Journey to a RTE-free X.509 parser

Arnaud Ebalard, Patricia Mouy, and Ryad Benadjila
prenom.nom@ssi.gouv.fr

ANSSI

Abstract. C programming language is a security nightmare. It is error-prone and unsafe, but, year after year, the conclusion remains the same: no credible alternative will replace C in a foreseeable future; all the more in low-level developments or for constrained environments.

Additionally, even though some C developers are keen to drop this language when possible for more robust ones like ADA or Rust, converting the existing code basis to safer alternatives seems unrealistic.

But one of the positive aspects with C is that its inherent flaws became a long time ago a full-time research topic for many people. Various static analysis tools exist to try and verify security aspects of C code, from the absence of run-time errors (RTE) to the verification of functional aspects.

At the time of writing, none of these tools is capable of verifying the full-fledged C source code from most well-known software projects, even the smallest ones. Those tools are nonetheless getting better, and they may be able to handle large software code bases in the near future.

Meanwhile, doing some steps in the direction of static analysis tools is sometimes sufficient to achieve full verification of a complex piece of code. This article details this kind of meet-in-the-middle approach applied to the development in C99 of a X.509 parser, and then its later verification using Frama-C.

1 Introduction

In a nutshell, the aim of the project was to develop a "guaranteed RTE-free X.509 parser", so that it could be used safely for syntactic and semantic verification of certificates before their usual processing in an implementation (e.g. in existing TLS, IKEv2, S/MIME, etc. stacks).

Common ASN.1 and X.509 parsers have a very poor track record when it comes to security (see [6] or [5] for instance), mainly due to their complexity. Since such parsers are usually used in security primitives to validate signatures, the consequences of a vulnerability can be even more disastrous due to the obvious critical and privilege levels of such primitives. Hence, these parsers appear as a perfect subject for an RTE-free development.

C was selected as the target language in order to allow integration in all kinds of environments, from high-level userland applications to embedded ones running on low power microcontrollers.

Our “guaranteed absence of RTE” goal has been achieved using Frama-C, an open-source C analysis framework. But at the beginning of the project, some specific rules were decided regarding the later use of this static analysis tool:

- No formal expertise expected from the developer
- Initial writing of the code without specific Frama-C knowledge (but considering that the code will be later analyzed)
- Frama-C analysis of the code with:
  - very limited rewriting of the code;
  - limited (simple) annotation of the code.

The main idea behind these rules – they will be described in more details later in the document – was to benchmark the ability for a **standard and average** – but motivated – C developer to successfully achieve verification with a limited additional investment.

Obviously, having already witnessed the limitations of some (most) static analysis tools, it was clear that being careless during the development phase would prevent the verification using a static analysis tool. For this reason, as explained in section 3.2, some care has been taken to make the code more suitable for a later analysis.

At this point, one may wonder why the term “guaranteed absence of RTE” is used to describe the result of the analysis instead of “proven” or “bug-free”. Qualifying the code as “proven” does not mean anything *per se*. Qualifying it as “bug-free” would require to ensure the absence of all kinds of bugs, including logical ones. Even if Frama-C can help verifying functional aspects of the code, which may provide some help in getting additional guarantees on what the code does, it has been used here only to verify the absence of **runtime errors** [1] (e.g. invalid memory accesses, division by zero, signed integer overflows, shift errors, etc.) on all possible executions of the parser. Even if care has been taken during the implementation to deal with logical ones (e.g. proper implementation of X.509 rules described in the standard), their absence has not been verified by Frama-C and is considered out of scope for this article. We nonetheless stress out that the absence of RTE is yet a first (non trivial) step paving the way towards a “bug-free” X.509 parser.

1. Runtime errors are denoted throughout the article as RTE.

The remaining of the article first gives a quick glance at the X.509 format, complexity and prior vulnerabilities in various projects. It then presents various aspects of parser development towards security and finally describes the incremental work with Frama-C up to a complete verification of the parser.

2 X.509

2.1 Introduction

Essentially, X.509 certificates contain three main elements: a subject (user or web site), a public key and a signature over these elements linking them in a verifiable way using a cryptographic signature.

But in more details, a lot of other elements are also present in a certificate such as an issuer, validity dates, key usages, subject alternative names, and various other kinds of optional extensions; enough elements to make certificates rather complex objects.

To make things worse, certificate structures and fields contents are defined using ASN.1 [8] and each specific instance is DER-encoded [9]. As discussed later, each high-level field (e.g. subject field) is itself made of dozens of subfields which are themselves made of multiple fields.

At the end of the day, what we expect to hold a subject, a key and a signature linking those elements together ends up being a 1.5KB (or more) binary blob containing hundreds of variable-length fields using a complex encoding.

This Matryoshka-like recursive construction makes parsing X.509 certificates a security nightmare in practice and explains why most implementations – even carefully implemented ones – usually end up with security vulnerabilities.

If this was not already difficult enough, parsing and validating an X.509 certificate does not only require a DER parser and the understanding of X.509 ASN.1 structures. Various additional semantic rules and constraints must be taken into account, such as those scattered in the various SHALL, SHOULD, MUST, etc. in the IETF specification [2]. This includes for instance the requirement for a certificate having its `keyCertSign` bit set in its `keyUsage` extension to have the `cA` boolean in its Basic Constraints to also be asserted (making it a Certification Authority (CA) certificate). Additional rules have also been put in place by the CA/Browser Forum [2].

2. `https://cabforum.org`

And then, because Internet is Internet, some may expect invalid certificates (regarding previous rules) to be considered valid because lax implementations have generated and accepted them for a long time.

2.2 ASN.1, BER and DER encoding

The X.509 format is built upon ASN.1 (Abstract Syntax Notation One), which basically defines a general purpose TLV (Tag Length Value) syntax for encoding and decoding objects. It is particularly useful for elements that must be marshalled over a transmission line (e.g. for network protocols).

At its basic level, ASN.1 defines *types* with rules encoding them as a binary blob. Among the defined types, we have **simple types** such as `INTEGER`, `OCTET STRING` or `UTCTime`. These types are *atomic* and represent the leaves when parsing ASN.1. **Structured types** such as `SEQUENCE` on the other hand encapsulate simple types and introduce the recursive aspect of ASN.1 objects.

ASN.1 introduces various ways of encoding the same type using BER (Basic Encoding Rules) [9]. The same element can be represented in a unique TLV object, or split across multiple TLV objects that, when decoded and concatenated, will produce the same ASN.1 object. Because of the ambiguity introduced by BER (many binary representations can be produced for the same object), the DER (Distinguished Encoding Rules) have been introduced. DER adds restrictions to the BER encoding rules that will ensure a unique binary representation of all ASN.1 types. From now on, we will only focus on DER encoding as it is the one specified by the X.509 standards.

Even though no ambiguity exists in DER, encoding and decoding ASN.1 is still quite complex and error-prone. Figure 1 provides a concrete example of the way a very simple object like the `INTEGER` 2 is DER-encoded (this is the version field found in X.509 v3 certificates).

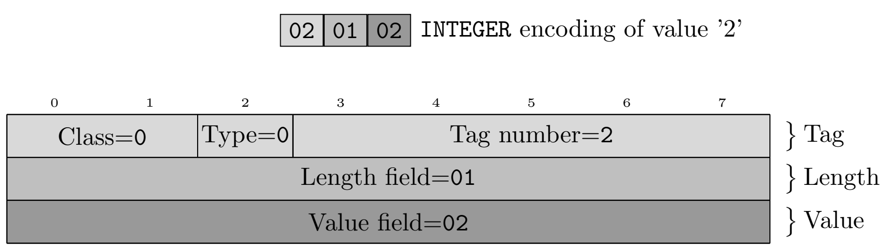

Fig. 1. ASN.1 simple `INTEGER` encoding.

The first byte is the tag of the TLV. It is made of three different subfields: the first two bits provide the Class, which is universal (0). The third bit being 0 indicates that the type is primitive (otherwise, it would have been constructed). Then, the last five bits of this first byte provide the Tag number which has value 2, and indicates that the element type is an `INTEGER`. Note that class values are not limited to the 32 values the 5 bits can encode; when the specific value 31 is used (all 5 bits set), the class is encoded on multiple following bytes.

The second byte is the beginning of the length field of the TLV. The type is primitive (first two bits of the first byte are 0), and the length is encoded using either the short form (one octet encoding) or the long definite form (two to 127 octets) depending on the length of the encapsulated value (below or above 127 bytes) [3] (section 8.1.3.3 of [9]). Because the version field contains only a small integer, its length is encoded using the short form, which can be deduced from the fact that the leading bit of the second byte is 0. This is a trivial case for which the length of the content is directly the value of the second byte, i.e. 1.

We now know that the content (i.e. value) of the `INTEGER` is encoded as a single byte following the length field, in big endian two's complement binary notation. The integer value is 2 in our case.

In the end, extracting this simple `INTEGER` value from those 3 bytes required parsing 3 fields, each of which contained multiple subfields capable of modifying the parsing logic. This very simple example is expected to show the reader the complexity of parsing DER-encoded structures.

2.3 X.509 format

At high level, an X.509 certificate is a signed ASN.1 structure holding few elements represented on listing 1. The standard [10] [4] defines all the X.509 types recursively until *simple or structured types* are reached, yielding a non ambiguous ASN.1 definition. The X.509 ASN.1 specification indicates that a `Certificate` structure is the signed version of a `TBSCertificate` structure, which is itself defined above as a `SEQUENCE` of various elements (that can be each one a `SEQUENCE` or a `SET` of other elements, and so on).

The first element in the `TBSCertificate` sequence is a field named `version` of type `Version`, which is defined as an `INTEGER` taking three

3. In generic non DER ASN.1 encoding, an indefinite form also exists for constructed types where no length is provided and the content octets are marked using a specific value named End-of-contents octets.

4. RFC5280 [2] contains the same description.

different values indicating the version of the certificate. If absent, the certificate is a v1 one.

```
Certificate ::= SIGNED{TBSCertificate}

TBSCertificate ::= SEQUENCE {
  version                     [0] Version DEFAULT v1,
  serialNumber                    CertificateSerialNumber,
  signature                       AlgorithmIdentifier{{
      SupportedAlgorithms}},
  issuer                          Name,
  validity                        Validity,
  subject                         Name,
  subjectPublicKeyInfo            SubjectPublicKeyInfo,
  issuerUniqueIdentifier      [1] IMPLICIT UniqueIdentifier OPTIONAL,
  ...,
  [[2: -- if present, version shall be v2 or v3
  subjectUniqueIdentifier     [2] IMPLICIT UniqueIdentifier OPTIONAL]],
  [[3: -- if present, version shall be v2 or v3
  extensions                  [3] Extensions OPTIONAL]]
  -- If present, version shall be v3]]
  }

Version ::= INTEGER {v1(0), v2(1), v3(2)}
...
Validity ::= SEQUENCE {
  notBefore Time,
  notAfter  Time,
  ... }
...
Time ::= CHOICE {
  utcTime         UTCTime,
  generalizedTime GeneralizedTime }
```

Listing 1. X.509 certificate high level structure.

As another example of ASN.1 complexity, the fifth element in the certificate is a `validity` field whose structure is defined below the `TBSCertificate` structure as a `SEQUENCE` of two elements (`notBefore` and `notAfter`). Both are defined using `Time` type which is itself defined as a CHOICE between two possible types (`UTCTime` and `GeneralizedTime`).

The last element of the certificate might achieve to convince of the inherent structural complexity of X.509. [2] defines the `extensions` field as presented on figure 2. The `extensions` field is a `SEQUENCE` of `Extensions`, which are themselves `SEQUENCE`s of 3 elements: an object identifier, a critical bit and a value encoded as an `OCTET STRING` and that can then be decoded specifically based on the object identifier. Additionally, the various extensions have structures that are more complex than the basic main certificate fields.

```
Extensions ::= SEQUENCE OF Extension

Extension ::= SEQUENCE {
  extnId  EXTENSION.&id({ExtensionSet}),
  critical BOOLEAN DEFAULT FALSE,
  extnValue OCTET STRING
     (CONTAINING EXTENSION.&ExtnType({ExtensionSet}{@extnId})
     ENCODED BY der),
  ... }

der OBJECT IDENTIFIER ::=
  {joint-iso-itu-t asn1(1) ber-derived(2) distinguished-encoding(1)}

ExtensionSet EXTENSION ::= {...}
```

Listing 2. X.509 extensions ASN.1 structure.

2.4 Vulnerabilities

This section provides a few examples of X.509 or ASN.1 parser vulnerabilities in order to illustrate the possible devastating impacts of errors in such parsers.

CVE-2017-7932 Various NXP ARM Systems On Chip (SoC) share a common mechanism called High Assurance Boot to secure their boot process by providing authenticity of firmware images. The mechanism is implemented in the BootROM of the SoC, a ROMed piece of code, which cannot be updated for existing chips. [5] describes a stack-based buffer overflow in the use of the `asn1_extract_bit_string()` function when parsing the content of the `keyUsage` extension. This vulnerability is exploitable using a certificate with a crafted `keyUsage` extension, allowing the attacker to redirect the PC register and execute arbitrary code embedded in the certificate. One of the demonstrated uses is the complete bypass of the secure boot mechanism of i.MX28, i.MX 50, i.MX 53, i.MX 6, i.MX7, Vybrid VF3xx, VF5xx, and VF6xx processors. The only way to get a fixed version of such processors was to wait for new hardware revisions. No valid workaround or fix exists to alleviate the issue for existing platforms that rely on this mechanism for their security.

3DS flawed ASN.1 parser In 2018, Scire and al. documented in [17] attacks on the BootROMS of the Nintendo 3DS, allowing to exfiltrate secret information from protected memory areas and gain persistent early code execution. The attack exploits a flaw in the RSA PKCS1v#1.5 padding ASN.1 parsing implementation where the bounds of the signed hash field embedded in an `OCTET STRING` are not verified. This allows an adversary to alter the parsing process and make the BootROM code check a crafted signed hash elsewhere on the stack in place of the one embedded in the signed firmware. An interesting element here is that this little crack in the 3DS security scheme is one of the only – yet fatal – flaws in a rather clean security architecture.

CVE-2016-5080 Objective Systems Inc. develops and sells an ASN.1 compiler for C/C++ called ASN1C, which generates ASN.1 parsing code. Generated code produced by version 7.0 or below contained a heap overflow vulnerability allowing a possible code execution on the targeted platforms. One of the vulnerable example implementations was a 3GPP API add-on in the ASN1C SDK.

CVE-2017-2781 InsideSecure MatrixSSL 3.8.7b contained an exploitable heap buffer overflow vulnerability when parsing `IssuerPolicy` `PolicyMappings` extension. This vulnerability allowed remote code execution.

CVE-2017-9023 ASN.1 `CHOICE` types were badly handled in StrongSwan ASN.1 parser when parsing X.509 certificates and resulted in an infinite loop. All versions before 5.5.2 were affected by this denial of service.

CVE-2017-2800 wolfSSL SSL/TLS library up to version 3.10.2 contained an exploitable off-by-one write vulnerability in their X.509 certificate parsing implementation. The impact was a possible remote code execution via a crafted X.509 certificate.

3 Parser development

3.1 Strategy for X.509 support

In an ideal world, verified RTE-free ASN.1 DER libraries would exist to serve as a groundwork for building parsers to target versatile ASN.1 syntaxes, like X.509 certificates.

Unfortunately, a simple query for “ASN.1 parser+static analysis” on any search engine provides a near empty set of results. One of the reasons behind this matter of fact is probably the inherent complexity of ASN.1 (even when considering only DER, its simplest encoding).

Additionally, because of the semantic complexity added by X.509, even if we had a clean DER ASN.1 parser, many requirements would have to be checked on top of the mere X.509 ASN.1 syntax implementation, to deal with the constraints not captured by the DER decoding.

For all these reasons, the development was performed by implementing in a progressive manner the minimal support functions to progress through the DER encoding of X.509 structures, while also taking into account the semantic elements of the specification [2].

The use of a vast representative test set, as discussed in section 3.3, helped a lot for implementation decisions to keep the parser capable of handling real-world certificates.

This pragmatic approach resulted in a limitation of the amount of code compared to a generic ASN.1 DER parser but also in a reduced complexity for the implementation. This was a first step towards Frama-C.

3.2 Development constraints

Various development constraints were selected to do some basic steps towards static analysis tools in general but not specifically towards Frama-C itself. These design patterns are usually an advised best practice when static analysis is planned.

Basic C99 without VLA. In practice, the need for C99 is mainly required by the use of designated initializers, missing in C89. All other fancy evolutions of C99 compared to C89 were considered useless and possibly dangerous, for example variable-length arrays (VLA).

No dynamic allocation. Care has been taken to not use dynamic allocation. This has been possible using various design decisions, based either on the analysis of the specification or on the analysis of real-world certificates. For instance:

— Most certificates are usually 1.5KB or so in length but there is basically no theoretical limit for their size. We decided to set an upper bound of 64 KB to certificates our parser will handle. Setting this upper bound on the whole structure also provides an upper bound on each field/structure/element that will be parsed. This helped eliminating the need for dynamic allocation.

— Another example is the handling of extensions in a certificate. There are basically no upper limits on the number of extensions in a certificate, even though most certificates only have a few. The analysis of our set of 200 million certificates shows that less than 200 different extensions exist in real life. Considering that [2] also requires each extension to appear only once in the certificate, enforcing this requirement with an upper bound of 200 extensions is a pretty easy way to avoid dynamic allocation. This would not have been possible when considering a huge or unlimited amount of extensions. In practice, an even lower bound is used in the parser.

One should also notice that avoiding dynamic allocation makes the parser more fitted to embedded devices tight constraints.

Limited use of function pointers. Function pointers are a useful tool in C but they must be used with care. For instance, the main loop handling the sequence of extensions in a certificate could incorporate a very large switch/case to call a specific handler but this would create a very large function. In practice, this is better achieved using `static const` structures associating function pointers with identifiers and additional useful data. Parser code makes limited use of this specific design pattern and prohibits the use of dynamic arrays of function pointers. This was expected to help static analysis tools follow the pointers.

No external dependencies. In order to avoid the possible security impact of external code and to facilitate the validation of this code in static analysis tools, the parser was built without dependencies to external libraries.

Use of `static`, `const` and alike qualifiers. The use of C qualifiers like `static` and `const` is very useful both to help compilers doing a better job but also to spot potential errors. They are obviously of great help for static analysis tools and require in the end only a minimal effort.

Use of unsigned integers of minimal length (`uint8_t`, `uint16_t`, etc.). `int`s are usually used without care in many C programs, for instance in situations where unsigned integers and even ones of a specific size (`uint8_t`, `uint16_t`, etc.) would be more suitable. The parser tries to use such specific integers when possible, in order for static analysis tools to benefit from the information embedded in the type (arithmetic sign, range of values, etc.). Exploring the effects of all the possible 256 values of an `uint8_t` is obviously far less complex than doing so for the 2^{32} values of an `uint32_t`.

Limited cyclomatic complexity. Both for human readability and to simplify later validation by static analysis tools, parser code has been written in order to keep functions as small as possible and to keep the cyclomatic complexity of the project low.

Strict compilation options. Before starting static analysis work, the feedback from compilers has been used as a useful tool during development to spot possible errors. This has been done using strict compilation options. As an example, the options used with `clang` to build the project are provided below:

```
clang -Weverything -Werror -Wno-reserved-id-macro \
      -Wno-unreachable-code-break \
      -Wno-covered-switch-default \
      -Wno-padded -pedantic -fno-builtin \
      -D_FORTIFY_SOURCE=2 -fstack-protector-strong \
      -std=c99 -O3 -fPIC -ffreestanding \
      -c x509_parser.c -o x509_parser.o
```

Because different tools provide different and complementary views of the project, the ability to build the project with `gcc` has been maintained.

No recursion. Obviously, recursion is both a discouraged coding practice in embedded devices and a disastrous construction for static analysis tools.

3.3 Testing and validating the X.509 parser

In order to experiment with the capabilities of the parser against real world certificates, a test suite has been gathered from various SSL/TLS test campaigns spanning from diverse sources over a few years, and the huge amount of certificates available from Certificate Transparency [5] logs.

This set of **200 million unique certificates** was used for various purposes in the project, including the computation of statistics on specific aspects of certificate content: real-world use of a given extension, possible alternative encodings of a specific field, recursion limits, etc. This also helped taking informed decisions on useless extensions, best ways to implement SHOULD of [2], and so on. Additionally, this set was a useful basis to measure the *performances* of the parser.

5. `https://www.certificate-transparency.org/`

Implementation decisions The RFC [2] is 150 pages long. This could seem rather small, but this represents nearly 400 SHOULD, SHALL, MAY and other MUST to implement for a valid X.509 parser.

As an example, the content of section 4.1.2.1 of [2], describing one of the most simple field in a certificate, the version field, is provided below. To be more specific, the following excerpt describes what the field should contain, but not how it should be encoded, which is given by ASN.1 notation and DER encoding. If you follow the RFC, you will have to support all possible version values and then ask yourself various questions like what to do from a version 1 certificate that includes extensions.

> *This field describes the version of the encoded certificate. When extensions are used, as expected in this profile, version MUST be 3 (value is 2). If no extensions are present, but a UniqueIdentifier is present, the version SHOULD be 2 (value is 1); however, the version MAY be 3. If only basic fields are present, the version SHOULD be 1 (the value is omitted from the certificate as the default value); however, the version MAY be 2 or 3.*
>
> *Implementations SHOULD be prepared to accept any version certificate. At a minimum, conforming implementations MUST recognize version 3 certificates.*
>
> *Generation of version 2 certificates is not expected by implementations based on this profile.*

Having a huge representative set gets interesting at that point, because you can take educated decisions about the content of the RFC. As shown on figure 2, keeping backward compatibility with v1 certificate does not make much sense since they are nearly non-present in the wild.

	Number	**Percentage**
v1	3890	0.002
v2	32	0.00002
v3	196467422	99.992
v4	11703	0.006

Fig. 2. Certificates version in our set.

In practice, we have almost 3 times more v4 certificates (what's that?) than v1 ones. In the end, considering the RFC and the information provided by our set, the decision was taken to accept only v3 certificates.

Beyond the simple version field issues, we briefly provide (non exhaustive) additional decisions we have made during the implementation using the experimental feedback of our test set.

Serial number field: certificate serial number is encoded as a positive integer. Both CAs and users are expected to support serial number field up to 20 octets. The set tells us that we have no certificate with a negative serial number, so we strictly follow the RFC on that aspect. Regarding serial number size, the set has certificates with serial number from 1 to 129 octets. Serials with a length above 20 bytes represent 0.02% of the set, i.e. they are marginal. For this reason, our implementation does enforce a maximum length of 20 bytes for serial number.

Subject Public Key Info: the set provides interesting statistics about the algorithms and what needs to be supported in the parser.

Extensions: the set tells us there are tens of different extensions in the certificates we have. We have no real reasons to try and support exotic extensions. The parser implements the most common extensions based on the statistics provided by the set.

In the end, even if the parser tries to follow the rules given in [2] as much as possible, unclear guidance and suggestions are handled using real world information using the set. Regarding the MUST, SHALL and so on requirements, the decisions taken and the status of the compliance with the standard are provided in a specific document of the project.

Unit and regression tests Having a huge set of different certificates is very useful. First, it allows to detect *regressions* in already implemented code (the number of validated certificates suddenly drops from 95% to 0 because a test was reversed). It is also used to validate new features as they are developed, providing some unexpected aspects of a feature (common or maximum number of element in a given `SEQUENCE`).

Another interesting aspect which is currently a work in progress is its use as an initial set for running AFL. This will be covered in the detailed version of the article [4].

4 Introduction to program analysis

4.1 Functional and security verifications, absence of RTE

When it comes to static analysis of programs, at least two kinds of properties are desirable.

Functional verifications. this is the task of verifying that an implementation conforms to its specification (i.e. the program behaves as it *should be*). For formal functional verification, the specification has to be expressed in a formal way and the verification has to be done for all possible runs. In Frama-C, the functional specification can be expressed by the user with function contracts and assertions. The kernel computes the validity status of each property with the information given by the called plugins to ensure the consistency of the complete verification process. A validated property means there is no concrete implementation that violates this property. This functional verification can concern functional behaviors (what the function is supposed to do) but also, more precise security properties on the implementation.

Security verifications, absence of RTE. RTEs are unfortunately common when programming with unsafe languages such as C and can be a fatal problem during execution. Such errors cover divisions by zero, invalid pointer accesses, integer overflows, etc. They can lead to a segmentation fault or an unexpected/erroneous execution but they can also be exploited for a malicious purpose (e.g. by tampering with the program execution flow). Safety and security are closely related especially when dealing with RTE detection, in order to avoid memory errors and undefined behaviors. The use of formal tools for the detection of RTE has been common for years for safety concerns, especially for critical systems [1]. In parallel, we can note growing interest for these tools but for security purposes [19].

4.2 Static and dynamic analyses, soundness and completeness

In this paper, we mainly focus on *static program analysis* techniques widely used to detect vulnerabilities, but *dynamic analysis* can also be used for this purpose. Dynamic analysis aims at verifying properties at runtime when executing paths of a given program [19] .

Most of the tools covered here are based more precisely on *abstract interpretation* [18]. Some of them deal with heuristics but only sound analyzers (e.g. Frama-C/EVA [13]) prove the complete absence of RTE.

The term *soundness* comes from formal, mathematical logic. The proof system is a set of rules with which one can prove properties (absence of RTE) about the model. Soundness refers to the fact that statements proven to be true using the tool's axiomatic logics and a proof system in a given model are indeed true. In that setting, there is a proof system and a model. The program (all of its executions) plays the role of the model and the static analysis plays the role of the proof system. The proof system behind the sound tools discussed in this article are proven to be sound: this does not mean that their implementation is indeed sound. Any bug in the proof system implementation yields in unusable results. This is also true when the model used for the proofs is not realistic (e.g. an over simple memory model) or when some ground axioms are trivially false. This induces real-world limitations for all the static analysis tools. However, one should be aware that although such limitations exist, the results of such tools are the best guarantees one can get on the absence of bugs (such as RTEs) in a program. These limitations also explain why beyond Frama-C, we have put the X.509 parser code under the scrutiny of other tools while intersecting the results. A static analysis tool is *unsound* if the tool claims a property holds when it does not in the program i.e. if there is at least one erroneous execution. False alarms are then a practical reality for sound tools but these tools can guarantee no missed errors (except for a bug in the tool as explained before). On the other side, we have *completeness*. A proof system is complete if it can prove any true statement about the model. Clearly, it means that complete tools never emits false alarms. For a valid program, a complete tool must not issue an alarm. *In practice, there is no tool that is both complete and sound.*

To reach a guaranteed RTE-free X.509 parser, a sound analyzer appears to be the appropriate approach.

Abstract interpretation is a static technique to compute over-approximations of all possible values during program execution for each memory location. In sound analyses, if a property is verified for all values in the over-approximation, *and only in that case*, then the property is validated for any concrete execution of the program. If a doubt persists, the tool will emit a *warning* on the concerned property and the remaining warnings have to be verified one by one, either with another analysis tool or directly by hand. If a property is required for an analysis, its validity is assumed but needs to be verified afterwards.

Sound analyzers are generally not so much used for code verification. Actually, these tools do not get a good press among developers because of various caveats: they are greedy in resources (time and space), they require some expertise to be handled, they generally do not offer user-friendly interfaces, and suffer from many limitations for the code to be analyzed.

Although some of these statements are purely subjective, such tools have indeed suffered from a lack of openness to users unfamiliar with formal methods. However, the situation has improved in the last years. In any case, these tools undoubtedly allow to get very strong guarantees on the analyzed code.

One of the purposes of this article is precisely to provide a feedback on how to make Frama-C converge towards the absence of RTE proof on a real-world example. Beyond the mere result, the path to get such working proofs is also discussed. The results provided by other static analyzers on the produced code are also discussed.

5 Working with Frama-C on the parser

5.1 Frama-C presentation

Frama-C (*Fra*mework for *m*odular *a*nalysis of *C* programs [14]) is an extensible and collaborative platform dedicated to source-code analysis and more specifically for C99 source code [6]. It is mainly co-developed at the Software Security and Reliability Laboratory of CEA-LIST and the Toccata team of INRIA Saclay. The Frama-C platform is open-source and allows to bring together several analysis techniques designed as plugins. It is also designed to be extensible and allows the user to design custom plugins in a relatively easy way depending on the type of analysis and on the platform.

The kernel provides a core set of features (basically the normalized AST [7] of the program) and allows these plugins to work together either in a parallel or serial fashion. Each plugin performs a precise analysis and/or an annotation of the source code shared or reused by the next analysis. Analyses done by Frama-C can be static or dynamic (resp. without or with the program execution), or both. For the vast majority of static analyses, Frama-C aims to be *sound* in the sense that it never misses a potential error in the class of bugs targeted.

5.2 ACSL code annotations in Frama-C

In Frama-C, the annotations of C programs are expressed in ACSL (ANSI/ISO-C Specification Language [12]), a formal specification language based on a first-order logical language and designed to express properties of a C program during its execution. ACSL is an easy-to-adopt specification language with a syntax close to the C syntax with some additional but

6. Frama-C also handles other front-ends beyond the scope of this article, but such analysis are not as mature as for the C code.

7. Abstract Syntactic Tree i.e a tree representation of the abstract syntactic structure of the source code.

explicit predicates. It clearly alleviates the writing of annotations for C programmers. Examples of the ACSL language can be found in [11]. Assertions are another feature allowing to express code properties that must be true at precise program points.

These ACSL annotations can be performed either automatically by Frama-C (e.g. by the RTE plugin that generates ACSL annotations to warn about RTEs) or by hand, directly by the user, to express properties based on function contracts. Function contracts allow the user to provide preconditions and postconditions for given functions. Preconditions (resp. postconditions) are the set of properties supposed to be true before the function is called (resp. at the end of the function execution).

Using ACSL and Frama-C allows to target a large range of functional and security verifications. Among them, proving safety properties and the absence of RTEs are historical ones and still remain the main objectives with Frama-C. Other security properties as well as formal behavioral modeling and specifications can also be expressed with the versatile framework.

5.3 ACSL by example

In order to illustrate this, let us take the *very simple* example of the `div` function that computes the euclidean division of `x` by `y` and stores the quotient in `*q` and the remainder in `*r` (see listing 3).

```
/*@ requires \valid(q) && \valid(r);
  @ requires 0 <= x && 0 < y;
  @ assigns *q, *r;
  @ ensures x == *q * y + *r && 0 <= *r < y;*/
void div(int x,int y, int * q, int * r)
{
  /*...*/
}
```

Listing 3. ACSL annotations of the `div` function.

The preconditions are introduced by the predicate `requires`, the postconditions by `ensures`: as we can see, the postcondition is expressed as the natural desired result of the euclidean division. The set of memory locations modified by the function is given with the `assigns` clause. If no `assigns` clause is defined for a function, the caller will have no information at all on this function's side effects and will over-approximate them. The keyword `valid` implies the verification of memory access for read and write here (for a read only access, the keyword `valid_read` is used). ACSL annotations can be directly written in C source files in comments starting with `/*@` or `//@`. They are used by Frama-C analyzers but do not interfere with the original code as they are classical C comments [8].

5.4 RTE, EVA and WP plugins

In this article, we specifically focus on three plugins: RTE [16], EVA [13] and WP [15], since only these three are used for the verification of our parser.

The RTE plugin systematically adds ACSL annotations to check potential Run Time Errors. It is an annotations generator and it does not perform the discharging of such annotations. This plugin is used to seed more advanced plugins such as WP. EVA [9] uses sound abstract interpretation. EVA proceeds to a complete value analysis of the analyzed program to warn about possible RTEs. In practice, the EVA plugin internally verifies RTEs and adds annotations only when it cannot prove them. The RTE plugin covers only a subset of RTE checks done by EVA so explicit calls to the RTE plugin can be skipped. EVA can also be used to prove simple explicit ACSL annotations or assertions in C code.

For more complex ACSL properties or assertions, another plugin, Frama-C/WP [10], is usually used. It implements deductive verification [3] calculus, a modular sound technique to prove that a property holds after the execution of a function if some other properties hold before it (pre/post condition as seen before). WP is able to verify more complex logical annotations and assertions using external automated or interactive provers (mainly *AltErgo*, *Why3* and *Coq*) but requires extra efforts with the code annotations including *loop annotations*. Indeed, to analyze a source code with loops, WP needs a specification for each of them or it uses an implicit specification which is equivalent to *"anything can happen"*.

A loop annotation is composed of a loop invariant (i.e. a general condition which is true, before and also *after* the loop and even, for each iteration), a loop variant (an integer expression that strictly decreases at each iteration and ensures the loop terminates) and possibly the list of assigned variables (as for function annotations, without an `assigns` clause, it means that potentially the loop modifies "everything").

The idea of weakest-precondition calculus is to build valid deductions based on Hoare logic [7].

8. We do not consider here the runtime verification and the executable ACSL annotations (E-ACSL).
9. Evolved Value Analysis.
10. Weakest Precondition.

5.5 Frama-C interactive and iterative workflow

We provide hereafter an overview of the workflow involving Frama-C and its plugin based on the expected results. Frama-C will accept C code, either with or without ACSL annotations: developers may be interested in annotating their code with expected functional or security properties.

Annotations may either help the work of the tool or make it more complex. For instance, manually adding loop annotations usually helps the tool to maintain precise information on manipulated elements. As a consequence, nearby functions and annotations may benefit or suffer from this additional information. Annotations that cannot be validated may impact the duration or the final status of the analysis.

The initial goal of the project is to prove the absence of RTE of the X.509 parser without specific functions or logical guarantees. This is why our main guideline was to focus on full verification of the code by using EVA and WP.

Frama-C can either be launched directly or using the GUI interface. In both cases, initial options for the analysis are provided on the command line as shown below:

```
frama-c-gui x509_parser.c -machdep x86_64 \
            -eva -wp-dynamic \
            -then \
            -wp -wp-dynamic
```

This instructs the tool to work on the `x509_parser.c` file, targeting the x86_64 architecture, and using first EVA plugin and then WP plugin. Because our code uses *function pointers* and associated annotations (`@calls`) which are discussed later, the `-wp-dynamic` option is required.

Running Frama-C on the current parser code **without annotations** generates 908 proof obligations: this means that an RTE check is added every 3.5 line of code on average (the parser is made of around 3,000 lines of effective code). After less than a minute the result is 134 proof obligations having an unknown status. This means, that, without any specific effort, the first 85% of proof obligations are validated.

Let's now try and improve the result of the analysis, still without performing any manual annotations yet.

For that purpose, the invocation of Frama-C was progressively improved:

```
frama-c-gui x509_parser.c -machdep x86_64 \
     -eva -eva-slevel 1 \
     -eva-slevel-function="find_dn_by_oid:100, \
                           find_curve_by_oid:100, \
                           find_alg_by_oid:200, \
                           find_ext_by_oid:200, \
                           parse_AccessDescription:400, \
                           parse_x509_Extension:400, \
                           parse_x509_Extensions:400, \
                           bufs_differ:200, \
                           parse_x509_tbsCertificate:400" \
     -eva-warn-undefined-pointer-comparison none \
     -wp-dynamic \
     -then \
     -wp -wp-dynamic -wp-unfold-assigns \
     -wp-par $(JOBS) \
     -wp-steps 100000 -wp-depth 100000 \
     -wp-split -wp-literals -wp-model typed_cast_ref \
     -wp-timeout $(TIMEOUT) -save $(SESSION)
```

Regarding EVA plugin, the following main options have been added:

- `-eva-slevel 1` and `-eva-slevel-function`: `slevel` is probably the main parameter for EVA operations. Increasing its value either globally or for a given function improves the precision of the analysis by making the analyzer unroll loops and propagate separately the states that come from the `then` and `else` branches of a conditional statement. This also has the side effect of making the analysis slower. Hence, a good strategy is to use a low global `slevel` value and specify higher values for functions that require using `-eva-slevel-function` option, as depicted above.
- `-eva-warn-undefined-pointer-comparison none` is used with care in order to silence undefined pointers comparisons. This is needed to prevent Frama-C from emitting warnings for all tests of function input parameters against `NULL`.

Regarding WP plugin, the following options have been added:

- `-wp-par`: this option limits the number of parallel processes runs for decision procedures.
- `-wp-split` splits conjunctions in generated proof obligations recursively into subgoals. It generates more but simpler goals.

— `-wp-literals`: this option exports string literals to provers.
— `-wp-model typed_cast_ref`: `"Typed+var+int+float"` default sound model is overriden. This specific option is discussed later.

Using these options, we reduce the proof obligations with an unknown status from 134 to 63, yielding in 7% unknown. An interesting observation is that skipping the WP pass with only EVA leaves 72 unknown obligations, meaning that WP does not help that much on reducing the number of RTE-added annotations after the EVA pass. When skipping the EVA pass and leaving only RTE and WP, 150 unknown obligations are left.

Sadly, Frama-C will not go any further by tweaking plugins' options. In order to move forward we had to help the tool by annotating the portions of the code that challenge the tool such as the *loop patterns* (`while`, `for`, etc). This specific manual interactive annotation phase to converge towards a fully proven code is described in the next sections.

5.6 Manual code annotations

An overview of the remaining proof obligations shows that they are almost all related to buffer accesses and initializations. Furthermore, a large amount of them are located inside loops. Listing 4 exhibits such a loop working on a buffer inside the `_extract_complex_tag()` function.

```
    for (rbytes = 0; rbytes < len; rbytes++) {
        t = (t << 7) + (buf[rbytes] & 0x7f);
        if ((buf[rbytes] & 0x80) == 0) {
            break;
        }
    }
```

Listing 4. Initial version of `_extract_complex_tag()` main loop.

Listing 5 shows how RTE/EVA rewrites the loop and their automatic annotation: they remain in an *unknown state* after EVA and WP passes.

```
rbytes = (unsigned short)0;
while ((int)rbytes < (int)len) {
  {
    /*@ assert rte: mem_access: \valid_read(buf + rbytes); */
    t = (t << 7) + (u32)((int)*(buf + rbytes) & 0x7f);
    /*@ assert rte: mem_access: \valid_read(buf + rbytes); */
    if (((int)*(buf + rbytes) & 0x80) == 0) {
      break;
    }
  }
  rbytes = (u16)((int)rbytes + 1);
}
```

Listing 5. Annotation by RTE of `_extract_complex_tag()` main loop.

As we can see, Frama-C plugins need help to understand that each read access to `buf[rbytes]` is valid during each iteration of the loop, whose number of iterations depends on `rbytes` and `len`. This is achieved by using dedicated ACSL annotations as shown on Listing 6:

— `loop invariant`, which provides a condition that remains true during each iteration of the loop[11]. In practice, multiple loop invariants can be specified;

— `loop assigns` that specify the elements allocated outside the loop but modified inside the loop;

— optional `loop variant` that provide a strictly decreasing non-negative integer value at each loop iteration.

```
/*@
  @ loop invariant 0 <= rbytes <= len;
  @ loop invariant \forall integer x ; 0 <= x < rbytes ==>
     ((buf[x] & 0x80) != 0);
  @ loop assigns rbytes, t;
  @ loop variant (len - rbytes);
  @ */
for (rbytes = 0; rbytes < len; rbytes++) {
    t = (t << 7) + (buf[rbytes] & 0x7f);
    if ((buf[rbytes] & 0x80) == 0) {
        break;
    }
}
```

Listing 6. Annotated version of `_extract_complex_tag()` main loop.

Even if such manual annotation will indeed help the plugins, out-of-bound accesses validation requires additional knowledge about the buffer validity and state when entering the loop. Since the buffer and its length are parameters of the function, a function contract for `_extract_complex_tag()` is needed and shown in listing 7. As discussed in section 5.2, this contract helps the tool to have preconditions, postconditions and side effects of the function. Frama-C plugins will use these elements when trying to validate the behavior of the function (manual annotations, RTE-added annotations, etc.). The `requires` clauses will be considered as a work hypothesis, and in this context `ensures` and `assigns` clauses will be validated. When a callee function `f1()` is encountered during the validation of a caller function `f2()`, the plugins will validate the requirements of `f1()` and benefit of the `ensures` properties in `f2()`.

The second `requires` regarding `((len > 0) && (buf != \null))` informs the plugins that when a non `NULL` buffer is passed to to the function with a positive length, all its `len` elements can be safely read.

11. The semantic of the `loop invariant` is a bit trickier than that, see [12].

```
/*@
  @ requires len >= 0;
  @ requires ((len > 0) && (buf != \null)) ==>
        \valid_read(buf + (0 .. (len - 1)));
  @ requires \separated(tag_num, eaten, buf+(..));
  @ requires \valid(tag_num);
  @ requires \valid(eaten);
  @ ensures \result < 0 || \result == 0;
  @ ensures (len == 0) ==> \result < 0;
  @ ensures (buf == \null) ==> \result < 0;
  @ ensures (\result == 0) ==> 1 <= *eaten <= len;
  @ assigns *tag_num, *eaten;
  @*/
static int _extract_complex_tag(u8 *buf, u16 len, u32 *tag_num, u16
    *eaten)
{
    u16 rbytes; u32 t = 0; int ret;
    if ((len == 0) || (buf == NULL)) {
        ret = -__LINE__;
        ERROR_TRACE_APPEND(__LINE__);
        goto out;
    }
    if (len > 4) { len = 4; }
    /*@
      @ loop invariant 0 <= rbytes <= len;
      @ loop invariant \forall integer x ; 0 <= x < rbytes ==>
          ((buf[x] & 0x80) != 0);
      @ loop assigns rbytes, t;
      @ loop variant (len - rbytes);
      @ */
    for (rbytes = 0; rbytes < len; rbytes++) {
        t = (t << 7) + (buf[rbytes] & 0x7f);
        if ((buf[rbytes] & 0x80) == 0) {
            break;
        }
    }
    /* Check if we left the loop w/o finding tag's end */
    if (rbytes == len) {
        /*@ assert ((buf[len - 1] & 0x80) != 0); */
        ret = -__LINE__;
        ERROR_TRACE_APPEND(__LINE__);
        goto out;
    }
    if (t < 0x1f) {
        ret = -__LINE__;
        ERROR_TRACE_APPEND(__LINE__);
        goto out;
    }
    *tag_num = t; *eaten = rbytes + 1; ret = 0;
out:
    return ret;
}
```

Listing 7. Annotated version of `_extract_complex_tag()` function.

Informally, the first `if` at the beginning of the function will ensure the conditions of the implication. It guarantees that just after this `if` `len` is positive and `buf` is not `NULL`, resulting in an ensurance that all `len` elements of `buf` can be read. The second `if` will limit the value of `len` to 4 if a buffer larger than that is provided. With these extra information on the validity of the buffer and the upper bound on its length, the plugins will be able to validate the loop annotations and use them to also validate the assert added by RTE inside the loop on buffer accesses. The plugins also maintain the `assigns` clause in the function contract to validate it upon return. An equivalent work is performed for `ensures` clauses. For the `_extract_complex_tag()` function, one important aspect given in the function contract is related to the value of `eaten` output parameter. When the function succeeds (return value is 0), `eaten` provides the number of elements in `buf` that were read and guarantees that the value is in the range `[1, len]`. Because the value of `eaten` is used by the caller upon success to progress in the buffer (i.e. skip `*eaten` first bytes), it is very useful for the plugins validating caller code to know how `eaten` and `len` are linked together.

With this example, one can see that annotating the code usually means:

— writing *basic* function contracts: in our parser code, the focus is put on buffer related information (validity, length, etc.). No functional property about what function does from a semantic standpoint is expressed nor validated in these contracts;

— writing loop annotations so that the plugins can maintain a precise state when handling loops and validate RTE-added annotations.

5.7 Dealing with function pointers

At various locations in the code, we use function pointers to access the right function. Using function pointers helps code factorization and structures versatility. However, Frama-C can have issues to handle them. We briefly describe hereafter how directed annotations can be used to validate function pointers.

In theory, Frama-C should be able to annotate functions pointers dereference and get the associated function depending on the context. Unfortunately, from our experience, the tool is not able to validate some of preconditions of the called functions. This is where the `calls` ACSL statement comes in actions: it is currently an undocumented feature as it is not part of [12], and is used to list possible values of a function pointer. Even if this manual annotation has the **side effect** of helping in the validation of the calls performed using a function pointer, it can also be used to provide guarantees on which functions can be called using a function pointer (see listing 8).

```
static int parse_AttributeTypeAndValue(const u8 *buf, u16 len, u16 *eaten)
{
    ...
    /*
     * Let's now check the value associated w/ and
     * following the OID has a valid format.
     */
    /*@ calls parse_rdn_val_cn, parse_rdn_val_x520name,
          parse_rdn_val_serial, parse_rdn_val_country,
          parse_rdn_val_locality, parse_rdn_val_state,
          parse_rdn_val_org, parse_rdn_val_org_unit,
          parse_rdn_val_title, parse_rdn_val_dn_qual,
          parse_rdn_val_pseudo, parse_rdn_val_dc;
      @*/
    ret = cur->parse_rdn_val(buf, data_len);
    if (ret) {
        ERROR_TRACE_APPEND(__LINE__);
        goto out;
    }
    ...
    ret = 0;
out:
    return ret;
}
```

Listing 8. Use of `calls` statement in `parse_AttributeTypeAndValue()`.

6 Results and feedback

6.1 Results overview

Our main result is that we have a working X.509 parser with RTE-free C code that is verified by Frama-C using and "EVA then WP" strategy.

Over the 8,000 lines of code (with comments) of the whole X.509 parser, 1,100 lines of annotations have been added (14% of the source code). Since this additional percentage is sufficient to guarantee a complete absence of RTE in the code, this seems like a reasonable investment.

But beyond the number of lines of annotation we had to introduce, an interesting indicator is also the amount of work and time that was necessary to obtain the expected results, as well as the learning curve of handling the Frama-C framework. Going from zero knowledge about Frama-C to the fully annotated and proven code took less than 5 calendar months (more on this in the feedback section) when most of the development of the parser spanned (with a similar effort level) over 12 calendar months.

Finally, another interesting indicator is the time Frama-C takes to execute its proofs. As we know, soundness comes at a cost, and some tools might take a tremendous amount of CPU time to converge towards a result. In our case, EVA and WP finish their processing in 15 minutes for the whole project on a common laptop with 8GB of RAM, which is very reasonable considering the amount of proof objectives of the project (up to around 18,000 when using `-wp-split`) and means that *anyone can reproduce validation* on their machine.

6.2 Annotation work complexity

The parser implementation contains a total of 99 defined functions and 193 functions calls (either directly or via function pointers). The number of decision points in the code is 674, among which 631 are `if` statements.

The whole implementation contains 35 loops, which are almost all used to progress in the ASN.1 main buffer during parsing. A few of these loops are used to iterate on global structures to find an entry (e.g. locate an entry with a given OID to call a function pointer provided by the associated entry).

Regarding annotations, the unique C file contains a total of 953 manually-added clauses, among which 112 are loop annotations. With a total of 35 loops in the code, this gives an average of 3 clauses for each loop. Function contracts represent most annotations with 768 clauses (336 ensures, 336 requires and 96 assigns). This gives an average of 8 clauses per function. The remaining annotations are 62 `assert` manually put in the code to help the tool insist on a specific aspect and 5 uses of `calls` clause where function pointer are dereferenced.

The conclusions that can be drawn from previous statistics are that the project has been developed in order to split all functions in smaller functions, thus reducing the complexity of the code. The cost of annotating the code has been limited to 3 annotations per loop and 8 on average per function contract.

6.3 Frama-C learning curve

Although this can be a subjective matter, we have found that the learning curve is pretty steep because a good understanding of some very classical quirks [12] is required (e.g. loop variants and invariants).

Self-discipline is also required for loops implementation in order to simplify annotations and efficiently get validation results. Complex loops with multiple elements evolving together are hard (if not impossible) to annotate, will possibly fail to be validated, and will increase or break the whole analysis time. This work shows that even a complex X.509 parser can be implemented using a limited amount of loop constructs (35). Additionally, all these loops can be written simply enough to be annotated and validated.

One interesting element regarding the complexity of the annotation work is the elements of ACSL language used for this purpose. When targeting the goal of the absence of RTE, the amount of elements required for annotations is a very limited subset of the specification [12]. Interestingly, this limited subset is sufficient to achieve RTE-free validation without requiring a thorough understanding of formal methods.

6.4 Feedback

Many static analysis tools do not require (or support) manual annotations. This is both an advantage and a disadvantage. On one hand, this reduces the time the developer has to spend but on the other hand, this makes it difficult to handle cases where the tool does not complete its analysis. ACSL annotations are very similar to C, which makes them straightforward to work with from a C developer perspective.

Frama-C is an actively developed framework with a responsive community and releases every 6 months. We indeed witnessed improvements on the analysis capabilities of the tool between consecutive versions (we essentially used Chlorine and Argon versions). There is an effort to keep up-to-date the various documentations for the tool and each plugin with each release even if, in certain cases, we failed to find all the useful information in these documentations. Fortunately, several public support options are available and provided by the Frama-C community. Another minor drawback is that external tutorial and examples, even if they help learning how to use the tool, can quickly get outdated.

When validating a complex piece of code, one of the downsides of the tool is that there is not always a clear strategy towards success. Even if the tool provides some interesting information (possible values for a given variable at a given point in the code, etc.), some experience and few attempts are sometimes required to get the right annotation and/or code modification. Having managed to validate a complete X.509 parser shows that this work remains feasible but it is not automatic and has probably been the most time-consuming task of this validation work. Things can also get frustrating when plugins options can either help verifying the code or completely destabilize the analysis (large increase of unproven goals or of the processing time).

7 Feedback with other tools

To complete the results obtained with the EVA then WP strategy, we have also tried another strategy and other tools.

Because of a lack of space, these detailed results are provided in the extended version of the article [4]. A valuable result of these tools is the confirmation of the absence of RTE (although false alarms have to be manually checked for some of them).

8 Conclusion

In a nutshell, we provide a RTE-free X.509 parser validated using the Frama-C framework (using an EVA then WP strategy).

We have shown that although such a formal tool can appear complex to handle at a first glance, it proves relatively intuitive and simple to use when compared to other sound solutions, even when using it on an existing code base. Specifically, the annotation system takes a reasonable amount of efforts to integrate in the code, provided that basic and obvious guiding coding rules are respected (simple functions, simple loops, etc.). From our standpoint, this *meet in the middle* strategy works well with Frama-C annotations capabilities and ultimately few minor rewriting of the code, which seems to be a suitable approach for C developers with little background in formal oriented tools. When compared to pure static analysis tools (without annotation), the annotations seem a better alternative that, assuming an initial learning and coding effort, avoid many hours of false positive results or non-convergence of the tool in a reasonable time.

Of course, absence of RTE does not mean absence of bug, but it surely is a big step forward when compared to the current situation of C-based parsers. The absence of RTE would nonetheless constitute a very interesting and possibly achievable goal using Frama-C via additional annotations, but this is left for future work.

An interesting parallel work would be to explore other languages than C with type-safeness and guarantees absence of RTE such as ADA and Rust. Comparing the time and coding efficiency to get an RTE-free working parser in such languages when compared to C plus Frama-C would provide a valuable feedback for the developers and the security community.

As a side note, a work in progress is the extension of the parser to support certificate signature validation and path validation to create a usable standalone X.509 stack.

References

1. J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static Analysis by Abstract Interpretation of Embedded Critical Software. *SIGSOFT Softw. Eng. Notes*, 2011.
2. S. Farrell S. Boeyen R. Housley W. Polk D. Cooper, S. Santesson. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. `https://www.ietf.org/rfc/rfc5280.txt`, May 2008.
3. E.W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *ACM*, 1975.
4. A. Ebalard, P. Mouy, and R. Benadjila. Journey to a RTE-free X.509 parser (extended). `https://www.sstic.org/2019/presentation/journey-to-a-rte-free-x509-parser/`, 2019.
5. K. Szkudłapski G. Delugré. Vulnerabilities in High Assurance Boot of NXP i.MX microprocessors. `https://blog.quarkslab.com/vulnerabilities-in-high-assurance-boot-of-nxp-imx-microprocessors.html`, 2017.
6. D. Benjamin H. Sidhpurwala, H. Böck. OpenSSL „Negative Zero“ issue. `https://www.openssl.org/news/secadv/20160503.txt`, 2016.
7. C.A.R. Hoare. An axiomatic basis for computer programming. *ACM*, 1969.
8. ITU-T. X.680: Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation. `https://www.itu.int/ITU-T/studygroups/com17/languages/X.680-0207.pdf`, 2002.
9. ITU-T. X.690: Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). `https://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf`, 2002.
10. ITU-T. X.509: Information technology – Open Systems Interconnection – The Directory: Public-key and attribute certificate frameworks. `https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.509-201610-I!!PDF-E&type=items`, 2016.
11. K. Hartig H. Pohl J. Burghardt, J. Gerlach. ACSL By Example Towards a Verified C Standard Library Version 5.1.0 for Frama-C Boron. `https://www.cs.umd.edu/class/spring2016/cmsc838G/frama-c/ACSL-by-Example-12.1.0.pdf`.
12. CEA LIST. ACSL: ANSI/ISO C Specification Language Version 1.13. `https://frama-c.com/download/acsl.pdf`.
13. CEA LIST. Eva - The Evolved Value Analysis plug-in. `https://frama-c.com/download/frama-c-eva-manual.pdf`.
14. CEA LIST. Frama-C User Manual. `http://frama-c.com/download/frama-c-user-manual.pdf`.
15. CEA LIST. Frama-C/WP. `https://frama-c.com/download/frama-c-wp-manual.pdf`.
16. CEA LIST. RTE - Runtime Error Annotation Generation. `https://frama-c.com/download/frama-c-rte-manual.pdf`.
17. D. Maloney M. Norman S. Tux M. Scire, M. Mears and P. Monroe. Attacking the Nintendo 3DS Boot ROMs. `https://arxiv.org/pdf/1802.00359.pdf`, 2018.

18. P. and R. Cousot. Abstract interpretation: "A" unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Annual ACM Symposium on Principles of Programming Languages*, 1977.
19. D. Pariente and J. Signoles. Static Analysis and Runtime Assertion Checking: Contribution to Security Counter-Measures. *SSTIC*, 2017.