ASN1*: Provably Correct Non-Malleable Parsing for ASN.1 DER

Haobin Ni
Cornell University

Antoine Delignat-Lavaud
Microsoft Research

Cédric Fournet
Microsoft Research

Tahina Ramananandro
Microsoft Research

Nikhil Swamy
Microsoft Research

Abstract

Abstract Syntax Notation One (ASN.1) is a language for structured data exchange between computers, standardized by both ITU-T and ISO/IEC since 1984. The Distinguished Encoding Rules (DER) specify its non-malleable binary format: for a given ASN.1 data type, every value has a distinct, unique binary representation. ASN.1 DER is used in many security-critical interfaces for telecommunications and networking, such as the X.509 public key infrastructure, where non-malleability is essential. However, due to the expressiveness and flexibility of the general-purpose ASN.1 language, correctly parsing ASN.1 DER data formats is still considered a serious security challenge in practice.

We present ASN1*, the first formalization of ASN.1 DER with a mechanized proof of non-malleability. Our development provides a shallow embedding of ASN.1 in the F* proof assistant and formalizes its DER semantics within the EverParse parser generator framework. It guarantees that any ASN.1 data encoded using our DER semantics is non-malleable. It yields verified code that parses valid binary representations into values of the corresponding ASN.1 data type while rejecting invalid ones.

We empirically confirm that our semantics models ASN.1 DER usage in practice by evaluating ASN1* parsers extracted to OCaml on both positive and negative test cases involving X.509 certificates and Certificate Revocation Lists (CRLs).

1 Introduction

Abstract Syntax Notation One (ASN.1) is a data type declaration language standardized by both ITU-T and ISO/IEC since 1984.[1] It is used for exchanging structured data between platforms in a variety of settings, notably in the X.509 [11] standard for public-key certificates. The latter forms the cornerstone of digital identities and secure communication on the Internet and, as such, the ASN.1 and X.509 standards and their implementations are security critical components of societal infrastructure.

The ASN.1 language supports describing structured data of many varieties, including a wide collection of base types, products, sums, sequences, and sets. For example, we give below an ASN.1 *declaration* for two-dimensional points, where the base type `INTEGER` denotes integers of arbitrary size.

```
Point2D ::= SEQUENCE { x INTEGER, y INTEGER }
```

ASN.1 declarations can be grouped into ASN.1 modules. For example, the format of X.509 certificates is one such ASN.1 module. We give below its top-level declaration, a triple of fields:

```
Certificate ::= SEQUENCE {
    tbsCertificate TBSCertificate,
    signatureAlgorithm AlgorithmIdentifier,
    signature BIT STRING }
```

where `tbsCertificate` is the certificate contents 'to be signed' using `signatureAlgorithm`, and `signature` is the resulting signature value.

ASN.1 decouples data type declarations from their formats. It provides several classes of *encoding rules* that govern the wire format of data types, one of which known as the *distinguished encoding rules* or DER, following the general tag-length-contents encoding pattern. For example, the point $(0, 0)$ is encoded into the 8-byte string "30 06 02 01 00 02 01 00" where 30 is the tag locally assigned to points in their ASN.1 module, 02 is the primitive tag of integers, and 06, 01, 01 encode their content lengths.

DER are designed to ensure that every value of a given ASN.1 type has a distinct, canonical wire format representation. That is, DER formats are intended to be *unambiguous* and *non-malleable*, in the sense that given a bit string b that encodes a value v, every parser will yield back v, whereas changing any bit in b either produces an invalid representation or yields a distinct value $v' \neq v$. These properties are particularly important in security applications, inasmuch as they depend on values v but apply cryptographic protection only on binary formats b. In particular, the X.509 standard[2] requires that certificates be formatted using DER, to prevent any ambiguity between the claims signed by the issuer in `tbsCertificate` and their interpretation by the relying party after verifying the signature.

Despite the maturity of the standard and the presence of libraries in several languages that support their use, ASN.1

[1] https://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx

[2] https://www.rfc-editor.org/rfc/rfc5280

and DER have a reputation for being difficult to master. Implementations have suffered from parsing bugs that have led to critical vulnerabilities. For example, Marlinspike [21] discovered that Microsoft's CryptoAPI component would incorrectly parse a string containing a null character in a domain name in the subject's Common Name (CN) field of an X.509 certificate, e.g., parsing the string `"a.com\0b.com"` as `"a.com"` thereby misinterpreting the certificate issuer's intent and enabling an attacker to spoof a certificate to carry out a man-in-the-middle attack. This is a classic example of security vulnerability due to the use of a malleable parser—the parser simply ignores the content of the string after the null character. We discuss other security vulnerabilities related to X.509 parsing in §4. Of course, many vulnerabilities discovered in implementations of X.509 and related standards involve software flaws beyond parsing (e.g., in certificate chain validation [5])—however, ensuring that parsing is correct and non-malleable is a necessary basic requirement.

ASN1⋆: A formalization of ASN.1 DER. Our long-term ambition is to provide high-assurance implementations of tools to parse and serialize data to and from ASN.1 DER, and to build provably correct cryptographic applications upon such tools. This paper presents a first milestone towards that long-term goal, namely ASN1⋆, a mathematical formalization of ASN.1 DER, deeply embedding its syntax and providing several related denotational semantics within the F⋆ proof assistant [29]. It provides a precise, mathematical basis on which to understand and further study a widely used Internet standard that has, to date, only been specified in several voluminous natural-language documents.

We formalize the syntax of ASN.1 DER as a family of mutually inductive indexed types, the primary one being declaration : set id_t → Type, the type of a single ASN.1 declaration. For example, `Point2D` and `Certificate` are represented in F⋆ as instances of declaration. The index on declaration enforces a well-formedness property on ASN.1 DER specifications, a form of static discipline discussed in §2.

We provide two related denotational semantics. First, a *type denotation* asn1_as_type : declaration s → Type that interprets every well-formed ASN.1 DER declaration as a type in the meta-language, i.e., F⋆. For example, the type denotation of `Point2D` is an F⋆ pair of mathematical integers, int & int. Second, a *parser denotation* that interprets every declaration as a pure function from a sequence of bytes (a DER wire format) to either a value of its type denotation or an error. Our main theorem, outlined below

```
val asn1_as_parser : (d:declaration s) → parser (asn1_as_type d)
```

establishes that our parser denotation can be typed as a `parser`, the type of correct, non-malleable parsers defined in the EverParse framework [25], applied to our type denotation. (§A provides background on F⋆ and EverParse.) That is, we show that every well-formed ASN.1 DER declaration can be interpreted both as an F⋆ type and a non-malleable parser from a sequence of bytes to that type.

A key technical contribution of our development is that it yields a *compositional* semantics of ASN.1 DER where, despite complications of the standard such as optional elements, default elements, and local retagging, (which require careful custom treatment) our top-level theorem still offers a clear, canonical correctness and non-malleability result in terms of EverParse's parser abstraction. To this end, we also contribute new parser combinators, notably for sequence, choice, and state-machine-based parsers, together with their proofs of correctness and non-malleability.

Validating ASN1⋆. To validate that our formalization corresponds to the practice of ASN.1 DER in existing standards and interfaces, we use F⋆'s extraction mechanism to produce, for selected ASN.1 declarations expressed as instances of v : declaration s, functions in OCaml that parses a sequence of bytes. We wrote ASN1⋆ format declarations for X.509 version 3 certificates, covering its most popular extensions, and tested our extracted OCaml parser on a corpus of more than 10,000 certificates, including both positive and negative test cases, confirming that we correctly handle them all. We also tested on a further ~2,000 (mostly ill-formed) certificates dataset produced by fuzzing, and again confirmed that we correctly handle them all. We also wrote a ASN1⋆ format declarations for Certificate Revocation Lists (CRLs) and evaluated our parsers on ~4,000 CRLs found in the wild.

Extensions and Limitations. Our formalization aims to cover a practical version of ASN.1 DER, sufficient to express many formats used in the wild. We support features that are not core to ASN.1 but are commonly used in informal side conditions. For example, many specifications prescribe additional formatting constraints in natural language, e.g., X.509 has a notion of *expansion lists*, which our formalization does cover. On the other hand, we do not support a form of set that is seldom used with DER and does not occur in our case studies (see §3.1.2).

Although our formalization offers executable OCaml code for parsing, we have not attempted to optimize this code at all, and make no claims about its efficiency. Indeed, as mentioned earlier, we see our work as "merely" the formal foundation towards producing in the future high-performance, provably correct, low-level implementations of ASN.1 DER parsers and serializers, and cryptographic applications to be built using them, including certificate chain and policy validation.

In summary, our contributions include:

1. The first formalization of ASN.1 DER, providing a basis on which to understand long-standing, widely used natural language standards. Our main theorem proves that all well-formed ASN.1 DER specifications induce non-malleable parsers.

2. New correct- and non-malleable-by-construction parser combinators for sequences, choice, and state-machine-based parsers.
3. An experimental validation of our formalization by evaluating the parsers from our semantics on a corpus of ASN.1 DER formatted data in the wild, including for X.509 and CRL, confirming that our semantics is faithful to the intent of the official standard.

The anonymized supplementary materials include our formal development and experimental datasets.

2 A Brief Primer on ASN.1 and DER

Figure 1 presents an informal summary of the concrete syntax of ASN.1, distilled from the ITU's X.680 standard [17]. Figure 2 shows an actual snippet of ASN.1 declaring the type of X.509 to-be-signed certificate contents introduced in §1. We use them to establish some basic concepts and intuitions, and to convey some of the challenges involved in their formalization, presented next in §3.

An ASN.1 module declares a collection of data types, including finite sums, dependent and non-dependent products, variable-length sets and lists over a collection of base types. Each module is a list of declarations; each declaration associates a name with either a constant value (such as an object identifier) or a data type, and may refer to prior declarations by name. In Figure 2, for example, Version and AlgorithmIdentifier refer to prior declarations in scope.

A data type is either a *terminal*, such as an integer, or a type constructed from more basic types: a SEQUENCE is the product of a given list of field names f_i and *decorated* declarations, where the decorations can marks a field as optional, provide a default value when the field is omitted, and modify its tag—we discuss this in detail shortly; a SEQUENCE OF is a list of an arbitrary number of t-typed elements; the CHOICE constructor is the sum of a given list of data types. ASN.1 also offers SET and SET OF constructors that are unordered analogs of SEQUENCE and SEQUENCE OF.

A design goal of ASN.1 is to decouple type declarations from their binary formats. To this end, ASN.1 settles on an encoding scheme where all data type values are encoded in binary as identifier-length-content (ILC) tuples—the precise form of these tuples varies between the different encoding rules that ASN.1 provides, DER, our focus, being among them. The identifier, or tag, mainly serves as an indicator for the type of the value, for example, to distinguish between different cases of sum. The length specifies the length of the content field in bytes and eliminates ambiguity when a binary string can be fragmented in different ways. Although the identifier and the length fields are not always necessary, they usually do not cause much overhead, and they enable applications to skip over contents in binaries.

Primitive ASN.1 types have their own built-in identifiers. For example, the type INTEGER has identifier 02 (in hex),

$$
\begin{array}{llll}
c & ::= & \mathsf{INTEGER} \mid \mathsf{BITSTRING} \mid \ldots & \textit{Terminals} \\
t & ::= & c \mid \mathsf{SEQUENCE}\ \{f_1\ \tau_1, \ldots, f_n\ \tau_n\} & \textit{Declarations} \\
 & \mid & \mathsf{CHOICE}\ \{f_1\ \tau_1, \ldots, f_n\ \tau_n\} & \\
 & \mid & \mathsf{SEQUENCE\ OF}\ t \mid \mathsf{SET\ OF}\ t \mid \ldots & \\
\tau & ::= & t \mid \tau\ \mathsf{OPTIONAL} \mid \tau\ \mathsf{DEFAULT}\ v & \textit{Decorated decls} \\
 & \mid & [n]\ \mathsf{EXPLICIT}\ \tau \mid [n]\ \mathsf{IMPLICIT}\ \tau &
\end{array}
$$

Figure 1. Informal syntax of ASN.1

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

Figure 2. An ASN.1 declaration from X.509

so 0 is in ASN.1 DER as 02 01 00, where the first byte is the identifier for integers, the second is the length of the content (1 byte); and 00 is the content itself.

ASN.1 allows users to *override* the (otherwise decoupled) binary encoding of identifiers for their declarations. For example, one can declare MYINT ::= [1] IMPLICIT INTEGER, and the encoding of 0 as a MYINT becomes 81 01 00. The identifier byte 81 expanded in binary digits is 10 0 00001, where the first two bits indicate that this is a context-specific user-defined identifier, the next bit indicates that the data type is primitive, and the last 5 bits encode the user-chosen constant 1.

Identifier formats are actually variable-length. For example, a long identifier such as [128] IMPLICIT takes 3 bytes: the first byte is 10 0 11111, where the first 3 bits are as before, but the last five signal a long-form identifier. The next two bytes are 1 0000001 and 0 0000000, where the leading bit of the first byte signals that more bytes are to follow, and the leading bit of the third byte signals that this is the final byte of the identifier, overall representing 8 bits spread across the last two bytes. Note that a correct parser must reject unnecessary long forms, as they would break non-malleability.

ASN.1 also allows to *wrap* an encoding within a custom ILC tuple. For example, the encoding of 0 as a WRAPPED_INT ::= [1] EXPLICIT INTEGER is A1 03 02 01 00, where the leading A0 in binary is 10 1 00001, representing a constructed user-defined short identifier; the length of the wrapped contents is 3; and the content itself is the built-in encoding of 0.

ASN.1 has further decorations to mark certain fields in sequence as optional, or optional with default values. For example, in a TBSCertificate the Version field may be omitted in binary format, which must be interpreted as the constant v1 (a value in scope), and any of the last three fields may also be omitted. This complicates parsing, and motivates the use of IMPLICIT and EXPLICIT identifiers to prevent any ambiguity. For example, when parsing the optional field Uid, if the next byte encodes the identifier for [1] IMPLICIT, then the content must be a Uid, but if it encodes the identifier for [3] EXPLICIT, then both Uid fields are absent, and one should start parsing the extensions. (Binary encodings of Extensions may start with any identifier, hence the need to wrap them.)

To ensure that a declaration can be unambiguously parsed there are various well-formedness conditions, e.g. all the fields in a consecutive block of OPTIONAL and DEFAULT fields, and the plain field that immediately follows them (if any) must have distinct identifiers. As such, not every syntactic instance of an ASN.1 declaration is admissible.

3 ASN1*

Figure 3 summarizes our formalization of ASN.1. In §3.1, we present an intrinsically typed syntax for ASN.1, whose typing constraints ensure the well-formedness of ASN.1 declarations. We offer some syntactic conveniences to help transcribe ASN.1 concrete syntax into our formal ASN1* notation, though the correspondence is only established empirically. In §3.2, we show that every well-formed ASN1* term can be denoted as an F* type. This part of our semantics is independent of the binary format, in keeping with the ASN.1 view that the type declarations and binary representations are to be decoupled. §3.3 contains the main formal result of the paper, namely that every ASN1* term has a denotation as a non-malleable parser for values of the type denotation. Our parser semantics yields OCaml code for parsing ASN.1 DER formatted data, and in §4 we test our code against concrete ASN.1 DER binary formatted data to confirm empirically that our semantics is faithful to the ASN.1 DER standard.

3.1 Syntax and Well-formedness of ASN.1

Figure 4 shows the formal syntax and well-formedness rules of ASN1*. We omit the definition of terminal_k, the language of terminal types, and their interpretation as F* types, terminal_t : terminal_k $\rightarrow$ Type. The content type is the core syntax of taggable content, while the declaration type associates an identifier with a content term—we leave the length out of the specification, since it is a dynamically computed value. The d_declaration type associates a decoration with an declaration value, and decorated and decorateds are just abbreviations. For compactness, we adopt a convention where free names are universally bound as implicit parameters at the top of the type of each constructor.

3.1.1 Identifiers. The type id_t below models identifiers, explained in §2. For example, the identifier [2] IMPLICIT encoded as byte 10 0 00010 has class CONTEXT_SPECIFIC, flag PRIMITIVE, and value 2. We bound identifier values to 32 bits, though we could have also chosen to use unbounded integers in F*—identifiers longer 32 bits are very uncommon.

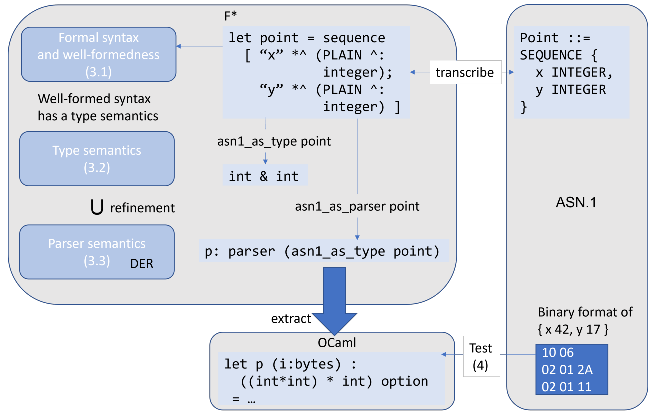

Figure 3. Architecture of our development

```
type id_class_t = | UNIVERSAL | APPLICATION | PRIVATE
                | CONTEXT_SPECIFIC
type id_flag_t = | PRIMITIVE | CONSTRUCTED
type id_t = {class:id_class_t; flag:id_flag_t; value:U32.t}
```

3.1.2 The content type. The TERMINAL constructor supports a form of decidable refinement. For example, to represent the type of natural numbers less than 4, one can write TERMINAL INTEGER ($\lambda v \rightarrow 0 \leq v$ && $v < 4$). While this is not strictly part of ASN.1, many common specifications express side conditions in natural language, so we include them in our formal language. SEQUENCE, SEQUENCE_OF, and SET_OF are just like their informal analogs in Figure 1.

PREFIXED models the wrapping of data types using EXPLICIT, e.g., ILC id (PREFIXED t) require that the inner type be wrapped with identifier id.

ANY_DEFINED_BY is the most complex content type. For example, the X.509 specification has a type for (mathematical) fields of characteristic two for some elliptic curves, given below in ASN.1 concrete syntax.

```
Characteristic-two ::= SEQUENCE {
  m INTEGER, - Field size 2^m
  basis OBJECT IDENTIFIER,
  parameters ANY DEFINED BY basis }
```

This declares a record of an integer m, followed by an object identifier basis, and then some parameters whose legal values are determined by the value of basis. The specification also includes (in natural language text) the basis/parameters pairs that are supported. In the constructor ANY_DEFINED_BY, the prefix represents fields (such as m) that precede the keys and values. The fields id and key are the identifier and type of the keys, which must be a terminal type (such as OBJECT IDENTIFIER). The field kvs represents the supported key-value pairs. The field def is an optional default value, which some specifications use to represent a default case not included in kvs. The final field is a proof obligation to

```
type decorator = | PLAIN | OPTION | DEFAULT
type content : Type =
| TERMINAL :
        k:terminal_k →
        is_valid:(terminal_t k → bool) →
        content
| SEQUENCE : decorateds → content
| SEQUENCE_OF : declaration s → content
| SET_OF : declaration s → content
| PREFIXED : declaration s → content
| ANY_DEFINED_BY :
        prefix:list decorated →
        id:id_t → key:terminal_k →
        kvs:list (terminal_t key & decorateds) →
        def:option decorateds →
        squash (wf_any prefix id kvs) →
        content

and declaration : set id_t → Type =
| ILC : id:id_t → content → declaration (singleton id)
| CHOICE_ILC :
    choices:list (id_t & content) →
    squash (no_repeats (map fst choices)) →
    declaration (as_set (map fst choices))
| ANY_ILC : declaration (complement empty)

and d_declaration : set id_t → decorator → Type =
| PLAIN_ILC : k:declaration s → d_declaration s PLAIN
| OPTION_ILC : k:declaration s → d_declaration s OPTION
| DEFAULT_TERMINAL :
    id:id_t →
    is_valid:(terminal_t k → bool) →
    defaultv:terminal_t k →
    squash (is_valid defaultv) →
    d_declaration (singleton id) DEFAULT

and decorated = s:set id_t & d:decorator & d_declaration s d
and decorateds = items : list decorated &
    squash (sequence_k_wf (map proj12 items))
```

Figure 4. Formal syntax and well-formedness
exclude repeats in the kvs list and its encodings. (squash p is the F★ type of proof-irrelevant proofs of p.)

Although ASN.1 includes a SET constructor, ASN1★ does not support it. Much like SEQUENCE, SET is used to declare a record, but with the intent that the ordering of its fields is unimportant. This is at odds with DER, which requires that binary representations of elements of SET and SET_OF be strictly sorted. We decided to fully support SET_OF but to ignore SET, since it does not occur in any of our case studies; it can usually be replaced with a SEQUENCE with the same fields and a simpler format; and it would require parsers for corner cases such as

```
SET { [2] IMPLICIT INTEGER,
      CHOICE { [1] IMPLICIT INTEGER,
               [3] IMPLICIT INTEGER }}
```

which declares a pair of integers, but insists their binary format order them by tags: either 1,2 or 2,3. (By contrast, SET OF declares sets where all elements have the same type, so we check their representations are strictly ordered but need not consider re-orderings.)

3.1.3 The declaration type.

The declaration s type associates an identifier with a content type, where the index s represents the set of valid first identifiers that may be encountered in the binary format of the type—this is used below in the well-formedness of decorated types. The CHOICE_ILC is for a sum and associates a distinct identifier with every content type in the sum. Finally, the ANY_ILC is used to represent any identifier-length-content tuple.

3.1.4 The decorated type.

The type d_declaration associates a decoration with an declaration type. The DEFAULT case supports refined terminals and requires a proof that the default value satisfies the refinement. Rather than using d_declaration, we use its packaged variants decorated and decorateds. The latter type enforces that all the fields in a consecutive block of OPTION and DEFAULT fields, and the PLAIN field that immediately follows them (if any) have distinct identifiers.

3.1.5 Smart constructors.

Writing a value of type declaration directly from its constructors can be tedious, especially due to the proof obligations on several of the constructors. To assist with this, we introduce a layer of smart constructors that internalize some of the proof obligations and provide tactics for them. These constructors enable writing specifications in our embedded declaration language in a style relatively close to the concrete ASN.1 syntax, while also formally capturing constraints that are typically left to natural language in concrete specifications. For example, we give below the specification in ASN1★ of the `Characteristic-two` declaration presented earlier, with an asn1_integer m as prefix, followed by the key name basis, and a choice between the three legal key-value pairs—the proof obligations are dispatched by seq_tac and choice_tac, tactics we developed for ASN1★.

```
let characteristic_two = asn1_any_oid_prefix
    ["m" *^ (PLAIN ^: asn1_integer)]
    "basis"
    [(gnBasis_oid, gnBasis_parameters);
     (tpBasis_oid, tpBasis_parameters);
     (ppBasis_oid, ppBasis_parameters)]
    (_ by (seq_tac())) (_ by (choice_tac()))
```

In the future, we may leverage user-defined syntax extensions proposed for F★ to streamline this further.

3.2 Denoting ASN1★ Declarations as F★ Types

```
let rec content_t (k:content) : Type = match k with
  | TERMINAL t is_valid → x:terminal_t t { is_valid x }
  | SEQUENCE ds → decorateds_t ds
  | SEQUENCE_OF k → list (asn1_as_type k)
  | SET_OF k → list (asn1_as_type k)
  | PREFIXED k → asn1_as_type k
  | ANY_DEFINED_BY prefix _ _ kv def _ →
    sequence_t prefix (choice_t (any_t kv) (def_t def))

and asn1_as_type (k:declaration s) : Tot Type (decreases k) =
  match k with
  | ILC id k → content_t k
  | CHOICE_ILC lc _ → choice_t (cases_t lc) ⊥
  | ANY_ILC → id_t & octetstring_t

and decorated_t (d:decorated) : Type =
  let (| _, _, dk |) = d in
  match dk with
  | PLAIN_ILC k → asn1_as_type k
  | OPTION_ILC k → option (asn1_as_type k)
  | DEFAULT_TERMINAL id is_valid defv → default_tv defv

and decorateds_t (| l, _|) = sequence_t l unit

and def_t d = match d with
  | None → ⊥
  | Some ds → decorateds_t ds

and any_t (ls:list (t & decorateds)) : Tot _ (decreases ls) =
  match ls with
  | [] → []
  | (x, ds) :: tl → (x, decorateds_t ds) :: any_t tl

and choice_t (lc:list (key & Type)) (def:Type) = k:key & assoc k lc def

and cases_t (lc:list (id_t & content)) : list (id_t & Type) =
  match lc with
  | [] → []
  | (x,y) :: t → (x, content_t y) :: cases_t t

and sequence_t (items:list decorated) (suffix_t:Type) : Type =
  match items with
  | [] → suffix_t
  | hd :: tl → decorated_t hd & sequence_t tl suffix_t
```

Figure 5. Denoting ASN.1 definitions as F⋆ types

Figure 5 shows our interpretation of ASN1⋆ syntax as F⋆ types, following the structure of the inductive type definitions in Figure 4. In the spirit of ASN.1, this first denotational semantics is independent of the binary representation.

3.2.1 Denoting content.

TERMINAL t v is interpreted as an F⋆ refinement of the denotation of t. SEQUENCE ds is interpreted as an n-ary tuple, where n is the length of ds, followed by a trailing unit (left here for simplicity, but optimized away in our implementation). SEQUENCE_OF and SET_OF are both denoted as lists. In principle, the latter could be quotiented by a relation that equates lists up to permutation, though F⋆ lacks native support for quotient types. PREFIXED only affects the binary format and has no effect on the type denotation. ANY_DEFINED_BY is represented as a tuple beginning with prefix followed by a sum defined by the kv association-list, with an optional default case.

3.2.2 Denoting declaration.

In an ILC id k, the identifier id concerns only the binary format. The CHOICE_ILC lc case maps the content_t interpretation over the list of cases, and then forms a (strong) sum type, *aka* a dependent pair, where the type is uninhabited in the case of an unexpected identifier. Finally, ANY_ILC is just a pair of an identifier and a string of bytes. Although we could have written helper functions like any_t and cases_t using combinators like map, F⋆'s termination checking rules make it much easier to write explicit, mutually recursive definitions in place. Additionally, the termination checker needs a couple of hints in the form of decreases annotations to accept this definition.

3.2.3 Denoting decorated types.

The denotation of decorated types is straightforward, with PLAIN having no impact; OPTION denoted as an option; and DEFAULT_TERMINAL denoted as default_tv defaultv, a refined form of option with constructors Default and Nondefault of (v:_ { v ≠ defaultv }).

3.2.4 Terminals.

Our semantics of terminals formally capture the properties of many ASN.1 types previously described only in natural language. We omit the details, and only discuss the UTF8String terminal, loosely defined in the standard as any byte string tagged with a special identifier, followed by 13 pages of English text for the actual specification. The intended usage is to first parse its contents for as a byte sequence, then to separately check that it is a valid UTF8String. Instead, we encode those constraints directly with F⋆ propositions and inductive types, and we prove that our parser, described next, only accepts values of this more precise type.

3.3 A Constructive Formalization of DER

The main formal result of this paper, summarized in this section, is that every ASN1⋆ type definition t : declaration s can be interpreted as a parser asn1_as_parser t of a byte sequence representation of asn1_as_type t. The specific format accepted by our parsers is intended to represent ASN.1 DER. We prove that the parser asn1_as_parser t is *injective*, i.e., for every v:asn1_as_type t there exists at most one valid binary representation.[3] We conclude that ASN.1 DER is a non-malleable format.

[3]The converse property, that every v:asn1_as_type t has at least one valid binary representation is not guaranteed by our proofs, though we test the non-triviality of the generated parsers empirically.

Injectivity of parsers is a *relational property* or a *hyperproperty* [10]. Proofs of hyperproperties are known to be challenging, with many special- and general-purpose logics proposed for various classes of hyperproperties [2–4, 16]. For the specific scenario of proving injectivity properties of parsers, the EverParse [25] library offers a family of injective-by-construction parser combinators. The library is structured around a type called parser k t, outlined below.

```
let parser (k:parser_kind) (t:Type) =
  p:(b:bytes → option (t & n:nat { n ≤ length b }) {
    has_kind k p ∧
    (∀ b₀ b₁. match p b₀, p b₁ with
       | Some (v₀, l₀), Some (v₁, l₁) →
         v₀ == v₁ ⟹ slice b₀ l₀ = slice b₁ l₁
       | _ → ⊤) }
```

In addition to injectivity, EverParse provides a language of *parser kinds* that characterize various other properties. For our purposes, we are interested in only two parser kinds, strong and weak, where strong parsers are insensitive to input extension. That is, appending any bytes to the input does not change the return value of a strong parser. We write weak_parser and strong_parser instead of parser weak and parser strong. Kinds are combined according to a small algebra, but we refer the reader to prior work on EverParse for the details.

EverParse provides several basic parsers and combinators to compose parsers, e.g., parse_u8 to parse a single byte, or nondep_then to parse two values in sequence while returning them as a pair. The type of combinators like nondep_then encodes a proof rule which ensures that the sequential composition of injective parsers is injective.

```
val parse_u8 : parser u8_kind U8.t
val nondep_then (p₀:parser k₀ t₀) (p₁:parser k₀ t₁)
  : parser (and_then_kind k₀ k₁) (t₀ & t₁)
```

In giving a parser denotation to ASN.1, the main challenge was to define a compositional semantics so that both their type-correctness (that they parse well-typed values according to the type denotation) and their injectivity follow structurally. In the process, we also extended EverParse with new general-purpose, injective-by-construction parser combinators, notably a combinator parameterized by a state machine, which should be of interest and applicability beyond the context of ASN.1 and DER.

3.3.1 Main theorem.

Figure 6 shows a few selected pieces from the parser denotation of ASN1★. The type of asn1_as_parser (reproduced below for clarity) is our main theorem: every ASN1★ declaration k:declaration s can be interpreted as a strong injective-by-construction parser returning a value of type asn1_as_type k, the type denotation of ASN1★. Since a parser is a total function, this proof is also constructive in the sense that it yields executable code for a parser for any ASN1★ type definition.

```
1  let dlc_parser t = lc:(id_t → strong_parser t) {cases_injective lc}
2  let twin_t t = strong_parser t & dlc_parser t
3  type twin = { d: decorated; ps:twin_t (undec_d_t d) }
4  let twins ds = lp : list twin_d{map (λ x → x.d) lp == ds}
5
6  let rec content_as_parser (k:content) : weak_parser (content_t k) =
7    match k with
8    | TERMINAL k v → weaken ((terminal_as_parser k) `filter` v)
9    | SEQUENCE (| ds, _ |) → mk_seq_parser (seq_as_twins ds)
10   . . .
11 and asn1_as_parser (k : declaration s) : strong_parser (asn1_as_type k) =
12   match k with
13   | ILC id k' → parse_ILC id (content_as_parser k')
14   . . .
15 and seq_as_twins (ds : decorateds) : twins ds
16   match ds with
17   | [] → []
18   | hd :: tl → decorated_as_twin hd :: seq_as_twin tl
19   . . .
20 and decorated_as_twin (d:decorated) : (tw:twin {tw.d == d}) =
21   let (| _, _, dk |) = d in
22   match dk with
23   | PLAIN_ILC k | OPTION_ILC k → { d; ps=asn1_as_twin k }
24   | . . .
25 and asn1_as_twin (k : declaration s) : twin_t (asn1_as_type k) =
26   match k with
27   | ILC id k' →
28     let p = content_as_parser k' in
29     ilc_twin_case_injective id p; (* lemma *)
30     parse_ILC id p, parse_ILC_twin id p
31   | CHOICE_ILC lc pf →
32     let lp = cases_as_parser lc in
33     choice_twin_cases_injective lc pf k lp; (* lemma *)
34     make_choice_parser lc pf k lp,
35     make_choice_parser_twin lc pf k lp
36   . . .
```

Figure 6. The parser denotation of ASN1★ (fragments)

```
val asn1_as_parser (#s:set id_t) (k : declaration s) :
  parser strong (asn1_as_type k)
```

The proof of this theorem is the bulk of our development, comprising about 6,000 lines of F★ code. Next, we summarize a few of the main ideas behind the proof.

3.3.2 Content, LC, and ILC Parsers.

At the top-level of our semantics (Figure 6 line 6) content_as_parser interprets a k:content as a weak_parser (content_t k). A bare content parser is not a strong parser—for example, a sequence parser would accept additional elements appended at the end of its input—but it can be strengthened by first parsing a length and then requiring that the content consume exactly the specified number of bytes. We thus define strong length-content (LC) parsers, using length field parsers and a combinator that invokes the content parser on the input byte sequence truncated to a specific length. We obtain an ILC parsers (line 13) by first parsing a leading identifier. The identifier parser itself involves a non-trivial, automata-like logic; it is based on a combinator described in §3.4.

For a given ASN.1 declaration, some identifiers may be fully determined by the context, and may thus be omitted. Some more compact ASN.1 encodings, e.g., the Packed Encoding Rules, include optimizations to eliminate redundant identifiers, but they are not widely adopted due to their increased complexity and marginal benefits. The DER does not include such optimizations.

3.3.3 Sequence parsers. Sequences would be simple to parse if all their fields were always present, but this is not the case with fields decorated with OPTION or DEFAULT. More generally, the well-formedness constraints on SEQUENCE ensure that any consecutive block of omittable fields and the plain field (if any) that immediately follows must have distinct identifiers, so one can use the next identifier value to tell which field comes next and which ones should take their default value. However, this breaks the one-to-one correspondence between identifier and ILC tuple, hence a first challenge for parsing sequences is handling *dangling identifier*, that is, single identifiers that determine the values of multiple fields. A second challenge is to handle omittable suffixes, since, for example, an empty string is a valid encoding of a sequence whose fields are all optional or default.

3.3.4 Dependent LC and twin parsers. To tackle the resolution of dangling identifiers, we introduce an alternate form of LC-parsers that depend on a previously-parsed identifier. That is, a p:dlc_parser t (Figure 6 line 1) expects an identifier i and ensures that p i is a strong_parser t, while guaranteeing that p i is injective in i—different values of i must return parsers that accept different values. By decoupling the parsing of identifiers and the length-content, we can construct sequence parsers while accounting for optional and default fields. When a block of omittable fields is encountered, our sequence combinator first parses an identifier and tries to match it against the set of identifiers for each field. If the identifier matches, the dlc_parser for the (undecorated) field is invoked, using the identifier that was just parsed. If the identifier does not match, the omittable field is filled with the default value and the dangling identifier is passed to the next field. In some cases it is useful to interpret a decorated type (line 20) both as a standard ILC parser as well as a dlc_parser for its underlying undecorated form—we call these *twin* parsers (line 3).

3.3.5 Defaultable parsers. We solve the problem of omittable suffixes with a new parser combinator called defaultable, which overrides the behavior of an existing parser when an empty string is encountered by returning a pre-determined value. To maintain injectivity, it requires the underlying parser to never return the default value.

3.3.6 Choice parsers. As we've seen, the type denotation of a CHOICE_ILC is a dependent pair. As such, if two different cases have the same underlying type, they are still distinguishable, since the identifier of the cases differs. The well-formedness condition on ASN1⋆ definitions ensures that the identifiers for all the cases must be distinct. We implemented the ASN.1 choice combinator with a generic tagged union combinator provided by EverParse, which first reads the identifier value, then looks it up in the list of cases. Once a match is found, the parser for the corresponding element is invoked to handle the rest of the input.

3.3.7 Any-defined-by parsers. Although ANY_DEFINED_BY also roughly assembles a tagged union, it differs from CHOICE in that it uses an explicit field (usually an object identifier) in the context of a sequence instead of a tag. Furthermore, its payload is a list of decorated sequence fields, instead of a single piece of content. We implemented a generic parser for ANY_DEFINED_BY by combining the techniques we used for choice and sequence parsers. First, a potential prefix of decorated fields is parsed (which may leave a dangling identifier), then the key field is parsed, its value is compared to the list of known values and, if a match is found, the corresponding continuation is invoked, otherwise the fallback parser is invoked.

3.4 Automata-Based Parser Combinator

While EverParse offers a variety of generic parser combinators, building multi-step parsers with branches and loops can be burdensome because relational proofs of parser kinds and injectivity must be provided for the continuation of each step before the combinators can be assembled. We developed a new parser combinator for generic, automata-based parsers that simplifies the construction of such proofs, and used this combinator to build parsers for several terminal types, including, notably, UTF-8 code points, which we use to illustrate the design of our automata parser combinator.

The ASN.1 specification requires handling the UTF8STRING terminal type, which is a sequence of valid Unicode code points, up to 21-bit values, each encoded in UTF-8, which takes between one and four bytes (see Table 1). A code point may have more than one representation, by using more bytes than necessary and filling the highest bits with 0s. To maintain non-malleability, the standard thus requires that each code point be encoded with the minimal number of bytes.

It is natural to structure a parser for UTF-8 code points as an automaton that reads one byte at a time and, whenever it accepts a code point, emits its value as an integer in $0..2^{21}-1$. For this, it is convenient to maintain auxiliary state that keeps track of the bit prefix of the code point parsed, rather than encoding this memory in the states of the automaton itself—we refer to this auxiliary state as a "buffer".

Range	Byte 1	Byte 2	Byte 3	Byte 4
≤ U+007F	0xxxxxxx			
≤ U+07FF	110xxxxx	10xxxxxx		
≤ U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
≥ U+10000	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Table 1. Encoding Unicode points to UTF-8

Bit Pattern	Action
0xxxxxxx	Accept, return the byte value
10xxxxxx	Reject: invalid first byte
1100000x	Reject: not using minimum number of bytes
110xxxxx	Transit to S_1 with buffer xxxxx
11100000	Transit to S_2' for extra checks
1110xxxx	Transit to S_2 with buffer xxxx
11110000	Transit to S_3' for extra checks
11110xxx	Transit to S_3 with buffer xxx
11111xxx	Reject: invalid first byte

Table 2. Transition table for Init

For example, Table 2 gives the transitions from the initial state, depending on the value of the first byte. Similarly, the other states have different transitions depending on the byte they read. They all check that their input is of form 10xxxxxx, then S_1 adds bits to the buffer and returns the content; S_2 and S_3 add bits to the buffer; S_2' and S_3' check the encoding is minimal and initialize the buffer with the correct bits. The transitions to $S2'$ and $S3'$ mention extra checks needed to ensure the uniqueness of representations, e.g., the 2 byte encoding allows representing code points encoded in 8–11 bits, while 3 bytes must only be used to encode values that require 12–16 bits.

Our automata combinator supports defining parsers with a "control plane" and a "data plane." The control plane contains the states, the alphabet (a single byte in this case), and the conditions for rejecting, accepting, and transitioning for each state. The data plane describes the behavior of the buffer. For example, the control plane of Table 2 is captured by three functions below, whereas the data plane for UTF-8 uses bitwise operations to reassemble the code points.

```
let reject_init (ch : byte) : bool
= (0b10000000 ≤ ch && ch ≤ 0b11000001) || 0b11111000 ≤ ch

let accept_init (ch : byte {reject_init ch = false}) : bool
= ch ≤ 0b01111111

let transit_init (ch:byte {reject_init ch = false && accept_init ch = false})
: state
= if (ch < 0b11100000) then S1
  else if (ch = 0b11100000) then S2'
  else if (ch < 0b11110000) then S2
  else if (ch = 0b11110000) then S3'
  else (* ch < 0b11111000 *) S3
```

Given the description of the automata and a parser for the alphabet (just a byte parser for UTF-8), the automata combinator assembles a parser that follows the specification of the state machine.

The main novelty is the way in which our combinator structures relational proofs of strong parser kinds and injectivity. The strong parser kind property directly follows from the byte parser having this property. Injectivity is proven by structural induction on the transitions of the automata. This induction is performed automatically by the automata combinator and reduces the goal to proving, for each state of the automata, the injectivity of the suffix that it parses. For UTF-8 code points, the initial state has three cases:

(1) *If the initial state accepts both bytes b_1 and b_2, and returns the same value, then $b_1 = b_2$.* This is trivial because the initial state returns the byte value.

(2) *If the initial state accepts b_1 but transits to another state on b_2, the final output will be different.* This holds because the initial state's return value is less than 2^7 while all other states eventually returns larger values (since they correctly reject over-long forms).

(3) *If the initial state transits to other states that return the same output values, then $b_1 = b_2$.* If the next states differ, then the return values differ because they have different number of bits. If the next states are the same, then the control bits in b_1 and b_2 are the same. The induction hypothesis that the suffix parser is injective shows the buffer contents must be the same, and thus $b_1 = b_2$.

Importantly, these proof goals are only propositions about the control and data plane, separate from the low-level parsing actions. In our implementation, all cases are automatically verified by the SMT solver backend of F⋆.

Our key insight of the automata combinator is the monotonicity innate to multi-step parsers. Each step parses some prefix of the input and "consumes" it such that the later steps can no longer depend on those bytes directly, but only through the control state and the partial output buffer. A necessary condition for injectivity is that each step must preserve enough information about the prefix parsed so far which also implies the information encoded in the control state and the output buffer must grow monotonically. This is what enables the use of structural induction and to decompose the goal into smaller goals about individual states. The manual proofs for each state verifies that the amount of information added in each step is equivalent to that in the prefix consumed.

4 Experimental Evaluation

We experimentally evaluate the precision and completeness of our model by writing in ASN1⋆ some of the most commonly used ASN.1 formats, and by executing our formally-verified parsers on large corpuses of inputs collected from real world internet usage, as well as synthetic invalid inputs created for security testing via systematic fuzzing.

The parsers we use in the experiments are extracted from the specification-level parsers derived from ASN.1 declarations (with `as_parser`) using the OCaml backend of F⋆, and thus, they are much less efficient than low-level in-place C validators available for some other combinators in the EverParse library. We leave the extraction of optimized C code to future work. All experiments are conducted on an Apple Macbook laptop from 2021.

4.1 X.509 Certificates

A major use case of ASN.1 from its conception is to represent cryptographic identities and credentials for internet communication. Like ASN.1, X.509 is a standard created by the International Telecommunication Union (ITU) in 1988 and used to this day to encode digital certificates, which associate entities to public keys and capture trust relations. X.509 certificates are critical to internet security: most websites, and many individuals, are issued certificates to authenticate themselves, for instance when creating a secure HTTP connection (indicated by a padlock icon in many browsers). There are certificate transparency logs that record the issuance of new certificates; at the time of writing (2022), they collect an average of 5 million new entries every day. Moreover, there is a long history of vulnerabilities in ASN.1 parsers causing major exploits in X.509 validation library. Surprisingly, although the format of certificates has not significantly evolved in the past 30 years, new vulnerabilities are routinely found in well-established ASN.1 parsers. For instance, looking at the history of documented attacks against OpenSSL, the most popular secure channel and cryptography library commonly used to validate certificates, new ASN.1 exploits[4] were found in 2003 (4 occurences), 2006, 2012, 2015 (6 occurences), 2016 (4 occurences), 2018 and 2021. Interestingly, the ASN.1 vulnerabilities are diverse: CVE-2021-3712 is a buffer overrun caused by functions wrongly assuming ASN.1-encoded strings are NULL-terminated (a problem similar to a famous exploit by Eliot Phillips at Black Hat 2009 that allows an attacker to impersonate any website using NULL bytes in the middle of domain names); CVE-2018-0739 results from recursive parsers causing stack overflows; CVE-2016-2108 is an interesting combination of vulnerabilities in the INTEGER parser (which can overflow when dealing with the incorrect negative encoding of 0) and the ASN.1 tag parser (which could misinterpret a large universal tag as a negative zero); CVE-2006-4339 is a famous attack by Bleichenbacher that relies on the ASN.1 parser accepting non-canonical serializations to forge RSA signatures. The same trend is observed when looking at MITRE's Common Vulnerabilities and Exposures (CVE) database, which lists ASN.1 vulnerabilities in the past 5 years in most operating systems (Linux, iOS, tvOS, macOS) and cryptographic libraries (OpenSSL, NSS, MatrixSSL, worlfSSL, RSA BSAFE, axTLS). Most are memory safety and functional correctness issues that could be prevented by formally verified parsers.

[4] https://www.openssl.org/news/vulnerabilities.html

Format Declaration. Figure 7 shows the top-level ASN1⋆ declaration for X.509 certificates, translated from the ASN.1 declaration in RFC 5280 shown in Figure 2. We make a few adaptations compared to the reference declaration; most notably, we try to capture data dependencies in a more precise way. The format of extensions and public keys depend on tags (typically object identifiers) whose possible values are not fully specified in the declaration (to leave the ability to define new ones in future revisions). For instance, extensions use an identifier to indicate their type, a boolean flag to indicate if the extension is critical, and an OCTET STRING that will contain the ASN.1 serialization of the extension payload, which depends on the extension type. An application is supposed to go over the list of extension, and further parse the payload using the right parser for this extension's type. If it encounters an extension with an unrecognized identifier, and the extension is marked critical, it must reject the certificate. It is useful to perform some of these application-level checks in the parser itself, thus limiting the chance that the checks are mishandled or omitted in the application. For example, we extend ANY DEFINED BY with a default definition, in case the identifier's value is not one of the specified ones. In this case, the fallback representation is the same as the generic definition, but requires the critical flag to be false:

```
(* Extension ::= SEQUENCE {
        extnID OBJECT IDENTIFIER,
        critical BOOLEAN DEFAULT FALSE,
        extnValue OCTET STRING *)
let extension_fallback = mk_gen_items [
    "critical" *^ (DEFAULT ^: critical_field_MUST_false);
    "extnValue" *^ (PLAIN ^: asn1_octetstring)]
    (_ by (seq_tac ()))
let extension = asn1_any_oid_with_fallback
    "extnId" supported_extensions extension_fallback
    (_ by (seq_tac ())) (_ by (choice_tac ()))
```

The altered definition parses all supported extensions in a single pass and guarantees critical unknown extensions are rejected during parsing. Overall, our X.509 module consists of 143 intermediate declarations in 608 lines of F⋆ code, and can be found in `ASN1.X509.fst`.

Datasets. To evaluate our X.509 module, we use one public dataset from the Electronic Frontier Foundation (EFF) consisting of certificates collected from the wild by scanning the IPv4 address space, and a second synthetic dataset of certificates that have been systematically altered to introduce DER and ASN.1 violations, and is used as part of the OpenSSL build tests to check for regressions.

The EFF dataset was created as part of the SSL Observatory effort in August 2010 by trying to initiate a TLS handshake with all reachable IPv4 addresses on port 443 (typically used

```
let x509_TBSCertificate= asn1_sequence [
  "version" *^ (PLAIN ^: (mk_prefixed (mk_custom_id
          CONTEXT_SPECIFIC CONSTRUCTED 0) version));
  "serialNumber" *^ (PLAIN ^: certificateSerialNumber);
  "signature" *^ (PLAIN ^: algorithmIdentifier);
  "issuer" *^ (PLAIN ^: name);
  "validity" *^ (PLAIN ^: validity);
  "subject" *^ (PLAIN ^: name);
  "subjectPublicKeyInfo" *^ (PLAIN ^: subjectPublicKeyInfo);
  "issuerUniqueID" *^ (OPTION ^: (mk_retagged
          (mk_custom_id CONTEXT_SPECIFIC PRIMITIVE 1) uId));
  "subjectUniqueID" *^ (OPTION ^: (mk_retagged
         (mk_custom_id CONTEXT_SPECIFIC PRIMITIVE 2) uId));
  "extensions" *^ (PLAIN ^: (mk_prefixed
          (mk_custom_id CONTEXT_SPECIFIC CONSTRUCTED 3)
          extensions))]
  (_ by (seq_tac ()))

let x509_certificate = asn1_sequence [
  "tbsCertificate" *^ (PLAIN ^: tBSCertificate);
  "signatureAlgorithm" *^ (PLAIN ^: algorithmIdentifier);
  "signatureValue" *^ (PLAIN ^: bitString)]
  (_ by (seq_tac ()))
```

Figure 7. Representing X.509 in ASN1*

Dataset	Total	Accept	Reject	Fail	Time
EFF	10138	9131	1007	0	198s
OpenSSL	2242	61	2181	0	30s
EFF CRL	4109	3388	703	18	68s
OpenSSL CRL	2063	15	2048	0	30s

Table 3. Results of running extracted X.509 and CRL parsers

for HTTPS), and capturing the collected certificate chains. The scan only captures objects that are at least recognized by OpenSSL at the time of processing as a certificate, which doesn't mean that it is valid or well-formed. Indeed, many of these certificates use undefined X.509 version numbers. The dataset is not labelled so we must manually inspect the rejected certificate to understand the cause of failure. Due to the large number of certificates in the dataset, we only carry this process on 10,138 of their X.509 version 3 certificates, (arbitrarily) selected by IP addresses that range from 108.0.100.238 to 109.95.49.5.

The OpenSSL dataset is used to check for regressions using `libfuzzer` each time the library is built. It cointains a corpus that captures all the known ASN.1 vulnerabilities found in previous versions, and many variants produced by fuzzing. By construction, all certificates in this dataset are invalid; however, in some cases the error doesn't appear during parsing but during signature validation instead. Since we only implement parsing, we do not detect errors introduced after RSA encryption, e.g. in the payload of signatures.

Analysis of results. The top part of Table 3 shows the results of running the X.509 module on the 2 datasets. For the EFF dataset, we manually inspect each of the 1007 rejected cert to determine what is the first error. We manage to attribute all failures to one of the following classes:

Default Field	Identifier	Terminal Type	Empty Sequence
710	196	65	36

Default field means that an optional field contains its default value, which is prohibited by the DER. This error appears either in the basic constraints extension, which is used to indicate if a certificate can sign other certificates or not, or in the parameters of RSA public key algorithm, which must be NULL. **Identifier** means the identifier don't match those stated in the standard. Again, these cases are often found inside an `ANY` structure. Issuer/subject fields of the certificate are prone to this kind of error. A typical case is that the standard requires a more restrictive string type, for instance the printable string, but the certificate uses a general one, for instance an ASCII string. **Terminal Type** is a class that includes all cases where a certain terminal type, such as boolean and integer, is not encoded correctly. A representative case is that of UTCTime, which require the letter Z to be used at the end of the representation to denote the Greenwich time instead of +/-0000 for non-malleability. For another example, a peculiar certificate encoded a very large integer but did not use least number of bytes for it. These kinds of errors are hard to detect for conventional parsers because they are niche cases for the implementation of a particular terminal parser while the tests are usually for the whole datatype. **Empty Sequence** occurs in certain sequence of structures that cannot be empty. We found this kind of error frequently shows up in extensions as well.

In summary, all 1007 rejected certificates are indeed invalid. Conversely, we cannot manually confirm the 9,131 accepted certificates are indeed valid. Instead, we rely on our results from the OpenSSL regression test. 97% of their certificates are indeed correctly rejected; we manually inspect each of the accepted certificates and confirmed that either the error only appears in the signature (which we cannot detect) or in an extension that we do not implement.

4.2 Certificate Revocation Lists

Format Declaration. Our CRL module consists of 8 declarations in 69 lines of F* and can be found in `ASN1.CRL.fst`.

Datasets. We did not find any large public corpus of revocations lists, so we wrote a script that extracts the URLs where the certification authority publishes their CRL from the "CRL distribution endpoint" certificate extension. We managed to collect 4,109 samples with this method.

The OpenSSL regression tests also includes tests for CRLs, which we use for negative testing. It contains 2,063 samples.

Analysis of results. The bottom part of Table 3 shows the results of running the CRL module on the 2 datasets. It is worth noting that CRLs can be much larger than certificates if a CA has revoked many certificates. This triggers a limitation in our OCaml extraction: since our byte buffers are modelled using F⋆ sequences, they extract to non-flat OCaml lists, which means some linear operations on flat buffers may be extracted to quadratic algorithms. Hence, in 18 cases we fail to execute our parsers. This can be fixed by using a flat memory representation for buffers, however extracting efficient OCaml parsers is not our goal and we would rather invest effort on extraction to C. The findings are very much aligned with the X.509 dataset: failures align with the 4 classes of errors in the EFF dataset. Similarly, the only OpenSSL samples that we do accept have errors in their signature or in extensions that we did not specify.

5 Related work and Conclusions

Formally verified parsers. While our work focuses on non-malleability of ASN.1 DER, formally verified parsers have covered various binary data formats and provided various properties on those formats and their implementations. Narcissus [12] is a library of parsing and serialization combinators verified in Coq and extracted to OCaml focused on the correctness of encoders with respect to decoders; it has been used to harden the network stack (TCP, UDP, IPv4, ARPv4, Ethernet) of the Mirage OS kernel [20]. Narcissus has also been used for Protocol Buffers [34]. EverParse [25] provides not only encoder correctness proofs, but also non-malleability, and extracts to C instead of OCaml, giving rise to efficient zero-copy C implementations proven memory safe and functionally correct with respect to the data format specifications. While EverParse was initially designed to support TLS handshake messages, our work is based on EverParse and extends it with ASN.1 parsing combinators with non-malleability proofs. Other extensions of EverParse such as EverParse3D for network virtualization packet formats [30] prove additional properties such as absence of double fetches to ensure secure efficient parsing on volatile input buffers where two reads from a given byte cannot be guaranteed to return the same value.

Formal studies of ASN.1. While ASN.1 predates many modern verification tools, there have been some early attempts to gain confidence in its security properties. Rinderknecht [26] proved properties of ASN.1 on paper such as non-malleability of a subset of “well-labeled” ASN.1 format descriptions, but without clearly relating this subset to DER. Conversely, Steckler [28] wrote an executable semantics of ASN.1 in Haskell but no associated formal proofs. DICE* [31] is an implementation of secure measured boot for IoT formally verified in F⋆ and extracted to C code to be run as part of the boot firmware of micro-controllers. As one of its main components, it includes a formal semantics of a small subset of ASN.1 used to create the unique certificate of a device. This subset cannot capture general purpose certificate as it lacks several important constructors such as CHOICE or ANY DEFINED BY. Tullsen et al. [32] formally verify C implementations of ASN.1 decoders and encoders for a vehicle-to-vehicle (V2V) messaging system, using the annotation-based SAW verification framework [13] turning annotated C programs into first-order formulae to be checked by SMT solvers. While their work provides both non-malleability and encoder correctness, their proofs focus on the C implementations for the purpose of the security of the enclosing V2V system, rather than a full formal specification of ASN.1 per se. In other words, they have not proven the functional correctness of their C encoders or decoders against any formal data format specification. Moreover, they do not support CHOICE. Pona and Zaliva [24] describe verification methodology challenges to verify an existing ASN.1 description compiler for C, ASN1C [33], by first formalizing the corresponding subset of ASN.1 in Coq, and then separately proving the functional correctness of ASN1C with respect to their specification using Appel’s Verified Software Toolchain [1]. However, we are not aware of any completed results from their effort yet.

Security of ASN.1 parsers. Because of the security-critical nature of the remaining applications of ASN.1 such as X.509 and the PKCS standards for encryption, signature, and wrapping, many techniques have been applied to find vulnerabilities in ASN.1 applications. Frankencert [5], Mucert [8] and Coveringcerts [18] are three domain-specific fuzzing tools to evaluate the security of real-world parsers and use various techniques to guarantee coverage and ensure that alterations pass through cryptographic integrity checks; general-purpose tools such as Nezha [23] and SAGE [15] have also been specialized for this purpose. Other papers such as Chen et al. [9] and Symcerts [7] attempt to detect non-compliance by discovering discrepancies between implementations, either by testing or by symbolic execution. Attacks that exploit the malleability of ASN.1 parsers to forge signatures have also been found in PGP [14], NSS [6], GnuTLS [22], Bouncy Castle [19], or even in the Nintendo 3DS boot ROM [27].

Conclusion. We have presented the first formalization of the semantics of ASN.1 and its Distinguished Encoding Rules, yielding parsers for binary formatted ASN.1 data that are type correct and non-malleable. Through testing, we have confidence that our formalized semantics matches the usage of ASN.1 in the wild, notably on X.509 certificates and certificate revocation lists. We aim to continue testing our semantics on more applications to further increase trust in our formalization. Additionally, we strive to use our semantics as a basis on which to build high-assurance cryptographic applications such as X.509 certificate chain validation.

References

[1] Andrew W. Appel. 2011. Verified Software Toolchain. In *Programming Languages and Systems*, Gilles Barthe (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–17.

[2] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2016. Product programs and relational program logics. *J. Log. Algebr. Meth. Program.* 85, 5 (2016), 847–859. https://doi.org/10.1016/j.jlamp.2016.05.004

[3] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. 2012. Probabilistic Relational Hoare Logics for Computer-Aided Security Proofs. In *11th International Conference on Mathematics of Program Construction (Lecture Notes in Computer Science, Vol. 7342)*. Springer, 1–6. http://hal.inria.fr/docs/00/76/58/64/PDF/main.pdf

[4] Nick Benton. 2004. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. ACM, New York, NY, USA, 14–25. https://doi.org/10.1145/964001.964003

[5] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. 2014. 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, 114–129. https://doi.org/10.1109/SP.2014.15

[6] Sze Yiu Chau. 2019. A Decade After Bleichenbacher'06, RSA Signature Forgery Still Works. *Black Hat USA* (2019).

[7] Sze Yiu Chau, Omar Chowdhury, Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. 2017. Symcerts: Practical symbolic execution for exposing noncompliance in X. 509 certificate validation implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 503–520.

[8] Yuting Chen and Zhendong Su. 2015. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 793–804.

[9] Yuting Chen and Zhendong Su. 2015. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 793–804.

[10] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (Sept. 2010), 1157–1210. https://www.cs.cornell.edu/fbs/publications/Hyperproperties.pdf

[11] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. 2008. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. https://www.ietf.org/rfc/rfc5280.txt

[12] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. 2019. Narcissus: correct-by-construction derivation of decoders and encoders from binary formats. *Proc. ACM Program. Lang.* 3, ICFP (2019), 82:1–82:29. https://doi.org/10.1145/3341686

[13] Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. 2016. Constructing Semantic Models of Programs with the Software Analysis Workbench. In *Verified Software. Theories, Tools, and Experiments*, Sandrine Blazy and Marsha Chechik (Eds.). Springer International Publishing, Cham, 56–72.

[14] Hal Finney. 2006. Bleichenbacher's RSA signature forgery based on implementation error. *http://imc.org/ietf-openpgp/mail-archive/msg14307.html* (2006).

[15] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012), 40–44. https://doi.org/10.1145/2093548.2093564

[16] Niklas Grimm, Kenji Maillard, Cédric Fournet, Cătălin Hriţcu, Matteo Maffei, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, and Santiago Zanella-Béguelin. 2018. A Monadic Framework for Relational Verification: Applied to Information Security, Program Equivalence, and Optimizations. In *The 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. https://arxiv.org/abs/1703.00055

[17] ITU. 2021. Abstract Syntax Notation One ASN.1: Specification of basic notation. https://www.itu.int/rec/t-rec-x.680/en

[18] Kristoffer Kleine and Dimitris E Simos. 2017. Coveringcerts: Combinatorial methods for X. 509 certificate testing. In *2017 IEEE International conference on software testing, verification and validation (ICST)*. IEEE, 69–79.

[19] Ulrich Kühn, Andrei Pyshkin, Erik Tews, and Ralf-Philipp Weinmann. 2008. Variants of Bleichenbacher's low-exponent attack on PKCS# 1 RSA signatures. *SICHERHEIT 2008* (2008).

[20] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 461–472. https://doi.org/10.1145/2451116.2451167

[21] Moxie Marlinspike. 2009. Null Prefix attack against TLS Server Certificates. https://nvd.nist.gov/vuln/detail/CVE-2009-2510

[22] Yutaka Oiwa, Kazukuni Kobara, and Hajime Watanabe. 2007. A new variant for an attack against RSA signature verification using parameter field. In *European Public Key Infrastructure Workshop*. Springer, 143–153.

[23] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. 2017. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on security and privacy (SP)*. IEEE, 615–632.

[24] Nika Pona and Vadim Zaliva. 2020. Research Report: Formally-Verified ASN.1 Protocol C-language Stack. In *2020 IEEE Security and Privacy Workshops (SPW)*. 308–317. https://doi.org/10.1109/SPW50608.2020.00065

[25] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(USENIX Security 2019)*. USENIX Association, USA, 1465–1482.

[26] Christian Rinderknecht. 1998. *Une formalisation d'ASN.1 - Application d'une méthode formelle à un langage de spécification télécom*. Ph. D. Dissertation.

[27] Michael Scire, Melissa Mears, Devon Maloney, Matthew Norman, Shaun Tux, and Phoebe Monroe. 2018. Attacking the Nintendo 3DS Boot ROMs. *arXiv preprint arXiv:1802.00359* (2018).

[28] Paul Steckler. 2007. A Formal Semantics for ASN.1 *(High-Confidence Software and Systems (HCSS))*.

[29] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. https://www.fstar-lang.org/papers/mumon/

[30] Nikhil Swamy, Tahina Ramananandro, Aseem Rastogi, Irina Spiridonova, Haobin Ni, Dmitry Malloy, Juan Vazquez, Michael Tang, Omar Cardona, and Arti Gupta. 2022. Hardening Attack Surfaces with Formally Proven Binary Format Parsers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 31–45. https://doi.org/10.1145/3519939.3523708

[31] Zhe Tao, Aseem Rastogi, Naman Gupta, Kapil Vaswani, and Aditya V. Thakur. 2021. DICE*: A Formally Verified Implementation of DICE Measured Boot. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1091–1107. https://www.usenix.org/conference/usenixsecurity21/presentation/tao

[32] Mark Tullsen, Lee Pike, Nathan Collins, and Aaron Tomb. 2018. Formal Verification of a Vehicle-to-Vehicle (V2V) Messaging System. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 413–429.

[33] Lev Walkin. 2003–2021. asn1c. https://github.com/vlm/asn1c/.

[34] Qianchuan Ye and Benjamin Delaware. 2019. A Verified Protocol Buffer Compiler. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP 2019)*. Association for Computing Machinery, New York, NY, USA, 222–233. https://doi.org/10.1145/3293880.3294105

A Background on F⋆ and EverParse

F⋆ is a programming language and proof assistant based on a dependent type theory (like Coq, Agda, or Lean). F⋆ also offers an effect system, extensible with user-defined effects, and makes use of SMT solving to automate some proofs. F⋆ syntax is roughly modeled on OCaml (val, let, match etc.) with differences to account for the additional typing features. Binding occurrences b of variables take the form x:t, declaring a variable x at type t; or #x:t indicating that the binding is for an implicit argument. The syntax $\lambda(b_1) \dots (b_n) \rightarrow t$ introduces a lambda abstraction, whereas $b_1 \rightarrow \dots \rightarrow b_n \rightarrow c$ is the shape of a curried function type. Refinement types are written b{t}, e.g., x:int{x≥ 0} is the type of non-negative integers (i.e., nat). As usual, a bound variable is in scope to the right of its binding; we omit the type in a binding when it can be inferred; and for non-dependent function types, we omit the variable name. The c to the right of an arrow is a *computation type*. An example of a computation type is Tot bool, the type of total computations returning a boolean. By default, function arrows have Tot co-domains, so, rather than decorating the right-hand side of every arrow with a Tot, the type of, say, the pure append function on vectors can be written #a:Type → #m:nat → #n:nat → vec a m → vec a n→ vec a (m + n), with the two explicit arguments and the return type depending on the three implicit arguments marked with '#'. We often omit implicit binders and treat all unbound names as implicitly bound at the top, e.g., vec a m → vec a n → vec a (m + n)

F⋆ programs are not executable per se. Instead, F⋆ extracts OCaml code from F⋆ code. To this end, F⋆ distinguishes between pure computations, which extract to OCaml, and *ghost* computations for proof purposes only (where use of axioms such as excluded middle or indefinite description is allowed), erased at extraction. (F⋆ also supports effectful code, and extraction to C via Low*, a fragment of F⋆ shallowly embedding a subset of C, but this is out of the scope of this paper.)

EverParse. EverParse is a formally verified library and toolchain to build verified parsers and serializers for binary data formats such as TLS or network virtualization protocols. Formal guarantees supported by EverParse include proofs of unique binary representation, a.k.a. non-malleability, for the purpose of secure authentication and hashing; proofs that serializer and parser are (partial) inverse of each other; bounds on the size of the byte representation. (EverParse also allows generating executable C code for such parsers, via the Low* fragment of F⋆, allowing some performance optimizations, for which EverParse proves memory safety, arithmetic safety, functional correctness with respect to the original parser specification.) To establish such guarantees, EverParse builds on its core component, called LowParse, a library of monadic parser and serializer combinators formally verified in F⋆. Such parser combinators supported by LowParse include dependent pairs (a.k.a. tagged unions), filter refinements, rewriters, lists, and data prefixed with its size in bytes. Such combinators were initially tailored to support formats such as TLS handshake messages. On top of LowParse, EverParse provides several front-ends: QuackyDucky [25] targeting TLS handshake messages, and 3D [30] targeting network virtualization packets. With those front-ends, EverParse allows users to define their data formats in a high-level descriptive language, and to push a button to automatically generate formally verified parser and serializer code for their formats, by assembling LowParse combinators, with zero user proof effort. Thus, EverParse as a toolchain is similar in spirit to recent efforts in automatic parser generation for binary data formats such as Protocol Buffers or Cap'n Proto, except that, contrary to EverParse, those two toolchains come with their own classes of supported data formats, excluding existing network protocol formats (one cannot, say, define the TLS handshake message formats in Protocol Buffers.) Moreover, EverParse distinguishes itself by generating formally verified code.