Towards Formal Verification of Contiki: Analysis of the AES–CCM* Modules with Frama-C

Alexandre Peyrard
IMT Lille Douai, France
alexandre.peyrard.pro@gmail.com

Nikolai Kosmatov
CEA, List, France
nikolai.kosmatov@cea.fr

Simon Duquennoy
RISE SICS, Sweden and
Inria Lille - Nord Europe, France
simon.duquennoy@ri.se

Shahid Raza
RISE SICS, Sweden
shahid.raza@ri.se

Abstract

The number of Internet of Things (IoT) applications is rapidly increasing and allows embedded devices today to be massively connected to the Internet. This raises software security questions. This paper demonstrates the usage of formal verification to increase the security of Contiki, a popular open-source operating system for the IoT. We present a case study on deductive verification of encryption-decryption modules of Contiki (namely, AES–CCM*) using Frama-C, a software analysis platform for C code.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Verification

Keywords

Formal Verification, AES, Frama-C, Security. Contiki

1 Introduction

Recent advances in embedded wireless communication have enabled the Internet of Things (IoT), where devices connect the physical world to the Internet. IoT is rapidly expanding into critical domains such as healthcare, energy and avionics. With the proliferation of IoT deployments, our daily life will become increasingly dependent on IoT systems. Therefore, security and safety concerns must be taken seriously. Though a number of efforts are being carried out to secure IoT networks [15][16][17], protection of communication links is not enough. The security of the underlying software, i.e., mechanisms to ensure correctness and absence of runtime errors, is also critical.

International Conference on Embedded Wireless Systems and Networks (EWSN) 2018
14–16 February, Madrid, Spain

ISBN: 978-0-9949886-2-1

Cyber-attacks such as Mirai botnets or Zigbee war-flying have already intruded IoT systems by exploiting software security breaches. Such attacks have potentially disastrous consequences, e.g., leakage of private data, or life-critical actuation.

In this paper, we present an ongoing case study applying formal verification to ensure software security of Contiki, an open-source OS for the IoT targeted primarily for recourse-constrained battery-powered devices [11]. The target modules of this study are the AES and CCM* encryption modules of Contiki, which we analyze using Frama-C [12], a software analysis platform for C code.

The paper is organized as follows. First, we present Contiki and the module under verification, as well as the basics of Frama-C that we used to prove the code of this module (Section 2). Then we describe our verification approach (Section 3) and discuss the results and lessons learned (Section 4). Next, we give some related work (Section 5). Finally, we present the concluding remarks and future work (Section 6).

2 Background

This section presents the necessary preliminary knowledge on Contiki and the target modules verified in this study, as well as the Frama-C platform and its specification language ACSL.

2.1 Contiki

Contiki [11] is an open-source operating systems for IoT devices. It was one of the first operating systems to provide full low-power IPv6 connectivity, including recent IETF stadnards such as 6TiSCH, 6LoWPAN, RPL or CoAP. The target devices for Contiki have an MCU ranging from 8 to 32-bit, with no MMU. These devices usually have a small low-power radio modules, some sensors, a few kilobytes RAM and tens of kilobytes ROM. Contiki is implemented in C language and has a kernel linked to platform-specific drivers at compile-time.

When Contiki was developed in 2002 by Adam Dunkels, no particular attention was paid to its security. The first security modules appeared for communication security via standard protocols such as IPSec or DTLS. However, little (if any) attention was given to software security aspects so far.

2.2 AES and CCM* Modules

Contiki implements the Advanced Encryption Standard (AES) [9], a symmetric encryption algorithm that replaced in 2002 the earlier Data Encryption Standard (DES) [6], which became obsolete in 2005. 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 and authentication is a very important ingredient of wireless communication in a network. CCM* is used for instance by the DTLS protocol to provide end-to-end confidentiality and integrity. However, it is also crucial for the source code implementing these modules to be dependable. Previous studies have shown that runtime errors (including invalid pointer accesses, out-of-bounds array accesses, integer overflows, etc.) are a major reason of many security vulnerabilities [18]. That is why we choose to analyze the AES and CCM* modules of Contiki for absence of potential runtime errors. Note that we assume the underlying algorithms to be dependable, as they follow the well-known standards and were carefully tested.

An additional reason for this choice of modules is their small size and their independence from other Contiki modules. Source code analysis by deductive verification is a time-consuming process, so focusing on a small yet critical part of a large software project is a practical way to obtain representative results. The two chosen modules are both tightly coupled, and consist in only a few hundreds lines of codes in four files: `aes-128.c`, `aes-128.h`, `ccm-star.c` and `ccm-star.h`. Further, note that as these modules expose a clear API, they are fairly independent from the rest of Contiki, and could be used in other projects.

Figure 1 shows code snippet from the AES module. It illustrates an intensive usage of pointer and array accesses and integer arithmetics representing a risk of runtime errors (such as an out-of-bounds access) that can potentially lead to a security breach.

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

  rcon = 0x01;
  memcpy(round_keys[0], key, AES_128_KEY_LENGTH);
  for(i = 1; i <= 10; i++) {
    round_keys[i][0] = sbox[round_keys[i - 1][13]]
    ^ round_keys[i - 1][0] ^ rcon;
    round_keys[i][1] = sbox[round_keys[i - 1][14]]
    ^ round_keys[i - 1][1];
    round_keys[i][2] = sbox[round_keys[i - 1][15]]
    ^ round_keys[i - 1][2];
    round_keys[i][3] = sbox[round_keys[i - 1][12]]
    ^ round_keys[i - 1][3];
    for(j = 4; j < AES_128_BLOCK_SIZE; j++) {
      round_keys[i][j] = round_keys[i - 1][j]
      ^ round_keys[i][j - 4];
    }
    rcon = galois_mul2(rcon);
  }
}
```

Figure 1. Implementation of AES key storage in Contiki (function `set_key`, file `aes-128.c`)

2.3 Frama-C: A Software Analysis Platform

Frama-C [12] is a rich toolset for static and dynamic analysis of C code developed by the List institute of the French Alternative Energies and Atomic Energy Commission in collaboration with the French Institute for Research in Computer Science and Automation (INRIA). Frama-C offers various analyzers developed as individual plug-ins and based on different techniques such as abstract interpretation, weakest precondition calculus, test generation, runtime verification, dependency and impact analysis, program slicing, etc. For instance, the Eva plug-in performs abstract-interpretation-based value analysis of C code. It can be used to compute potential values of program variables at each program point and to detect potential errors in the program.

Frama-C also provides a specification language called ACSL (ANSI/ISO C Specification Language) [5] to write annotations (or contracts) that constitute a program *specification*. ACSL annotations are written in special comments `/*@ <annotation>*/` or `//@ <annotation>` in the C source code. Thanks to this way of inserting annotations, the annotated code remains compilable and equivalent to the original code: the annotations are simply ignored during compilation.

In this paper, we use WP, the Weakest-Precondition-based plug-in of Frama-C for deductive verification of C programs. WP can be used to establish a formal, mathematical proof that a given C program respects its specification. Given a C program with an ACSL specification, WP translates the required properties into theorems to be proved, called *verification conditions* or *goals*. When verification conditions are rather simple, the WP plug-in can be capable alone to prove them. However, it often happens that WP cannot validate all necessary properties. In this case, it calls external provers like SMT solvers to prove them. In our analysis of the AES–CCM* modules, we use three SMT solvers: Alt-Ergo [8], CVC4 [3] and Z3 [10].

2.4 Example of ACSL Specification

The ACSL language is a behavioral specification language used in Frama-C to define the expected (i.e. correct) behavior of a given C program. Any C function can be provided a *function contract* specifying its preconditions (i.e. conditions supposed to be true on entry) and *postconditions* (properties on the memory state and return value that the function should ensure on exit). These properties can deal with the values of program variables, validity of memory addresses and arrays. More complex user-defined predicates can be used as well. We say that the function respects its specification if for any possible function call such that the program state respects the precondition before the call, the program state satisfies the postcondition after the execution of the function.

```
1  #include <limits.h>
2  /*@
3   requires INT_MIN < val;
4   ensures \result >= 0;
5   ensures (val >= 0 ==> \result == val) &&
6    (val < 0 ==> \result == -val);
7  */
8  int abs(int val){
9   if(val < 0) return -val;
10  return val;
11 }
```

Figure 2. An absolute value function in C with ACSL annotations

Figure 2 shows a function that computes the absolute value of an argument. It illustrates how the function contract written in ACSL is divided into a **requires** clause (line 3) and two **ensures** clauses (lines 4–6). The **requires** clause expresses the precodition – in this example, the fact that the value of parameter `val` must be greater than the predefined constant `INT_MAX`[1]. The **ensures** clauses specify two postconditions.

3 Verification Approach

Computer programs often contain unintentional errors (or bugs) due to the fact that they are written by humans. Such bugs can be difficult to find. The purpose of software verification and validation is to provide confidence that the program does not have bugs – either by detecting and fixing them, or by proving their absence.

One traditional technique of software validation is testing: a program is executed on selected test inputs, and its results are checked with respect to the expected behavior. Depending on the required level of confidence, tests can become very complex, activating various parts of the source code. However, since the program cannot be in general tested for all possible input states, testing cannot prove the absence of bugs.

Another approach is to ensure that the program is correct using formal verification. Deductive verification can be used to prove that the program respects a given specification. Depending on the level of detail of the provided specification, one can prove functional correctness or just the absence of runtime errors. In this work, we follow the latter approach and apply the WP plug-in of Frama-C to prove the absence of runtime errors in the AES–CCM* modules of Contiki.

3.1 Verification of the AES–CCM* Modules

As mentioned in Section 2.2, the AES module is composed of one main C file `aes-128.c` and its header `aes-128.h`. It is in the C file that we have written an ACSL specification. To use Contiki's AES function, one must use a variable of type **const struct** `aes_driver_128`, shown in Figure 3, which defines the two AES functions to be used as `set_key` and `encrypt`.

The `set_key` function shown in Figure 1 implements an algorithm to register a given AES key, whereas the `encrypt`

[1]to avoid an arithmetic overflow at line 9 since `-INT_MAX` cannot be represented in the **int** type

```
const struct aes_128_driver aes_128_driver = {
 set_key,
 encrypt;
}
```

Figure 3. The aes_driver_128 structure

function encodes a given message. We have written a function contract as explained previously for each of these functions.

This work is aimed at verifying the absence of runtime errors such as out-of-bounds memory accesses in the target modules. To achieve this goal we specify properties related to the variables used in the function of the file `aes-128.c` that can potentially be responsible for such errors. As mentioned above, we do not verify functional correctness of the AES algorithm in the `encrypt` function with Frama-C/WP.

Similarly, for the CCM* module, we also focus on runtime errors, so we do not specify the CCM* algorithm in ACSL but write properties on the variables used in this algorithm in order to ensure the absence of errors. As the CCM* algorithm builds on AES, we find a similar structure, **const struct** `ccm_star_driver`, composed again of two functions. This structure includes a key recording function named `set_key` and an encryption function called `aead`.

We have verified both the AES and CCM* C files with Frama-C/WP to prove the absence of runtime errors thanks to the provided ACSL specifications. We have also created a test file representing the regular use of the AES and CCM* modules by the Contiki users. As we study only a part of Contiki, verifying a module in a realistic context demonstrates the possibility to extend the verification to other modules in the future. This approach is detailed in Section 3.2.

3.2 Consistency of the Specification

Ensuring that all covered functions respect their contracts is not enough. One must also verify that the calling functions satisfy the preconditions. As AES is basically only used by CCM*, we were able to verify all preconditions for function calls to the AES module. CCM*, however, can be called from many different places, including user code. As long as the whole code of Contiki is not verified, there is still a risk that some of these functions can be called with illegal parameters (i.e., that do not comply with the preconditions). In this case, the function can have a different behavior (since its input state does not verify the precondition) that can potentially lead to a runtime error.

Since a complete formal verification of Contiki is not yet performed, we propose a way to get confidence that our specification is consistent with the calling code. More specifically, we simulate regular usage of the target module by Contiki users in several test cases and verify the consistency of the preconditions we provided for these typical use cases. In order to do that, we have created a test file which represents the most likely usage of AES–CCM* functions.

This test file, called `wp_tests.c`, contains three tests inspired from existing Contiki application code examples. The first test `test_aes_128` is a simple use of the two main

Table 1. ACSL line count from the AES-CCM* modules

C Files studied	ACSL lines	ACSL and C lines
aes-128.c	47 (23%)	208
ccm-star.c	56 (29%)	192

Table 2. Results of analysis for the file `aes-128.c` where each next prover is applied to yet unproven goals

Frama-C/WP provers	Number of goals proved
Simplifier Engine (Qed)	166
Alt-Ergo	93
CVC4	13
Z3	3
All provers	275

functions of the AES module, `set_key` and `encrypt`. The two other tests are more complex because we need to simulate a frame that will be encrypted with the CCM* encryption. That is why we need to put annotations in the `packetbuf.c` file that we used in these tests.

We also run Frama-C/WP on this test file to ensure that the provided preconditions are indeed consistent with the typical call contexts. It gives confidence that the perimeter of verification will be extended to a larger set of modules using AES–CCM* in the future.

4 Results

To prove the specifications of the AES–CCM* modules, we have used the Magnesium-20151002 version of Frama-C/WP with the SMT solvers Alt-Ergo 0.99.1, CVC4 1.4 and Z3 4.4.1. The ACSL specification of the `aes-128.c` file contains 47 lines of code, for a total of 208 lines of annotated code without top comments (so the specification takes 23% of the annotated code). Regarding the `ccm-star.c` file, there are 56 lines of ACSL specification for 192 lines of annotated C source code without top comments (29%).

After writing the ACSL specification, we start the analysis of the modules by Frama-C/WP. We run the analysis once for each C file, `aes-128.c`, `ccm-star.c` and a test file, `wp_tests.c`. The command line used for the first analysis for the file `aes-128.c` is as follows:

```
$ frama-c-gui -wp -wp-rte -wp-model "Typed+Cast"
-wp-prover alt-ergo,cvc4,z3
-cpp-extra-args='-I./core -I./platform/native/
-I./cpu/native/' core/lib/aes-128.c
```

Notice that we use the `-wp-rte` option in order to verify the absence of possible run-time errors in all instructions of the corresponding file. Moreover, we use two memory models provided by WP, called *Cast* and *Typed*, for our analysis. A *memory model* defines the way the tool represents variables and memory locations during the verification process. The Typed model is the default memory model for WP that stores variables of different types in different arrays. The Cast model supports casts between pointers.

The results of the analysis of the AES module show that for a total of 275 goals generated, all goals are proved by the Frama-C/WP provers for a total processing time of 27.98 seconds. These goals were proven by Qed, the simplifier engine of WP, and three SMT provers. Qed proved 166 goals. Alt-Ergo, CVC4 and Z3 proved, respectively, 93, 13 and 3 goals. The provers are used successively in this order to validate the remaining goals, that is, a next prover was applied to yet unproven goals. Table 2 sums up the results of Frama-C/WP for this file. A timeout per goal was set to 10 seconds.

The results of the CCM* module show that out of 467 goals generated by WP, all are successfully proved meaning that the AES-CCM* modules are validated. As in the previous analysis, the provers are used one after another. First, Qed proved 280 goals, then Alt-Ergo, CVC4 and Z3 proved, respectively, 119, 64 and 4 goals. The results are presented in Table 3. The total processing time of this analysis is 48.97 seconds.

The last analysis is performed on the test file `wp_tests.c`. As it includes all files containing ACSL specifications, its analysis includes analysis of the previous ones. The annotation of the C code for the file `wp_tests.c` requires to annotate some other Contiki C files such as `packetbuf.c` belonging to other modules of Contiki. That is why the analysis of this test file is still in progress. It is expected to show that the chosen specifications of the AES–CCM* modules are compatible with their typical use cases.

Thanks to formal verification using the WP plug-in of Frama-C, we can guarantee that there are no out-of-bounds memory accesses in Contiki's AES and CCM* modules, as long as they are used on admissible inputs with respect to their contracts. This last property will be demonstrated by the verification of the test file that is still ongoing, and confirmed during the verification of other modules. It will give us strong guarantees of absence of out-of-bounds memory accesses in the AES-CCM* modules.

5 Related Work

Formal verification of OS and Cloud hypervisors has already been successfully applied in several projects. For example, Klein et al. [13] describe formal verification for seL4, a microkernel allowing devices running it to achieve the highest level of the Common Criteria. Formal verification of a microkernel is described in [2]. In these projects, verification relies on interactive, machine-assisted and machine-checked proof with the theorem prover Isabelle/HOL. [4]

Table 3. Results of analysis for the file `ccm-star.c` where each next prover is applied to yet unproven goals

Frama-C/WP provers	Number of goals proved
Simplifier Engine (Qed)	280
Alt-Ergo	119
cvc4	64
z3	4
All provers	467

presents a verification of a model of virtualization. Both implementation and verification are done in the Coq proof assistant. Verification of the translation lookaside buffer (TLB) virtualization, a core component of modern hypervisors, is presented by Alkassar et al. [1]. Frama-C/WP has also been applied in a previous work [7] to formally verify the virtual memory system of a hypervisor. The proof is performed mostly automatically, only a few lemmas have been proved interactively in Coq.

In the context of IoT software, formal verification was rarely applied since this application domain was not considered as critical for a long time. Recently, a complete formal verification using Frama-C/WP of the memory allocation module of Contiki, `memb`, was presented in [14].

The present work continues the previous efforts and presents a case study on formal verification of a critical module of Contiki in Frama-C/WP.

6 Conclusion and Future Work

In this paper, we have presented a recent verification effort of proving in Frama-C/WP the absence of runtime errors in the AES and CCM* modules of Contiki, two of its most critical parts responsible for data encryption. We have verified the absence of runtime errors, such as invalid memory accesses. As all properties have been proved by Frama-C/WP, we have a complete analysis of the AES-CCM* modules.

Frama-C/WP is a powerful verification tool capable of formally proving that a given program respects its specification. Depending on the level of detail of the specification, the verification engineer can prove a complete or partial functional correctness of the program, and/or the absence of runtime errors.

However, to use the tool, the engineer has to acquire a sufficient expertise in writing formal specification in the ACSL language, selecting an appropriate WP memory model and using a suitable SMT solver. Depending on their initial background and previous experience, this training phase can take between a few days and several months. Debugging of the specification can represent a particular issue since proof failures can often be difficult to understand.

It is important to continue formal verification of Contiki. Even though a previous study proved functional properties of Contiki's memory allocation module `memb` [14] and this work has shown the absence of security-related memory errors in the AES–CCM* modules, formal verification of a significant part of the source code of Contiki remains to be done. Future work includes extending the perimeter of verification to a larger range of modules in Contiki. Another future work direction is to extend Frama-C in order to automatically generate a subset of relevant annotations, in particular, for loop invariants.

7 Acknowledgments

This work was partially supported by the Vessedia project, funded from European Union's Horizon 2020 research and innovation programme under grant agreement No 731453. It was also supported by a grant from CPER Nord-Pas-de-Calais/FEDER DATA and the distributed environment Ecare@Home funded by the Swedish Knowledge Foundation 2015-2019.

8 References

[1] E. Alkassar, E. Cohen, M. Kovalev, and W. J. Paul. Verification of TLB virtualization implemented in C. In *Proc. of the 4th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2012)*, volume 7152 of *LNCS*, pages 209–224. Springer, 2012.

[2] E. Alkassar, W. Paul, A. Starostin, and A. Tsyban. Pervasive verification of an OS microkernel. In *Proc. of the 3rd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2010)*, volume 6217 of *LNCS*, pages 71–85. Springer, 2010.

[3] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proc. of the 23rd International Conference on Computer Aided Verification (CAV 2011)*, volume 6806 of *LNCS*, pages 171–177, 2011.

[4] G. Barthe, G. Betarte, J. D. Campo, J. M. Chimento, and C. Luna. Formally verified implementation of an idealized model of virtualization. In *Proc. of the 19th International Conference on Types for Proofs and Programs (TYPES 2013)*, pages 45–63, 2013.

[5] P. Baudin, P. Cuoq, J.-C. Fillitre, C. March, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language*. http://frama-c.com/acsl.html.

[6] E. Biham and A. Shamir. *Differential Cryptanalysis of the Data Encryption Standard*. Springer, 1993.

[7] A. Blanchard, N. Kosmatov, M. Lemerre, and F. Loulergue. A case study on formal verification of the anaxagoros hypervisor paging system with frama-c. In *Proc. of the 20th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2015)*, volume 9128 of *LNCS*, pages 15–30. Springer, June 2015.

[8] S. Conchon et al. The Alt-Ergo Automated Theorem Prover. http://alt-ergo.lri.fr.

[9] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[10] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, pages 337–340. Springer, 2008.

[11] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proc. of the IEEE Conference on Local Computer Networks (LCN 2014)*. IEEE, 2004.

[12] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.

[13] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, 2010.

[14] F. Mangano, S. Duquennoy, and N. Kosmatov. A memory allocation module of Contiki formally verified with Frama-C. A case study. In *Proc. of the 11th International Conference on Risks and Security of Internet and Systems (CRiSIS 2016)*, volume 10158 of *LNCS*, pages 114–120. Springer, 2016.

[15] S. Raza, S. Duquennoy, J. Höglund, U. Roedig, and T. Voigt. Secure Communication for the Internet of Things - A Comparison of Link-Layer Security and IPsec for 6LoWPAN. *Security and Communication Networks, Wiley*, 7(12):2654–2668, Dec. 2014.

[16] S. Raza, T. Helgason, P. Papadimitratos, and T. Voigt. Securesense: End-to-end secure communication architecture for the cloud-connected internet of things. *Future Generation Computer Systems (Elsevier)*, 77:40–51, December 2017.

[17] S. Raza, L. Seitz, D. Sitenkov, and G. Selander. S3K: Scalable Security with Symmetric Keys - DTLS Key Establishment for the Internet of Things. *IEEE Transactions on Automation Science and Engineering*, 13(3):1270–1280, July 2016.

[18] V. van der Veen, N. dutt-Sharma, L. Cavallaro, and H. Bos. Memory errors: The past, the present, and the future. In *Proc. of the International Symposium on Research in Attacks, Intrusions, and Defenses*, volume 7462 of *LNCS*, pages 86–106. Springer, 2012.