Journey to a RTE-free X.509 parser

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

ANSSI

Abstract. The 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; especially 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.

One of the positive aspects with C is nevertheless 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.

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.

Table of Contents

Journey to a RTE-free X.509 parser 1
A. Ebalard, *P.* Mouy, *R.* Benadjila
1 Introduction 4
2 X.509 5
2.1 Introduction 5
2.2 ASN.1, BER and DER encoding 6
2.3 X.509 format 8
2.4 Vulnerabilities 8
CVE-2017-7932 9
3DS flawed ASN.1 parser 10
CVE-2016-5080 10
CVE-2017-2781 10
CVE-2017-9023 11
CVE-2017-2800 11
3 Parser development 11
3.1 Strategy for X.509 support 11
3.2 Development constraints 12
3.3 Testing and validating the X.509 parser 14
Implementation decisions 14
Unit and regression tests 20
4 Introduction to program analysis 20
4.1 Functional and security verifications, absence of RTE 20
Functional verifications: 20
Security verifications, absence of RTE 21
4.2 Static and dynamic analyses, soundness and completeness . 21
5 Working with Frama-C on the parser 22
5.1 Frama-C presentation 22
5.2 ACSL code annotations in Frama-C 23
5.3 ACSL by example 25
5.4 Rte, EVA and WP plugins 25
5.5 Frama-C interactive and iterative workflow 26
5.6 Manual code annotations 29
5.7 Dealing with function pointers 32
6 Results and feedback 33
6.1 Results overview 33
6.2 Annotation work complexity 34

6.3 Frama-C learning curve 34
6.4 Conclusions about Frama-C usage 35
7 Analyses with other tools 35
7.1 Sound and fully automatic strategy to prove the absence of RTE 36
Frama-C EVA 36
Code Prover 40
7.2 Other tools 42
8 The story of a logical bug 44
9 Conclusion 48

1 Introduction

In a nutshell, the project was initiated with the intent 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 (for instance 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 [9] or [7] 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 disastrous due to the critical privilege level of such primitives. Hence, these parsers appear as a perfect target 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 goal of a "guaranteed RTE-free" parser has been achieved using Frama-C, an open-source C analysis framework. 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 taking into account the need to analyze the code at a later statge)
- 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 detail later in the document - was to test the ability for a **standard and average** - but motivated - C developer to successfully achieve verification with a limited additional investment.

From the observation of the limitations of 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 was taken to make the code amenable for a later analysis.

At this point, one may wonder why the term "guaranteed absence of RTE" is used instead of "proven" or "bug-free" to describe the result of the analysis. 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, shifts errors, etc.) on all possible executions of the parser. Even if care has been taken during the implementation to deal with logical errors (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.

The remaining of the article first gives a quick overview of the X.509 format and its 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

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.

In more details, a lot of other elements are also present such as an issuer, validity dates, key usages, subject alternative names, and various other optional extensions which contribute to make certificates rather complex objects.

To make things worse, certificate structures and fields content are defined using ASN.1 [11] and each specific instance is DER-encoded [12]. As it will be 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.

1. hereafter referred as RTE

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

If this was not already difficult enough, parsing and validating a 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 [5]. 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].

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. This 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 to encode 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) [12]. 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

2. https://cabforum.org

now on, we will only focus on DER encoding as it is the one specified by the X.509 standard.

Even though no ambiguity exists in DER, encoding and decoding ASN.1 is still quite complex and error-prone. Fig. 1 provides a concrete example on how a very simple type like the `INTEGER` 2 is DER encoded (found in X.509 v3 certificates).

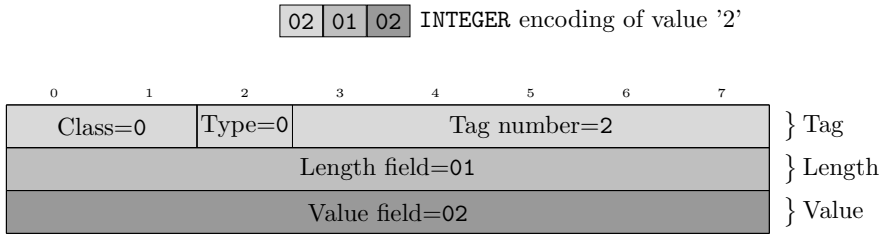

Fig. 1. ASN.1 simple `INTEGER` encoding

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 Tag 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. Because the type is primitive (first two bits of the first byte are 0), the specification requires the length to be encoded using what is called the definite form [3], which means that *"the length octets shall consist of one or more octets, and shall represent the number of octets in the contents octets using either the short form or the long form as a sender's option."* (section 8.1.3.3 of [12]). 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.

3. In the indefinite form, no length is provided and the content octets are marked using a specific value named End-of-contents octets.

We now know that the content (i.e. value) of the `INTEGER` is encoded on 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 Fig. 2. The standard [13] [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 (each one being possibly a `SEQUENCE` or a `SET` of other elements in a recursive fashion).

The first element in the `TBSCertificate` sequence is a field named `version` of type `Version`, which is defined as an `INTEGER` taking three different values indicating the version of the certificate. If absent, the certificate version is v1.

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 a certificate showing the inherent structural complexity of X.509. [11] is the `extensions` field as presented on Fig. 3. The `extensions` field is a `SEQUENCE` of `Extension`, which are themselves `SEQUENCEs` of 3 elements: an object identifier, a criticality bit and a value encoded as an `OCTET STRING`, that can then be decoded specifically based on the object identifier value. Additionally, the various extensions have structures that are more complex than the main certificate fields.

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.

4. [5] contains the same description.

```
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 }
```

Fig. 2. X.509 certificate high level structure

CVE-2017-7932 Various NXP ARM Systems On Chip (SoC) share a common mechanism called High Assurance Boot (HAB) 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. [7] describes a stack-based buffer overflow in the use of `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 of such an exploit 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 bootrom version for such processors was to switch to new

```
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 ::= {...}
```

Fig. 3. X.509 extensions ASN.1 structure

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 [20] 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 PKCS#1v1.5 padding of the 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 instead 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 bug.

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 very few results. One of the reasons behind is probably the inherent complexity of ASN.1 (even when considering that DER is the simplest encoding).

Additionally, because of the semantic complexity added by X.509, even if we were able to use a clean DER ASN.1 parser, many requirements would have to be checked regarding the X.509 implementation dealing with the certificate structure.

For all these reasons, the development was performed by implementing incrementally manner the minimal support functions to progress through the DER encoding of X.509 structures while also taking into account the semantic elements of [5].

The use of a vast representative test set, as discussed in subsection 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 implementation complexity. This was a first step towards using Frama-C.

3.2 Development constraints

Various development rules were followed in order to ease the use of static analysis tools. They were not tailored specifically to Frama-C. These design patterns are usually advised best practices when a 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 like variable-length arrays (VLA).

- ***No dynamic allocation***: care has been taken not to 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 for 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 reducing the need for dynamic allocation.
  — Another example is the handling of extensions in a certificate. There is no upper limit 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 [5] also requires each extension to appear only once in the certificate, enforcing this requirement with an upper bound of 200 extensions is pretty easy without 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 the tight constraints of embedded devices.

- ***Limited use of function pointers***: function pointers are a useful tool in C but 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 specific handlers 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 a 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 pointers.

- ***No external dependencies***: in order to avoid both the possible security impact of external code and the ability to validate 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 to use them.

- ***Use of unsigned integers of minimal length (`uint8_t`, `uint16_t`, etc.)***: `int` 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 (signed arithmetic, 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 for simplifying later validation by static analysis tools, parser code has been written in order to keep functions as small as possible and 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, here are the options used with `clang` to build the project:

```
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 retained.

- ***No recursion***: recursion is both a discouraged coding practice in embedded environments as well as 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 was 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 [5], and so on. Additionally, this set was an useful basis to measure the performances of the parser.

Implementation decisions The RFC [5] is 150 pages long. This could seem rather small, but this represents nearly 400 SHOULD, SHALL, MAY and other MUST in order to obtain a valid X.509 parser.

Version

As an example, let us analyze the content of section 4.1.2.1 of [5], describing one of the most simple fields in a certificate, the version field, is provided below. To be more specific, the following describes what the field should contain, but not how it should be encoded, since this is determined by ASN.1 notation and DER encoding. If one follows the RFC, one will have to support all possible version values and then ask yourself various questions like what to do from a version 1 certificate that

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

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 really helps at this point, because you can take educated decisions about the content of the RFC. As shown on Figure 5, keeping backward compatibility with v1 certificates does not make much sense because they are infrequent in the wild.

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

Fig. 4. Certificates version in our set

In our set, 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 made to accept only v3 certificates.

Beyond the simple version field issues, we briefly provide a non exhaustive list of additional decisions that we made during the development 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 fields 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 numbers.

Subject and Issuer fields

Subject and issuer fields must contain a Distinguished Name (DN) which is itself made of Relative Distinguished Names (RDN). Each RDN has an OID defining its type and a specific value, which depends of its type. A parser has to understand and validate the value and its encoding.

Here is what section 4.1.2.4 of [5] expects for Issuer (and also for Subject field):

> *Standard sets of attributes have been defined in the X.500 series of specifications [X.520]. Implementations of this specification MUST be prepared to receive the following standard attribute types in issuer and subject (Section 4.1.2.6) names:*
>
> * *country,*
> * *organization,*
> * *organizational unit,*
> * *distinguished name qualifier,*
> * *state or province name,*
> * *common name (e.g., "Susan Housley"), and*
> * *serial number.*
>
> *In addition, implementations of this specification SHOULD be prepared to receive the following standard attribute types in issuer and subject names:*
>
> * *locality,*
> * *title,*
> * *surname,*
> * *given name,*
> * *initials,*
> * *pseudonym, and*
> * *generation qualifier (e.g., "Jr.", "3rd", or "IV").*

The parser currently supports all the 14 attributes above (MUST and SHOULD) as a starting point. Because an unknown attribute cannot have its content validated, a certificate with such an unknown attribute is simply refused.

The set provides interesting statistics about the attributes that are commonly found in practice on the Internet, as presented below. The attributes expected by [5] are colored in orange.

# of occurences	OID value	OID name
126647362	2.5.4.3	commonName
76190507	2.5.4.11	organizationalUnitName
73984568	2.5.4.6	countryName
71709230	2.5.4.10	organizationName
50480910	2.5.4.7	localityName
50150074	2.5.4.8	stateOrProvinceName
4595889	2.5.4.5	serialNumber
1227210	1.2.840.113549.1.9.1	emailAddress
1103699	1.3.6.1.4.1.311.60.2.1.3	MS EV JOICountryName[6]
999503	2.5.4.17	postalCode
553996	0.9.2342.19200300.100.1.25	domainComponent
376837	2.5.4.13	description
259124	2.5.4.15	businessCategory
220997	2.5.4.4	surname
21506	1.2.840.113549.1.9.2	unstructuredName
20566	2.5.4.42	givenName
18509	2.5.4.9	streetAddress
10373	0.9.2342.19200300.100.1.1	userid
2391	2.5.4.97	organizationIdentifier
1500	2.5.4.46	dnQualifier
1092	1.2.840.113549.1.9.8	unstructuredAddress
569	2.5.4.12	title
437	0.0	zeroDotZero
399	2.5.4.41	name
217	2.5.4.18	postOfficeBox
81	2.5.29.17	subjectAltName
56	1.3.6.1.4.1.311.60.2.1.1	MS EV JOILocalityName[7]
44	2.5.4.45	uniqueIdentifier
36	1.3.6.1.4.1.18838.1.1	Spanish national ID
36	0.9.2342.19200300.100.1.3	rfc822Mailbox
23	1.3.6.1.4.1.16533.30.1	???
21	2.5.4.43	initials
19	1.3.6.1.7	mail
15	2.5.4.45.17	???
12	2.5.4.20	telephoneNumber
11	1.3.6.1.4.1.311.60.2.1.2	MS EV JOISOPName[8]
8	2.5.4.54	dmdName
7	2.5.4.65	pseudonym
5	1.3.6.1.4.1.23267.2.3	???
5	2.5.4.16	postalAddress
3	0.9.2342.19200300.100.1.4	info
2	1.3.6.1.4.1.23727.1.1.1	id-nat-uri
2	2.5.29.19	basicConstraints
2	2.5.4.72	role
...	...	...

Fig. 5. Statistics of RDN types in Issuer and Subject fields

The statistics above first tells us that the main extensions expected to be supported by [5] are the most represented.

The `emailAddress` attribute is less represented in the set and described in the following way in [5]:

> *Conforming implementations generating new certificates with electronic mail addresses MUST use the rfc822Name in the subject alternative name extension (Section 4.2.1.6) to describe such identities. Simultaneous inclusion of the emailAddress attribute in the subject distinguished name to support legacy implementations is deprecated but permitted.*

At the moment, we implement a strict strategy regarding the appearance of an `emailAddress` in subject or issuer fields.

Another interesting aspect provided by the statistics is related to the two attributes in red: some people generated funny certificates with subject or issuer fields with attributes using the OID used for the `subjectAltName` and `basicConstraints` extensions. Welcome to the Internet.

In the end, the statistics also tell us which additional attributes would be needed to validate most certificates in our set.

Validity

Dates in notBefore and notAfter elements of validity field can either be encoded using `generalizedTime` or `utcTime` encoding, based on their value. Most certificates (more than 99%) currenlty have dates encoded using `utcTime`. This was expected because most dates are currently before 2050. 26 certificates of the set have `notBefore` values that are after `notAfter` value.

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 reason to try and support exotic extensions. The parser implements most common extensions based on the statistics provided by the set. Additionally, any given type of extension can appear only once in a certificate. That rule is enforced in the parser. The set tells us that there are 13 certificates that rejected because of this rule.

In the end, even if the parser tries to follow the rules given in [5] 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, a dedicated document describes the decisions taken and the compliance status with respect to the standard.

As an example, our implementation currently does not support the Subject Information Access extension: we only have 135 certificates with it in our set. Requirements associated with this extension are marked as unfulfilled in the compliance document. We also do not support the Name constraints extension: it is complex and rare, considering all the statistics we have using the CA certificates in our set.

Unit and regression tests Having a large set of different certificates is very useful for testing. 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` for instance).

Another interesting use of the test set which is currently a work in progress is its use as an initial set for running AFL. This may be covered in a future version of the article [3].

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*). 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 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 address 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 in 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. It has been a few years since the use of formal tools for safety concerns has become common, especially for critical systems [4]. There is also a growing interest in these tools to tackle security properties [25].

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 [25] .

Most of the tools covered here are based on *abstract interpretation* [24]. Some of them use heuristics but only sound analyzers (e.g. Frama-C/EVA [16]) 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 may produce invalid results. This is also true when the model used for the proofs is not realistic (e.g. an oversimplified 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 regarding 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. 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 a practical reality for

sound tools but these tools can guarantee that there are no missed errors (except because of 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. It means that complete tools never emit false alarms. For a valid program, a complete tool must not issue any alarm. *In practice, there is no complete and sound tool.*

To build 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 there is any remaining doubt, the tool will emit a *warning* on the involved 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 used so much for code verification. Actually, these tools are not much liked among developers because of the various caveats: they are resource-hungry (time and space), they require some expertise to be handled, they generally do not offer user-friendly interfaces, and suffer from many limitations.

Although some of these statements are purely subjective, such tools were not easily mastered by users unfamiliar with formal methods. However, the situation improved in the recent years. In any case, these tools undoubtedly allow one 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 provide a proof of RTE absence 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) [17] is a modular and collaborative platform dedicated to source-code analysis and more specifically for C99 source code [9]. It is mainly co-developed at

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

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. Fig. 6 gives an overview of the open-source plugins done by CEA-LIST and this gallery is a good illustration of the power (and the complexity) of this platform. It is also designed to be expandable and allows the user to design custom plugins in a easy way depending on the type of analysis and on the platform.

The kernel provides a core set of features (basically the normalized AST [10] of the program) and allows plugins to work together either in a parallel or serial way. Each plugin performs a precise analysis and/or an annotation of the source code available for the next analyses. Analyses done by Frama-C can be static or dynamic (resp. with or without 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 [11] [15], a formal specification language based on a first-order language and designed to express properties of a C program. ACSL is an easy-to-adopt specification language with a syntax close to C with some additional but explicit predicates. It clearly alleviates the writing of annotations for C programmers. Examples of the ACSL language can be found in [14]. 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

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

11. ANSI/ISO-C Specification Language

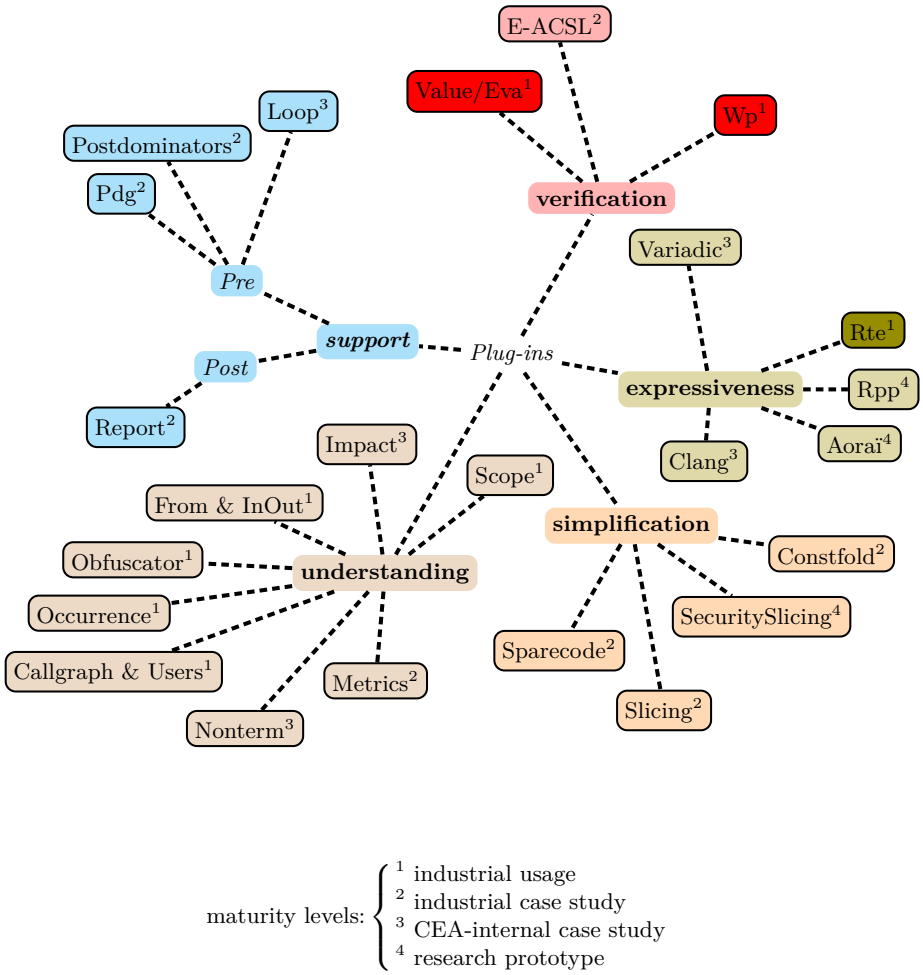

Fig. 6. Open-source CEA-List's Frama-C Plug-in Gallery.

with Frama-C. Other security properties as well as formal behavioral modeling and specifications can also be expressed with this 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 1).

```
1 /*@ requires \valid(q) && \valid(r);
2   @ requires 0 <= x && 0 < y;
3   @ assigns *q, *r;
4   @ ensures x == *q * y + *r && 0 <= *r < y;*/
5 void div(int x, int y, int * q, int * r)
6 {
7   /*...*/
8 }
```

Listing 1. ACSL annotations of `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 [12].

5.4 Rte, EVA and WP plugins

In this article, we specifically focus on three plugins: Rte [19], EVA [16] and WP [18], 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 can be used

12. We do not consider here the runtime verification and the executable ACSL annotations (E-ACSL).

to feed more advanced plugins such as WP. EVA [13] 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. Rte plugin is thus useless with EVA. 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, WP (Weakest Precondition), is used. It implements deductive verification [6] 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, during and also *after* the loop), 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 [10].

5.5 Frama-C interactive and iterative workflow

We provide hereafter an overview of the workflow involving Frama-C and its plugins based on the expected results. Frama-C accepts 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 of the analysis, create additional timeouts and prevent completion of the analysis.

13. Evolved Value Analysis

The initial goal of the project is to prove the absence of RTE of the X.509 parser without logical guarantees. This is why our main guideline was to focus on full verification of the code 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 3000 lines of real code). After less than a minute the result is 134 proof obligations having an unknown status. This means, that, without any specific effort, 85% of the proof obligations are validated.

Let's now try to improve the result of the analysis, still without performing any manual annotation yet. For that purpose, the invocation of Frama-C is progressively improved using options:

```
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 additional 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 it 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`. [14]

Regarding WP plugin, the following options have been added:

— `'-wp-par'`: this options limits the number of parallel processes runs for decision procedures.
— `'-wp-split'`: this option splits conjunctions in generated proof obligations recursively into subgoals. This has the effect of generating 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 rate. 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 have to help the tool by annotating the portions of the code that challenge it such as the *loop patterns* (`while`, `for`, etc). This specific manual interactive annotation phase aimed at converging towards a fully proven code is described in the next sections.

14. The strongest hypothesis that EVA relies on is that it is possible to pass from one address to another if and only if the two addresses share the same base address.

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 2 exhibits such a loop working on a buffer inside the `_extract_complex_tag()` function.

```
1     for (rbytes = 0; rbytes < len; rbytes++) {
2         t = (t << 7) + (buf[rbytes] & 0x7f);
3         if ((buf[rbytes] & 0x80) == 0) {
4             break;
5         }
6     }
```

Listing 2. Initial version of `_extract_complex_tag()` main loop

Listing 3 shows how Rte/EVA rewrites the loop and their automatic annotation: they remain in an *unknown state* after EVA and WP passes.

```
1   rbytes = (unsigned short)0;
2   while ((int)rbytes < (int)len) {
3     {
4       /*@ assert rte: mem_access: \valid_read(buf + rbytes); */
5       t = (t << 7) + (u32)((int)*(buf + rbytes) & 0x7f);
6       /*@ assert rte: mem_access: \valid_read(buf + rbytes); */
7       if (((int)*(buf + rbytes) & 0x80) == 0) {
8         break;
9       }
10    }
11    rbytes = (u16)((int)rbytes + 1);
12  }
```

Listing 3. 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 4:

- `loop invariant`, the construct provides a condition that remains true during each iteration of the loop [15]. In practice, multiple loop invariants can be specified.
- `loop assigns`, the construct specifies the elements allocated outside the loop but modified inside the loop.
- optional `loop variant`, the construct provides a strictly decreasing non-negative integer value at each loop iteration.

15. The semantic of the `loop invariant` is a bit trickier than that, see [15].

```
    /*@
      @ 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 4. 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 5. As discussed in section 5.2, this contract helps the tool to grasp preconditions, post-conditions 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 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 from the `ensures` properties in `f2()`.

```
/*@
  @ 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)
{
```

```
16      u16 rbytes; u32 t = 0; int ret;
17      if ((len == 0) || (buf == NULL)) {
18          ret = -__LINE__;
19          ERROR_TRACE_APPEND(__LINE__);
20          goto out;
21      }
22      if (len > 4) { len = 4; }
23      /*@
24        @ loop invariant 0 <= rbytes <= len;
25        @ loop invariant \forall integer x ; 0 <= x < rbytes ==>
26           ((buf[x] & 0x80) != 0);
27        @ loop assigns rbytes, t;
28        @ loop variant (len - rbytes);
29        @ */
30      for (rbytes = 0; rbytes < len; rbytes++) {
31          t = (t << 7) + (buf[rbytes] & 0x7f);
32          if ((buf[rbytes] & 0x80) == 0) {
33              break;
34          }
35      }
36      /* Check if we left the loop w/o finding tag's end */
37      if (rbytes == len) {
38          /*@ assert ((buf[len - 1] & 0x80) != 0); */
39          ret = -__LINE__;
40          ERROR_TRACE_APPEND(__LINE__);
41          goto out;
42      }
43      if (t < 0x1f) {
44          ret = -__LINE__;
45          ERROR_TRACE_APPEND(__LINE__);
46          goto out;
47      }
48      *tag_num = t; *eaten = rbytes + 1; ret = 0;
49  out:
50      return ret;
51  }
```

Listing 5. Annotated version of `_extract_complex_tag()` function

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

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 a 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, based on our experience, the tool is not able to validate some preconditions of the called functions. This is where the `calls` ACSL statement comes into play: it is currently an undocumented feature as it is not part of [15], 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 regarding which functions can be called using a function pointer (see Listing 6).

```
1 static int parse_AttributeTypeAndValue(const u8 *buf, u16 len, u16 *
      eaten)
2 {
3     ...
4     /*
5      * Let's now check the value associated w/ and
6      * following the OID has a valid format.
7      */
8     /*@ calls parse_rdn_val_cn, parse_rdn_val_x520name,
9           parse_rdn_val_serial, parse_rdn_val_country,
```

```
10              parse_rdn_val_locality, parse_rdn_val_state,
11              parse_rdn_val_org, parse_rdn_val_org_unit,
12              parse_rdn_val_title, parse_rdn_val_dn_qual,
13              parse_rdn_val_pseudo, parse_rdn_val_dc;
14        @*/
15      ret = cur->parse_rdn_val(buf, data_len);
16      if (ret) {
17          ERROR_TRACE_APPEND(__LINE__);
18          goto out;
19      }
20      ...
21      ret = 0;
22  out:
23      return ret;
24  }
```

Listing 6. 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 9000 lines of the whole X.509 parser, about 5000 lines are real code (without comment and blank lines) and 1200 lines of annotations have been added (24% of the source code). Considering this additional percentage is sufficient to guarantee a complete absence of RTE in the code, this seems like a reasonable investment.

Beyond the number of lines of annotations we had to introduce, an interesting indicator is the amount of work and time that was necessary to obtain the expected results, as well as the learning curve for handling the Frama-C framework. Going from no 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 the same effort level) 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 (≈ 18000 [16]) and means that *anyone can reproduce validation*) on their machine.

16. when using '`-wp-split`'

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 project has been developed in order to split all functions in smaller functions, thus reducing the complexity of the code. As can be seen from the above statistics, 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 is required (e.g. loop variants and invariants) [15].

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 [15]. Interestingly,

this limited subset is sufficient to achieve RTE-free validation without requiring a thorough understanding of formal methods.

6.4 Conclusions about Frama-C usage

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. Indeed, 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 to learn 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 several 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 free 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 Analyses with other tools

To complete the results obtained with the "EVA then WP" strategy, we have also tried another strategy to find RTEs but also other tools to find defects in our parser.

7.1 Sound and fully automatic strategy to prove the absence of RTE

The previous strategy is based on a combination of Frama-C plugins to obtain a fully automated verification of our parser. As explained before, we have chosen to spend time annotating our code at the beginning of the verification to obtain a fully proven RTE-free code. It is not the usual workflow. Generally, an automatic tool is directly used on code without user interaction (as code annotations) and at the end of the analysis, the chosen tool (based or not on sound analysis) emits a list of warnings including false positives. In the ideal case, the tool is based on sound analysis and no warning is emitted but in practice, on real code, this scenario does not happen. The second part of the classical workflow consists in a manual investigation of all these warnings. The real RTE are, of course, fixed and the analysis is done again. The difficulty is about the false positives: for each of them, a human investigation has to be performed to confirm it is not an actual RTE. This manual and tedious work has to be done and redone at each code modification.

In the interest of a fair comparison, we have first decided to provide the results of Frama-C with this strategy: no manual annotation in a fully automated way only by using EVA.

Frama-C EVA As explained before, EVA is often used to automatically prove the absence of RTE with minimal user interaction (ideally in a fully automatic way): contrary to WP, EVA does not require code annotations. Each annotation has to be verified by EVA and too many annotations can even complicate the analysis. So, for this new strategy, we activated the `-no-annot` option in order to ignore ACSL annotations. This approach to analyze our code in a "EVA only" strategy is similar to those of other sound tools.

The invocation and the results of this fully automated way to use EVA are given in Fig. 7 where `-machdep x86_64` defines the machine dependent configuration and `-eva-ilevel 64` indicates the number of elements below which sets of integers should be precisely represented.

An important point to note about EVA is that code annotations are added in the source only where a potential RTE remains; it means only if EVA failed to prove the RTE cannot occur at that specific location. This implies we can not reason in terms of valid checks but only in terms of remaining warnings or errors.

Tool Version	Errors	Warnings	Time Stats
18.0 Argon	0	227	4.5 s

EVA invocation: `frama-c x509-parser.c -no-annot -machdep x86_64 -eva-ilevel 64 -eva`

Fig. 7. EVA: results and invocation of the first run

By adding the `-eva-warn-undefined-pointer-comparison none` option to suppress the warnings emitted by EVA about the comparison of pointers with `NULL` [17], 178 warnings remain.

Because some headers in the standard C library provided with Frama-C use ACSL annotations and because the previous `-no-annot` option also make the tool ignore the use of these dedicated annotations, we then changed our way to use EVA. We simply removed all the manual annotations from the source code and also removed the `-no-annot` option. ACSL annotations from the standard C library are then considered and we run a new analysis. This way, the number of remaining warnings decreases to 69 in 4.3 seconds as shown in Fig. 8.

Tool Version	Errors	Warnings	Time Stats
18.0 Argon	0	69	4.3 s

EVA invocation: `frama-c x509-parser.c -machdep x86_64 -eva-ilevel 64 -eva-warn-undefined-pointer-comparison none -eva`

Fig. 8. EVA: results and invocation before tweaking of options and domains

To go further and improve the results, we have then used the `slevel` option and added the recommended domains as explained in the EVA manual and tutorial (cf [16], [1] or [2]).

To explain a little more the path taken to select EVA options, Fig. 9 gives detailed results, options and domains for various runs. The interesting option values or domains which improved EVA results on our parser verification are emphasized with blue color in Fig. 9.

Some options, not mentioned in this figure (for readability) are used, such as options about the architecture, the size of integers, the `-eva-warn-undefined-pointer-comparison none` option, ...

17. One of the main hypothesis done by EVA is to only allow manipulations between pointers with a same base address.

We then run EVA iteratively with the options and domains appearing to be good candidates to improve the analysis according to the available documentation of the tool.

In Fig. 9, additional options are present compared to our "EVA then WP" strategy as `-eva-symbolic-locations-domain`. Indeed, these options were not necessary to achieve the complete verification of our code but were useful to improve the results in the "EVA only" strategy. The `-eva-symbolic-locations-domain` option - highly recommended in EVA documentation - performs a special analysis for reused left-values from a conditional and then allows to refined results. The `-eva-split-return auto` option tells Frama-C to automatically split the states according to the function return. For this "EVA only" strategy, we have also changed some values of the default options: in practice `-eva-slevel 12` and `-slevel-function` fixed to 400 only for the function `parse_x509_Extensions`.

In the successive EVA experiments exposed in Fig. 9, we can also note various combinations of options and domain do achieve, more or less, the same result. This is, in a way, a revealing element of the large amount of options of EVA and the difficulty to find the correct way to proceed; indeed, no real procedure or even hints are available. We do not detail all these options in this paper; the interested reader will find them in the tool documentation.

Because of the myth of `-eva-slevel 1000000`[26], we also increased significantly the value of this parameter, but in a reasonable way i.e. up to 1000. Because of this global `slevel` set to a high value, we have suppressed the `slevel-function` in these invocations. We did not try a higher value because we were not inclined to accept a longer analysis time especially without a gain in the analysis result.

Our best result with EVA was 58 warnings in less than 2 minutes (112s). The associated invocation is given in Fig. 10

All these warnings are about memory access and more precisely the verification that a memory location is valid for reading.

To go further, we increased the amount of checks using the `-warn-right-shift-negative`, `-warn-signed-downcast`, `-warn-unsigned-downcast`, `-warn-unsigned-overflow` options. Indeed, by default, Frama-C only checks RTE which correspond to undefined behaviors according to the C99 standard and these options correspond to defined behaviors and are then not set by default. By taking the previous

26. Private joke in the Frama-C community

First tweaking of `slevel` value:

1								69	4.5s
10								66	34.0s
50								62	229.3s
100								62	689.3s
30								62	120,3s
20								64	71,7s

Tweaking of domains

slevel [18]	esld [19]	ebd [20]	esr [21]	sf [22]	eed [23]	eqtc [24]	egd [25]	#W	Time
1	✓							67	7.1s
30	✓							62	164.3s
1		✓						69	5.9s
30		✓						62	146.9s
1			✓(full)					69	5.1s
30			✓(full)					62	7936.8s
1			✓(auto)					69	5.2s
30			✓(auto)					58	151.2s
1					✓			69	6.3s
30					✓			62	186.8s
1						✓(all)		69	5.0s
30						✓(all)		62	119.4s
1						✓(formals)		69	5.0s
30						✓(formals)		62	117.9s
1							✓	69	6.3s
30							✓	62	154.8s

Combination of - visibly helpful - domains and options:

30	✓		✓(auto)					58	221.4s

Addition of `slevel-function`:

1	✓		✓(auto)	♣				60	95.0s
30	✓		✓(auto)	♣				58	203.7s
20	✓		✓(auto)	♣				58	183.5s
10	✓		✓(auto)	♣				58	122.7s
1	✓		✓(auto)	♠				65	221.3s
3	✓		✓(auto)	♠				64	45s
10	✓		✓(auto)	♠				61	87.8s
20	✓		✓(auto)	♠				58	186.9s
12	✓		✓(auto)	♠				58	111.9s

With significant increase of `slevel` option:

300	✓		✓(auto)					58	2867.2s (>47mn)
500	✓		✓(auto)					58	4332.7s (>2h)
1000	✓		✓(auto)					58	59936.6s (>16h)

- esld ≡ -eva-symbolic-locations-domain
- ebd ≡ -eva-bitwise-domain
- esr ≡ -eva-split-return auto/full
- sf ≡ -slevel-function
- eed ≡ -eva-equality-domain
- eetc ≡ -eva-equality-through-calls full/formal
- egd ≡ -eva-gauges-domain
- ♣ ≡ same functions and values than in "WP then EVA" strategy
- ♠ ≡ slevel-function value fixed at 400 only for parse_x509_Extensions (N.B. we tryed each function one by one to select the best value for the option - only this one seems to be necessary to get best results on our analysis)

Fig. 9. "EVA only" strategy experiments

```
frama-c x509_parser.c \
-machdep x86_64 \
        -eva-ilevel 64 \
        -eva-warn-undefined-pointer-comparison none\
        -eva-symbolic-locations-domain \
        -eva-split-return auto \
        -slevel-function="parse_x509_Extensions:400" \
        -slevel 12 \
        -eva
```

Fig. 10. EVA: the optimal invocation

invocation (in Fig. 10) and adding these additional options, new warnings are emitted by EVA as shown in Fig. 11.

Tool Version	Errors	Warnings	Time Stats
18.0 Argon	0	158	2392.3s (>39mn)

Fig. 11. EVA: results for the last run (with tweaking of options and domains and with additional checks)

As depicted above, the total number of warnings and the duration both significantly increased. The new warnings emitted by the tool are false positives and our previous strategy ("EVA then WP") effectively proved their absence.

Code Prover Polyspace Code Prover [22] is similar to EVA: a sound static analysis tool based on abstract interpretation to prove the absence of RTE. The analysis is fully automated and does not require code annotation. The workflow used by Code Prover is very close to the "EVA only" strategy.

A first run of Code Prover was performed, with no additional information on the code and with the default values for the analysis. The summary of this run is given in Fig. 12.

The results obtained with Code Prover in this first run are quite similar with those obtained with EVA: no error found, the same parts of code are indicated unreachable and remaining warnings correspond (before tweaking EVA options).

However, if some hypotheses are given to Code Prover, for example that the buffer is a well-defined and initialized array, then the analysis

Tool configuration:

Tool version	Precision level	Verification level
9.10	2	Safety analysis level 2

Results:

Total checks	Red checks	Grey checks	Orange checks	Green checks	Time Stats
3057	0	51 (1.5%)	153 (5.2%)	2853 (93.3%)	65 s

— Red check ≡ error
— Grey check ≡ unreachable code
— Orange check ≡ warning
— Green check ≡ proven

Fig. 12. Code Prover: results of the first run (without tweaking of options)

can be much more precise and raises only 52 warnings. We also increased the precision and verification levels in the tool. The summary of this new run is done in Fig 13.

Tool configuration:

Tool version	Precision level	Verification level
9.10	3	Safety analysis level 4

Results:

Total checks	Red checks	Grey checks	Orange checks	Green checks	Time Stats
3002	0	48 (1.6%)	52 (1.7%)	2902 (96.7%)	245s

Fig. 13. Code Prover: results of the second run (with tweaking of options)

In Fig. 14, the results of a last run is done with additional detection of unsigned integer overflows.

Tool configuration:

Tool version	Precision level	Verification level
9.10	3	Safety analysis level 4

Results:

Total checks	Red checks	Grey checks	Orange checks	Green checks	Time Stats
4295	0	49 (1.1%)	202 (4.7%)	4044 (94.2%)	248 s

Fig. 14. Code Prover: results of the last run (with tweaking of options) and with the detection of unsigned integer overflows

The report and interface of Code Prover are more user-friendly than in Frama-C but the tweaking is more limited. This is quite logical: on

the one hand, Code Prover is a commercial tool to automatically verify code without user interaction and on the second hand, Frama-C is an academic platform which offers to the user multiple ways of interactions and multiple kind of analyses. Code Prover is clearly really easy to use. The GUI is useful to explore the warnings emitted by the tool and one has the necessary information needed to understand them (execution path, computed values, ...). The tool provides a good way to visualize the analysis results and explore them, by presenting those results in a clear way and with contextual information to investigate them. This effort towards end-user deserves to be emphasized. Results obtained are quite interesting: in an easy and fast way (less than 4 mn), Code Prover verifies our parser code and only 52 potential RTE remain. These ones are actually false positives and the user has to investigate each warning and give a manual justification about the absence of RTE. Code Prover is clearly a good and efficient tool to find RTE and with a high level of guarantee.

7.2 Other tools

To go further, we also analyzed our parser with various tools, not dedicated to RTE-detection, but detection of vulnerabilities or defects in C code.

We started with three commercial tools: Bug Finder (Polyspace) [21], Codesonar (Grammatech) [8] and Coverity (Synopsys) [26] which are static analysis tools based on unsound approaches in order to scale analysis to larger code bases. These tools are used to detect software defects including, among others, the compliance to coding standards as MISRAC:2012 [23].

The facility to use Bug Finder and the GUI are the same as Code Prover. A detailed report is generated at the end of the analysis which is really useful to investigate the reported defects.

Codesonar is a whole program static analysis tool focused on safety and security critical software. It has a wide array of checkers, including checkers for 3^{rd} party API misuse as well as concurrency. It can quickly analyze and validate the source code as wall as binary code. It has a mathematical background based on dataflow analysis, symbolic execution and advanced theorem provers. We were surprised - in the good way - by some warnings of the tool alerting on unusual code constructions. The GUI is really helpful and guides the result analysis.

The use of Coverity is also easy and fast. The results found by this tool are mostly accurate and comprehensive and the GUI enables developers to quickly detect and fix defects in the code.

These three tools took a similar time to analyze our code (about 60 s). Results are given in Fig. 15. For all the tools, we have set the maximum level of detection (all the defects for Bug Finder, aggressive way for Codesonar and so on).

Type	Bug Finder (R2019a)	Codesonar (5.0)	Coverity (2019.03)
Hard-coded loop bound	5		
Redundant conditions		9	
Empty if statements		2	
Mixed enum type			2
Useless if (condition always verified)	4		
Identification of useless parts:			
Useless assignments		36	
Unused values		22	23
Write without further read	28		
Identification of unreachable parts:			
Dead code	2		4
Code deactivated by false condition	2		
Unreachable computation		3	
Total	41	72	29

Fig. 15. Results of Bug Finder, Codesonar and Coverity

Defects returned by these three tools are indeed present in our code but do not pose security risk. Of course, we have investigated these warnings and one of them, not related to a RTE, allowed us to find a real defect in our parser. Because of the limited number of defects emitted by Coverity, we started by investigating them in this tool and we easily found this logical bug as explained in Section 8. This defect, identified as "Unused value", shows the overloading of the return variable of `parse_DisplayText()` function resulting in the acceptation of invalid strings. The two other tools have also identified this defect during the analysis, pointed as “Useless assignments” or “Write without further read” depending on the associated tool.

To continue our code exploration, we have also tested some additional open-source tools.

Flawfinder [27] is an open-source tool for scanning source code (C or C++) to find potential flaws sorted by risk level. It works with a flaw pattern database of known problems (buffer overflow, string format, race conditions...). It can be compared to an extended "grep" which can also take into account user comments to ignore some false positives (`/*`

`Flawfinder: ignore */`). This tool is fairly simple to use: it works through a command line interface receiving the directory with the files to analyze as parameter. The complete list of rules used by the tool can be found directly on Flawfinder homepage [27]. For version 2.0.8, 223 rules were present in the ruleset with five different risk levels. For our parser, the analysis returned 1 hit in 0.12s: `"x509-parser.c:26: [4] (format) printf: If format strings can be influenced by an attacker, they can be exploited (CWE-134). Use a constant for the format specification."` due to the use of `printf` in a deactivated macro to trace errors in our parser.

The scan is easy and fast but without surprise, inconclusive. The approach of those purely lexical or pattern matching tools can detect the use of potentially insecure C functions, like `strcpy()`, `strcat()` etc. But, this method may produce a massive amount of false positives and false negatives. Indeed, this method ignores the data and control flow data and is purely syntactic. For example, using variable names similar to dangerous functions names increases significantly the number of false positives. In the same way, the tool does not discriminate a patched version of a dangerous function with a vulnerable one. As clearly explained by D. Wheeler on the Flawfinder HomePage, the use of Flawfinder can be useful to find easily security vulnerabilities but it is not sufficient and the use of others static analysis tools is strongly encouraged.

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). We can also observe the importance of not relying on an unique analysis or tool to gain an high degree of assurance.

8 The story of a logical bug

Inside qualifiers of Certificate Policies structures that can be found in a certificate, the `organization` field of the `NoticeReference` sequence and the `explicitText` field of the `UserNotice` sequence both use the `DisplayText` type defined in the following way:

```
DisplayText ::= CHOICE {
     ia5String        IA5String      (SIZE (1..200)),
     visibleString    VisibleString  (SIZE (1..200)),
     bmpString        BMPString      (SIZE (1..200)),
     utf8String       UTF8String     (SIZE (1..200))
}
```

A `DisplayText` is a string with four possible types. Each specific string type has a limited charset which the parser needs to verify.

In the parser, this task is done by a function called `parse_DisplayText()`. At one point during the implementation, the function `parse_DisplayText()` was implemented as depicted in Fig. 7.

The function is pretty simple in its design. It parses the beginning of the buffer to get the string length it contains and then calls a function dedicated to the parsing of the specific encountered string type.

Before going any further, we must stress that this annotated function was fully validated by Frama-C, guaranteeing the absence of RTE.

```
1  /*@
2    @ requires len >= 0;
3    @ requires ((len > 0) && (buf != \null)) ==> \valid_read(buf + (0
       .. (len - 1)));
4    @ requires \valid(eaten);
5    @ requires \separated(eaten, buf+(..));
6    @ ensures \result <= 0;
7    @ ensures (\result == 0) ==> (*eaten <= len);
8    @ ensures (len == 0) ==> \result < 0;
9    @ ensures (buf == \null) ==> \result < 0;
10   @ assigns *eaten;
11   @*/
12 static int parse_DisplayText(const u8 *buf, u16 len, u16 *eaten)
13 {
14     u16 hdr_len = 0, data_len = 0;
15     u8 str_type;
16     int ret;
17
18     if ((buf == NULL) || (len == 0)) {
19         ret = -__LINE__;
20         ERROR_TRACE_APPEND(__LINE__);
21         goto out;
22     }
23
24     str_type = buf[0];
25
26     switch (str_type) {
27     case STR_TYPE_UTF8_STRING:     /* UTF8String */
28     case STR_TYPE_IA5_STRING:      /* IA5String */
29     case STR_TYPE_VISIBLE_STRING: /* VisibileString */
30     case STR_TYPE_BMP_STRING:      /* BMPString */
31         ret = parse_id_len(buf, len, CLASS_UNIVERSAL, str_type,
32                    &hdr_len, &data_len);
33         if (ret) {
34             ERROR_TRACE_APPEND(__LINE__);
35             goto out;
36         }
37
38         buf += hdr_len;
```

```
39
40          switch (str_type) {
41          case STR_TYPE_UTF8_STRING:
42              ret = check_utf8_string(buf, data_len);
43              if (ret) {
44                  ERROR_TRACE_APPEND(__LINE__);
45              }
46              break;
47          case STR_TYPE_IA5_STRING:
48              ret = check_ia5_string(buf, data_len);
49              if (ret) {
50                  ERROR_TRACE_APPEND(__LINE__);
51              }
52              break;
53          case STR_TYPE_VISIBLE_STRING:
54              ret = check_visible_string(buf, data_len);
55              if (ret) {
56                  ERROR_TRACE_APPEND(__LINE__);
57              }
58              break;
59          case STR_TYPE_BMP_STRING:
60              ret = check_bmp_string(buf, data_len);
61              if (ret) {
62                  ERROR_TRACE_APPEND(__LINE__);
63              }
64              break;
65          default:
66              ret = -__LINE__;
67              ERROR_TRACE_APPEND(__LINE__);
68              break;
69          }
70
71          *eaten = hdr_len + data_len;
72
73          break;
74      default:
75          ret = -__LINE__;
76          ERROR_TRACE_APPEND(__LINE__);
77          goto out;
78          break;
79      }
80
81      ret = 0;
82
83  out:
84      return ret;
85  }
```

Listing 7. `parse_DisplayText()` function

Sadly, the attentive reader will notice that the switch/case is missing a bunch of `goto out;` at lines 43, 49, 55, 61 in the tests after each call to `check_*_string()` functions, and also at line 67. These missing clauses result in `ret` being overloaded at line 81. The net result is that the verdict

of string parsing is never used and invalid strings are just accepted. This is a good example of a logical bug.

A possible fix for the bug is presented below in a diff format.

```
diff --git a/src/x509_parser.c b/src/x509_parser.c
index 69450c0ee2f3..22c9f0848230 100644
--- a/src/x509_parser.c
+++ b/src/x509_parser.c
@@ -5549,7 +5549,7 @@ static int parse_DisplayText(const u8 *buf,
    u16 len, u16 *eaten)
 {
        u16 hdr_len = 0, data_len = 0;
        u8 str_type;
-       int ret;
+       int ret = -1;

        if ((buf == NULL) || (len == 0)) {
                ret = -__LINE__;
@@ -5578,29 +5578,34 @@ static int parse_DisplayText(const u8 *buf,
    u16 len, u16 *eaten)
                        ret = check_utf8_string(buf, data_len);
                        if (ret) {
                                ERROR_TRACE_APPEND(__LINE__);
+                               goto out;
                        }
                        break;
                case STR_TYPE_IA5_STRING:
                        ret = check_ia5_string(buf, data_len);
                        if (ret) {
                                ERROR_TRACE_APPEND(__LINE__);
+                               goto out;
                        }
                        break;
                case STR_TYPE_VISIBLE_STRING:
                        ret = check_visible_string(buf, data_len);
                        if (ret) {
                                ERROR_TRACE_APPEND(__LINE__);
+                               goto out;
                        }
                        break;
                case STR_TYPE_BMP_STRING:
                        ret = check_bmp_string(buf, data_len);
                        if (ret) {
                                ERROR_TRACE_APPEND(__LINE__);
+                               goto out;
                        }
                        break;
                default:
                        ret = -__LINE__;
                        ERROR_TRACE_APPEND(__LINE__);
+                       goto out;
                        break;
                }

```

```
@@ -5614,8 +5619,6 @@ static int parse_DisplayText(const u8 *buf,
    u16 len, u16 *eaten)
                break;
        }

-       ret = 0;
-
 out:
        return ret;
 }
```

Listing 8. fix for `parse_DisplayText()` function

9 Conclusion

In this work, 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 at a first glance complex to handle, 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 only few minor rewriting of the code are needed, which makes it 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 avoids spending many hours handling false positive results or non-convergence of the tool in a reasonable time, assuming an initial learning and coding effort.

Of course, the absence of RTE does not mean absence of bugs, but it is surely a big step forward when compared to the current situation of C-based parsers. The absence of RTE would nonetheless constitutes a very desirable 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 inherent type-safeness and RTE-freeness properties 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 valuable data 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. **Frama-C/Eva applied to the Chrony source code: a first analysis**. https://blog.frama-c.com/public/chrony/report-eva-chrony.pdf, 2018.
2. **Running the first EVA analysis** . http://blog.frama-c.com/index.php?tag/tutorial, 2018.
3. **Journey to a RTE-free X.509 parser (extended)**. https://www.sstic.org/2019/presentation/journey-to-a-rte-free-x509-parser/, 2019.
4. 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.
5. 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.
6. E.W. Dijkstra. **"Guarded Commands, Nondeterminacy and Formal Derivation of Programs"**. *ACM*, 1975.
7. 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.
8. Grammatech. **CodeSonar Static Analysis**. https://www.grammatech.com/products/codesonar.
9. D. Benjamin H. Sidhpurwala, H. Böck. **"OpenSSL „Negative Zero" issue"**. https://www.openssl.org/news/secadv/20160503.txt, 2016.
10. C.A.R. Hoare. **"An axiomatic basis for computer programming"**. *ACM*, 1969.
11. 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.
12. 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.
13. 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.
14. 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.

15. CEA LIST. **ACSL: ANSI/ISO C Specification Language Version 1.13**. https://frama-c.com/download/acsl.pdf.
16. CEA LIST. **"Eva - The Evolved Value Analysis plug-in"**. https://frama-c.com/download/frama-c-eva-manual.pdf.
17. CEA LIST. **"Frama-C User Manual"**. http://frama-c.com/download/frama-c-user-manual.pdf.
18. CEA LIST. **"Frama-C/WP"**. https://frama-c.com/download/frama-c-wp-manual.pdf.
19. CEA LIST. **"RTE - Runtime Error Annotation Generation"**. https://frama-c.com/download/frama-c-rte-manual.pdf.
20. 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.
21. MathWorks. **Polyspace Bug Finder**. https://fr.mathworks.com/products/polyspace-bug-finder.html.
22. MathWorks. **Polyspace Code Prover**. https://fr.mathworks.com/products/polyspace-code-prover.html.
23. MISRA. **MISRA C:2012**. https://www.misra.org.uk/MISRAHome/MISRAC2012/tabid/196/Default.aspx.
24. 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.
25. D. Pariente and J. Signoles. **Static Analysis and Runtime Assertion Checking: Contribution to Security Counter-Measures**. *SSTIC*, 2017.
26. Synopsys. **Coverity Static Analysis**. https://www.synopsys.com/content/dam/synopsys/sig-assets/datasheets/SAST-Coverity-datasheet.pdf.
27. D. Wheeler. **Flawfinder Homepage**. https://dwheeler.com/flawfinder/.