linked list is not this trivial as linked data-structures are known to be hard to handle. We experimented two different ways to model the lists. The first one relies on ghost code, the second on ACSL logic lists.

4.7.2.1 Using ghost arrays

Ghost Code

Before we describe the approach we used for the verification, it seems to be important to introduce briefly what is ghost code. Ghost code is, in Frama-C (and ACSL)[3], some regular source code that is introduced in the code we want to verify using ACSL annotations. The purpose of this type of annotations (that contain standard C code) is to ease the verification by saving some useful information that require computation in some variable, thus, transforming some implicit properties (that would require potentially a lot of reasoning) into explicit properties since they are directly modelled by variables. Thus, ghost code can *observe* the actual source code but must not modify its behaviour.

[3] Ghost code is a general notion in verification, and many tools propose such a feature, here we concentrate on how it is used in Frama-C

A ghost variable can be modified and read in ghost code, it cannot be used (neither read nor written) in the actual code. An actual variable can be read (observed) in ghost code but cannot be written since it would mean that we are modifying the behaviour of the program from ghost code.

For the verification, we use a ghost array to model the list we are considering in the different functions. Some ghost code is added to mirror the operations that are done in the actual code.

Ghost Arrays to model linked lists

Since reasoning about arrays is common and quite easy using Frama-C/WP, an appealing idea is to model lists using companion ghost arrays that contain the elements in the list at any moment.

Reasoning about linked data-structures such as lists generally requires to reason by induction since we, only knowing the C structure, do not have a global view of the data structure. For example, with lists, the C structure only mentions the first element of a given list and then refers to the other elements using a pointer. So, we basically do not know how many of them exist. Thus, to state a property *P* about a list, we have to consider two cases: either the list is empty or the list contains an element. In the second case, we generally say that some property holds about the first element and then that *P* must hold on the remaining part of the list.

For verification, Frama-C/WP relies on Satisfiability Modulo Theories (SMT) solvers, in order to maximize automation. However, reasoning by induction is not possible for most of the SMT solvers, thus inductive properties are generally not handled very well. Here, we propose to define equivalence between the list and an array. Defining this equivalence still requires writing an inductive predicate, but once it is done, most interesting properties can be expressed on the array without requiring induction.

Figure 6 - A linked list modelled with a ghost array illustrates the idea of the equivalence we state. For a list starting with the cell A, and ending at an (excluded) bound, we determine a starting location in the companion ghost array that models it. Each consecutive elements of the list must be (contiguously) found in the companion array in the same order as we can find them in the actual list.

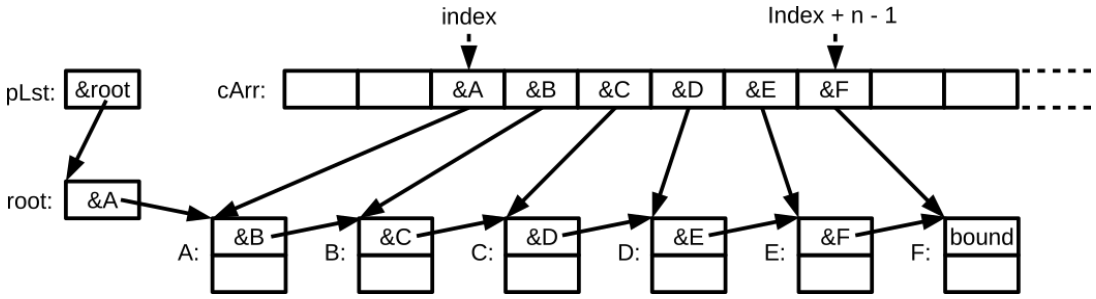

Figure 6 - A linked list modelled with a ghost array

Of course that means that any modification performed in the list must be reflected in the companion array. For example, an operation like `list_push` first ensures that the element in not in the list by removing it (if it is not inside, the operation keeps the list as it is) and then adds it at the beginning of the list. Thus we have to perform similar operations in the companion array.

The verification mainly consists in proving that the equivalence is always maintained during the different operations on the list, which allow deriving the important properties of the list API:

- Removal operations only removes the provided element,
- Adding an element to the list ensures that the element is at the expected location,
- Items unicity and validity is maintained.

In order to ensure that the equivalence relation is maintained, we have to reason about an inductive property. To maximize automation, we have added 24 lemmas that are proved by induction interactively with the Coq proof assistant, allowing avoiding inductive reasoning for all the other proofs.

More details can be found in the article accepted at NASA Formal Methods 2018 about this verification [5].

While verifying the module with the ghost arrays approach allowed us to verify most functions of the module (see details in Section 4.7.4), dealing with separation was a hard task. The reason is mainly that ghost arrays are not differentiated of the actual memory of the program, thus requiring from us to write a lot of guiding annotations to guide the proof. Thus a more abstract model of the list could be considered.

4.7.2.2 Using logic lists

Since modelling linked lists with arrays produced a too concrete model (thus hard to differentiate from the actual data-structures for the verifier), one could use a more mathematical view. Recently, the support of ACSL lists has been greatly improved in the WP plugin of Frama-C, thus we tried to perform a similar verification using this logic type.

ACSL logic lists

This datatype, denoted by `\list<type>`, is parameterized by `type`, the type of a list element. It has two usual constructors: `\Nil` (also written `[]`), the empty list, and `\Cons(type, \list<type>)`, that builds a new list `\Cons(e, l)`, also denoted by `e::l`, from a given list `l` and an element `e` added at the beginning. Instead of `\Cons(item, \Nil)`, a singleton list can be written `[|item|]`. It also provides some other features, for example concatenation (denoted by `l1 ^ l2`), length, n^{th} element, etc. In our case, we use logic lists of type `\list<struct list*>`, that will contain the addresses of elements of the linked list.

Logic lists have the advantages to make specification more concise (often avoiding to create universally quantified variables) and to be natively handled by most SMT solvers thus leading to efficient automatic proof.

Logic lists to model linked lists

Again because of the recursive nature of the C data-structure, we need to state the equivalence between the C structure and the logic list that models it using an inductive definition. Figure 7 - A linked list modelled with a logic list gives again the general idea of the equivalence which is quite similar the ghost array version. The two main differences are first the fact that we do not have to consider any index and second, of course, that we use here a logic type and not an actual C type.

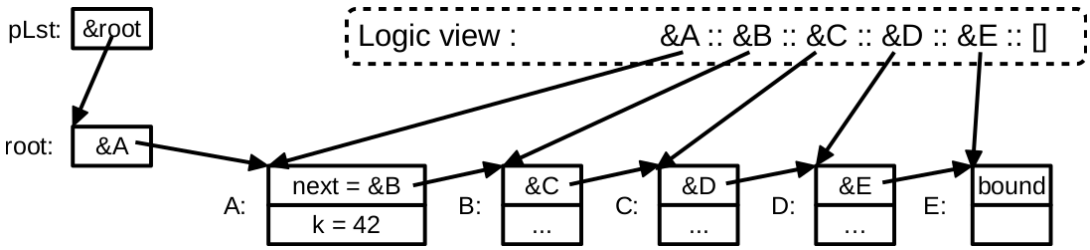

Figure 7 - A linked list modelled with a logic list

On the aspect of the specification, this method has one small drawback. While with the ghost approach it was easy to bind the linked list to its modelling value through ghost variables, we do not have this possibility with ACSL logic lists. Most specification languages allow to quantify variables on a complete contract for this purpose, this is however not the case of ACSL. Thus, we created a logic function that can build, from a C linked list, the corresponding logic list. While writing this function is quite easy (and is in fact close to the inductive equivalence predicate), it forced us to write more lemmas to help the verification as we will detail in Section 4.7.3.1 and further in Section 4.7.4.2.

More details can be found in the article accepted at the 34th Symposium on Applied Computing (SAC 2019) [7].

4.7.3 Verification process

In this section, we detail some aspect of the verification process. The Section 4.7.3.1 does not aim at being complete. We detail some aspects of the methodology that are necessary to understand

the results exposed in Section 4.7.4. The two remaining sections gives however some more details about two methods that have been developed during the verification of the module.

4.7.3.1 General remarks

When verifying such a module that requires inductive predicates it is common to use lemmas to help verifiers. Indeed, reasoning about these predicates generally requires to reason carefully by induction, and SMT solvers are bad at this work.

Thus, a common way to proceed, in order to maximize automation is to state some lemmas that allows to directly deduce properties from the inductive predicates. These lemmas can be directly instantiated without requiring reasoning by induction (thus they can be used efficiently by SMT solvers), while the proof of these lemmas (requiring reasoning by induction) are done using a proof assistant like Coq [9]. The idea is then to declare the minimal set of lemmas that are sufficient to complete the rest of the proof automatically, as writing an interactive proof is time consuming task.

4.7.3.2 Auto-active proof

As interactive proof is a time consuming task, we want to get rid of it. A way to do that is to use what is called "auto-active proof". In fact, auto-active proof is already the method that is classically used to prove programs with Frama-C and WP, as it consists in providing additional annotations (and note only function contracts) to guide the verifier. However, most of the time, only assertions and loop invariant are used in the case of Frama-C and WP. But, one can go further with auto-active proof by using more ghost code in order to provide for example induction schemes to prove inductive properties. A classic example used in other tools is to create what we call "lemma functions" that are basically ghost functions that act as lemmas that we can call at the right place to ease the proof process.

By using this method, we were able to perform the entire proof performed with the ghost arrays without writing any Coq proof, thus maximizing the automation of the proof process. We also succeeded in doing it for the MEMB module (presented in Section 4.8) and the book ACSL by Example [10].

For more details about this work, please refer to [8].

4.7.3.3 Runtime verification

While the formal verification of the module was necessary, verifying completely client modules with WP might be too hard, or unnecessary. For this one could rely on runtime verification. However, some constructs used in the specification (namely inductive predicates and axiomatic definitions) are not currently handled by the E-ACSL plugin of Frama-C. For them, we need to find a solution to get an executable code.

Instead of providing a way to generate an executable code from inductive and axiomatically defined properties, we propose to define a new version of the specification that is executable and to prove its equivalence with the previous one. In the original specification, two properties were defined using ACSL constructs that are not supported by E-ACSL: the equivalence between the list and its companion array (using an inductive predicate), and a function that specifies the index of a particular element in the list (using an axiomatically defined function).

For both of these properties, we define:

- A new predicate/function expected to be equivalent to the one used for the proof,
- A lemma that states the equivalence between the non-executable and executable version for any input.

The corresponding lemma is interactively proved using the Coq proof assistant. We then build a new specification for the linked list module which is the same as the previous one except that we have replaced each occurrence of the features that cannot be translated by the equivalent executable versions. As a result, we obtain an executable specification without breaking the previous proof results. For example, starting from the inductive predicate `linked_n` that inductively defines the equivalence between a list and its companion array, we created a new version `linked_n_exec`,

which is proved to be equivalent. In the specification, each use of the `linked_n` predicate is replaced by a use of the `linked_n_exec` predicate.

Note however that this approach is currently not directly supported by E-ACSL. Our proposal in [6] is to provide a way in E-ACSL to specify that we need an equivalent version of some property in order to let the tool generate the required lemmas and to automatically replace occurrences of the property in the specification during the generation of the executable version of the specification.

4.7.4 Results

4.7.4.1 Bug found and fixed in `list_insert`

The verification helped to identify an inconsistency in the function `list_insert`, with respect to the assumptions of other functions. This function adds an element `new_item` to a list just after a given element `prev_item`. If the element `prev_item` is `NULL`, the function directly calls `list_push`, meaning that if the element is already there, it is removed and then added to the start. However, if `prev_item` is present in the list, the function directly adds `new_item` after `prev_item` without removing a previous instance of `new_item` from the list (if any). It shows that the uniqueness property is in general not preserved by the function, but in some cases it is. Thus this function does not respect a contract consistent with the other functions.

The issue can be found on the GitHub of Contiki-NG[4].

In the entire code of Contiki, we have found only one call to `list_insert` and not a single one in the core part of the system.

Unit tests may identify such a bug. However, one difficulty with functional tests (that is also the cause of many security bugs) is the fact that we tend to test valid scenarios rather than invalid ones.

4.7.4.2 Comparison of the methods

Proved functions

Table 2 - Summary of proved functions with each method

Function	Ghost arrays	Ghost arrays AA*	Logic lists
`array_pop`	Proved	Proved	Unnecessary
`array_push`	Proved	Proved	Unnecessary
`array_find`	Proved	Proved	Unnecessary
`list_add`	Proved	Proved	Proved
`list_chop`	Proved	Proved	Proved
`list_copy`	Proved	Proved	Proved
`list_head`	Proved	Proved	Proved
`list_init`	Proved	Proved	Proved
`list_insert`	Not proved	Not Proved	Proved
`list_item_next`	Proved	Proved	Proved
`list_length`	Proved	Proved	Proved
`list_pop`	Proved	Proved	Proved

[4] : https://github.com/contiki-ng/contiki-ng/issues/254

Function	Ghost arrays	Ghost arrays AA*	Logic lists
`list_push`	**Proved**	**Proved**	**Proved**
`list_remove`	**Proved**	**Proved**	**Proved**

*** AA: auto-active.**

It is important to notice that "`array_*`" function do not need to be proved for the logic lists version, indeed as we do not use ghost variables we do not have to update them and consequently no function is needed for that.

Furthermore, while it was expected that `list_insert` would be too hard to prove with ghost arrays because of memory separation, we succeeded in proving it using the ACSL logic lists.

Proof efficiency

Table 3 - Summary of proof figures (excluding the `list_insert` function)

	Ghost arrays	Ghost arrays AA	Logic lists
Generated goals	**805**	**1631**	**503**
Interactive proofs	**24**	**1**	**33**
Automatic proofs	**781**	**1630**	**470**
% of automation	**97**	**100 (*)**	**93**
Total time	**24 min**	**21 min**	**5 min 30**
Time per goal	**1.8 s**	**0.77 s**	**0.7 s**

To measure proof efficiency, we exclude the `list_insert` function that was not proved with the versions with ghost arrays. For the ghost arrays + auto-active version, we consider a hundred percent proof automation. Indeed, while one interactive proof appears, the corresponding proof is in fact a Coq "one-liner" that relies on the structure of inductive predicate, that is to say: it is based on an axiom. Thus, such a proof should be done for free by WP since it is trivial to generate.

Discussion

The proof with logic lists was the most efficient. Indeed, this verification generates the less proof obligations and at the same time, the automation remains quite good and the automatic proofs are fast. It is coherent with the fact that this proof requires less guiding annotations and the fact that the logic lists are natively handled by SMT solvers, making their job easier. However, this method is also the one that requires the most interactive proofs and these proofs were harder to perform than the ones we had to do for the proof with ghost arrays without the full auto-active methodology.

Auto-active proof allowed us to reach a full automation of the proof. However, this version is also the one that requires the most guiding annotations. Note that while the figures seems to indicates that it doubles this work compared to the base version, it is in fact not the case, for more details we refer to the corresponding article [8].

Determining if the ghost arrays with full auto-active proof or the logic lists version is the more suitable for an analysis, that is to say: concrete model or mathematical model, is a matter of skills available in the verification team. Basically, in a team where nobody masters a proof assistant, the concrete approach is probably the most suitable as it allows to reach a complete automation without requiring to learn the use of another tool in addition to Frama-C and WP. In a team where an expert of Coq or Isabelle is available the mathematical view would probably allow to get faster results, and simpler

specification. Furthermore, some proofs that are today extremely hard with ghost arrays (in our case, the `list_insert` function) were realized quite easily with the logic lists approach.

4.7.4.3 Ghosts and separation

Most of the annotations we had to write to guide the proof were due to memory separation (as opposed to VeriFast, Frama-C/WP does not use separation logic). While a part of these required annotations, namely the separation between the different elements of the list, were expected to be necessary, some of them were unexpected and unnecessarily complex to handle. Indeed, we often had to state that the ghost memory (that represents the list) and the elements of the list are separated.

While for verification, this need is inconvenient, from the current Frama-C's implementation of ACSL, it is necessary. Indeed, ghost code must not interfere with the actual code, but this fact is not checked by Frama-C. Thus, a tool that has to deal with ghost cannot just assume that the separation exists: this has to be checked explicitly.

What we propose is a refinement of the notion of ghost in Frama-C. Instead of indicating that some variable is ghost or not, we would be able to define more precisely what belong to ghost or not even in a single variable, essentially in the same way we can use the keyword `const` in C.

For example, in:

```
//@ int * ghost p1 ;
//@ int ghost * ghost p2 ;
```

The pointer `p1` is a ghost pointer to a memory location that is not ghost. So in a ghost code, `p1` can be modified but the pointed memory location `*p1` cannot. On the opposite, `p2` is a ghost pointer that points to a ghost memory location so both `p2` and `*p2` are mutable in ghost code.

A plugin has been developed (and is part of deliverable D2.3) and is able to verify this kind of property using typing. The plugin ensures that the ghost does not interfere with the actual code by a first analysis. And then, we can use the fact that we know that the property is ensured and provides more assumptions to the plugin that further analyses the source code. For example, we could add an assumption in WP that states the fact that ghost and actual memory areas are separated.

4.7.4.4 Logic lists and auto-active proof

It could be appealing to join the best of the three methods that we have used so far, that is to say: using a full auto-active proof approach with logic lists. However, with the current state of Frama-C, this is impossible. Indeed, to be able to perform such a proof, it is necessary to have the possibility to use variables with logic types in annotation, which is currently not possible.

ACSL is supposed to authorize that but currently the kernel of Frama-C (and the abstract syntax tree) cannot accept such annotations.

4.7.4.5 Logic lists and runtime verification

While, thanks to recent improvement of the platform, it is almost directly possible to generate executable code from the specification using ghost arrays with the E-ACSL plugin of Frama-C, it is currently unclear whether it could be done with logic lists as the plugin currently does not support this data-structure.

4.7.4.6 Observation function

Finally, let us elaborate on another modelling possibility. It could be possible to use observation functions to model the content of the list. Instead of building a list, or an array that represents the actual list that is currently manipulated by some function, we define a function that associates to each valid index, the element that can find at this location in the list.

This solution could combine both the advantages of the abstract list and the ghost view. Indeed, while the view is extremely abstract, allowing an efficient proof with automatic solvers if the suitable lemmas are provided, it does not require a complex data structure, thus the specification currently

seems to be easy to transform into an executable specification. However, we have not currently tried to prove mutating algorithms with this approach, so this is still a work in progress.

4.8 The memory allocation module (deductive verification)

The MEMB module is a memory allocation module. It allows allocating and deallocating chunks of pre-allocated memory blocks. The idea is to first allocate a certain number of resources in a static C array and then to acquire and release them on demand. Basically, any module that needs to manage some memory resources relies on this module, making it critical.

4.8.1 Preparation

Frédéric Mangano verified this module during his internship at CEA [11]. This verification was a functional verification performed with the WP plugin of Frama-C, so it required annotating the program with an ACSL specification. The main challenge was to deal with the genericity of the module, since it allows managing any kind of C objects.

The verification of the 77 lines of code that compose the module required to write 150 lines of ACSL annotations, 115 of them being function contracts. Some test client functions were written to validate the specified behaviour. It allowed to check that correct caller functions were indeed proven correct, as well as to check that we were unable to prove incorrect function calls, which gives some confidence in the consistency of the specification.

4.8.2 Results

While most interesting properties have been proved:

- The module correctly keeps track of free and allocated blocks,
- The different blocks of memory are spatially separated,
- We cannot allocate more memory than what we have.

Some properties cannot currently be proved with Frama-C and the WP plugin due to unsupported ACSL features, namely the notion of *fresh* memory location, or the clause *frees* of a function contracts that allows specifying that a function makes previously allocated memory location allocable again.

This work was easy to integrate into Contiki-NG. We only had to update the few proofs that are written in Coq, and that was due anyway to the new versions of Frama-C and not to Contiki.

Chapter 5 Lessons Learnt

5.1 Global analysis of a configurable system

Analysing a highly configurable system is a hard task. And most of the difficulties we had to face for the parts of the use case presented in Sections 4.4, 4.5 and 4.6, as well as some of the remaining aspects that we present in Section 5.4, are due to configuration problems and the extreme complexity of the building process of such a system. While the absence of unit tests on the platform is certainly also a big problem when starting formal analysis of such a system, we strongly believe that an important research topic is the verification of configurable system. Such problems are currently not addressed by the VESSEDIA tools.

5.2 On the aspect of runtime verification and libraries

Our experiment on the runtime verification of the operating system (presented in Section 4.5) allowed identifying some conditions that a runtime verification tool should satisfy when instrumenting a system, and in particular when such a system is, or uses, a library.

It seems to be important that the instrumentation should be local to the verified system as much as possible and that some kind of introspection is necessary to be able to get all required information about the execution without relying on an external tool, or at least not more than the compiler used to build it. Maybe a good solution in the case of E-ACSL, for example, should be to add information using a dedicated plugin to the compiler.

However, remaining local is maybe not so trivial. We build system using libraries in order to interact with them and thus, we have to deal with these interactions that may for example make the frontier (in memory) between those components not as impermeable as we would like.

5.3 The difficulty of deductive verification

While deductive verification remains difficult, we strongly believe that it becomes more and more affordable. SMT solvers are powerful tool and for task that they are usually bad at, either some experimental tools start to provide new solutions (for example new SMT solvers able to perform simple inductive proofs) or the verifiers that rely on them can be used efficiently as powerful workarounds.

The auto-active approach that we have experimented with, while perfectible, seems to be a very promising way of doing deductive proof. Indeed, one of the hardest part with tools like the WP plugin is the fact that the SMT solvers on which they rely are highly heuristics, thus making maintenance sometimes hard when new versions of the tools are published. Auto-active proofs make all of this more predictable since the different steps of the proof becomes explicit in the annotations thus less dependent to the heuristics and easier to read for a verification engineer, without reaching the degree of complexity of an interactive proof.

5.4 Remaining aspects

Some aspects that were originally targeted for this project have finally not been treated.

First, we wanted to provide a generic abstract physical platform corresponding to Contiki on which users could rely for the verification of their client code. This would have been the result of the work presented in both Sections 4.4 and 4.5. However, the difficulties related to the configuration and the build process of the operating system made this task intractable during the time available for this project. Furthermore, we strongly believe that one of the reasons why it is so difficult is the lack of documentation available about these aspects of the code base and that writing this kind of documentation is not the scope of the VESSEDIA project.

Second, protothread have finally not been analysed during the project. Such a verification would have been done with both the creation of a new plugin of Frama-C and the WP plugin, that is to say using deductive verification (as the absence of runtime errors is not necessarily a big deal because of a how short is the protothread module). While this kind of analysis is still extremely appealing, we decided during the project that comparing different proof methods for the verification of a linked data-structure was a more important contribution for VESSEDIA. Indeed, protothreads are extremely specific to the Contiki operating system, thus this result would have been useful only for it and not for a more global audience.

Chapter 6 Summary and Conclusion

In this deliverable, we have reported on the realization of the use case of the Contiki operating system which is an operating system for the internet of things. This operating system supports a lot of platform with important resource constraints, thus resulting in a highly configurable, highly optimized source code.

We have treated different aspects related to safety and security objectives, from verification of the absence of runtime errors to the functional correctness of different modules. Namely, we have proved the absence of runtime errors in small modules of the core of the operating system, as well as in the AES-CCM cryptography module.

An important work has been performed on the complete operating system to check whether the tools scale on such a piece of software. On this aspect, the success is partial. We were able to analyse representative instances of Contiki. But not to use the results to detect bugs due to the too large amount of alarms for the static analysis and due to the fact that we could not cover a large part of the OS for runtime verification during the execution. While the tools could be improved (handling of libraries for E-ACSL, handling of recursive functions for EVA), the main problems do not seem to be related to the used plugins of Frama-C themselves but to the fact that no plugins or tool is currently able to deal with the question of the configuration, this could be a major concern in the future.

We have proved the functional correctness of the MEMB module and the linked list module using different proof techniques. We think that the results we have obtained for these modules (and the future and ongoing work about other techniques) will help future verification and allow verification engineers to choose the best option when starting the verification of a complex module depending of the skills available in their team.

Chapter 7 List of Abbreviations

Abbreviation	Translation
6LoWPAN	IPv6 Low-power Wireless Personal Area Networks
ACSL	ANSI C Specification Language
AES	Advanced Encryption System
AES-CCM	Implementation of CCM that uses AES for security
API	Application Program Interface
CBC-MAC	Cipher Block Chaining Message Authentication Code
CCM	Counter with CBC-MAC
CPS	Cyber Physical System
DDoS	Distributed Denial of Service
DTLS	Datagram Transport Layer Security
E-ACSL	Executable ACSL
GCC	GNU Compiler Collection
IoT	Internet of Things
IPv6	Internet Protocol version 6
JNI	Java Native Interface
JVM	Java Virtual Machine
MCU	Micro Controller Unit
MPU	Memory Protecttion Unit
MMU	Memory Management Unit
OS	Operating System
SFR	Security Functional Requirement
SMT	Satisfiability Modulo Theories
TOE	Target of Evaluation
TSF	TOE Security Functions, typically cryptographic functionality

Chapter 8 Bibliography

[1] German Federal Office for Information Security (BSI), "Common Criteria - Operating System Protection Profile (BSI-CC-PP-0067)," 2010. [Online]. Available: https://www.commoncriteriaportal.org/files/ppfiles/pp0067b_pdf.pdf.

[2] G. Montenegro, M. Corporation, N. Kushalnagar, I. Corp, J. Hui, D. Culler et A. R. Corp, «RFC 4944 - Transmission of IPv6 Packets over IEEE 802.15.4 Networks,» September 2007. [Online]. Available: https://tools.ietf.org/html/rfc4944.

[3] «IEEE Standard for Low-Rate Wireless Networks,» *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011),* 2016.

[4] A. Peyrard, N. Kosmatov, S. Duquennoy et S. Raza, «Towards Formal Verification of Contiki: Analysis of the AES--CCM* Modules with Frama-C,» *RED-IOT 2018 - Workshop on Recent advances in secure management of data and resources in the IoT,* 2018.

[5] A. Blanchard, N. Kosmatov et F. Loulergue, «Ghosts for Lists: A Critical Module of Contiki Verified in Frama-C,» *Nasa Formal Methods,* 2018.

[6] F. Loulergue, A. Blanchard et N. Kosmatov, «Ghosts for Lists: from Axiomatic to Executable Specifications,» *Test And Proofs,* 2018.

[7] A. Blanchard, N. Kosmatov et F. Loulergue, «Logic against Ghosts: Comparison of Two Proof Approaches for a List Module,» *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC 2019),* pp. 2186-2195, 2019.

[8] A. Blanchard, F. Loulergue et N. Kosmatov, «Towards Full Proof Automation in Frama-C Using Auto-active Verification,» *NASA Formal Methods - 11th International Symposium (NFM 2019),* pp. 88-105, 2019.

[9] The Coq development team, «The Coq proof assistant,» [Online]. Available: http://coq.inria.fr.

[10] J. Burghardt, A. Carben, R. Clausecker, L. Gu, K. Hartig, T. Lapawczyk, H. W. Pohl, J. Soto et K. Völlinger, ACSL by Example, 2018.

[11] F. Mangano, S. Duquennoy et N. 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,* 2016.