Systematic Parsing of X.509:
Eradicating Security Issues with a Parse Tree

Alessandro Barenghi [a], Nicholas Mainardi [a] and Gerardo Pelosi [a,*]

[a] *Department of Electronics, Information and Bioengineering – DEIB. Politecnico di Milano,Italy*
E-mails: alessandro.barenghi@polimi.it, nicholas.mainardi@polimi.it, gerardo.pelosi@polimi.it

Abstract. X.509 certificate parsing and validation is a critical task which has shown consistent lack of effectiveness, with practical attacks being reported with a steady rate during the last 10 years. In this work we analyze the X.509 standard and provide a grammar description of it amenable to the automated generation of a parser with strong termination guarantees, providing unambiguous input parsing. We report the results of analyzing a 11M X.509 certificate dump of the HTTPS servers running on the entire IPv4 space, showing that 21.5% of the certificates in use are syntactically invalid. We compare the results of our parsing against 7 widely used TLS libraries showing that 631k to $1,156$k syntactically incorrect certificates are deemed valid by them (5.7%–10.5%), including instances with security critical mis-parsings. We prove the criticality of such mis-parsing exploiting one of the syntactic flaws found in existing certificates to perform an impersonation attack.

Keywords: Digital Certificates, X.509, Transport Layer Security, TLS, Parsing, Security Vulnerabilities

1. Introduction

Digital certificates are the mainstay of public key authentication in secure network communications since the introduction of the Secure Sockets Layer (SSL) and Transport Security Layer (TLS) protocols. The requirement for interoperability called for the adoption of a standardized format specifying both the information which should be contained in them and their encoding.

Such a standardization action took place within the X.500 standard series by the International Telecommunication Union (ITU), and resulted in the X.509 standard for digital certificates. The standard has been further described in a series of Request for Comments (RFCs) memoranda [1–6] by the Internet Engineering Task Force (IETF) and has grown to a considerable complexity since its first version designed in 1988. As a consequence, performing sound validation of the contents of an X.509 certificate has become a non trivial and security critical task. Indeed, X.509 certificate validation has shown consistent lack of effectiveness in implementations due to the large number of constraints to be taken into account and the complexity of the involved data structure. Practical attacks against the X.509 certificate validation have been pointed out for the last 10 years, leading to effective impersonations against TLS/SSL enabled software.

Some among the most renown security issues involve certificates which are deemed valid to be binding a public key to the identity of a Certification Authority (CA), while such an information is contradicted either by the values contained in the certificate [7] or by misinterpretations in the subject name contained in it [8], both leading to effective impersonation of an arbitrary identity. More recently, in [9]

---

it was shown that inconsistent validations were performed by different TLS libraries, due to integer overflows in the recognition of some X.509 certificate fields, providing ground for attacks. Broken certificates are common even among the Alexa top 1M visited sites [10], and the diversity in the Application Program Interface (API) exposed by the existing TLS/SSL libraries was proven a further source of security issues [11]. The latest among the reported issues on X.509 validation shows that, due to a misinterpretation issue of the encoding, it was effectively possible to get certificates with forged signatures accepted [12, 13].

An interesting point to be noted is that all the aforementioned issues do not stem from a cryptographic vulnerability of the employed primitives, but rather from a non systematic approach to the syntactic recognition of the certificate. Indeed, mainly due to the high complexity of the data format, no methodical approach at content format recognition and syntactic verification, i.e., parsing, has been either proposed or employed in the use of existing X.509 digital certificates. All the existing available implementations dealing with X.509 certificates employ ad-hoc handcrafted code to parse the certificate contents, in turn resulting in software artifacts which are difficult to test for correctness. A practical validation of such issue is reported in [14] where the authors employed a tool to generate pseudo-random X.509 certificates obtained by splicing a set of valid ones and inserting intentional errors.

The results of testing common SSL/TLS implementations against such datasets, instead of purely random inputs alone, helped to uncover a significant number of incorrect recognition issues. However, such an approach does not provide a constructive guarantee that a parsing strategy for X.509 certificates is indeed sound in its action.

A classical and sound approach to the syntactic validation of a given input format is the one regarding it as the problem of parsing a language specified by a given grammar. Such an approach allows to provide a synthetic description of the format to be recognized from which an implementation of the actual parser can be automatically derived, minimizing the implementation effort and providing guarantees on the correctness of the recognition action. Despite the aforementioned advantages of tackling the issue by means of a language theoretic approach, such a strategy has never been employed to parse existing X.509 certificates. We ascribe this both to the complexity of the X.509 standard when considered as a language over an arbitrary byte alphabet and to the fact that it was pointed out in [9] that X.509 is at least *context sensitive*, a feature preventing the use of most parser generation techniques. An interesting proposal for a new regular format is presented in [15]. We report that a language theoretic analysis similar to the one we present for X.509 was performed on the OpenPGP message format in [16].

Contributions.
In this work, we tackled the task of analyzing the standard specification of X.509 certificates from a language theoretic standpoint, highlighting which ones of its features present a hindrance for a decidable and effective parsing. After performing a critical analysis of the X.509 standard, which is specified as a combination of Abstract Syntax Notation 1 (ASN.1, see ITU-R X.680) [17] data type descriptions and natural language defined constraints, we designed a predicated grammar defining a language which contains all the X.509 certificates employing standardized algorithms and smaller than 4 GiB.

We automatically derive a parser for the said X.509 language from the aforementioned grammar specification, employing the ANTLR parser generator [18]. We validate the parsing capability of our parser analyzing 11M X.509 certificates obtained as a survey of servers running the HyperText Transfer Protocol over Secure Socket Layer (HTTPS) on the entire space of the Internet Protocol version 4 (IPv4), showing that $\approx 21.5\%$ of the X.509 certificates are syntactically incorrect.

We compare our syntactic recognition capabilities against the certificate validation performed by 7 widely employed TLS libraries, and show how a significant number of X.509 certificates which are

syntactically invalid are not recognized as such by all of the current TLS libraries. We validate the criticality of the detected flaws exploiting one of them to lead a successful impersonation attack against the OpenSSL and BoringSSL libraries.

2. Preliminaries

In this section we report the notions needed to describe a digital certificate compliant with the X.509 standard in a form amenable to an automated parser generation. In particular, we recall both the key properties of formal languages and the efficiency of their recognizers. Subsequently, since the X.509 specification is given employing the Abstract Syntax Notation 1 (ASN.1) meta-syntax, we provide a bridge among its notation and the usual formal grammar one.

2.1. Parser Generation

Given a finite set of symbols $\mathbf{V}_t$, known as *alphabet*, a language $\mathbf{L}$ is defined as a set of elements, named *sentences*, obtained by concatenating zero or more symbol of $\mathbf{V}_t$. Conventionally, the *empty sentence* is denoted as ε, while a portion of a sentence is known as a *factor*.

Given a language it is also possible to describe it via a generative formalism, i.e., a *grammar*. A grammar is a quadruple $G : (\mathbf{V}_t, \mathbf{V}_n, \mathbf{P}, S)$, with $\mathbf{V}_t$ the alphabet of the generated language, $\mathbf{V}_n$ a finite set of symbols named *nonterminals*, $\mathbf{P}$ a set of *productions* defined as pairs of strings obtained by concatenating elements of $\mathbf{V}=\mathbf{V}_t\cup\mathbf{V}_n$, and $S\in\mathbf{V}_n$ the *axiom* of the grammar. The productions are denoted as usual as $\alpha\rightarrow\beta$; $\alpha\in\mathbf{V}_n^+$, $\beta\in\mathbf{V}^*$, where $\mathbf{V}^+$ represents the closure with respect to concatenation over $\mathbf{V}$, and $\mathbf{V}^*=\mathbf{V}^+\cup\{\varepsilon\}$. A sentence of a language $\mathbf{L}$ is generated by the grammar G if it is possible to obtain it starting from a string $u\in\mathbf{V}^*$ made of the grammar axiom alone (i.e., the nonterminal S), and iteratively substituting a portion of u appearing as the left hand side of a production in $\mathbf{P}$ with the right hand side of the said production, until the result is made only of elements of $\mathbf{V}_t$. The grammar is said to be *ambiguous*, if it is possible to generate a sentence of the language through two different sequences of substitutions. Languages are classified according to the Chomsky hierarchy [19], depending on which constraints are holding on the productions of the grammar generating them. Language families generated from grammars with stricter constraints are included in all the families with weaker constraints. In the following, we describe the language families starting from the least constrained one. Grammars with no restrictions on the left and right sides of its productions are denoted as *unrestricted* grammars, and generate *context sensitive with erasure* (CS-E) languages. Determining if a generic string over $\mathbf{V}_t$ belongs to a CS-E language is not decidable.

Restricting the productions of a grammar so that the sequence of substitutions does not allow the erasure of symbols yields *context sensitive* (CS) grammars, which generate CS languages. Determining if a generic string over $\mathbf{V}_t$ belongs to a CS language is decidable, i.e., it is always possible to state whether a string over $\mathbf{V}_t$ can be generated by a CS grammar. Restricting the productions of a grammar to have a single element of $\mathbf{V}_n$ on their left hand side yields the so-called *context-free* (CF) grammars, which generate CF languages. Deterministic CF (DCF) languages can be recognized by a Deterministic PushDown Automaton (DPDA) and their generating grammars constitute a proper subset of non ambiguous CF grammars. Finally, restricting the productions of a grammar either to the ones of the form $\{A\rightarrow a, A\rightarrow aA\}$ or to the ones of the form $\{A\rightarrow a, A\rightarrow Aa\}$, where $A\in\mathbf{V}_n$, $a\in\mathbf{V}_t\cup\{\varepsilon\}$, yields *right-* or *left-linear* grammars, which generate *regular* (REG) languages. We will denote such grammars as *linear* whenever the direction of the recursion in their productions is not fundamental.

Given a grammar G generating the language $\mathbf{L}$ and a generic terminal string $w \in \mathbf{V}_t^*$, the process of *parsing* consists both in determining if $w \in \mathbf{L}$ and in computing the sequence(s) of applications of the productions of G which generate(s) w. The sequence(s) of productions are depicted as a tree, called either *parse tree* or *abstract syntax tree*, with the axiom of the grammar being the root and the terminal symbols being the leaves. There exists a straightforward parsing algorithm of a CS language which is complete and correct, but its running time is exponential in the length of the input. As a consequence, a number of higher efficiency automated parser generation techniques were defined for subsets of the CS grammar family. CF grammars are the mainstay of parser generation as it is possible to automatically generate a recognizer automaton for any language generated by them. In particular, given a generic DCF grammar it is possible to automatically generate a deterministic parser for the strings of the corresponding language [19]; such parser enjoys worst-case linear space and time requirements in the length of input. Linear grammars are optimal from a parsing standpoint, as it is possible to derive from them a parser which runs with constant memory and linear time requirements in the length of input. Moreover, a parser for a REG language can be generated starting from the definition of a *regular expression* [20].

We aim at obtaining a grammar definition for the X.509 language such that the parsing is decidable. In addition, we require that such grammar is expressed as much as possible in terms of right-linear productions so that the parser will only require a constant amount of memory (besides the one to contain the input). In the remainder of this work, the terminal alphabet $\mathbf{V}_t$ will be the set of 256 values which can be taken by a byte, unless pointed out otherwise. Each terminal symbol will be denoted by two hexadecimal digits (e.g., the byte taking the decimal value 42 will be denoted as `2A`). We will also employ the Extended Backus-Naur Form (EBNF) to write the right hand sides of grammar productions.

2.2. ASN.1 Overview

The ASN.1 meta-syntax was introduced by the ITU to specify structured data as abstract data types (ADTs), and it is described in the set of ITU Recommendations (ITU-R) X.680-X.683 [17]. The encoding schemes for the ADTs are specified in ITU-R X.690–X.696 [17]. The ASN.1 provides the means to express both the syntax of the ADT at hand, in a form akin to a grammar, and some semantic constraints concerning the values taken by an instance of such an ADT. As our systematic parsing strategy will rely on a parser generated from a grammar specification for the X.509 ADT, we first provide a mapping between the syntactic-structure-specifying keywords of ASN.1 and the corresponding productions of an EBNF grammar. Subsequently, we highlight how the remaining, non purely syntactic features of ASN.1 act as constraints on the language of the instances of the described ADT in terms of semantics and describe the meta-language facilities exposed to ease the definition of a non ambiguous specification.

Syntactic Elements.
An ASN.1 ADT can be regarded as a construct equivalent to a single EBNF grammar generating all the possible concrete data type instances as its language. The user-defined name of the ADT corresponds to the axiom of the EBNF grammar, while the productions are represented as structured data definitions with the `::=` operator separating the left and right hand side, in lieu of the common $\rightarrow$.

The structure of an ADT may either be a single element, in case the type is *primitive*, or a composition of other types employing ASN.1 constructs in case it is *constructed*.

Primitive types in ASN.1 represent terminal rules of an EBNF grammar, i.e., rules where $\alpha \rightarrow \beta$, $\alpha \in \mathbf{V}_n^+, \beta \in \mathbf{V}_t^*$. The right hand side of a primitive type definition is described by a single line ended by a specific keyword (e.g., `INTEGER, BOOLEAN, OBJECT IDENTIFIER, OCTET STRING, BIT STRING`),

Table 1

Equivalence between the notation of the ASN.1 right hand side of a constructed type definition and the right hand side of an EBNF grammar production, with a set of sample ASN.1 types `u,v,x` matching the identically named strings $u, v, x \in (\mathbf{V}_t \cup \mathbf{V}_n)^*$ in EBNF

ASN.1 right hand sides	EBNF right hand sides
`SEQUENCE {u,v,x}`	uvx
`CHOICE {u,v,x}`	$u\lvert v\rvert x$
`u OPTIONAL`	$u\lvert\varepsilon$
`SET {u,v,x}`	$uvx\lvert uxv\lvert vux\lvert vxu\lvert xuv\lvert xvu$
`SEQUENCE OF u`	u^*
`SET OF u`	u^*
`SEQUENCE SIZE(1 .. MAX) OF u`	u^+
`SET SIZE(1 .. MAX) OF u`	u^+
`u(2 .. N)`	$u^2\lvert u^3\lvert \ldots \lvert u^N$
`ANY`	Arbitrary definition
`ANY DEFINED BY u`	Arbitrary ASN.1 definition

specifying completely its nature. An `OBJECT IDENTIFIER` is a sequence of decimal numbers separated by dots, of which a single decimal number is known as an *arc*.

Constructed types may either have a single user-defined name appearing on the right hand side of their definition, in which case they act as the copy rules of an EBNF grammar (i.e., rules where $\alpha \rightarrow \beta, \alpha, \beta \in \mathbf{V}_n$), or an arbitrary ASN.1 syntactic construct may be used. The only exception to the aforementioned constructed type definition is the possibility of turning a primitive `OCTET STRING` or `BIT STRING` type into a constructed one through appending the keyword `CONTAINING` to it, followed by the description of its contents. Deriving a constructed type from an `OCTET STRING` forces its value in an ADT instance to be byte-aligned, and allows the designer to enforce the choice of the encoding rules to be employed for it with the `ENCODED BY` keywords followed by the encoding name (see ITU-R X.682) [17]. Table 1 shows a mapping between the ASN.1 structures appearing on the right hand side of the data types which are considered in this work and their matching EBNF notation, expressed with a set of sample ASN.1 types `u,v,x` and identically named strings $u, v, x \in (\mathbf{V}_t \cup \mathbf{V}_n)^*$ in EBNF.

The ASN.1 notation specifies the common concatenation and alternative choice via the `SEQUENCE` and `CHOICE` keywords, respectively, while the syntactic constraint indicated by the `SET` keyword mandates that a set of ADTs may appear in any order, without repetitions. Postfixing an ADT appearing in a right hand side of a definition with the `OPTIONAL` keyword allows it to be either missing or present only once. Given an ASN.1 ADT `u`, the syntactic constraints imposed by the `SEQUENCE OF` and `SET OF` constructs, indicate a concatenation of zero or more instances of `u`, matching the star operator in EBNF. The ternary range operator in ASN.1, having the syntax `t(low .. high)`, where `t` is an ADT and `low,high` the range boundaries, is employed with two purposes: specifying the range of possible values of the instances of the primitive ADT to which they are appended, or indicating the concatenation of any number `low`$<n<$`high` of instances of the user-defined ADT preceding them. The bounds of a range operator are either constant values or the keywords `MIN` and `MAX`, which indicate that the minimum (resp., maximum) of the given range, is interpreted as the smallest (resp., greatest) value which can be taken by the ADT on their left. ASN.1 allows to specify *size constraints* through the use of the keyword `SIZE`,

followed by a range of allowed sizes. A common idiom in ASN.1 ADT declarations is to employ `MAX` as an upper bound for `SIZE`s to indicate that there is no upper bound on the size. As a consequence, the two common ASN.1 `SEQUENCE SIZE(1 .. MAX) OF` and `SET SIZE(1 .. MAX) OF` idioms in Table 1 are representing a concatenation of one or more instances of the involved ADT u, corresponding to the EBNF cross construct. Finally, the ASN.1 keyword `ANY` allows to delegate the definition of the structure of a given ADT to another document, potentially not expressed in ASN.1. Specifying further the effect of the `ANY` construct with the `DEFINED BY` keywords, followed by the name of an ADT, enforces the fact that the specification should be expressed in ASN.1.

Semantic Elements and Disambiguation Constructs.
ASN.1 allows to describe semantic information on instances of an ADT either specifying a constant value for a given *primitive type* element, appending such value between round brackets on the line where the type appears, or specifying a so-called `DEFAULT` value. Appending the `DEFAULT` keyword allows to indicate that, in case an element is missing in an instance of the ADT, the recognizer should assume its presence nonetheless, and assign the semantic value present in the ASN.1 specification to it. The expressive power of the ASN.1 allows the designer to specify an ADT corresponding to an *inherently ambiguous* language, i.e., a language for which no unambiguous grammar exists. An illustrative example of such a case is the following ADT t:

```
t ::= CHOICE {
 s1 SEQUENCE{u(i..i),v(i..i),x(j..j)},
 s2 SEQUENCE{u(j..j),v(i..i),x(i..i)}}
i ::= INTEGER 1 .. MAX
j ::= INTEGER 1 .. MAX
```

which has its instances belonging to the intrinsically ambiguous language $\mathbf{L}=\{u^i v^i x^j \ \lor \ u^j v^i x^i$ s.t. $u, v, x \in \mathbf{V}_t, \ i \geqslant 1, j \geqslant 1\}$. To provide a convenient way to cope with ambiguities, ASN.1 introduces the so-called user-defined *tag* elements. A *tag* is a syntactic element, denoted as a decimal number enclosed in square brackets, which is prefixed to an ADT appearing in the right hand side of a data description. Proper use of *tag*s minimally alters the language of accepted ADT instances, while effectively curbing ambiguities. The inherent ambiguity of the sentences of the language of the previously shown ADT t can be eliminated adding two *tag*s to its description:

```
t ::= CHOICE {
 s1 SEQUENCE{ [0] u(i..i),v(i..i),x(j..j)},
 s2 SEQUENCE{ [1] u(j..j),v(i..i),x(i..i)}}
i ::= INTEGER 1 .. MAX
j ::= INTEGER 1 .. MAX
```

It is worth noting that there is no algorithmic way to check whether a given set of *tag*s introduced in an ASN.1 specification is enough to suppress all parsing ambiguities, as the expressive power of ASN.1 is sufficient to define a generic CF grammar, for which determining if it is ambiguous is a well known undecidable problem [19].

2.3. Distinguished Encoding Rules

The encoding rules for an ASN.1 data type instance (see ITU-R X.690–X.696 [17]) define several formats portable across different architectures by mandating bit and byte value conventions and ordering

of the encoded contents. We tackle the Distinguished Encoding Rules (DER), as a *constructed typed* field in the X.509 standard certificate ADT requires its DER encoding via the ENCODED BY keyword. Consequentially, DER is the encoding rule employed in the overwhelming majority of X.509 certificate instances found. DER encoded material may be further mapped to the fully printable base64 encoding, resulting in the so-called Privacy-enhanced Electronic Mail (PEM) format [21]. The DER encoding strategy represents an ASN.1 ADT instance as a stream of bytes which is logically split up into three fields: *identifier octets*, *length octets*, *contents octets*.

The *identifier octets* field is employed to encode the ASN.1 tag value and whether the ADT instance at hand is a primitive or a constructed ASN.1 type. The tag value may be either the disambiguating user-defined one present in the ASN.1 ADT definition, or a so-called *universal tag* assigned by the DER standard to all ASN.1 primitive types and to the SEQUENCE, SEQUENCE OF, SET, SET OF constructs. If a user-defined tag is present, its encoding is stored in the *identifier octets* field, while the encoding of the tagged ADT is stored in the *contents octets* field. To provide a more succinct encoding, ASN.1 allows to specify that a given user-defined tag is IMPLICIT, i.e., that it should replace the tag of the tagged ADT in the encoding in the *identifier octets* field. On the other hand, the EXPLICIT keyword states that a user-defined tag should be encoded according to the default behavior.

The *contents octets* field contains the actual encoding of the ADT instance at hand, while the *length octet* one stores the size of the content field as a number of bytes. The number of bytes constituting the *length octets* field varies in the 1 to 126 range. The encoding conventions for the length value are stated in the ITU-R X.690 [17]. A short form and a long form for the encoding of a length value are possible. The short form mandates the encoding of the length field as a single octet in which the most significant bit is 0 and the remaining ones encode the size of the *contents octet* field from 0 to 127 bytes. The long form consists of one initial octet followed by one or more subsequent octets containing the actual value of the *length octet* field. In the initial octet, the most significant bit is 1, while the remaining ones encode the number of length field octets that follow as an integer varying from 1 to 126. Thus, the number of bytes for the *contents octet* field is at most $2^{126 \cdot 8}$. The *length octet* field value equal to 128 encoded as a single byte is forbidden in DER, while in other standard encoding rules it is reserved to indicate that an indefinite number of bytes will follow. It is worth noting that the requirement to validate correctly the information in the *length octets* field against the actual length of the *contents octets* one can be done via a regular language matcher, as the lengths have an upper bound. Nevertheless, the corresponding minimal finite state automaton (FSA) has a number of states which cannot be represented (see Section 4).

2.4. Description of X.509 Certificate Structure

The structure of an X.509 certificate is described as an ASN.1 ADT in the ITU-R X.509 [22], and in the RFC 4210 [23] and its complements [1–6]. Its structure evolved over time, as its first version dates back to 1988. In this section, we provide a synthetic description of its contents.

Certificate Abstract Data Type.
Figure 1 reports a shortened version of the X.509 standard, defining the main Certificate ADT. In the figure, field names start with a lowercase letter, while ASN.1 user-defined ADT names start with capital letters. The Certificate ADT is a concatenation of three fields: the material to be signed typed as TBSCertificate, the identification data for the signature algorithm typed as AlgorithmIdentifier, and a BIT STRING field containing the actual signature value.

```
Certificate ::= SEQUENCE {
tbsCert            TBSCertificate,
signatureAlgorithm AlgorithmIdentifier,
signatureValue     BIT STRING }

TBSCertificate ::= SEQUENCE {
version [0] EXPLICIT Version DEFAULT v1,
serialNumber       CertSerialNumber,
signature          AlgorithmIdentifier,
issuer             Name,
validity           Validity,
subject            Name,
subjectPublicKeyInfo SubjectPubKeyInfo,
issuerUniqueID  [1] IMPLICIT UniqueId  OPTIONAL,
subjectUniqueID [2] IMPLICIT UniqueId  OPTIONAL,
extensions      [3] EXPLICIT Extensions OPTIONAL }

AlgorithmIdentifier ::= SEQUENCE {
algorithm  OBJECT IDENTIFIER,
parameters ANY DEFINED BY AlgorithmP OPTIONAL}

SubjectPublicKeyInfo  ::=  SEQUENCE  {
algorithm         AlgorithmIdentifier,
subjectPublicKey  BIT STRING }
```

Figure 1. Portion of the description of the X.509 `Certificate` ADT and its fields

TBSCertificate Abstract Data Type.
Considering the contents of the `TBSCertificate` ADT, the first two fields contain a version number typed as `Version` (a value constrained `INTEGER`), and an integer value which must be unique among all the certificates signed by the same certification authority (CA), which is typed as `CertSerialNumber`. The third field, typed as an `AlgorithmIdentifier`, contains the information to uniquely identify the cryptographic primitive employed to sign the certificate.

The `AlgorithmIdentifier` ADT is a concatenation of two fields: an ASN.1 `OBJECT IDENTIFIER` (OID) typed field `algorithm` and an optional `parameters` field, typed as `ANY DEFINED BY AlgorithmP`. The OID value allows to uniquely label the signature algorithm to allow the description of the structure of its parameters, done in [1–3, 5, 6] for a set of standardized signature algorithms. The `issuer`, `validity` and `subject` fields contain information on the CA issuing the certificate and the subject to whom it has been issued, together with the validity time interval of the certificate. `issuer` and `subject` fields are typed as a `Name` ADT: a `SEQUENCE OF` of `SET OF` structures containing a concatenation of two fields typed as OID and `ANY`, respectively. Despite the quite baroque definition, the `Name` ADT is indeed employed to represent a list of names for both the issuer and the subject which are typically expressed as printable strings prefixed with a standardized OID value stating their meaning (e.g., organization, state).

The `subjectPublicKeyInfo` field provides both the public key binded to the subject identity, and information on the employed cryptographic primitive in the form of a `BIT STRING` and an `AlgorithmIdentifier` typed field, respectively. Following the `subjectPublicKeyInfo` field, the `TBSCertificate` ADT includes two deprecated extra optional fields, containing further information about the issuer and the subject. These fields are tagged with tags [1] and [2] respectively, preventing a possible parsing ambiguity arising from only one of them being present.

```
Extensions ::= SEQUENCE SIZE(1..MAX) OF Extension

Extension ::= SEQUENCE {
extnID    OBJECT IDENTIFIER,
critical  BOOLEAN DEFAULT FALSE,
extnValue OCTET STRING CONTAINING ...  ENCODED BY der }
```

Figure 2. X.509 ASN.1 description reporting the definition of the last field of the `TBSCertificate` ADT

Extension Abstract Data Type.
The `extensions` field concludes the definition of the `TBSCertificate` ADT. Most of the information of modern certificates is contained in it, and its presence is mandatory in the current version (v3) of X.509 certificates. As reported in Figure 2 the `Extensions` ADT is a sequence of one or more `Extension` typed fields. Each `Extension` ADT is composed of an OID typed field identifying it unambiguously, and a `critical` field typed as `BOOLEAN` indicating, if `True`, that the certificate validation should fail in case the application either does not recognize or cannot process the information contained in the subsequent `extnValue` field. An example of information contained in the `extnValue` field is the so-called `KeyUsage`, i.e., information stating which is the legitimate purpose of the subject public key in the certificate at hand (e.g., signature validation or encryption).

3. Analysis of the X.509 Certificate Language

In this section, we analyze the X.509 certificate structure from a language theoretic standpoint. In particular, we highlight the portions of the certificate which hinder and harden the design of a grammar amenable to automatic parsing generation algorithms. These issues will be tackled in the next section, in order to achieve our goal of obtaining an effective and decidable parser for X.509 digital certificates.

Undecidability of The Parsing.
Some portions of an X.509 certificate requires an unrestricted grammar in order to be generated, in turn implying the parsing undecidability. First, consider the `signature` field in the `TBSCertificate` ADT, which is typed as an `AlgorithmIdentifier` ADT. This is composed by an OID, identifying the signature algorithm, and by optional `parameters`, whose structure depends on the OID. The complete freedom in the specification of the `parameters` field via the `ANY` keyword of ASN.1 meta-syntax allows the description to be arbitrarily complex, in particular allowing natural language constraints to be specified on eventual fields of `parameters`. This, in turn, requires an *unrestricted* grammar to generate the language and results in the undecidability of the certificate parsing. A similar issue arises in `extnValue` field of the `Extension` ADT. This field is typed as an `OCTET STRING` and is turned into a constructed type by the keyword `CONTAINING` (see Section 2.2). The specific structure of the `extnValue` field depends on the value of the `extnID` field, which is an OID identifying the extension type. The standard allows the definition of custom extensions, which may specify an arbitrary content for it [22]. Such a lack of constraints allows the checks on this content to be arbitrary complex, i.e., representable by an unrestricted grammar.

Context-Sensitiveness.
Some portions of X.509 certificate structure cannot be generated by a context-free grammar, in turn implying that for these portions an efficient automatically generated parser cannot be derived. The context-sensitiveness is introduced by the same kind of constraint in 2 different portions of the certificates. This constraint is the need to check repetitions of arbitrarily long strings. First, consider the

`signatureAlgorithm` field in `Certificate` ADT and the `signature` field in `TBSCertificate` ADT, both typed as `AlgorithmIdentifier`. While the latter is found in the portion of the certificate which is signed, the former is not. Therefore, it is expected that these 2 fields have the same content, in turn requiring a Context-Sensitive (CS) recognizer to check this equality. Similarly, some portions of the certificate must be present if the certificate is self-issued, which means that the issuer and the subject are the same entity. Recall that both `issuer` and `subject` are typed as `Name` ADT, which is an arbitrarily long sequence of names, which are generally printable strings, referring to the issuer or subject entity, coupled with an OID identifying their meaning. Verifying if a certificate is self-issued or not requires to match the content of `issuer` and `subject` fields, which are arbitrarily long string of bytes.

Parsing Ambiguities and Inconsistencies.
Due to looseness in standard specification of some fields, there are also parsing ambiguities in the format. First, consider the `signatureValue` field in the `Certificate` ADT. Despite the X.509 standard is typing this field as a primitive `BIT STRING`, some standardized signature algorithms require it to be a constructed field (e.g., the DSA and ECDSA cryptographic primitives [5, 6]), in turn giving way to an ambiguity in its interpretation. The same kind of issue arises in `SubjectPublicKeyInfo` field in `TBSCertificate` ADT. Indeed, the public key field is typed as a `BIT STRING`, instead of a constructed ADT. These ambiguities may lead to security issues when parameters for a given cryptographic primitive are either misinterpreted as valid for another one or simply parsed incorrectly [12, 24]. A further issue is introduced by the `extnValue` field in `Extension` ADT, which is typed as `OCTET STRING` but contains the extension data, which is usually a constructed ADT. Nevertheless, DER forbids the encoding of `OCTET STRINGS` as constructed types (see ITU-R X.690) [17], in turn forcing an inconsistency in the way `OCTET STRINGS` containing the value of an `extnValue` field should be treated during parsing (due to the `CONTAINING` keyword in the X.509 specification). We note that a less problematic definition of the field would have involved a dedicated constructed ADT for the `extnValue` field, which should have had its structure specified according to the value of the `extnID` field.

Unmanageable Number of Rules.
Recall that the `Extensions` ADT is a sequence of `Extension` ADTs. Such a sequence of ADTs has no constraint on their order, as they all share the same ADT. However, a constraint expressed in natural language in the X.509 standard mandates that 2 `Extension` typed field instances with the same `extnID` field cannot appear, in turn providing a concrete hindrance to its representation in grammar form. Indeed, an exponential number of productions would be required to generate unique `Extension` instances in any possible order. Considering that the number of X.509 standardized extensions is 17 [22], the required grammar would have at least 2^{17} productions, which is hardly manageable by a designer.

3.1. Summing Up Hindrances to X.509 Parsing

In the following statements we provide a synthesis of the undecidability, context sensitiveness and ambiguity problems arising from the X.509 certificate standard and recall the issue about the size of the grammar representation reported in Section 2.3.

Statement 1. (UNDECIDABILITY CONTEXT SENSITIVENESS AND AMBIGUITY) *The following issues should be tackled lest no unambiguous, correct parsing of X.*509 *is possible:*

i) Cope with the undecidability arising from the potential definition of arbitrary structures for both the `AlgorithmP` *ADT in the right-hand side of the* `AlgorithmIdentifier` *ADT and the value of*

the `extnValue` *field in the right-hand side of* `Extension` *ADT, without requiring an* unrestricted *grammar to specify them.*

ii) *Cope with the context sensitiveness of the X.*509 *grammar, arising from the equality checks required on the* `signature` *field of the* `TBSCertificate` *ADT against the* `signatureAlgorithm` *field of* `Certificate` *ADT, and the need to match arbitrarily long* `subject` *and* `issuer` *fields of the* `TBSCertificate` *ADT.*

iii) *Cope with the ambiguity introduced by the definition of the* `extnValue` *field as a constructed* `OCTET STRING`, *while its encoding is forced to mark it as a primitive one. Similarly cope with the issue of the* `signatureValue` *field and the* `subjectPublicKey` *field where information represented as a constructed data type is encoded as a primitive* `BIT STRING`.

In addition to the aforementioned issues, some of the portions of the X.509 language which can be defined with a linear grammar are still unpractical to be fully specified.

Statement 2. (UNMANAGEABLE GRAMMAR REPRESENTATION) *The presence of uniqueness constraints among the instances of the* `Extension` *ADT standardized in [4] can only be expressed with a linear grammar having* $>2^{17}$ *productions.*

We consider the aforementioned issues to be among the main causes of the current approach to X.509 parsing, which sees fully handcrafted implementations of the parser as an alternative to the development of an automatically generated one. However, given the complexity of the X.509 certificate standard specification, and the impossibility of methodically checking the correctness of a human-implemented recognizer (due to context-sensitive portions of the X.509 language), the current state-of-the-art parser implementations are exhibiting a non-trivial amount of security critical errors in the recognition of X.509 sentences [7–9].

4. Systematic X.509 Parsing

In this section we describe our approach to realize an automatically generated X.509 parser, starting from a grammar description. The obtained parser enjoys termination guarantees, correctness and a worst-case computational complexity which is quadratic in the length of the certificate.

We strive to minimize the non-regular portion of X.509 language, allowing an efficient parser to be systematically generated. However, since it is not possible to turn X.509 into a regular language without either significant precision loss in recognition, or substantial functionality loss in the parser, we retain a small CS portion of its description. Such a CS portion, indeed made of two binary string equality checks and a single boolean condition, is implemented by hand, as relying on the generic CS parsing algorithm would entail an exponential running time even for such rather trivial operations.

4.1. Coping with Undecidability, Context Sensitiveness and Ambiguity

The first step to obtain a useful grammar description of the X.509 standard is to cope with its parsing undecidability problem. We restricted the `AlgorithmIdentifier` ADT descriptions to the ones which are explicitly and fully described in the standards [1–3, 5, 6]. As a consequence we remove the possibility of having an unrestricted syntactic structure of their fields allowed by the ASN.1 keyword

`ANY` in the standard. This decision implies that we will recognize certificates employing only algorithms for which a standardized `AlgorithmIdentifier` description exists, which is typically a sign of wide acceptance of the cryptographic security margin offered by them. Examining all the standardized `AlgorithmIdentifier` descriptions we note that their structures are described by regular languages. This restriction may prevent correct recognition of valid X.509 certificates found in the wild. During our practical evaluation campaign, we observed just a negligible percentage of certificates being affected by this restriction, as detailed in Section 5. We tackled the similar problem caused by the `ANY`-typed `extnValue` field by parsing the syntactic structure of all the standard defined extensions [4] strictly.

However, in contrast with the decision taken for the non standard `AlgorithmIdentifier` structures, we accept user-defined extensions, considering them opaque byte sequences, and performing only a syntactic check on the correctness of the OID field and the length of the unspecified structure field. Indeed, in case of a custom extension for which the Boolean `critical` field is set, a validation failure must be signaled by the application logic, in case it is not able to handle the contents of the `extnValue` field [22]. Taking into account the described restrictions, the resulting X.509 specification is a CS language $\mathbf{L}_{\text{X.509}-\text{r1}}$ where the string membership problem is decidable.

The second step in producing a grammar description of X.509 is to tackle its context-sensitiveness. To this end, we consider the language $\mathbf{L}_{\text{X.509}-\text{r1}}$ as the intersection of three languages $\mathbf{L}_{\text{X.509}-\text{r1}} = \mathbf{L}_{\text{AId}-\text{match}} \cap \mathbf{L}_{\text{SI}-\text{match}} \cap \mathbf{L}_{\text{CF}}$, where $\mathbf{L}_{\text{AId}-\text{match}}$ is the language containing sentences with two matching `AlgorithmIdentifier` type instances, $\mathbf{L}_{\text{SI}-\text{match}}$ is the one where whenever the certificate `subject` and `issuer` match, the `keyIdentifier` field of the `AuthorityKeyIdentifier` extension ADT should be present, and $\mathbf{L}_{\text{CF}}$ is the CF language where all the remaining X.509 linguistic constraints are holding on the sentences. Thus, parsing $\mathbf{L}_{\text{X.509}-\text{r1}}$ can be done analyzing the candidate string (i.e., a certificate instance) with three parsers, each one recognizing one of the three aforementioned languages. We choose to start with the parser $\mathcal{A}_{\text{CF}}$ for $\mathbf{L}_{\text{CF}}$, as it can be automatically generated from its grammar description. The parse tree produced by $\mathcal{A}_{\text{CF}}$ is then employed by a handwritten string matcher, which checks the conditions required for a string to belong both to $\mathbf{L}_{\text{AId}-\text{match}}$ and to $\mathbf{L}_{\text{SI}-\text{match}}$. The code auditing of such a simple handwritten string matcher can be confidently performed by direct inspection to check its correctness. Indeed, these pieces of code are small and definitely simpler than a full hand-written parser, dramatically decreasing the complexity of the code auditing process. Therefore, spotting any vulnerabilities in these hand-written parsers is expected to be easier. Finally, we remark that this code has no influence on the automatic generated parser for $\mathcal{A}_{\text{CF}}$, since these matchings are performed when the parse tree for $\mathcal{A}_{\text{CF}}$ has already been built. Consequently, it is not possible that a subtle vulnerability in this code affects the correctness of the $\mathcal{A}_{\text{CF}}$ parser.

The last step to achieve unambiguous, decidable recognition of the X.509 certificates is to cope with the ambiguity issues stemming from the primitive encoding forced by DER on constructed `OCTET STRING`s in the `extnValue` field and the `BIT STRING`s containing signature and public key material specified by the standard to be primitive. Concerning the issue of the `extnValue` field, we choose to eliminate the ambiguity relying on the fact that the standards define the contents of the constructed type binding it to a specific OID: this in turn allows to parse unambiguously an ADT instance. Concerning the `BIT STRING` fields, we eliminate the parsing ambiguity recognizing them as constructed or primitive types according to the individual cryptographic primitive needs as specified in the standards [1–3, 5, 6]. Such an approach will indeed pick only a single way of interpreting the data contained in the field, preventing security critical parsing issues such as the ones in [24].

Once such ambiguities are removed from $\mathbf{L}_{\mathrm{CF}}$, the resulting language is indeed regular and will be denoted as $\mathbf{L}_{\mathrm{REG}}$ from now on, while the resulting restricted X.509 language will be denoted as $\mathbf{L}_{\mathrm{X.509-r2}}=\mathbf{L}_{\mathrm{AId-match}} \cap \mathbf{L}_{\mathrm{SI-match}} \cap \mathbf{L}_{\mathrm{REG}}$.

Lemma 1. *The proposed parser for* $\mathbf{L}_{\mathrm{X.509-r2}}$ *terminates on any input string.*

Proof. Since $\mathbf{L}_{\mathrm{X.509-r2}}=\mathbf{L}_{\mathrm{AId-match}} \cap \mathbf{L}_{\mathrm{SI-match}} \cap \mathbf{L}_{\mathrm{REG}}$ is the result of the intersection of two CS languages and a REG language, the closure property of the CS family w.r.t. set intersection implies that $\mathbf{L}_{\mathrm{X.509-r2}}$ is CS, a necessary condition for the termination of its parsing process. To prove the termination guarantee of the proposed parser, it is sufficient to observe that all the three parsers, sequentially executed to recognize $\mathbf{L}_{\mathrm{X.509-r2}}$, have guarantees on their termination, namely: the parser for $\mathbf{L}_{\mathrm{REG}}$ is a FSA, while the string matchers act on arbitrarily large, but not infinite strings.   □

Lemma 2. $\mathbf{L}_{\mathrm{X.509-r2}}$ *is non intrinsically ambiguous, and all its sentences can be parsed unambiguously.*

Proof. The only source of ambiguity in the original X.509 specification is the one caused by interpretation issues of the X.509 standard, which are no longer present in $\mathbf{L}_{\mathrm{X.509-r2}}$. The first stage of the proposed parser matches the sentences of $\mathbf{L}_{\mathrm{X.509-r2}}=((\mathbf{L}_{\mathrm{REG}} \cap \mathbf{L}_{\mathrm{AId-match}}) \cap \mathbf{L}_{\mathrm{SI-match}})$ by means of a deterministic finite state recognizer, which can only have a single recognition path for each one of them. The CS recognizer stages simply discard some of the strings deemed valid by the first stage, and thus do not introduce ambiguity.   □

4.2. Managing Grammar Representation and Recognizer Generation

Having dealt with the linguistic issues which could either have prevented the parsing (i.e., the undecidability) or made its result unusable (i.e., the ambiguity), we now tackle the issues concerning the practical construction of a grammar representation for X.509 and the generation of its recognizer. Among the recognizers required to match $\mathbf{L}_{\mathrm{X.509-r2}}=((\mathbf{L}_{\mathrm{REG}} \cap \mathbf{L}_{\mathrm{AId-match}}) \cap \mathbf{L}_{\mathrm{SI-match}})$ the last two suffer from no issues either in their syntax definition or in their recognizer generation.

In order to tackle the issues coming from Statement 2 in Section 3.1, we consider $\mathbf{L}_{\mathrm{REG}}$ as the intersection of three regular languages: $\mathbf{L}_{\mathrm{REG}} = \mathbf{L}_{\mathrm{REG-1}} \cap \mathbf{L}_{\mathrm{REG-2}} \cap \mathbf{L}_{\mathrm{REG-3}}$. $\mathbf{L}_{\mathrm{REG-1}}$ is the language of byte strings containing sequences of `Extensions` instances where each `Extension` instance appears at most once, $\mathbf{L}_{\mathrm{REG-2}}$ is the language of DER encoded ASN.1 fields with the correct constraints holding between the *length octet* field and the *contents octets* field, and $\mathbf{L}_{\mathrm{REG-3}}$ contains all the remaining regular constraints to obtain a sentence in $\mathbf{L}_{\mathrm{REG}}$ when considered in intersection with $\mathbf{L}_{\mathrm{REG-1}}$ and $\mathbf{L}_{\mathrm{REG-2}}$.

A straightforward definition of $\mathbf{L}_{\mathrm{REG-1}}$ with a linear grammar requires 17! rules, a number which can be reduced to a little over 2^{17} through careful rewriting. We deem this number of productions too high to be described and checked reliably, thus we match a relaxation of $\mathbf{L}_{\mathrm{REG-1}}$ at first, which allows the `Extensions` to be duplicated, and perform a subsequent uniqueness check on the result of the parsing action. This choice allows us to retain the description of the structure of the individual `Extension` instance as a purely syntactic one, and perform the check for duplicates on their parsed OID fields. We implemented the duplicate checking logic as an handwritten portion of our parser, as we deem the code to be small and simple enough to allow effective code auditing of it.

Providing an effective and efficient grammar definition for $\mathbf{L}_{\mathrm{REG-2}}$ is similarly difficult. Indeed, it is enough to consider a simpler but structurally equivalent language $\mathbf{L}_{\mathrm{REG-2-P}}$, where only the length constraints on DER encoded ASN.1 primitive types are enforced, to obtain an unmanageable number
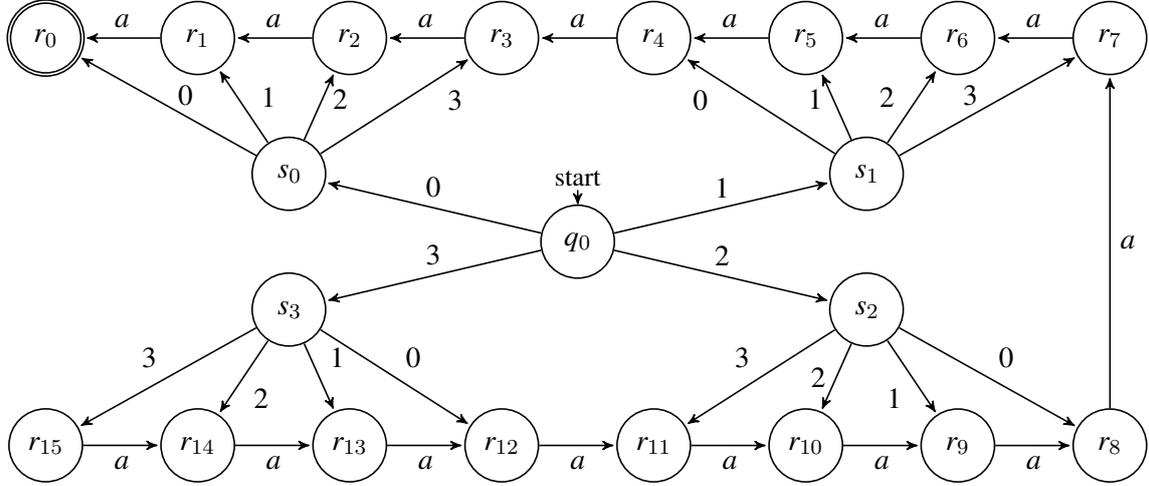
Figure 3. Strawman example of finite state recognizer accepting strings of *a* characters appended to their lengths expressed as two radix-4 digits. The initial state q_0 accepts the first digit, and memorizes it by means of the finite state memory, and moves to one of the four s_i states, which in turn read the second digit of the length and transition to an r_i state. r_i states are linked by a countdown chain of transitions, where i indicates the amount of remaining *a* characters to be recognized before acceptance

of productions. $\mathbf{L}_{\mathrm{REG-2-P}}$ is composed of strings of the form *lv*, where *l* is a byte string containing a unique encoding of a length in $[0, 2^{126\cdot 8}]$ and *v* a string of arbitrary bytes with the given length. A straightforward grammar representation of it requires at least a grammar production for each valid value of the byte string *l*. Such a number of productions is well beyond the realm of feasibility in realization. The first observation concerning this issue is that the possible values taken by the *l* string in practice will be significantly less than the possible ones, as the sensible lengths for an X.509 certificate are typically far smaller than $2^{126\cdot 8}$B. As a consequence, we deemed reasonable to restrict the set of valid represented lengths (both in $\mathbf{L}_{\mathrm{REG-2-P}}$ and in $\mathbf{L}_{\mathrm{REG-2}}$) to the $[0, 2^{32} - 1]$ interval, which entails accepting certificates of size up to 4 Gib. Note that, restricting further the encoded length values to a number which allows handwriting of the corresponding linear grammar by a human designer (i.e., in the hundreds range at most) would critically reduce the usefulness of the parser, as it would be able to match only contents smaller than a couple of hundreds bytes. We chose the aforementioned conservative upper bound of 2^{32} to prevent a parser that preallocates memory for the contents of the soon-to-be-recognized string in advance from being subject to easy Denial-of-Service attacks, via incorrect certificates overstating their lengths. Since the bound imposed by the practical usefulness on the number of rules of the grammar is still insufficient to allow a manageable specification of the grammar itself, we resort to analyze the parser automaton of $\mathbf{L}_{\mathrm{REG-2-P}}$ to derive a more manageable implementation of the recognizer, and adapt it to recognize the entire $\mathbf{L}_{\mathrm{REG-2}}$.

To illustrate how the problem of implementing the recognizer for $\mathbf{L}_{\mathrm{REG-2-P}}$ is solved in practice, we first provide a toy example, recognizing the language of words *lv*, where the content length *l* is expressed by two, radix-4 digits, and is followed by the content *v* which is a sequence of *a*.

The toy Finite State Automaton (FSA) recognizer operating on the set of bytes $\Sigma = \{0, 1, 2, 3, a\}$ is defined as $\mathcal{A}_{\mathrm{REG-2-P}} : (\Sigma, \mathbf{Q}, \delta, q_0, \mathbf{F})$, where $\mathbf{Q}$ is the set of states of the automaton, q_0 the initial state, $\mathbf{F}$ the set of final states, and $\delta : \mathbf{Q} \times \Sigma \rightarrow \mathbf{Q}$ the transition function. A depiction of the recognizer FSA is provided in Figure 3. The key idea underlying its functioning is count the *a* symbols in *v* employing a

chain of states, of which only the last is an accepting one (depicted in figure as $r_i, i \in [0, 15]$). The recognition of the length l is performed while decoding which one is the state of the chain of counting states where the computation should start recognizing the content v. The decoding is performed memorizing the number read digit by digit, yielding a single sequence of transitions starting from the initial state q_0 and ending in a counting state for each one of the latter.

The main issue in implementing an automaton with the structure depicted in Figure 3 is to provide a way to implement the transition function δ different from the common tabulated form, which would require an amount of space proportional to the maximum length of the represented string multiplied by the alphabet size; i.e., a number of entries $\geqslant 2^{32}{\cdot}256$ for the recognizer of $\mathcal{A}_{\mathrm{REG}-2-\mathrm{P}}$.

Our approach to cope with this issue is to employ a computational form of the said δ, where the states are encoded as unsigned integers, and the destination state can be computed in closed form. In the toy example, the δ function, which has its description as $\delta(q_0, t) = s_t, t \in \{0, 1, 2, 3\}$; $\delta(s_i, t) = r_{4i+t}$, $t \in \{0, 1, 2, 3\}$; $\delta(r_i, t) = r_{i-1}, t = a, i \in [1, 15]$, can be transformed in computational form encoding the r_i states with the unsigned integer i, the s_i states with $2^4 + i$ and q_0 as 2^5, and computing it as

$$
\begin{aligned}
\delta(q_0, t) = s_t, t \in \{0, 1, 2, 3\} \quad &\rightarrow \quad \delta(q, t) = 2^4 + t, \ t \in \{0, 1, 2, 3\}, q = 2^5 \\
\delta(s_i, t) = r_{4i+t}, t \in \{0, 1, 2, 3\} \quad &\rightarrow \quad \delta(q, t) = (q - 2^4) \cdot 4 + t, \ t \in \{0, 1, 2, 3\}, q = 2^4 + i, i \in [0, 3] \\
\delta(r_i, t) = r_{i-1}, i \in [1, 15] \quad &\rightarrow \quad \delta(i, t) = i - 1, \ t = a, i \in [1, 15]
\end{aligned}
$$

Following the same line of reasoning employed to obtain a computational form for the delta of our toy example automaton, we devised the representation of the transition function for the recognizer $\mathcal{A}_{\mathrm{REG}-2-\mathrm{P}}$. In particular, such a length function deals with the possible variable length encoding of the length field dedicating a specific transition path for each one of the lengths:

$$
\delta(q_0, a) = \begin{cases}
a \text{ with } 0 \leqslant a \leqslant 127 \\
2^{32} \text{ with } a = 129 \\
2^{33} \text{ with } a = 130 \\
2^{34} \text{ with } a = 131 \\
2^{35} \text{ with } a = 132
\end{cases}
$$

This split in the parsing paths also allows us to encode in the state the actual value of the length counter, accumulating it in case it is longer than a single byte. All the length-field recognition paths terminate with a transition moving to the state encoding the corresponding number of *content octets* field bytes to be read, upon reading the last byte of the length field, following the same structure of the toy automaton. Such a transition function is shown in Figure 4.

Finally, the remaining portion of the transition function takes care of counting the *content octet* field bytes while they are being read simply decrementing the counter encoded in the state as $\forall \, a {\in} \mathbf{V}_t, \ \delta(q, a) {=} q - 1$ with $1 \leqslant q \leqslant 2^{32} - 1$ until the only final state of the automaton $\mathbf{F} {=} \{0\}$ is reached. All the unspecified transitions are leading to the error state. This δ can be conveniently implemented in any programming language employing a single 64-bit unsigned integer as storage for the current state.

In order to recognize $\mathbf{L}_{\mathrm{REG}-2}$ to its full extent, we build on $\mathcal{A}_{\mathrm{REG}-2-\mathrm{P}}$ a machine able to take into account ASN.1 constructed types. Since the content of a constructed type is a list of other ASN.1 types, a nesting structure is introduced, requiring different counters depending on the nesting level. In our case, it is still possible to do so, while still employing a FSA recognizer as the maximum nesting depth is bounded, as a consequence of the capping imposed on the value of the outermost counter. As a consequence, we represent the state of $\mathcal{A}_{\mathrm{REG}-2}$ as a vector of states of $\mathcal{A}_{\mathrm{REG}-2-\mathrm{P}}$, employing one for each

$$
\substack{\forall\, a \in \mathbf{V}_t, \\ 0 \leqslant i \leqslant 255} \qquad
\delta(q,a) = \begin{cases}
a & \text{with } q{=}2^{32},\ 128{\leqslant}a{\leqslant}255 \\
2^{33}{+}a{+}1 & \text{with } q{=}2^{33} \\
a{+}(q{-}(2^{33}{+}1)){\cdot}2^8 & \text{with } q{=}\delta(2^{33},i) \\
2^{34}{+}1{+}a & \text{with } q{=}2^{34} \\
2^{34}{+}1{+}a{+}(q{-}2^{34}{-}1){\cdot}2^8 & \text{with } q{=}\delta(2^{34},i) \\
a{+}(q{-}2^{34}{-}1){\cdot}2^8 & \text{with } q{=}\delta(\delta(2^{34},i),i) \\
2^{35}{+}1{+}a & \text{with } q{=}2^{35} \\
2^{35}{+}1{+}a{+}(q{-}2^{35}{-}1){\cdot}2^8 & \text{with } q{=}\delta(2^{35},i) \\
2^{35}{+}1{+}a{+}(q{-}2^{35}{-}1){\cdot}2^8 & \text{with } q{=}\delta(\delta(2^{35},i),i) \\
a{+}(q{-}2^{35}{-}1){\cdot}2^8 & \text{with } q{=}\delta(\delta(\delta(2^{35},i),i),i)
\end{cases}
$$

Figure 4. Transition function for the recognizer $\mathcal{A}_{\mathrm{REG}-2-\mathrm{P}}$

nesting level, and transitioning from one to the other upon recognizing the beginning and end of a constructed type. It is thus possible to implement the recognizer FSA for $\mathbf{L}_{\mathrm{REG}}$ taking care of performing the product among the automata $\mathcal{A}_{\mathrm{REG}-1}$, $\mathcal{A}_{\mathrm{REG}-2}$ and $\mathcal{A}_{\mathrm{REG}-3}$ employing any parser generator allowing to do so (e.g., multistate lexers in GNU Flex [25], or predicate grammars in ANTLR [18]).

4.3. Recognizer Implementation

Willing to implement the proposed recognizer by means of a tool generating the actual code from a synthetic-grammar based description of the $\mathbf{L}_{\mathrm{X.509}-\mathrm{r2}}$ language, we choose the formalism of *predicated grammars* [26] which can be employed as an input to the ANTLR parser generator to obtain an efficient LL(*) parser [18]. The LL(*) grammars are obtained as an extension of the common subset of the DCF grammars known as LL(k) grammars, i.e., the ones amenable to recursive descent parsing without backtracking [19]. Our choice of an LL(*) predicated grammar is justified by the expressiveness of the formalism, which allows us to combine the computational implementations of the transition functions of the length validating automaton $\mathcal{A}_{\mathrm{REG}-2}$, together with the two CS checks and the generated portion of the parser via the so-called *semantic predicates*. We note that, although it is possible in principle to provide an equally functional description of the X.509 employing a GNU Flex generated parser, exploiting the multi-state lexer functionality to combine the different transition functions and through a generous use of the semantic actions, such a description is less manageable as it would be made of a huge regular expression defining $\mathbf{L}_{\mathrm{REG}}$. Moreover, LL(*) parsers inherit from LL(k) ones the capability of providing a useful syntax error reporting, which is useful to diagnose the nature of certificate faults.

Predicated grammars are defined as augmented LL(k) grammars adding to the usual quadruple (see Section 2) $G : (\mathbf{V}_t, \mathbf{V}_n, \mathbf{P}, S, \Pi, \mathbf{M})$ two sets: the set of side-effect free *semantic predicates* Π and the set of *mutators* $\mathbf{M}$. Semantic predicates are employed as a prefix to the right hand side of a production, so that such a production is considered during the corresponding recursive descent parsing action only if the predicate evaluated on the state of the parser is true [26]. Mutators are the way predicated grammars formalize semantic actions acting on the state of the parser, which should be taken upon the matching of the right hand side of a rule [18].

The parser generation process of ANTLR produces a modified recursive descent LL parser, where in lieu of the common sets of finite-length lookaheads the decisions are taken relying on the fact that the language of the lookaheads (i.e., the language of the suffixes of the strings) can be split into a *regular partition*. This allows the LL(*) parser to employ FSAs to recognize the lookaheads, effectively

enhancing its recognition power, although at the possible cost of examining the entire remainder of the string for each decision to be taken. Therefore, while this tends to happen quite rarely, it is possible for a parsing action to take quadratic time in the size of the input.

In case the input grammar cannot be recognized by an LL(*) automaton, ANTLR falls back to an exponential-time backtracking based strategy for the generated parser. ANTLR also supports *semantic actions* to manipulate the results of the parsing action, after a successful match of the right hand side of a rule, in the same style as GNU Flex [25] and GNU Bison [27].

We implemented the recognizer described in the previous sections, employing semantic actions to perform the CS checks required to match $\mathbf{L}_{\text{AId}-\text{match}}$ and $\mathbf{L}_{\text{SI}-\text{match}}$, introducing them as soon as the portion of the parse tree matched contains the required information. The implementation of the δ functions for $\mathbf{A}_{\text{REG}-1}$ and $\mathbf{A}_{\text{REG}-2}$ FSAs and the product with the FSA portion of the LL(*) matcher automaton are performed exploiting semantic predicates for $\mathbf{A}_{\text{REG}-2}$ with the aim to forbid the transitions which would be valid according to the matcher of $\mathbf{A}_{\text{REG}-3}$ alone, and performing the uniqueness check for $\mathbf{A}_{\text{REG}-1}$ in an appropriate semantic action, upon matching of the required portion of the parse tree. We also employed semantic predicates to obtain an efficient implementation of the choice of the recognition path in the parser, when we employed the value of the OIDs to disambiguate whether an `OCTET STRING` contained in an `extnValue` field of the `Extension` ADT is indeed composite or primitive as described in Section 4.1. The same strategy was applied to disambiguate `BIT STRING` encodings in the `subjectPublicKeyInfo` and the `signatureValue` fields. The recognizer generation process of ANTLR confirmed that no fallback to the backtracking based parser is made, and that the grammar is indeed LL(*) and non ambiguous.

LL(*) parsers achieve the same correctness guarantees of classical automatically generated parsers: that is, they recognize the same language generated by the grammar from which the parser was derived. Therefore, we can claim our parser is correct, given that the grammar specification is compliant with the standard [4]. We validate the compliance of the grammar we propose by classifying the parsing errors reported, exploiting the accurate information obtainable with an LL(*) recognizer, and by manually inspecting the certificate to verify that there is actually the syntactic failure reported by our parser. We are able to report that no syntactic error detected by other libraries escapes our parser on the entire corpus of 11M X.509 certificates in use for TLS.

5. Experimental evaluation

We conduct an experimental campaign to validate the effectiveness of our parser against a comprehensive certificate set and to compare it to the recognition capabilities of 7 different TLS libraries chosen among the most widely used in securing HTTPS transactions. Our systematically generated parser allowed us to discover a significant amount of parsing errors, among which security critical issues are present. We validate such a criticality showing how one of the parsing flaws allows a common user owning a leaf certificate to act as a CA, effectively forging valid and trusted certificates for any subject. Differently from the aforementioned TLS libraries, our analysis does not perform any semantic check on the values of the fields of the considered X.509 certificates, as it is out of the scope of this work.

Experimental Settings and Libraries.
In our experimental campaign we generated the LL(*) recognizer employing the `C` backend of ANTLR [28] version 3.5.1, as no backends for statically compiled languages are available in the most recent ANTLR v4. The choice of employing the `C` backend in ANTLR was made to allow an easier

		$\mathcal{C}_2$ **chain validity**	
		Valid	**Error x**	**Error y**
$\mathcal{C}_3$ **chain validity**	**Valid**	*Valid*	*Does not happen*	*Does not happen*
	Error x	*Invalid*	*Valid*	*Invalid*
	Error y	*Invalid*	*Invalid*	*Valid*

(a) Example of two chains for certificates $\mathcal{C}_3$ and its CA certificate $\mathcal{C}_2$

(b) Validity outcome for the differential analysis of the certificate $\mathcal{C}_3$, given the validity of its chain, and the one of its CA. **Error x** and **Error y** are two distinct errors among the possible ones
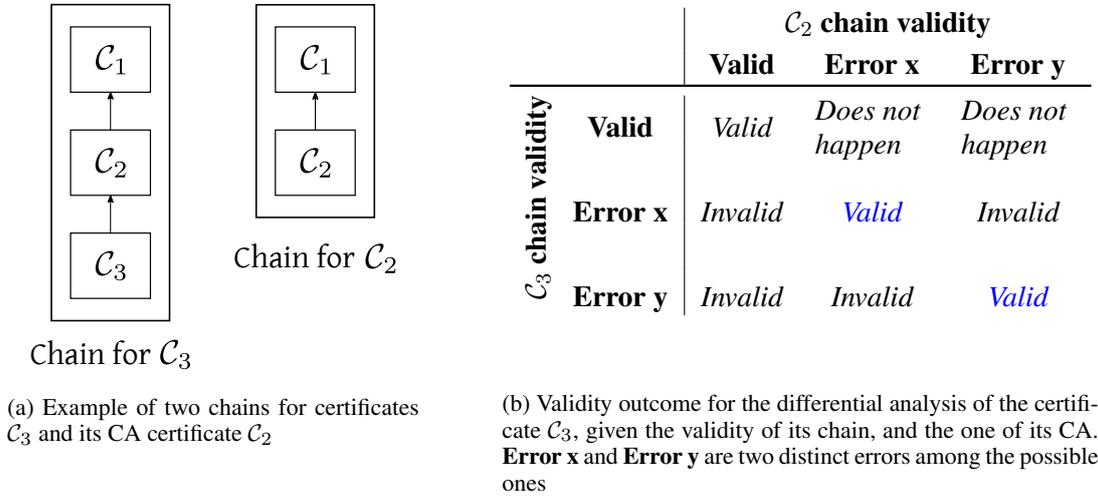
Figure 5. Differential chain analysis approach employed to prevent errors taking place in an intermediate certificate of a chain from overshadowing others. Sub-figure (a) reports two samples of certificate chains, the first having as leaf $\mathcal{C}_3$, the second one having its CA, $\mathcal{C}_2$ as leaf. Sub-figure (b) reports a summary of the validation outcome assigned to the leaf certificate of a chain, according to the validity reported by a library on both the chain terminating with it, and the one terminating with its CA

future integration of the generated parser within the most common programs employing TLS libraries, and to provide a parser implementation with good parsing performance.

We compare our parser against the following libraries: OpenSSL (v1.0.2g) [29], which is employed in both the Apache and NGINX web server SSL/TLS modules; BoringSSL (commit: ef7dba6ac, 2016-01-26) [30], the fork of OpenSSL re-engineered by Google and used as default TLS library in Chrome, Chromium and Android system binaries; GNUTLS (v3.3.17) [31], the Free Software Foundation TLS library employed in the GNOME desktop environment facilities and the Common Unix Printing System (CUPS); Network Security Services (NSS) (v3.22.2 Basic ECC) [32], the TLS implementation of the Mozilla Foundation employed in Firefox; SecureTransport (v57031.1.35) [33] and CryptoAPI (v10.0.105 86.0) [34], the proprietary implementations of TLS by Apple and Microsoft respectively, employed for OS updates by both of them; and BouncyCastle (v1.54), the default security provider in Android apps since v2.1 [35]. These libraries cover the overwhelming majority of the employed TLS providers: while other ones are available, they are either employed in more restricted application scenarios, or are obtained as a fork of one of the analyzed libraries, without changing the X.509 parsing logic (e.g., LibreSSL, Amazon s2n). For each one of the chosen libraries we realized a minimal client able to process and validate an X.509 certificate given its certification path.

We ran all the validators, including our parser, on a Linux Gentoo 13.0 amd64 host based on a six-core Intel Xeon E5-2603v3 endowed with 32 GiB DDR-4 DRAM, save for the CryptoAPI and SecureTransport clients, which were run on a Windows 10 machine, equipped with Intel i7 2600, 3.4 GHz 64-bit processor and 8 GiB DDR-3 DRAM, hosting a VMWare virtual machine running Mac OSX 10.10 Yosemite with 5.3 GiB RAM. We stored the results of the validation by all the recognizers in a MySQL database, together with base64 encoded DER certificates to ease the processing.

We choose as our dataset the one provided by the Internet-Wide Scan Data Repository [36] obtained as a result of a scan of the entire IPv4 address space on TCP port 443 on the 17th of December 2016, which encompasses $10, 999, 727$ X.509 digital certificates employed to secure HTTPS connections. A

statistical analysis of the certificates shows that their sizes are in $[0.31, 31.5]$ kiB, with an average of 1.41 kiB and a standard deviation of 0.46 kiB.

All the validation operations were made indicating the date of the certificate collection as the current one to the TLS library to avoid unwanted certificate rejections due to expiration. To the end of providing a fair comparison on the syntactic error recognition capabilities of the existing libraries on a single certificate we need to cope with their choices in terms of user interface. First of all, the existing libraries provide two different procedures, one to parse the certificate and retrieve the information stored, and another one which actually performs the validation of such information (e.g., checks signatures, checks expiration date) and builds its certification path. However, according to the errors reported by the two routines, it is quite evident that the syntactic error recognition is distributed between both of them. Consequentially, to provide a comparison of the complete syntactic detection capabilities of the libraries, we are forced to run also some or all of their semantic checks on the certificate, and its certification path, employing the validation procedure. To allow a comparison with our approach, we analyze the errors raised by these routines for each library, and we classify them in three categories:

- *syntactic*, which are related to a structural error of the certificate (e.g., `Duplicate Extensions Not Allowed` reported by Bouncy Castle). These are the relevant errors for the comparison with our parser;
- *validation*, which are related to the content of the certificate or to the certification path building (e.g., `Cert Expired` in GNUTLS or `Unable to Get Issuer Certificate` in OpenSSL). These are not relevant errors for the comparison with our parser, since they are out of its scope;
- *generic*, which are errors whose description is too loose to understand the issue they refer to, hardening the classification in one of the 2 previous categories (e.g., the `kSecTrustResultInvalid` error code reported by Apple Secure Transport). We cannot relate these errors to our parser due to this lack of information on their source.

We provide our classification of the errors reported by all the libraries in Appendix A, Table A.1.

A second point of the validation process of the libraries which we must cope with is that they provide a single answer for the validation of an entire certification path. While this is correct in terms of compliance to the TLS standard behavior, where a certificate is invalid if any of the ones composing its certification path fails validation, this behavior results in an incorrect parent certificate shadowing with its errors the potential correctness of the certificates for which it acts as a CA. In order to have a clearer insight of the extent of such a behavior we also report an ancillary set of results on the certificate correctness derived via a differential analysis strategy, of which the gist is reported in Figure 5.

The main idea of such a strategy is that if an error pertains to a certificate in a certification path, then the error disappears if the certificate is no longer in the path, and this fact can be used as a detection strategy. The differential analysis starts from validating all the possible certification paths, both the ones having an actual leaf certificate as the last one ($\mathcal{C}_3$, in Figure 5(a)), and all the ones having its CAs as the last certificate in the path ($\mathcal{C}_2$, in Figure 5(a)). The results are stored, and the leaf certificate is deemed valid or invalid according to the policy reported in Figure 5(b). In particular, a certificate is deemed invalid only if the error reported for the chain ending with it differs from the one reported for the chain ending with its CA. In case the errors reported for a certificate and its CA during the chain validation by a library match, the leaf certificate is deemed valid (see the two outcomes highlighted in blue in Figure 5(b)), as the differential analysis considers the match in errors as an overshadowing effect of the error in the CA certificate, which in turn causes an early exit in the validation procedure. The only precision penalty paid in this process is to deem a leaf certificate valid in case it is actually affected by

Table 2

Classification of parsing outcomes for the $10, 999, 727$ X.509 certificates stored in the *Internet-Wide Scan Data Repository* (U. Michigan), obtained from the differential analysis of $19, 024, 812$ certificate chains

Library	No. of Accepted Certificates	No. of Rejected Certificates		
		Syntactic Rejections	Validation Rejections	Generic Rejections
OpenSSL [29] (*OpenSSL Foundation*)	$3, 656, 634$	70	$7, 342, 978$	45
BoringSSL [30] (*Google Inc.*)	$3, 656, 906$	70	$7, 342, 706$	45
GNUTLS [31] (*Free Software Foundation*)	$4, 686, 475$	$4, 545$	$6, 308, 691$	16
NSS [32] (*Mozilla Foundation*)	$3, 457, 412$	33	$7, 520, 844$	$21, 436$
SecureTransport [33] (*Apple Inc.*)	$3, 574, 491$	0	$7, 389, 531$	$35, 705$
CryptoAPI [34](*Microsoft Corporation*)	$4, 197, 421$	591	$6, 801, 687$	28
BouncyCastle [37] (*Legion of Bouncy Castle*)	$4, 159, 585$	$5, 795$	$6, 723, 630$	$110, 724$
our systematic parser	$8, 638, 063$	$2, 361, 664$	$-$	$-$

the same error as its CAs, which is not expected to be a common case, given the significant variety of possible errors.

However, given the tight interleaving of syntactic and semantic checks in validation routines performed by the tested libraries, and the need to feed such routines with the whole certification path, we claim that no further information can be obtained unless a substantial re-engineering of the codebase of each library (when available) is performed. We remark that the practice of interleaving syntactic (input) validation and semantics, while being profitable from a performance standpoint, was identified as a pitfall from a security standpoint, since complex validation checks, which are more prone to vulnerabilities, may affect the parsing process. Indeed, rejecting the certificates upon parsing alone reduces the attack surface of the TLS/SSL libraries, as it does not expose internal calls to un-sanitized input.

Health State of the X.509 Certificate Set.
For each one of the considered libraries, Table 2 reports the outcome of the differential analysis of the $19, 024, 812$ certification paths and sub-chains included in the dataset. We note that only 31%–43% of the entire set is deemed valid by any one of the considered libraries, which in turn discard the overwhelming majority of the rejected certificates on the basis of validation error in their contents (e.g., invalid signature). Syntax errors are indeed composing a negligible part of the causes for rejection over the whole set of certificates (at most $\approx 0.05\%$), confirming the concerns on the soundness of the parsing actions of the libraries. We note that, due to the interleaving of the syntactic and semantic checking in common libraries, there is no way to understand which and how many syntax checks have been run when the semantic validation takes place. As a consequence, for the certificates deemed semantically invalid, we cannot provide a more precise comparison purely between the syntax checks of our parser and the ones in the libraries. A further tiny portion of the certificates (at most $\approx 1.6\%$) is rejected by the libraries with errors named as *generic*.

By contrast, our systematically generated parser recognizes 78.5% of certificates as syntactically X.509 abiding and, most interestingly, is able to discard 21.5% of the whole set of $10, 999, 727$ certificates via an appropriate language analysis only. Such a result provides strong evidence supporting a precise parsing action before the contents of the certificate are employed to perform validation.

Exploiting the accurate error reporting allowed by LL(*) parsers, we classified the syntactic issues found. Figure 6 reports errors found by our parser among the $2, 361, 664$ certificates deemed syntactically
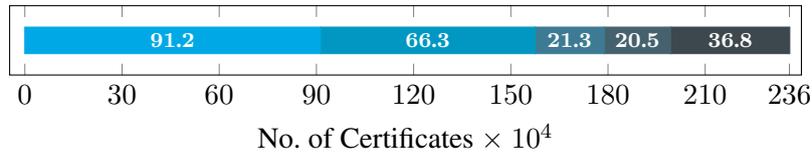
Figure 6. Syntactic errors recognized by our parser $(2,361,664)$. The bars shows the number of certificates suffering from: ▬ MISSING `keyIdentifier` IN NOT SELF-ISSUED CERT; ▬ BAD DNS/URI/EMAIL; ▬ MISSING `subjectKeyId`; ▬ `keyUsage` VIOLATION ON PK ALGORITHM ▬ includes 41 different kinds of errors

invalid. As shown, only 4 errors are covering about 84% of the rejected certificates, that are: MISSING `keyIdentifier` IN NOT SELF-ISSUED CERT, BAD DNS/URI/EMAIL, MISSING `subjectKeyId`, and `keyUsage` VIOLATION ON PK ALGORITHM.

Among these errors, the first and the third most frequent ones are not security critical, since they report missing fields which are required in order to help certification path building, thus the worst outcome may be a chain building error. By contrast, the second and fourth most frequent errors can be a source of security issues. In particular, malformed DNS, URI or email addresses may be exploited for malicious purposes by generating a malformed string which is actually deemed valid but interpreted differently by two implementations. Practical impersonation attacks exploiting differences in the interpretation of the same string among different implementations were proven a realistic threat in [9]. Concerning the fourth most common flaw in the certificates, violations in the `keyUsage` policy, it allows a public key to be employed in a cryptographic operation the corresponding primitive is not designed for (e.g., a Diffie-Hellman public key employed to check digital signatures), with unpredictable and potentially harmful consequences.

We remark our parser identifies 45 different errors on the dataset, showing a high variety of syntactical issues in the certificates analyzed, and a highly specialized error recognition capability. Among these different errors, one of them refers to the presence of an `AlgorithmIdentifier` ADT with an unrecognized algorithm OID. Recall that we impose this restriction on our grammar both to get rid of undecidable portions of the language, as well as relying only on standardized algorithm, which have generally undergone an higher level of scrutiny. We determined that such a restriction affects only $23,427$ certificates, that is, 0.2% of the dataset. We thus consider such a restriction acceptable in our implementation, especially considering that additional `AlgorithmIdentifier`s may be added to the parser by an implementor who has specific needs – e.g., maintenance reasons arising from the fact that the ITU or one of its authorized subsidiaries approved a supplementary (non-standard) algorithm identifier value.

Comparison of Parsing Effectiveness.
Table 3(a) and Table 3(b) report a comparative analysis on the effectiveness of our parser against the validation capability of the other libraries. The results in Table 3(a) show that no actual syntax errors are detected by the tested libraries on the set of $8,638,063$ certificates accepted by our parser. Indeed, a small number of syntax errors are reported by them; however, through manual inspection of these few certificates we confirmed that such errors are due to the lack of support for some features of the X.509 standard by the the said libraries. In particular, both OpenSSL and CryptoAPI do not support the recognition of some fields of `generalNames` in `NameConstraints` extension and have only partial support for the algorithms included in the Russian standard suite GOST. Moreover, NSS does not match the OID for the MD5 hash algorithm unless explicitly forced to do so via a flag set during the validation process. While discarding MD5 signed certificates is a sound choice from a semantic standpoint, as MD5 is known to be cryptographically weak, not matching the OID syntactically diverges from the standard

Table 3

Sub-table (a) reports the outcomes for the $8,638,063$ certificates accepted by our parser. The number of certificates having standard features which are unsupported by the tested library is reported in brackets. Sub-table (b) reports the outcomes for the $2,361,664$ certificates rejected by our parser

(a) Libraries outcomes on the set of $8,638,063$ certificates accepted by our parser

Library	Syntactic Rejection	Validation Rejection	Generic Rejection	Differential Acceptance
OpenSSL	0 (8)	5,613,323	38	3,024,694
BoringSSL	0 (8)	5,613,122	38	3,024,895
GNUTLS	0	5,108,243	1	3,529,819
NSS	0 (2)	6,081,980	261	2,555,820
SecureTransport	0	6,003,250	2,509	2,632,304
CryptoAPI	0 (60)	5,491,470	0	3,146,533
BouncyCastle	0	5,508,977	8,341	3,120,752

(b) Libraries outcomes on the set of $2,361,664$ certificates rejected by our parser

Library	Syntactic Rejection	Validation Rejection	Generic Rejection	Differential Acceptance	Whole Chain Acceptance	Invalid Certification Paths
OpenSSL	62	1,729,655	7	631,940	608,838	1,942,174
BoringSSL	62	1,729,584	7	632,011	608,919	1,942,251
GNUTLS	4,545	1,200,448	15	1,156,656	711,663	3,204,136
NSS	33	1,438,864	21,175	901,592	478,681	2,758,092
SecureTransport	0	1,386,281	33,196	942,187	483,291	3,398,365
CryptoAPI	531	1,310,217	28	1,050,888	611,524	3,419,740
BouncyCastle	5,795	1,214,653	102,383	1,038,833	605,552	3,412,395

recommendations. The absence of syntactic errors reported by other libraries on certificates accepted by our parser is really meaningful, since it entails that there are no syntactic flaws, identified by other libraries, which are missed by our parser. This is a practical outcome further validating the compliance of our designed grammar to the X.509 standard. The high number of certificates accepted by our parser and rejected by the libraries due to semantic validation errors confirms the sensible intuition that checking the semantics of a certificate content (e.g., via a signature check) is a crucial point in its validation process.

Table 3(b) shows how the tested libraries fare on analyzing the certificates which our parser deems syntactically incorrect. The first column of the said table shows that the portion of certificates recognized by the available libraries as syntactically incorrect is remarkably small (i.e., less than 0.25% of $2,361,664$ certificates), while a significant amount of the considered set of certificates (around 56%–73.2%) are detected as invalid during the validation phase of the libraries (see the element-wise sum of the 2nd and the 3rd column in Table 3(b)). These results confirm the fact that the certificate validation performed by libraries tightly blends syntactic validation and semantic checks, instead of clearly splitting the two phases. Nonetheless, even the richness of semantic checking is not sufficient for the tested libraries to detect all syntactically invalid certificates. Indeed, around 26.8%–44% of the set of $2,361,664$ syntactically incorrect certificates are accepted by the tested libraries employing the differential analysis technique described before (4th column in Table 3(b)). Such a result can be fruitfully interpreted comparing it with the one of the rejections caused on the entire certification path of a given

Table 4

Syntactic issues on the $468,052$ certificates of the entire dataset, which are differentially accepted by all the tested libraries

Security Critical	Number		Non Security Critical	Number
keyCertSign in leaf certificates w/o basicConstraints	$1,369$		keyCertSign encoding	2
			Empty value field	3
keyUsage violation on PK algorithm	$87,524$		Wrong OID in Distinguished Name	10
keyCertSign in leaf certificates	97		pathLenConstraint in not critical basicConstraints	15
Wrong string type	$1,289$		Wrong extnId	24
Char Set Violation	$14,155$		generic error	56
Bad DNS/URI/email format	$341,348$		missing subjectKeyId	61
Lexing Error	15		not critical basicConstraints	65
Empty Issuer Distinguished Name	368		wrong OID	83
Duplicated Extension	1		pathLenConstraint in leaf certificates	193
Unexpected NULL in AlgorithmIdP	1		empty generalNames	266
OID Arc overflow	5		empty string	401
Wrong algorithm	21		invalid Distinguished Name	$2,575$
Invalid date	122		missing keyIdentifier in self-issued certificate	$17,983$

certificate (5th column in Table 3(b)). Indeed, we recall that differential analysis accepts a certificate even if the outcome of the library validation flags it as invalid, if the reason for the rejection is the same as another certificate on its certification path. Consequentially, comparing the results of the whole chain acceptance with the differential acceptance ones points to the possibility that the effectiveness in rejecting syntactically incorrect certificates of existing libraries is a consequence of the rejection of a different, invalid certificate present in the same certification path of a syntactically wrong certificate. Indeed, the only cases where the differential acceptance may underestimate the recognition capability of the libraries are the ones where in a certification path two certificates with the same flaw are present. Nonetheless, even considering the maximum possible selectivity for the existing libraries, i.e., deeming a certification path invalid as a whole, between 20.2% and 30.3% of the syntactically invalid certificates are still deemed valid by existing libraries.

Finally, to quantify the extent of the missing rejections in the existing libraries we computed the set of certificates which are deemed differentially accepted by them, but rejected by our parser. Having derived this set of certificates we counted how many of the certificates in the entire certificate collection are either in the aforementioned set, or are issued by one of the subjects of the certificates of the said set (i.e., a certificate in the differentially accepted wrong certificates set is their ancestor in a certificate path). The last column of Table 3(b) reports the amount of such certificates for each library, providing the number of certification paths where at least one certificate is deemed invalid by us, and valid by the existing libraries, i.e., the amount of leaf certificates which are deemed valid when, according to the recursive validation strategy, they shouldn't be.

Table 5

Syntactic issues on the $468,087$ certificate chains accepted by each one of the tested libraries

Security Critical	Number
`keyUsage` violation on PK algorithm	$83,033$
`keyCertSign` in leaf certificates	1
Wrong string type	81
Char Set Violation	$12,167$
Bad DNS/URI/email format	$366,536$
Lexing Error	13
Invalid Date	1

Non Security Critical	Number
Wrong OID in Distinguished Name	2
Empty value field	2
empty `generalNames`	5
missing `subjectKeyId`	6
`pathLenConstraint` in not critical `basicConstraints`	15
`basicConstraints` not critical	38
empty string	199
invalid Distinguished Name	$1,145$
missing `keyIdentifier` in self-issued certificate	$4,852$

5.1. Detected Security Vulnerabilities

Table 4 reports the outcomes of our certificate parsing on the set of $468,052$ certificates deemed differentially valid by all the other libraries (i.e., on the intersection of the sets accounted for in the 4th column of Table 3(b)), while the data reported in Table 5 represents the outcomes of our parser on certificates belonging to a chain correctly validated by all the considered libraries (i.e., on the set of $468,052$ certificates obtained as the intersection of the sets accounted for in the penultimate column of Table 3(b)). Despite the reduction in the variety of errors reported in Table 5 with respect to the ones in Table 4, we note that the errors appearing only in the latter are not guaranteed to be safely detected by each library, while the ones in Table 5 are definitely evading detection by all the libraries. [1]

Both tables categorize the errors splitting them in two groups, according to their potential in generating exploitable security vulnerabilities. In the following, we detail the possible exploitation for each of these security critical errors, providing a rationale for such criticality. While we deem relevant the detailed investigation of each of the highlighted security flaws reported in this work, we provide only a single full proof-of-concept attack employing them, as the purpose of this work is to build a sound X.509 parser preventing all of them altogether. We spur the detailed investigation in this regard by releasing publicly our parser implementation [38].

The first and the third entry in Table 4 report issues on the `keyCertSign` bit being improperly set (either missing the `BasicConstraints` extension or in a leaf certificate) potentially allowing their subject to act as a malicious CA. We actually prove this attack to be feasible against some of the libraries, using certificates which exhibit this kind of error. Character set violations and strings of a type different from the one expected may be exploited in the same way as the already discussed issues about malformed DNS, URI or email addresses. Therefore, these issues may lead to impersonation attacks arising from different interpretations of the same string. Lexing errors state an incorrect DER encoding of ASN.1

---

[1] The outcomes of the syntactic analysis performed by our parser on the set of certificates accepted by the considered libraries separately, are reported in Appendix A, Table A.2 for OpenSSL and BoringSSL, Table A.3 for GNUTLS, Table A.4 for Secure Transport, Table A.5 for Bouncy Castle, Table A.6 for CryptoAPI, and Table A.7 for NSS

Table 6

Number of hosts, distinguished via their names, for which a certificate deemed incorrect by our parser is accepted by a library, considering the result of differential analysis

Library	# Certificates	# Names Affected
OpenSSL	$631,940$	$2,143,635$
BoringSSL	$632,011$	$2,143,812$
SecureTransport	$942,187$	$3,890,999$
CryptoAPI	$1,050,888$	$4,253,895$
GNUTLS	$1,156,656$	$4,543,605$
NSS	$901,592$	$3,786,986$
BouncyCastle	$1,038,833$	$4,196,435$

structures which may lead to a variety of attacks depending on the type of violation. Examining more closely some of them, we discovered that some implementations consider null character of byte strings such as `0x02 0x01 0x03 0x00` in the DER encoded INTEGER (tag 2) of length 1 with value 3, which suggest such implementations may be employing the outcome of the `strlen` C library function with the value of the actual length field, or employ C string based input reading functions. More serious issues such as flawed checks on length fields may lead to forgery attacks such as the ones reported in [12, 13]. The errors named as "Empty Issuer Distinguished Names" imply that a certificate has no issuer, which may lead to security issues depending on how the issuer of the certificate is retrieved by the implementation. The presence of duplicated extensions may lead to impersonation attacks, since implementations may employ the contents of one duplicate only, at their choice. A critical case for such a behavior is the one of a certificate with two `basicConstraints` extensions, one with `ca` flag set to `true` while the other one set to `false`. In such a case, it depends on an implementation-related choice whether to take into account only at the first extension, and thus considering the subject a CA, or to consider the second extension valid deeming the subject an end entity. Such misinterpretation may lead to powerful attacks where an end entity acts as a malicious CA. Unexpected `NULL` in `algorithmP` means that there are no parameters for an algorithm, even if they are expected, potentially weakening the security guarantees provided by some primitives (e.g., missing elliptic curve parameters in ECDSA may lead to backtracking to an unsafe default choice). OID *arc* overflows indicate that a single arc of an OID is indeed exceeding the maximum value set by the standard. While mishandling of arcs due to a short integer representation is known to lead to practical attacks in flawed implementations [9], we note that an OID arc overflow is potentially more dangerous as even standard abiding libraries will likely fail to manage it. Wrong Algorithm errors imply that a non standard algorithm is employed. While the employed algorithm may still be a sound one, we note that standardized algorithms have usually undergone a higher level of scrutiny. Finally, we report that some certificates have non-existing dates in their validity specification (e.g., the 29th of February 2022), potentially resulting in an erroneous expiration check.

5.2. Analysis of the Certificate Statuses on Distinct Hosts

Following the quantitative analysis on how many certificates are affected by potentially security threatening syntactic flaws, we want to analyze the effect of such flaws when reflected onto the hosts which are employing the said certificates. To this end, we analyzed the contents of the `Name` ADT of the certificates, which usually contains one or more URLs of the host for which the certificate is valid. Moreover, there is a particular standardized extension, called `Subject Alternative Names`, which contains a

```
KeyUsage ::= BIT STRING {
        digitalSignature       (0),
        nonRepudiation         (1),
        keyEncipherment        (2),
        dataEncipherment       (3),
        keyAgreement           (4),
        keyCertSign            (5),
        cRLSign                (6),
        encipherOnly           (7),
        decipherOnly           (8) }

BasicConstraints ::= SEQUENCE {
        cA                     BOOLEAN DEFAULT FALSE,
        pathLenConstraint      INTEGER (0..MAX) OPTIONAL }
```

Figure 7. `KeyUsage` and `BasicConstraints` extensions

further set of names (we consider DNS, email addresses, URLs and URIs) which are related to the subject of the certificate. To evaluate the practical impact of flaws in the certificates we report in Table 6 the amount of names bound to flawed certificates which are not detected by a library, but deemed incorrect by our parser. As emerging from the data, on average 4 common names are present for each certificate, thus pointing to a four-fold increase in the number of impacted hosts with respect to the rough estimate which can be provided considering each certificate as used by a single host.

5.3. Parsing Vulnerability Exploitation

Willing to validate the practical exploitation of the security issues emerged we focus on the syntactic problem of a certificate having no `BasicConstraints` (BC) extension, while having the `keyCertSign` bit set in the `KeyUsage` extension (there are $1,369$ such instances as reported in Table 4). Refer to Figure 7 for the detailed structure of these two extensions.

The `BasicConstraints` extension contains a single boolean field indicating whether the subject is a CA or not, and an optional constraint on the maximum length of the certification path. The `KeyUsage` extension is a 9-bit string storing flags which indicate the legitimate uses for the public key of the certificate subject. Semantically, the `keyCertSign` flag allows the public key of the subject to be used to verify certificate signatures, but the subject must be a CA. Such an information is contained in the `cA` Boolean field of BC, which has a default value of `FALSE`. Thus, if the `BasicConstraints` extension is missing, the subject cannot be considered a CA and the certificate must be rejected. An incorrect behavior in this case was reported in the recent OpenSSL bug report [39]: indeed OpenSSL versions prior to 1.0.2d allowed such syntactically flawed certificates to be used as CA ones. The version of OpenSSL employed in our experimental evaluation includes the bug fix reported in [39]; however such a fix is not addressing the root cause of the issue, as a number of syntactically incorrect certificates are still deemed valid.

We reproduced the issue at hand in a certificate of which we own the private key, employing the said private key to sign a forged certificate for `paypal.com`. The syntactically flawed certificate was signed with a private key of a CA bootstrapped by us, and such a CA certificate was added to the trusted storages of the verifying clients, completing the reproduction of the situation found in the wild. We provided the certification path of our forged `paypal.com` certificate to OpenSSL both via the programming API, and via command-line client, employing the default validation options, succeeding in getting it accepted as a valid certificate. An interesting remark is that such attack is mitigated if the `x509_strict` option
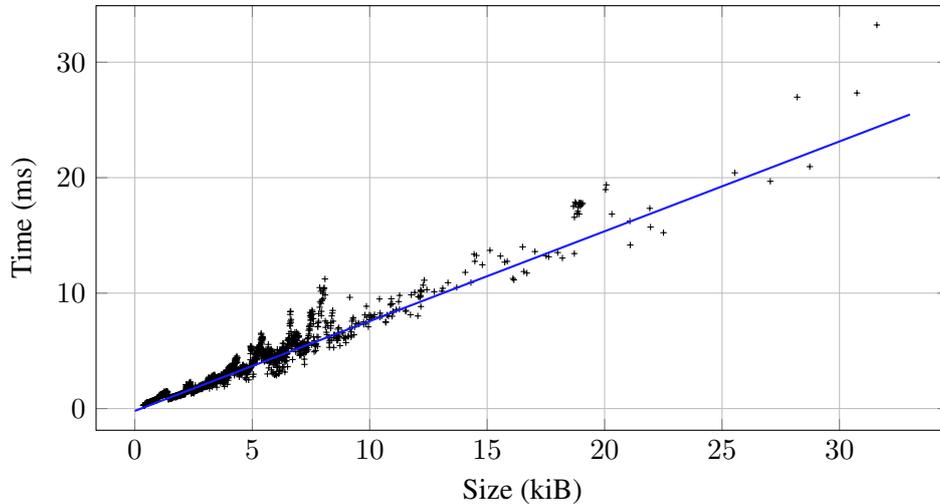
Figure 8. Parsing times for all the certificates present in our dataset. The blue line marks the linear interpolation of the obtained data points, with equation `Time=0.778·Size−0.203`

is set for the validation algorithm. However, this option is not enabled by default, leaving all entities relying on default OpenSSL settings (which are expected to be the majority) vulnerable to this serious attack. We were able to get the said certification path to be accepted also by BoringSSL, while the other libraries discard the chain. Nonetheless, we confirm the practicality of the threat on two of the most used TLS implementations, including the one employed by Chrome/Chromium. As a consequence, any one among the owners of the $1,369$ certificates of the dataset with the said vulnerability are likely to be able to exploit it successfully, as such items have been signed by a legitimate and trusted root CA. An inspection of such certificates show that some have been generated with the Open Directory framework of Mac OS X Server, providing a pointer to a real world TLS implementation generating syntactically incorrect certificates. Such implementations can be exploited by an attacker who could require a legitimate certificate for its own identity to such implementations. The generated certificate would suffer the mentioned syntactic issue and could be used as an intermediate certificate to sign other certificates for an arbitrary identity. Such certificates can later be used to perform a Man-In-The-Middle attack where a TLS session is successfully established impersonating the subject of the fake certificate.

5.4. Performance Analysis

Finally, although our work focuses on the effectiveness of the proposed certificate parsing approach, we provide some efficiency results of our parser. Figure 8 reports the performance results obtained running our parser on a 3.2GHz Intel i5-6500 based desktop, running Gentoo Linux 13.0 (x86_64). The reported timings show how the vast majority of the certificates can be recognized in less than 10 ms, a timing we deem acceptable for practical use, and which may be improved employing a more efficient C code generation backend with respect to the current one in ANTLR3. In addition to the absolute timings reported, it is interesting to note that the practical parsing complexity appears to be substantially linear, despite the theoretical quadratic worst case of LL(*) parsers. Such a fact represents a validation of our claim that the lookahead which the LL(*) parser requires is indeed far shorter than the entirety of the remaining input, indeed resulting in near optimal (i.e., linear) parsing complexity.

6. Concluding Remarks

We presented a systematic approach to the parsing of X.509 digital certificates, analyzing the standard and providing a description of the ADTs to be matched in terms of a predicate grammar. We generated systematically a parser from the given grammar representation, and analyzed with it 11M X.509 certificates in use to secure TLS connections. We report that 21.5% are syntactically incorrect and 7 of the most common TLS libraries deem authentic 5.7%–10.5% of them. We provided a validation of the practicality of the threat represented by mis-parsed certificates, providing a proof of concept of an effective impersonation attack stemming from them. We hope this work will encourage the integration of our systematically generated parser in widespread TLS libraries, although it is also possible to employ it as a stopgap measure in programs dynamically linked to existing libraries, including it within a wrapper to their API calls. Although we provided a single proof-of-concept exploit of a security vulnerability among the ones we identified, we believe exploring all the remaining ones will provide a concrete evaluation of the extent of the current issues in X.509 certificates.

Appendix A

Table A.1

Classification of the outcomes of the certificate chain validation procedure for each one of the tested TLS libraries as syntactic, validation or generic errors

SecureTransport		
	Syntactic	n/a
	Validation	`kSecTrustResult` deny
	Validation	`kSecTrustResult` recoverable trust failure
	Validation	`kSecTrustResult` fatal trust failure
	Generic	`kSecTrustResult` other error
	Generic	`kSecTrustResult` invalid

GNUTLS		
	Syntactic	Insecure Algorithm
	Validation	Certificate signature failure
	Validation	Certificate signer not CA
	Validation	Certificate expired
	Validation	Certificate signer not found
	Validation	Invalid Certificate
	Generic	Signer constraints failure

OpenSSL, BoringSSL		
	Syntactic	Unsupported constraint type
	Syntactic	No explicit policy
	Validation	Path length exceeded
	Validation	Permitted violation
	Validation	Certificate not yet valid
	Validation	Unhanded critical extension
	Validation	Unable to get issuer certificate
	Validation	Self-signed certificate in chain
	Validation	Certificate signature failure
	Validation	Certificate has expired
	Validation	Unable to get issuer certificate locally
	Validation	Depth zero self-signed certificate
	Generic	Invalid CA
	Generic	Invalid policy extension
	Generic	Unable to decode issuer public key

NSS		
	Syntactic	Signature algorithm disabled
	Validation	Untrusted certificate
	Validation	Bad signature
	Validation	Certificate not in namespace
	Validation	Untrusted issuer
	Validation	Expired certificate
	Validation	Unknown issuer
	Generic	Invalid arguments
	Generic	Inadequate key usage
	Generic	Inadequate certificate type

CryptoAPI		
	Syntactic	Not supported name constraint
	Syntactic	NTE bad algorithm identifier
	Syntactic	Invalid date
	Validation	Forbidden name constraint
	Validation	Revoked
	Validation	Off-line revocation
	Validation	Weak signature
	Validation	Signature not valid
	Validation	Time not valid
	Validation	Partial chain
	Validation	Untrusted root
	Generic	Invalid arguments
	Generic	Inadequate key usage
	Generic	Inadequate certificate type

BouncyCastle		
	Syntactic	Forbidden extension in v2 certificates
	Syntactic	Empty `issuerDN`
	Syntactic	URI must include scheme
	Syntactic	Empty `subjectDN` is not allowed in v1 certificates
	Syntactic	Invalid URI name
	Syntactic	Duplicate extensions are not allowed
	Syntactic	Signature algorithm mismatch
	Syntactic	No more data allowed for v1 certificate
	Syntactic	Incomplete X.509 certificate: empty subject field, and absent `SubjectAlternativeName` extension
	Syntactic	Incomplete X.509 certificate: `SubjectAlternativeName` extension must be marked as critical, when subject field is empty
	Validation	`targetConstraints` mismatch
	Validation	No issuer found in certification path
	Validation	Unable to find certificate chain
	Generic	certificate validation failed
	Generic	Certification path could not be validated

Table A.2

Syntactic issues on the $608, 838$(resp. $608, 919$) certificates of the entire dataset differentially accepted by OpenSSL(resp. BoringSSL)

Security Critical	OpenSSL	BoringSSL
`keyUsage` violation on PK algorithm	$126, 174$	$126, 182$
`keyCertSign` in leaf certificates	17	10
Wrong string type	140	139
Char Set Violation	$13, 913$	$13, 925$
Bad DNS/URI/email format	$455, 539$	$455, 536$
Lexing Error	57	57
Wrong algorithm	1	11
Extension found but version $\neq 3$	1	0

Non Security Critical	OpenSSL	BoringSSL
Empty value field	2	3
Wrong OID in Distinguished Name	3	2
`pathLenConstraint` in not critical `basicConstraints`	10	11
generic error	$4, 909$	$4, 909$
missing `subjectKeyId`	59	80
not critical `basicConstraints`	65	74
wrong OID	88	89
`pathLenConstraint` in leaf certificates	27	25
empty `generalNames`	14	16
empty string	278	277
invalid Distinguished Name	$1, 368$	$1, 370$
missing `keyIdentifier` in self-issued certificate	$6, 168$	$6, 196$
Bad `BIT STRING` encoding	1	1
Redundant Trailing Bytes	2	2
not critical `basicConstraints` and missing `subjectKeyId`	1	3
Empty sequence in `Authority/Subject Information Access` exts	1	1

Table A.3

Syntactic issues on the 711, 663 certificates of the entire dataset, which are differentially accepted by GNUTLS

Security Critical	Number		Non Security Critical	Number
keyUsage violation on PK algorithm	151, 727		Empty value field	2
keyCertSign in leaf certificates	20		Wrong OID in Distinguished Name	3
Wrong string type	397		pathLenConstraint in not critical basicConstraints	12
Char Set Violation	16, 637		generic error	4, 907
Bad DNS/URI/email format	514, 645		missing subjectKeyId	52
Lexing Error	14		not critical basicConstraints	57
Wrong algorithm	2		wrong OID	96
Duplicated Extensions	2		pathLenConstraint in leaf certificates	124
			empty generalNames	12
			empty string	487
			invalid Distinguished Name	1, 638
			missing keyIdentifier in self-issued certificate	20, 825
			Redundant Trailing Bytes	1
			not critical basicConstraints and missing subjectKeyId	1
			Empty sequence in Authority/Subject Information Access extensions	1
			Wrong extnId	1

Table A.4

Syntactic issues on the $483,291$ certificates of the entire dataset, which are differentially accepted by Secure Transport

Security Critical	Number
`keyUsage` violation on PK algorithm	86,276
`keyCertSign` in leaf certificates	4
Wrong string type	84
Char Set Violation	12,558
Bad DNS/URI/email format	377,788
Lexing Error	14
Wrong algorithm	1

Non Security Critical	Number
Empty value field	2
Wrong OID in Distinguished Name	2
`pathLenConstraint` in not critical `basicConstraints`	8
missing `subjectKeyId`	31
not critical `basicConstraints`	85
empty `generalNames`	5
empty string	207
invalid Distinguished Name	1,163
missing `keyIdentifier` in self-issued certificate	5,060
Redundant Trailing Bytes	2
Empty sequence in `Authority/Subject Information Access` extensions	1

Table A.5

Syntactic issues on the 608,038 certificates of the entire dataset, which are differentially accepted by Bouncy Castle

Security Critical	Number
`keyUsage` violation on PK algorithm	126,232
`keyCertSign` in leaf certificates	3
Wrong string type	137
Char Set Violation	13,710
Bad DNS/URI/email format	452,630
Lexing Error	13
Wrong algorithm	1

Non Security Critical	Number
Empty value field	2
Wrong OID in Distinguished Name	2
`pathLenConstraint` in not critical `basicConstraints`	8
generic error	4,907
missing `subjectKeyId`	25
not critical `basicConstraints`	57
wrong OID	88
`pathLenConstraint` in leaf certificates	26
empty `generalNames`	6
empty string	274
invalid Distinguished Name	1,355
missing `keyIdentifier` in self-issued certificate	6,073
Redundant Trailing Bytes	2
Empty sequence in `Authority/Subject Information Access` extensions	1

Table A.6

Syntactic issues on the 611, 524 certificates of the entire dataset, which are differentially accepted by CryptoAPI

Security Critical	Number
`keyUsage` violation on PK algorithm	121, 686
`keyCertSign` in leaf certificates	10
Wrong string type	194
Char Set Violation	14, 105
Bad DNS/URI/email format	462, 238
Lexing Error	14
Wrong algorithm	1
Wrong `keyCertSign` ASN.1 encoding	1

Non Security Critical	Number
Empty value field	3
Wrong OID in Distinguished Name	2
`pathLenConstraint` in not critical `basicConstraints`	12
generic error	4, 907
missing `subjectKeyId`	23
not critical `basicConstraints`	94
wrong OID	105
`pathLenConstraint` in leaf certificates	29
empty `generalNames`	7
empty string	523
invalid Distinguished Name	1, 421
missing `keyIdentifier` in self-issued certificate	6, 146
Redundant Trailing Bytes	2
Empty sequence in `Authority/Subject Information Access` extensions	1

Table A.7

Syntactic issues on the $478,681$ certificates of the entire dataset, which are differentially accepted by NSS

Security Critical	Number
`keyUsage` violation on PK algorithm	$87,187$
`keyCertSign` in leaf certificates	2
Wrong string type	98
Char Set Violation	$12,832$
Bad DNS/URI/email format	$372,132$
Lexing Error	14
Wrong algorithm	1

Non Security Critical	Number
Empty value field	2
Wrong OID in Distinguished Name	2
`pathLenConstraint` in not critical `basicConstraints`	7
missing `subjectKeyId`	20
not critical `basicConstraints`	54
wrong OID	62
`pathLenConstraint` in leaf certificates	1
empty `generalNames`	6
empty string	202
invalid Distinguished Name	$1,177$
missing `keyIdentifier` in self-issued certificate	$4,881$
Redundant Trailing Bytes	1

References

[1] T. Polk, R. Housley and L. Bassham, Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, RFC, 3279, 2002.

[2] R. Housley, B. Kaliski and J. Schaad, Additional Algorithms and Identifiers for RSA Cryptography for use in the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL), RFC, 4055, 2005.

[3] S. Leontiev and D. Shefanovski, Using the GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms with the Internet X.509 Public Key Infrastructure Certificate and CRL Profile, RFC, 4491, 2006.

[4] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley and T. Polk, Internet X.509 Public Key Infrastructure Certificate and CRL, RFC, 5280, 2008.

[5] Q. Dang, S. Santesson, K.M. Moriarty, D.R.L. Brown and T. Polk, Internet X.509 Public Key Infrastructure: Additional Algorithms and Identifiers for DSA and ECDSA, RFC, 5758, 2010.

[6] S. Turner, D.R.L. Brown, K. Yiu, R. Housley and T. Polk, Elliptic Curve Cryptography Subject Public Key Information, RFC, 5480, 2009.

[7] M. Marlinspike, Internet Explorer SSL Vulnerability, 2002. https://moxie.org/ie-ssl-chain.txt.

[8] M. Marlinspike, Null Prefix Attacks against SSL/TLS Certificates, 2009. https://moxie.org/papers/null-prefix-attacks.pdf.

[9] D. Kaminsky, M.L. Patterson and L. Sassaman, PKI Layer Cake: New Collision Attacks against the Global X.509 Infrastructure, in: *Financial Cryptography and Data Security, 14th International Conference, FC 2010, Tenerife, Canary Islands, January 25-28, 2010, Revised Selected Papers*, R. Sion, ed., Lecture Notes in Computer Science, Vol. 6052, Springer, 2010, pp. 289–303. ISBN 978-3-642-14576-6. doi:10.1007/978-3-642-14577-3.

[10] N. Vratonjic, J. Freudiger, V. Bindschaedler and J. Hubaux, The Inconvenient Truth about Web Certificates, in: *10th Annual Workshop on the Economics of Information Security, WEIS 2011, George Mason University, Fairfax, VA, USA, June 14-15, 2011*, 2011.

[11] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh and V. Shmatikov, The most dangerous code in the world: validating SSL certificates in non-browser software, in: *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, T. Yu, G. Danezis and V.D. Gligor, eds, ACM, 2012, pp. 38–49. ISBN 978-1-4503-1651-4. doi:10.1145/2382196.2382204.

[12] A. Delignat-Lavaud, BERserk Vulnerability Part 1: RSA Signature Forgery Attack Due to Incorrect Parsing of ASN.1 Encoded DigestInfo in PKCS1 v1.5, 2014. http://www.intelsecurity.com/resources/wp-berserk-analysis-part-1.pdf.

[13] A. Delignat-Lavaud, BERserk Vulnerability Part 2: Certificate Forgery in Mozilla NSS, 2014. http://www.intelsecurity.com/resources/wp-berserk-analysis-part-2.pdf.

[14] C. Brubaker, S. Jana, B. Ray, S. Khurshid and V. Shmatikov, Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations, in: *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, IEEE Computer Society, 2014, pp. 114–129. ISBN 978-1-4799-4686-0. doi:10.1109/SP.2014.15.

[15] A. Barenghi, N. Mainardi and G. Pelosi, A Novel Regular Format for X.509 Digital Certificates, in: *Information Technology - New Generations*, S. Latifi, ed., Advances in Intelligent Systems and Computing, Vol. 558, Springer International Publishing, 2018, pp. 133–139. ISBN 978-3-319-54977-4. doi:10.1007/978-3-319-54978-1.

[16] A. Barenghi, N. Mainardi and G. Pelosi, A Security Audit of the OpenPGP Format, in: *2017 14th International Symposium on Pervasive Systems, Algorithms and Networks 2017 11th International Conference on Frontier of Computer Science and Technology 2017 Third International Symposium of Creative Computing (ISPAN-FCST-ISCC)*, S. Jarvis, L. Liu, B. Javadi and S. Guo, eds, IEEE, 2017, pp. 336–343. ISBN 978-1-5386-0841-8. doi:10.1109/ISPAN-FCST-ISCC.2017.35.

[17] International Telecommunication Union, Data networks, Open System Comm.s and Security Recommendations, 2016. https://www.itu.int/rec/T-REC-X.

[18] T. Parr and K. Fisher, LL(*): the foundation of the ANTLR parser generator, in: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, M.W. Hall and D.A. Padua, eds, ACM, 2011, pp. 425–436. ISBN 978-1-4503-0663-8. doi:10.1145/1993498.1993548.

[19] M.A. Harrison, *Introduction to Formal Language Theory*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1978. ISBN 0201029553.

[20] G. Berry and R. Sethi, From Regular Expressions to Deterministic Automata, *Theor. Comput. Sci.* **48**(3) (1986), 117–126.

[21] J. Linn, Privacy Enhancement for Internet Electronic Mail: Message Encryption and Authentication, RFC, 1421, 1993.

[22] International Telecommunication Union, Recommendation ITU-T X.509: Open Systems Interconnection - Public-key and Attribute Certificate Frameworks, 2012. https://www.itu.int/rec/T-REC-X.509-201210-I.

[23] C. Adams, S. Farrell, T. Kause and T. Mononen, Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP), RFC, 4210, 2005.

[24] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J.A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S.Z. Béguelin and P. Zimmermann, Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice, in: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications*

Security, Denver, CO, USA, October 12-6, 2015, I. Ray, N. Li and C. Kruegel, eds, ACM, 2015, pp. 5–17. ISBN 978-1-4503-3832-5. doi:10.1145/2810103.2813707.

[25] V. Paxson, GNU Fast Lexical Analyzer Generator (FLEX) - Release 2.6.0, 2015. http://flex.sourceforge.net.

[26] T.J. Parr and R.W. Quong, Adding Semantic and Syntactic Predicates To LL(K): Pred-LL(K), in: *Proceedings of the 5th International Conference on Compiler Construction*, CC '94, Springer-Verlag, London, UK, 1994, pp. 263–277. ISBN 3-540-57877-3.

[27] R. Corbett, GNU Bison - Release 3.0.4, 2015. https://www.gnu.org/software/bison.

[28] J. Idle, ANTLR3 C Runtime API and Usage Guide, 2010. http://www.antlr3.org/api/C/main.html.

[29] OpenSSL Foundation, OpenSSL: Cryptography and SSL/TLS Toolkit, 2016. https://www.openssl.org/.

[30] Google Inc., BoringSSL, 2016. https://boringssl.googlesource.com/boringssl/.

[31] Free Software Foundation, Inc., The GnuTLS Transport Layer Security Library, 2016. http://www.gnutls.org/.

[32] Mozilla Foundation, Network Security Services, 2016. https://wiki.mozilla.org/NSS.

[33] Apple Inc., Secure Transport API, 2016. https://developer.apple.com/library/mac/documentation/Security/Reference/secureTransportRef/.

[34] Microsoft Corporation, Microsoft Cryptography API, 2016. https://msdn.microsoft.com/en-us/library/windows/desktop/aa380255%28v=vs.85%29.aspx.

[35] M. Egele, D. Brumley, Y. Fratantonio and C. Kruegel, An empirical study of cryptographic misuse in android applications, in: *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, A. Sadeghi, V.D. Gligor and M. Yung, eds, ACM, 2013, pp. 73–84. ISBN 978-1-4503-2477-9. doi:10.1145/2508859.2516693.

[36] Z. Durumeric, J. Kasten, M. Bailey and J.A. Halderman, Analysis of the HTTPS certificate ecosystem, in: *Proceedings of the 2013 Internet Measurement Conference, IMC 2013, Barcelona, Spain, October 23-25, 2013*, K. Papagiannaki, P.K. Gummadi and C. Partridge, eds, ACM, 2013, pp. 291–304. ISBN 978-1-4503-1953-9. doi:10.1145/2504730.2504755.

[37] Legion of the Bouncy Castle, Bouncy Castle Java Cryptography APIs, 2016. https://www.bouncycastle.org/java.html.

[38] N. Mainardi, NAXiPP: Not Another X.509 imProper Parser, 2018. https://github.com/nicholas-mainardi/NAXiPP.

[39] A. Langley and D. Benjamin, OpenSSL Security Advisory - Alternative Chains Certificate Forgery, 2015. http://openssl.org/news/secadv/20150709.txt.