D5.1

Inria's use case intermediate 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 / D5.1 / 1.0
Work package contributing to the deliverable:	WP 5
Due date:	June 2018 – M18
Actual submission date:	29th June, 2018

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

Abstract:	Inria's use-case intermediate report at M18. Applying the VESSEDIA tools to the verification of the Contiki Operating System, first contributions.
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

Armand Puccetti (CEA)

Gergely Eberhardt (SLAB)

Nathalie Mitton, Rehan Malak (INRIA)

Disclaimer

Executive Summary

IoT (Internet of Things), which denotes connected devices and services, are on a rapid increase, 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 period M7-M18 of the project on the Contiki operating system. This verification is conducted using the Frama-C platform.

We present the changes that happened in Contiki since the beginning of the project, in particular the new version, as well as previous verification effort now ported on this new version, Contiki-NG. We also present the verification effort performed on a critical module of Contiki with Frama-C/WP, the linked list module, and the preliminary work to be able to use Frama-C/Eva on the complete operating system. This report also introduces ongoing and future tasks.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Goal of this 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 depending on the criticality and the complexity of the modules to verify. This will allow 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" and then analysed using the Frama-C WP plugin with the "verification-service" approach developed in Task 3.3, as a fast alternative to value analysis. In this way, we will gather 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 even if some first experiments have been also conducted on the networking stack. We describe this verification effort, as well as ongoing and future work planned for the verification of Contiki.

1.2 Structure of the document

After this introduction, Chapter 2 explains why we switched from Contiki 3 to Contiki-NG (Next Generation) and the differences that exist between those two versions of Contiki. It sums up the new organization of the project and updates the table of priority of verification that we had presented in deliverable D1.2. It also presents previous (or early) verification effort made on Contiki 3 and now integrated into Contiki-NG.

Chapter 3 presents the functional verification of the linked list module of Contiki which is a critical module of Contiki that is intensively used in the core part of the operating system. After introducing the approach, we present the results, some discussion, and ongoing and future work on this module.

Chapter 4 explains how we integrated the Frama-C analysis scripts to Contiki and the first results we get on this aspect of the verification. In particular, it describes a current problem we have to analyse complete instances of Contiki and the possibilities we have to tackle it.

Chapter 5 describes the current status of the hardware application program interface (API) of Contiki and what we want to do with it in order to have platform independent analysis, and to provide the users with useful guidelines to adapt the verification environment to their own use-case.

Finally, Chapter 6 sums up this document and concludes.

1.3 Related deliverables

This deliverable is related to deliverable D1.2 that provides the security requirements for the different use-cases of VESSEDIA including the Contiki OS. It will also influence the different deliverables of Work Package 3 that is related to the enhancement of the tools and methods used for verification.

Chapter 2 From Contiki 3 to Contiki-NG

Recently, Contiki 3 has been forked to Contiki-NG[1]. Since then, Contiki 3 is not substantially updated, for example none of the bugs we have reported is fixed in Contiki 3. 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.

In this chapter, we detail what aspects of the operating system have changed in Contiki-NG, more particularly the differences that exist in terms of features and organization of the project, as well as the impact on the verification tasks we had planned in deliverable D1.2. We also detail some verification previously performed on Contiki 3, and that are now part of the verification of Contiki-NG.

2.1 Added and removed features

From Contiki 3 to Contiki-NG some features have been added, some existing features and examples have been removed. We list the essential features in this section.

2.1.1 Core libraries

In the core libraries, the GCR (Group Coded Recording) module and the Manchester encoding module have been removed. The verification effort with minimal contracts (see 2.3.3) that was invested on GCR is then not relevant for Contiki-NG. The managed memory module (MMem) has also been removed since it was not used a lot in practice.

In Contiki 3, some high-level libraries, meant to be used by client code but not mandatory by most use cases of Contiki, were previously separated of the core part of Contiki. Since they are platform independent, they have been added to the core libraries of Contiki. However, we will not consider them for the verification since they are high level libraries that are not critical, nor used by any use case of the operating system.

Different data-structures have been added:

- Queues and stacks,
- Double linked lists,
- Circular lists (and doubly-linked version).

The verification of queues and stacks will simply consist in the instantiation of the verification performed on the linked list module (queues and stacks functions are direct calls to the linked list module), that we present in Chapter 3, so the specification and verification should be easy.

The double-linked lists module is more challenging, the verification could be inspired of the one performed on the linked list module but some memory separation properties could be hard to maintain with a double-linked data structure. The circular lists module requires a new formalization and thus a new verification, which cannot be directly derived from the verification of the linked list module. This verification will start after we complete our study of the linked list module and associated approaches.

Finally, a module has been added to manage heap memory allocation. This last module seems to be particularly challenging. Indeed, it uses a double-linked list (and this kind of data structure is generally already hard to handle), but does not use the API provided by the double-linked list module to manage it (which means that we cannot directly reuse a verification results of this module, nor the

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

ACSL formalization). In the same time, we have to keep track, in the verification, of the different allocated and freed blocks, which is complex.

2.1.2 System libraries

The only feature that has been removed from the system libraries is the argument handling module that meant to ease the transmission of argument when creating processes and was not used in any example of Contiki instance. Some features related to concurrency have been added to the system libraries, more particularly mutexes and critical sections for multi-core processors. However, since Contiki mainly targets mono-core currently, it will not impact our verification.

2.1.3 Netstack library

The netstack library has mainly been reorganized. The most important change was that the IPv4 features have been completely removed. Some non-standard MAC modules have also been removed, as well as the implementation of the RIME protocol. Again, some features that were previously considered as high level features have been moved into the netstack library. This is for example the case for different app-layer libraries such as HTTP or MQTT.

Most of these features were not considered as critical in our previous analysis, and no verification was attempted on them.

2.1.4 Other significant changes

A lot of examples have been removed from Contiki. It does not impact the verification itself, but it was a good way to test our verification scripts on a lot of different examples. We will probably reintegrate some of them to Contiki-NG.

While it is not relevant for our verification, it is worth noticing that many architectures have been completely removed from Contiki-NG. While Contiki 3 platform directory contained 37 platforms (or variations of platforms), Contiki-NG only has 9 of them.

2.2 New structure of the project

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.

Module	kLOC	Description	Priority
dev	1.3	Platform-independent parts of drivers.	LOW
lib	39.3	Different general purpose librairies	

Module	kLOC	Description	Priority
|- 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.

2.3 Previous verification ported to Contiki-NG

Some verification has been performed on Contiki 3 before and during the beginning of the VESSEDIA project. In this section, we present the different verification use-cases that are now integrated in the verification of Contiki-NG.

2.3.1 The MEMB module 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.

Frédéric Mangano verified this module during his internship at CEA [1]. 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. 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 that we have.

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

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.

This work was easy into integrate in Contiki-NG. We only had to update the few proofs that are written in Coq [2], and that was due anyway to the new versions of Frama-C and not to Contiki.

2.3.2 The AES-CCM* module verification

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.

This module was verified by Alexandre Peyrard [3]. This verification consisted mainly in the analysis of two modules: the AES module and the CCM module. The verified property is the absence of runtime errors and has been established using the WP plugin of Frama-C. This verification was done by using a notion of *minimal-contracts*, introduced in the deliverable D1.1, that is: we do not provide a full functional specification of the module, only what is necessary to ensure the absence of runtime errors.

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. We also executed Frama-C/Eva on the corresponding test, that also validated the absence of runtime errors.

2.3.3 Minimal contracts for a subset of the core part of Contiki

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/WP. The main advantage of this method is that a verification with Frama-C/WP validates any correct use of the module. Whereas a verification using Frama-C/Eva 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 a full functional verification.

The corresponding contracts have been integrated to our verification of Contiki-NG.

2.3.4 Minimal contracts for the SICSLOWPAN module

This module is an implementation of the 6LoWPAN [4] 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, previously described, to ensure security and authentication.

The 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 are a lot of type casting and that some functions are long, consequently, performing the verification requires to add a lot of annotations to guide the provers. Further investigation is ongoing to understand if it is possible to handle this kind of module using minimal contracts.

Chapter 3 Verification of the linked list module

In this section, we present two published studies about the verification of the linked list module of Contiki as well as ongoing improvement to this verification. 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], and the second is the adaptation of the corresponding ACSL specification to make it executable using the E-ACSL plugin of Frama-C [6].

3.1 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 1 - The linked-list API of Contiki

The API of the module is given in Figure 1. Technically, it differs from many common linked list implementations in several regards. First, 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, that 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 2, 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".

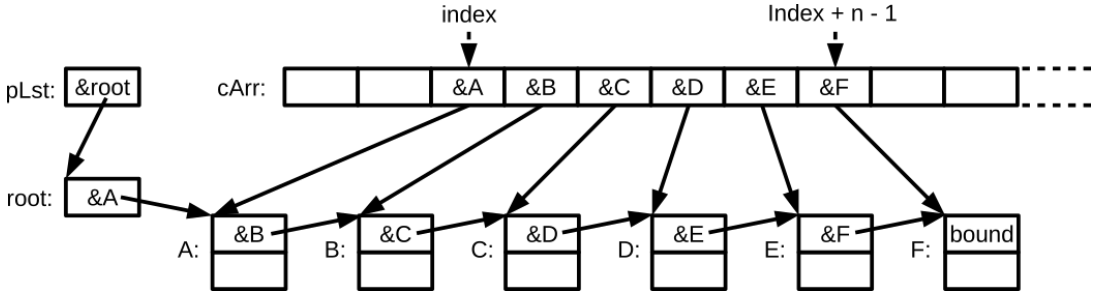

Figure 2 - Ghost model of the actual linked list

Second, 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.

3.2 Formal verification using Frama-C/WP and ghost code

In order to verify the linked list module, we need to formalize the behaviour of the module, thus to find a good representation to reason about the different elements of the list. Our verification relies on ghost code. Since reasoning about arrays is common and quite easy using Frama-C/WP, we model lists using companion ghost arrays that contain the elements in the list at any moment.

3.2.1 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)[2], 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.

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.

3.2.2 Verification approach

Reasoning about linked data-structures such as lists generally requires to reason by induction since we do not have, only knowing the C structure, 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.

[2] 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

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 an 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 2 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.

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, that allow to derive 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 to avoid 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].

3.2.3 Verification results

3.2.3.1 Functional verification of the module

Except for the list insertion function, all the functions of the module have been verified to respect the functional specification we stated. In total, the module is composed of 176 lines of code (excluding macros). We have written 46 lines of ghost code, and about 1400 lines of annotations, including about 500 lines for contracts and 240 lines for logic definitions and lemmas.

For this annotated version of the module, the verification using Frama-C/WP generates 798 goals. This number includes 108 goals for the verification of absence of runtime errors that are often responsible for security vulnerabilities and have also been carefully checked by Frama-C/WP. It also includes 24 auxiliary lemmas (that is, in total only about 3.3% of properties). The 24 lemmas are proved using Coq v.8.6.1. Out of the 774 remaining goals, almost all are automatically discharged by SMT solvers, except for 4 goals that are proved interactively. In this work, we used Frama-C v.16 Sulfur and the solvers Alt-Ergo v.1.30 (with direct translation from WP and via Why3), as well as Z3 v.4.5 and CVC3 v.2.4.1 (via Why3).

3.2.3.2 Bug found in the list insertion function

The verification helped to identify an inconsistency for the remaining 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 at on the github of Contiki-NG[3].

Moreover, 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.

3.3 Making specification executable for E-ACSL

3.3.1 Verification of client modules

As we previously said, a lot of client modules use the linked list module of Contiki. However, we do not necessarily want to formally verify all of them using Frama-C/WP. Moreover, some other users of Contiki could be interested in validating their own code that use the linked list module only using dynamic verification.

Now that we have a precise specification for the module, the checks to perform are well-defined. However, they are written in ACSL and still have to be translated to C code if we want to execute these checks at runtime. This is the purpose of the E-ACSL plugin of Frama-C which can automatically translate any specification written in E-ACSL (which is an executable subset of ACSL) into actual C code.

However, the specification is not in the executable fragment of ACSL. Indeed, it uses an inductive predicate and an axiomatically defined function. While those two features were convenient to reason using WP and the Coq proof assistant (for example providing the right induction principle), they are not in the E-ACSL fragment of ACSL and are not planned to be added. So we have to express these properties with other features, namely recursive predicates and functions. While this kind of properties is currently not implemented, it is planned to be. But, we do not want to lose the verification effort invested in the linked list module, so we use an approach inspired by the verification by code transformation.

3.3.2 Approach to enable E-ACSL

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

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

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.

3.4 Improving the verification

Some aspects of the verification can be improved. Most of them are related to the efficiency of the proof. While possible, the proof was hard to handle with Frama-C and WP so we experiment other ways to perform this proof. At the same time, some parts of the specification are not totally satisfactory since they make the proof of client modules complex. This section is essentially a work in progress.

3.4.1 Assigns specification

In ACSL, a function contract comprises a specific part that is dedicated to the specification of the side effects that are produced during the execution of the function. It allows from the caller perspective to determine what remains unchanged in the memory after a call to the function. This information is provided using the `assigns` clause in ACSL, that lists the different memory locations that *can* be modified by the function.

However, in our case, this specification is imprecise. Let us illustrate this by an example. The function `list_remove` allows removing an element from a linked list. To perform this operation in the middle of the list, we have to modify the previous element to make its `next` field point to the element that follows the element to remove. So we basically have to specify in the assign clause, considering that the element to remove is the i^{th} one, that the $(i\text{-}1)^{th}$ element can be modified by the function. However, when the element to remove is the first one, this element does not exist.

While in ACSL, we can use the notion of behaviour to specify assigns clauses for different subcases of the allowed input (here: there is no element in the list, the element is the first one in the list, the element is not in the list, the element is in the list but is not the first one), WP currently does not consider these subcases from the caller point of view for side-effects, if multiple assigns clauses are specified in different behaviours, they are merged into a single one, so it is hard to deduce what part of the list has not been modified.

Improving this part of the verification thus requires to improve the WP plugin of Frama-C.

3.4.2 Handling of ghost 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 were expected to be necessary, namely the separation between the different elements of the list, 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.

What we propose is a refinement of the notion of ghost in Frama-C. Instead of indicated 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.

We are producing a plugin that is able to verify this kind of property using typing, so 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.

3.4.3 Using ACSL logic lists

Generally, formal proof engineers tend to prefer a more abstract view of an API for verification than what we get by using ghost arrays. A good way to model lists is for example to use logic lists. The support of ACSL logic lists has been recently added to WP. It leads to a more abstract view of the module, and the absence of ghost data structure could ease the verification of the memory separation. However, the gap between a specification using logic lists and an executable specification will probably be harder to bridge. We are currently performing a new proof of the linked list module with this approach to compare it with the approach of the ghost companion.

Currently, all functions that do not involve mutation have been proven with this other approach, and we are working on the mutating algorithms (`list_remove`, or `list_add` for example). These first experiments tend to show that we need more lemmas (thus, more interactive proofs) while automatic proofs are significantly more efficient in the verification of the function of the API. These experiments also helped to identify some bugs in WP with the treatment of the logic functions that allows to describe logic lists in ACSL.

3.4.4 Using partial contiguous functions

Another way to get a more abstract view of a sequence of values than the ghost array is to use the notion of contiguous partial functions. 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 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.

Chapter 4 Integration of the Frama-C analysis scripts

The previous chapter focused on the functional verification of a particular module. We also started to analyse Contiki in order to show the absence of runtime errors on complete instances of the operating system.

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.

4.1 Analysis target added to the Makefile of Contiki

The common way to verify a real world software with Frama-C Eva 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/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.

4.1.1 Considered platform and configuration

For the analysis, we currently target the CC2538DK[4] 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] http://www.ti.com/tool/CC2538DK

4.1.2 Preparation of an instance and execution of the analysis

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
```

Which we consider 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.

4.2 Whole OS verification, first results

4.2.1 Analysis of alarms

We were able to run Frama-C/Eva on the different examples available in the Contiki repository. While we have not classified reported alarms as false or actual alarms, 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 assure 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 actual functionalities of Contiki. We still have to determine if those warnings are actual errors or due to the imprecision of the analysis.

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, however, 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, however, provided that 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.

Chapter 5 Identification of the platform API

Currently, the verification of Contiki with Frama-C/Eva is enabled for a particular hardware platform: CC2538DK. However, it is not completely satisfying for two main reasons. First, we want our current verification task to be useful for any user of Contiki, that is: getting confidence in the platform independent part of Contiki. Today, such a verification would not achieve this goal:

- Having the hardware platform during the verification increases the risk of imprecision,
- Many configurations are set and the verification could be invalidated with other ones.

The second aspect that we would like to improve before going further in the verification would be to ease the configuration of Frama-C for any hardware platform by giving the ability to the user to validate the instance of Contiki for their own instance of Contiki.

5.1 Hardware platform API and verification

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 the same time, some tasks are directly implemented in the hardware drivers while one would expect them to be implemented in the platform independent part of the OS, calling specific driver operations. This is for example the case for some networking tasks in the CC2538DK that are implemented as a particular thread that directly accesses buffers in the networking stack, while we would expect the networking stack to call the networking driver when needed.

We would like to simplify this view of the operating system to make the verification easier and future use of Contiki simpler.

5.1.1 Approach to make a platform agnostic verification

To make the verification independent of any hardware platform, we want to develop some *abstract* platform. That is, a platform that would not be an actual implementation of the needed hardware API but only a skeleton: a set of functions that are required by Contiki to work correctly. Basically, we would remove all the code from the hardware platform, leaving the functions completely empty. By doing this, we would be sure that we do not deal with bugs inside the platform implementation.

However, the absence of code would not allow a precise model of the behaviour of a platform. On this aspect, we plan to explore two approaches:

- Specifying the behaviour of each function using ACSL,
- Writing a simple simulation code.

While the first option is probably the more elegant one, some properties could be hard to model, in particular if we need some temporal properties. Moreover, Frama-C/Eva does not support a lot of complex ACSL constructs. However, it would allow a precise specification of what is expected of each functionality which would be interesting not only for verification but also for platform developers.

The second option would allow modelling some behaviours more precisely, however there is a risk to write buggy code or to get a code that is finally more complex than the one we had for a particular platform.

Finally, a lot of numerical constants are provided using macros, which means that these values are statically defined for each instance of Contiki, and thus the verification is made for a particular set of

values. This aspect could be improved with a support of "static const" variables in Frama-C, that would allow to reason about these variables without concrete values, thus making the verification correct for any value.

5.1.2 Provide guidelines to platform developers

Among what we already mentioned about the need of a clear documentation and specification of what is required for a platform API to implement for Contiki, formal verification is still a challenge for most users.

While Frama-C/Eva is close to a "push-button" approach, getting the right configuration to make the verification meaningful is still hard. On this aspect, we want to provide two different benefits for the user. First, by restricting our verification to the core part of Contiki, we can already produce a good set of options and configurations for Contiki in order to limit the amount of alarms, and classify the remaining alarms (including actual bugs that would be signalled to the Contiki team). Second, by also analysing Contiki for some actual hardware platforms, we can determine which options are useful to modify depending on the features that are provided by the platform.

5.2 Approach and results

5.2.1 Selection of a bunch of examples

The continuous integration for Contiki-NG is provided by Travis-CI. The minimum for a (non trivial) "Pull-Request" to be reviewed (and eventually accepted) is to pass all the regression tests running in a minimal Ubuntu Docker image. These tests are automatically launched by the Travis-CI plugin on Github and consist in the compilations and the runs of a selection of examples.

As we would like to define a suitable hardware API, we choose to define a minimal abstract hardware platform for Contiki-NG to validate it. As previously mentioned in 5.1.1 we need to identify the mandatory C functions allowing the compilation for all the examples or at least the not too platform-specific ones. For now, we do not give them a precise behaviour: we only provide an empty implementation.

There is a specific script, namely `tests/compile-all/build.sh` which compiles more examples but neither performs the runs, nor the cooja simulations. We modified it and also the Makefiles to catch the symbols defined in the object files coming from the `arch/` directory and called by the object files coming from the `os/` directory. This script is not used by the Travis-CI, so running it and simultaneously using more recent compilers with more restrictive compilation flags than those of the Docker image already revealed some coding errors (for example a structure initialized with the wrong number of arguments).

Indeed, compiling on 64 bits instead of the 32 bits Docker image revealed a subtle bug in the implementation of the neighbour discovery protocol cause by the different possible implementations of the `memcmp` function (note that `memcmp` is in the `libc`, but can sometimes also be provided directly by the compiler). So, we would like to use even more conservative flags of compilation, as this problem could have been easily detected with the `-Wconversion` flag.

One of the difficulty is that all platforms are either using the `--gc-sections` linker flag and/or build an archive with the GNU `ar` tool. By default, the GCC pull only the needed symbols from an archive (unless it is used with the `--whole-archive` flag). So all the unused code is currently removed automatically at the linking phase. If we remove the `--gc-sections` flag, some examples do not compile because of the non-coherent and/or non-systematic use of some macros (the so-called `#define` symbols) in the netstack `os/net/ipv6` and in few other places. For example, `#ifdef UIP_TCP` should probably be used in `os/net/ipv6/tcp-socket.c`.

5.2.2 A first shot for an abstract platform

Looking at the undefined symbols when we removed the `--gc-sections` linker flag and parsing the symbols in the map file (created by GCC thanks to the flags `-Wl,-Map=example.map,--cref` forwarded to the linker), we defined a minimal abstract platform with one default board with its buttons and sensors :

```
arch/platform/abstract/contiki-conf.h
arch/platform/abstract/platform.c
arch/platform/abstract/dev/board.h
arch/platform/abstract/dev/board-buttons.c
arch/platform/abstract/dev/board-sensors.c
```

and incorporating what is needed by the abstract CPU and GPIOs:

```
arch/cpu/abstract/clock.c
arch/cpu/abstract/dbg.c
arch/cpu/abstract/int-master.c (interruptions)
arch/cpu/abstract/low-level-stubs.c (place here compiler specific code)
arch/cpu/abstract/rtimer-arch.c
arch/cpu/abstract/rtimer-arch.h
arch/cpu/abstract/slip-arch.c
arch/cpu/abstract/watchdog.c
arch/cpu/abstract/dev/gpio-hal-arch.c
arch/cpu/abstract/dev/spi-arch.c
```

This organization is not definitive. It is only a first shot of a platform that can be compiled for all the examples called by `tests/compile-all/build.sh`. This approach adds more symbols than the obvious ones we can meet in `os/contiki-main.c` (and implemented in `clock.c`, `watchdog.c` and `rtimer-arch.c`). Note that some of the examples are skipped thanks to some Makefile variables (namely `PLATFORM_EXCLUDE`, `PLATFORM_ONLY`, `PLATFORM_ACTION`).

We are also working on an abstract radio driver:

```
arch/cpu/abstract/abstract-def.h
arch/cpu/abstract/dev/abstract-rf.c
arch/cpu/abstract/dev/abstract-rf.h
```

but it turned out, as mentioned in 5.1 that no symbols in `arch/` related to the network stack are currently called from `os/`. The platform-specific code can be very intricate with the non-specific one, there is here a substantial task of reorganization to do. For example, we could determine what are the functions called from `arch/` in `os/` as, for example, some features are directly defined in platform specific code using protothreads (that is somewhat an inversion of the dependency expected between `os/` and `arch/`).

One of the advantages of this abstract platform for future analyses with Frama-C is that it is stand-alone: contrary to the native platform we do not use any Linux headers, so for example, the parsing phase will be easier.

In future work we plan to validate our hardware API. More examples should be able to compile with our abstract platform, even the ones that were removed during the transition from Contiki 3.x to Contiki-NG.

Currently some of the combination of platform, examples and user Makefile variables are not coherent. We have to provide a tool that makes the configuration easier with all the dependencies taken in account.

Chapter 6 Summary and Conclusion

We presented the work done during the period M7-M18 of the VESSEDIA project for the Task 5.1 of the Work Package 5, about the verification of the Contiki Operating System.

We described the differences between the previous version of Contiki and the one we target right now: Contiki-NG. Contiki-NG is the recent fork of Contiki 3 and the only one to be currently maintained and updated. The port of the previous verification effort has been done and required a very moderate effort. This comprises:

- The functional verification of the MEMB module with Frama-C/WP
- The verification of absence of runtime-errors of the AES-CCM module with Frama-C/WP
- The verification of absence of runtime-errors of multiple small modules with Frama-C/WP

We detailed the functional verification performed on the linked list module of Contiki, that lead to two publications [5] and [6]. This verification relies on an approach that uses ghost arrays to model the lists. The verification has been done with Frama-C/WP and the Coq proof assistant. We also proposed an executable version of the specification of the list that is close to be handled by the E-ACSL plugin of Frama-C.

We presented two aspects of the verification of entire operating system instances using Frama-C/Eva. This includes the work done on the configuration of Frama-C for Contiki, that allowed to identify some blocking problems and possible solutions, and what we plan to build for Contiki: an abstract hardware API on which we can reason without platform dependent details.

To conclude, the work on the verification of Contiki is globally on track, even if enabling the verification with Frama-C/Eva will require more work than expected. We already have interesting ideas to improve Frama-C and its tools, for example a better handling of ghost annotations that is a work in progress.

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
E-ACSL	Executable ACSL
GCC	GNU Compiler Collection
IoT	Internet of Things
IPv6	Internet Protocol version 6
OS	Operating System
SMT	Satisfiability Modulo Theories

Chapter 8 Bibliography

[1] 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.

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

[3] 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.

[4] 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.

[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.