Proving a soundness property for the joint design of ASN.1 and the Basic Encoding Rules

Christian Rinderknecht
Groupe Léonard de Vinci
École Supérieure d'Ingénieurs Léonard de Vinci
Christian.Rinderknecht@devinci.fr

June 2004

Abstract

The Abstract Syntax Notation One (ASN.1) can be used to model types of values carried by signals in SDL or MSC but is also directly used by network protocol implementors. In the last few years, the press has reported several alleged vulnerabilities of ASN.1 and the Basic Encoding Rules (BER) related to network protocols like SNMP and, more recently, OpenSSL. In reality it has been shown that the security issues (theoritically denial of service attacks) were due to low-quality and poorly-tested compiler implementations. We use some formal methods to go further. We review formally the design of the BER themselves and prove that, under some assumptions, it is flawless whatever the network protocol is and whatever the values to be transmitted are. More precisely, we start with a formal modeling of the BER which abstracts away low-level details but captures the design principles. Then we define a soundness property stating that the composition of encoding and decoding yields a value which is equivalent to the original. Finally we prove that this property holds for all values specified with ASN.1.
Key words: Abstract Syntax Notation One, ASN.1, Basic Encoding Rules, BER, protocol, specification, vulnerabilities, formal methods.

1 Introduction

The wide variety of software and hardware architectures in distributed systems and telecommunications makes it valuable to use a common high-level data notation in protocol specifications. To fulfill this need, the ISO organization and the International Telecommunication Union

(ITU) defined the Abstract Syntax Notation One (ASN.1) series of standards. ASN.1 [2–6] is a language for data types allowing the protocol designer to capture numerous networking concepts, such as protocol data units, without worrying about the possible environment and implementation heterogeneity of the peers. The peers share a set of ASN.1 modules and agree upon a set of *encoding rules*, such as [7, 8], which is a method for encoding values produced at run-time by the communicating applications, into series of bits. ASN.1 has been adopted for a wide range of applications, such as network management, secure e-mail, mobile telephony, air traffic control etc.

In the last few years, the press has reported several alleged vulnerabilities of ASN.1 and the Basic Encoding Rules (BER) related to network protocols like SNMP and, more recently, OpenSSL. Each time, an accurate description of the problem has been finally published, showing that the weakness lay in *implementations* poorly written and insufficiently tested. The real vulnerabilities were almost all related to improper decoding of ill-formed BER encodings (or *codes*) causing buffer overflows, unspecified (non-deterministic) behaviours, stack corruptions and, in the end, a possible denial of service.

From now on, it is important to understand and remember that ASN.1 and the BER, intrinsically, have nothing to do with security or cryptographic protocols. Both are used for modeling and handling the data part of protocols, not the control. As a consequence, the soundness property we aim at in this article must not be considered as a security property about *control* but as mere correctness of composition of encoding and decoding with the BER of *values* specified by means of ASN.1. For instance, there are no attackers, no nonces etc. here. Nevertheless, the difficulty is not lesser.

More precisely, in this work we want to prove that the design of the BER themselves is flawless, whatever the network protocol is and whatever the values to be transmitted are. To achieve this goal we need the support of formal methods. We start by a formal modeling of the BER which abstracts away low-level details but captures the design principles. Then we define a soundness property representing the security warranty we require and finally we prove that this property holds for all values that can be specified with ASN.1.

2 Modeling

An ASN.1 compiler accepts a set of ASN.1 modules representing the *Protocol Data Unit* (PDU) and, according to a given set of encoding rules and a peer-specific target programming language, produces a set of data type definitions in that programming language, together with

a codec (encoder/decoder) for the values to be exchanged. Then these pieces of source code are compiled and linked separately against the communicating application. Let us make some remarks and assumptions.

- The peers share a set of ASN.1 modules and the assumption that the encoding rules are the BER. Without loss of generality, we can reduce the common knowledge to one module and even a unique ASN.1 type.
- In order to be independent from the application programming languages, we shall assume that both peers express directly their values in ASN.1 (in reality they are produced in memory at runtime).
- At this stage, it is important not to be drawn into too much details due to encoding and decoding series of bits. Instead, we chose to represent codes with a more abstract syntax than bits, which will allow us to easily reason by induction. That way we can convince ourselves that the underlying principles of the BER are sound. In a second stage we can study separately the encoding and decoding between our abstract codes and the transmitted bits.
- The standard document specifying the BER [7] says nothing about the decoding procedure except "It is implicit in the specification of these encoding rules that they are also used for decoding." We shall then explicitly propose a decoding from our abstract codes to ASN.1 (accordingly with the two previous assumptions).
- *The BER encodings may not be unique for a given value.* Indeed the BER allow the sender to choose independently from the receiver different encodings for a class of types. For instance, the encoding of the boolean value `TRUE` can be any non-zero octet [7, §8.2.2] and the encoding of a `SET` value imposes no order on the component encodings [7, NOTE in §8.11.3]. Mathematically, the BER define an application, not a function. (The restricted form of the BER, called the Canonical Encoding Rules (CER) and the Distinguished Encoding Rules (DER), are functions.) This leads us to require an *equivalence relationship between codes* which would be enough discriminative but would nevertheless make equivalent all the encodings of a value.

Proposition 1. *All the BER encodings of a given value, according to a given type, are equivalent.*

Network. We assume that the network transfer does not alter the codes, despite the publicised vulnerabilities mentioned in the introduction being due to possibly forged BER codes. We ignore this point

precisely because it has been shown that these vulnerabilities were due to non-robust decoders, and our aim here is to prove that the BER *themselves* are not flawed.

Well-formedness. The front-ends of the ASN.1 compilers must check that the type T and the value v are well-formed. These properties are intrinsic to ASN.1 and include, for the types, the uniqueness of names and tags of component types. For instance `T ::= CHOICE {a INTEGER, a REAL}` and `U ::= CHOICE {a INTEGER, b INTEGER}` are not well-formed. Indeed, the encoding of `t T ::= a : 0` would be ambiguous (i.e. non-deterministic) since `0` can denote either an `INTEGER` or a `REAL` value and `u U ::= a : 0` would make the decoding non-deterministic because the tags of fields `a` and `b` are identical (`INTEGER`'s tag). In both examples, there would be no way for the encoder or the decoder to solve the ambiguity.

Value equivalence and soundness. As we mentioned previously, the standard says little about the receiver's behaviour, but, since the BER embed all the tags in the codes, the uniqueness of tags is clearly intended to make the decoding a function, i.e. it returns always the same value on the same code. This is not stated explicitly in the standard and it is imaginable that the decoder sorts some decoded parts before passing the whole input to the application, but the standard seems to favour an asymmetric model in which the sender may spend some time reorganising the encoded data (i.e. not following strictly the order of the ASN.1 specification) and the receiver fastly decodes them as they arrive, without any subsequent processing. With the same asymmetrical focus, we believe that the receiver is the peer who is mostly concerned with security: the soundness property we propose consists in defining an *equivalence relationship between ASN.1 values* (therefore independent from the BER) and in stating that the decoded value is equivalent to the (unknown) one the sender emitted.

Theorem 1 (Soundness)**.** *Let v be a well-formed value of the well-formed type* T*. Then the BER decoding of* any *BER encoding of v is equivalent to v.*

Code equivalence. The figure 1 shows the model we described so far. We understand better now why it is important for the equivalence on codes to be enough discriminative: otherwise many codes would be equivalent despite their ASN.1 values not being related. As we said, the BER embed all the tags (collected from the type of the value) in the codes, so, if the type is well-formed, the codes would capture enough structure (from the type) to allow a rather natural and discriminative equivalence relationship to be defined. Moreover, the equivalence will not need the knowledge of the type to be decided (tags in the codes suffice). We already identified the need for an equivalence relation

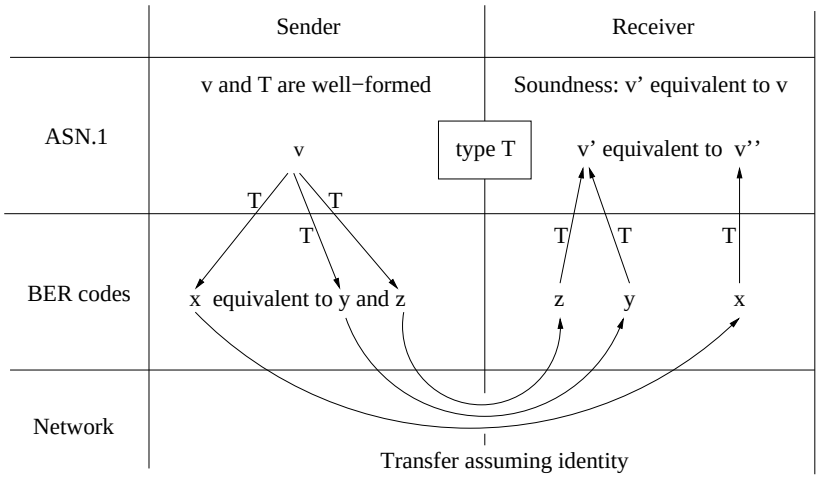

Figure 1: Soundness property with equivalence entailment

on ASN.1 values in order to express a soundness property, and since, according to our method, we define separately an equivalence relation on codes, we need the following property to be satisfied.

Proposition 2 (Equivalence entailment)**.** *Let c_1 and c_2 be two equivalent codes. Then the decoding of c_1 is equivalent to the decoding of c_2 assuming the same type.*

This way, we can maintain the soundness property despite the encoding procedure is not a function. In particular we suggest that the decoding is a non-injective function (decoding of two different codes can lead to the same value, *e.g.,* `TRUE`).

Typing. In figure 1 we annotate the arrows between the "ASN.1" and the "BER codes" layers with T to mean that a value is encoded following the type T or a code is decoded assuming the type T. The encoding and decoding of a value assumes that this value is of a given type. This does not imply that we need to formalise the typing relation independently, it actually means that part of the typing is embedded in the encoding and decoding relations. In other words, the encoding only does the type-checking needed to allow the decoding with the same typing assumption.

Subtyping. The BER do not take into account the subtyping constraints. Since these constraints restrict the set of values of a given type, the set of values considered by the BER is greater than the specified PDU. The Packed Encoding Rules [8] (PER) consider the subtyping constraints and define a notion of *PER-visibility* upon them.

This also amounts to making an approximation of the exact set of values. These behaviours are not a design flaw. Indeed, when the encoder receives a value from its application, it should first check whether this value fits the PDU and, if so, it would be encoded after. The decoder, on the other hand, when receiving a code, decodes it first, then checks whether the value fits the PDU and, if so, passes it to its application. Keep in mind also that the encoding rules try to minimise the length of the codes according to different strategies (contrast BER and PER), so they *must* approximate the data in order to find some regularities — as a cloud of points can be compactly approximated by its convex hull.

It is up to the ASN.1 compiler, not to the encoding rules themselves, to generate the code checking whether a value fits the PDU. The great expressiveness of the ASN.1 subtyping paradigm makes it very difficult to calculate the exact set of values of a subtype, even in particular to detect and reject empty PDUs [9]. However, the attacks mentioned earlier were based on forged BER codes which were not out of the PDU but merely ill-formed or took advantage of recursive types in order to overflow the receiver's stack. In any case, the decoders (generated by ASN.1 compilers) must be robust and the limits we just mentioned about determining the exact set of values of the PDU has more to do with ASN.1 modules validation rather than soundness of data transmission — at least until now. Thereupon, the BER can take into account the structural subtyping constraints (requiring a component to be `ABSENT`, `PRESENT` or to remain `OPTIONAL`).

Core ASN.1 Next, we note that *the BER only apply to a subset of ITU-T Rec. X.680* [3] (X.680 does not contain information objects, non-subtyping constraints and parameterization). For instance, the BER standard does neither consider `COMPONENTS OF` clauses in ASN.1 types nor selection types as well. The tagging policy (`EXPLICIT`, `IMPLICIT` or `AUTOMATIC`) is not considered either. Another example is `BIT STRING` values which are supposed not to be specified with named bits. All this suggests that the whole ASN.1 can be reduced to an inner subset which has the same expressivity, i.e. a sub-language which can express all what can be expressed with the whole language and nothing more. For the sake of brevity, in this paper we shall cope with X.680 and show that a simpler sub-language exists by giving a series of rewriting rules which preserves the set of values of a given type. In fact, it is even useful to reduce further our sub-ASN.1 (we call it *BER domain* in figure 2) into a smaller one that we call *core ASN.1.* The purpose is to get rid of some more syntactic constructs which are not fundamental, but are mere facilities, and thus to ease the formalisation and ensure some properties. One technical side effect is that the equivalence on values does not require the knowledge of their type, be-

cause *the values in core ASN.1 are not syntactically ambiguous* (*e.g.,* `0` is a value for both `REAL` and `INTEGER` types in full ASN.1, but in core ASN.1 it is only of `INTEGER` type — in the other case it is rewritten `0.0`). Another very interesting property is the following.

Theorem 2 (BER termination)**.** *The encoding of core ASN.1 values with the BER always terminates.*

The reason is that we detect and reject as illegal the infinite values, i.e. the recursive values, during the reduction phase. If we want to convince ourselves that the design of the BER is sound, we need to understand well ASN.1 and how to reduce it to a manageable kernel.

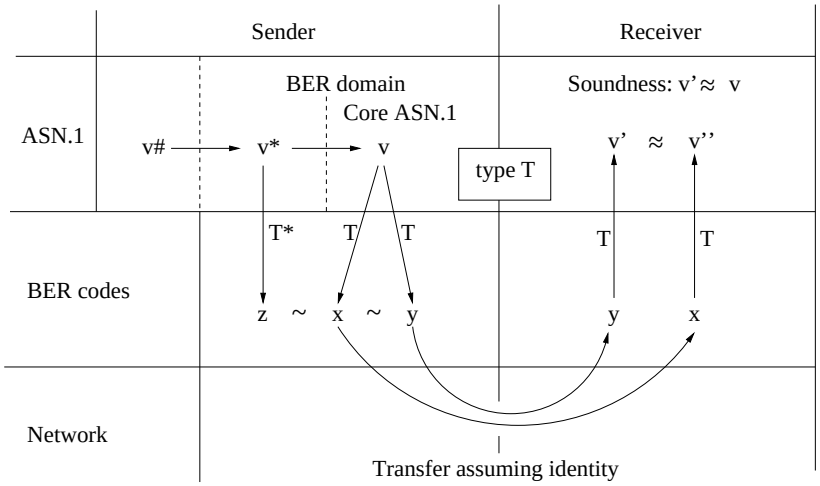

Figure 2: Core ASN.1 and soundness property

Figure 2 gives the final model we arrived at. We note now $v^{\#}$ and $\mathrm{T}^{\#}$ the values and types in X.680, v^{*} and T^{*} the values and types in the BER domain, and simply v and T when they are in core ASN.1. Let us note $v' \approx v$ the proposition "Value v' is equivalent to value v." Let us note $c' \sim c$ the proposition "Code c' is equivalent to code c." The figure 2 makes it clear that we need to guarantee that *all the encodings of* v^{*} *are equivalent to all the encodings of* v.

3 Core ASN.1

ASN.1 syntax is involved because it aims at allowing the specification of as many networking concepts as possible. For instance types, values and subtyping constraints may depend on each other: a type may contain constraints (on components) and values (*e.g.,* default values),

a value has a type and constraints rely upon types (*e.g.,* inclusion constraint) and values (*e.g.,* value constraint). We define core ASN.1 such that

- the default tagging mode of the module is `EXPLICIT TAGS`;
- tags obey the standard rules, like alternative types in `CHOICE` having distinct tags etc.;
- tags appear only at the top-level, i.e. just after the symbol '`::=`';
- the built-in types are explicitly tagged `IMPLICIT` and `UNIVERSAL`;
- tags are explicitly either `IMPLICIT` or `EXPLICIT`;
- `IMPLICIT` tags apply only to untagged types;
- tag values are numeric `INTEGER` values (not value references);
- component types are references and references are component types;
- there are no `DEFAULT` component types;
- there is no `COMPONENTS OF` clause;
- constraints appear only at the top-level (not in component types);
- there are no `ABSENT`, `PRESENT` or `OPTIONAL` component constraints;
- there is no selection type, *e.g.,* no `T ::= i < U`;
- `SET OF` and `SEQUENCE OF` apply to references;
- the `BIT STRING` and `INTEGER` type do not define constants, *e.g.,* no `INTEGER {c(1)}` or `BIT STRING {a(x)}`;
- the only `BIT STRING` values are series of bits, *e.g.,* `'1110'B`;
- `ENUMERATED` types define constants with explicit numeric integers;
- `REAL` values are not legal tokens for `INTEGER` values and conversely (*e.g.,* `0` is *only* of type `INTEGER`);
- `REAL` values do not use the *(mantissa, base, exponent)* form;
- there are no references in values (thus no recursive values).

We relax the first assumption we made in section 2 and assume now that we have one ASN.1 module, syntactically correct with respect to X.680. It is reduced to core ASN.1 by applying the following series of rewritings which do not commute in general. For we lack of room to give the formal rewriting rules, we only illustrate the process on short examples.

1. We remove the selection types, taking care of tags:

$$\left\{\begin{array}{l}\texttt{A ::= [0] i < [1] B}\\ \texttt{B ::= [2] C}\\ \texttt{C ::= [3] CHOICE\{i [4]D\}}\\ \texttt{D ::= [5] INTEGER}\end{array}\right. \rightarrow \left\{\begin{array}{l}\texttt{A ::= [0][4][5] INTEGER}\\ \texttt{B ::= [2] C}\\ \texttt{C ::= [3] CHOICE\{i [4]D\}}\\ \texttt{D ::= [5] INTEGER}\end{array}\right.$$

Note that the selection types that do not define a unique type lead to recursive type definitions whose pattern is `X ::= X`, as in

$$\begin{array}{l}\texttt{T ::= CHOICE \{a a < T\}}\\ \longrightarrow \left\{\begin{array}{l}\texttt{T ::= CHOICE \{a A\}}\\ \texttt{A ::= a < T}\end{array}\right. \longrightarrow \left\{\begin{array}{l}\texttt{T ::= CHOICE \{a A\}}\\ \texttt{A ::= A}\end{array}\right.\end{array}$$

2. The top-level type references are unfolded, i.e. the type references at the declaration level are replaced by the type they reference, as in

$$\left\{\begin{array}{l}\texttt{T ::= U (C)}\\ \texttt{U ::= REAL (D)}\end{array}\right. \longrightarrow \left\{\begin{array}{l}\texttt{T ::= REAL (D ^ C)}\\ \texttt{U ::= REAL (D)}\end{array}\right.$$

Beware of the case of constrained references to `SET OF` types:

$$\left\{\begin{array}{l}\texttt{A ::= SET OF C}\\ \texttt{B ::= A (SIZE (7))}\end{array}\right. \longrightarrow \left\{\begin{array}{l}\texttt{A ::= SET OF C}\\ \texttt{B ::= SET (SIZE (7)) OF C}\end{array}\right.$$

The result `B ::= SET OF C (SIZE (7))` would be wrong!

This step is difficult because it removes all recursive types declarations that do not lead to a uniquely defined type, like `T ::= T` or `T ::= CHOICE {a a < T}` etc. (See step 1.)

3. The default values are expanded and the `DEFAULT` annotation is replaced by `OPTIONAL`, like in the following example

$$\left\{\begin{array}{l}\texttt{v T ::= \{\}}\\ \texttt{T ::= SET \{a U DEFAULT w\}}\end{array}\right. \rightarrow \left\{\begin{array}{l}\texttt{v T ::= \{a w\}}\\ \texttt{T ::= SET \{a U OPTIONAL\}}\end{array}\right.$$

4. The `COMPONENTS OF` clauses are expanded:

$$\left\{\begin{array}{l}\texttt{T ::= SET \{COMPONENTS OF [6] A\}}\\ \texttt{A ::= SET \{a REAL\}}\end{array}\right. \rightarrow \left\{\begin{array}{l}\texttt{T ::= SET \{a REAL\}}\\ \texttt{A ::= SET \{a REAL\}}\end{array}\right.$$

If the tagging mode is `AUTOMATIC TAGS`, we must *previously* compute the current component tags and then insert the components referred by `COMPONENTS OF`.

```
{ PDU DEFINITIONS AUTOMATIC TAGS ::=
    A ::= SET {a SET OF B, COMPONENTS OF B}
    B ::= SET {b [2] INTEGER}
  END
```
$\longrightarrow$
```
{ PDU DEFINITIONS AUTOMATIC TAGS ::=
    A ::= SET {a [0] SET OF B, b [1][2] INTEGER}
    B ::= SET {b [2] INTEGER}
  END
```

5. `INTEGER` and `BIT STRING` constants are replaced by their definition and removed from their defining type:

```
{ T ::= INTEGER {c(x)}      ⟶ { T ::= INTEGER
{ v T ::= c                   { v T ::= x
```

   This step may reveal some recursive values, as in

```
{ T ::= INTEGER {c(v)}      ⟶ { T ::= INTEGER
{ v T ::= c                   { v T ::= v
```

6. For `BIT STRING` values which are specified by means of a series of bit names, we unfold their associated references and replace the value by an equivalent string of bits:

```
{ T ::= BIT STRING {msb(x),lsb(y)}  →
{ v T ::= {msb,lsb}
        { T ::= BIT STRING
        { v T ::= '10000001'B
```

   assuming the excerpt `x INTEGER ::= 7  y INTEGER ::= 0`
   Also, values in hexadecimal form are translated into binary form:

```
x U ::= 'A'H ⟶ x U ::= '1010'B
```

7. We unfold the value references, disallowing at the same time the recursive values, like `v T ::= {v}`
8. We unfold the `ENUMERATED` constants and add the missing integers:

```
{ T ::= ENUMERATED{a(v),b}  →
{ v INTEGER ::= 3
{ T ::= ENUMERATED{a(3),b(4)}
{ v INTEGER ::= 3
```

9. We unfold the tag values (this always terminates because there are no more recursive values since step 7), checking that they are syntactically integers:

$$\left\{\begin{array}{l}\texttt{T ::= [APPLICATION v] IMPLICIT REAL}\\ \texttt{v INTEGER ::= 3}\end{array}\right. \longrightarrow \left\{\begin{array}{l}\texttt{T ::= [APPLICATION 3] IMPLICIT REAL}\\ \texttt{v INTEGER ::= 3}\end{array}\right.$$

10. The tagging mode becomes `EXPLICIT TAGS`, like

$$\left\{\begin{array}{l}\texttt{PDU DEFINITIONS IMPLICIT TAGS ::=}\\ \texttt{\quad A ::= SET \{a [0] SET OF B\}}\\ \texttt{\quad B ::= [1] CHOICE \{b [2] REAL\}}\\ \texttt{END}\end{array}\right. \longrightarrow$$

$$\left\{\begin{array}{l}\texttt{PDU DEFINITIONS EXPLICIT TAGS ::=}\\ \texttt{\quad A ::= SET \{a [0] IMPLICIT SET OF B\}}\\ \texttt{\quad B ::= [1] EXPLICIT CHOICE \{b [2] IMPLICIT REAL\}}\\ \texttt{END}\end{array}\right.$$

11. We make explicit the tags of the built-in types:

`A ::= INTEGER` $\longrightarrow$ `A ::= [UNIVERSAL 2] IMPLICIT INTEGER`

12. We reduce the `IMPLICIT` tags, as

```
T ::= [0] IMPLICIT [1] EXPLICIT [UNIVERSAL 9]
  IMPLICIT REAL ⟶ T ::= [0] IMPLICIT REAL
```

13. We apply and reduce the structural subtyping constraints `ABSENT`, `PRESENT` and `OPTIONAL`, like

```
T ::= CHOICE {a REAL, b REAL}
             (WITH COMPONENTS {a(PRESENT)})
  ⟶ T ::= CHOICE {a REAL}
```

(General case complex but tractable.)

It is important to understand that in core ASN.1 it is still possible that

- types have only infinite values: `T ::= SET {a T}`
- values are ill-typed: `v REAL ::= ""`
- values do not conform to all additional X.680 requirements, like

$$\left\{\begin{array}{l}\texttt{T ::= SEQUENCE \{a BOOLEAN, b INTEGER\}}\\ \texttt{t T ::= \{b 7, a TRUE\}\quad -- illegal}\end{array}\right.$$

- subtyping constraints are inconsistent: `T ::= REAL (SIZE(7))`
- subtypes are empty:
  `T ::= SET ((SIZE(1))^(SIZE(2))) OF REAL`
- subtypes have no value set: `T ::= REAL (ALL EXCEPT T)`

The reason why this is not a problem is that core ASN.1 has been defined with the BER modeling in mind, in particular we do not aim here at a full validation of ASN.1.

Abstract grammar. We formally define the constructs of core ASN.1 by means of an *abstract grammar* implemented with the algebraic data types of the functional programming language OCaml [1], which is a full-fledged programming language, as well as, historically, a logic meta-language. The core ASN.1 parser output is a pair of a type environment and a value environment. The former is a mapping from type names to types, corresponding to the type declarations in the ASN.1 specification, and the latter is a mapping from value names to values, corresponding to the value declarations. The types and values are *abstract syntax trees*, complying with the abstract grammar. We except from the abstract grammar the `OBJECT IDENTIFIER` and `RELATIVE-OID` types and values for the sake of brevity. We also ignore the extension markers and the subtyping constraints beacause they play no role in the BER [7, §8.1.1.4] (however we considered some constraints at step 13).

Values. The abstract grammar for core ASN.1 values is defined as follows. Firstly, we assume that the parser removes the ambiguity between enumeration constants [3, §19] and value references [3, §11.4]. For instance, in `a T ::= b`, the token `b` can denote either an enumeration constant or a value reference, depending on the definition of type `T`. The ambiguity can always be removed just by looking at the type definition (this is easy in core ASN.1). The type *item* is used later in the enumerated constants and the type *label* denotes component names.

type *item* = *string* **and** *label* = *string*
type *core_value* = [‘SetOf **of** *core_value list* | ‘SeqOf **of** *core_value list* | ‘Set **of** (*label* × *core_value*) *list* | ‘Seq **of** (*label* × *core_value*) *list* | ‘TRUE | ‘FALSE | ‘Enum **of** *item* | ‘Int **of** *int* | ‘Real **of** *float* | ‘NULL | ‘MINUS_INFINITY | ‘Chosen **of** *label* × *core_value* | ‘String **of** *string* | ‘BitStr **of** *bool array* | ‘PLUS_INFINITY]

where ‘SetOf corresponds to values of `SET OF` and ‘SeqOf to values of `SEQUENCE OF` types [3, §25, §27]; ‘Set models values of the `SET` type and ‘Seq models values of `SEQUENCE` types [3, §24, §26] (the argument is a mapping from labels to values); ‘TRUE and ‘FALSE are obvious; ‘Enum models enumerated constants; ‘Int and ‘Real stand for `INTEGER` and

`REAL` values (for simplicity, we assume they fit the built-in arithmetic of OCaml); ‘NULL models the special `NULL` value [3, §23]; ‘PLUS_INFINITY and ‘MINUS_INFINITY correspond to `PLUS-INFINITY` and `MINUS-INFINITY`; ‘Chosen corresponds to `CHOICE` values [3, §28] (thus its argument is a pair of a label and a value); ‘String concentrates all kinds of character strings; ‘BitStr represents `BIT STRING` constants [3, §21] and `OCTET STRING` values [3, §22]. OCaml values of type core_value will be noted v.

type *tagged_type* = *tag list* × *core_type*
and *tag* = (*tag_class* × *int*) × *tag_mode*
and *tag_class* = UNIVERSAL | PRIVATE | APPLICATION | Context
and *tag_mode* = EXPLICIT | IMPLICIT
and *core_type* = [‘CHOICE **of** *label* → *tagged_type* | ‘OCTET_STRING | ‘SET **of** *components* | ‘SEQUENCE **of** *components* | ‘BIT_STRING | ‘SET_OF **of** *tagged_type* | ‘SEQUENCE_OF **of** *tagged_type* | ‘NULL | ‘ENUMERATED **of** *item* → *int* | ‘INTEGER | ‘BOOLEAN | ‘REAL | ‘String | ‘TRef **of** *string*]
and *components* = (*label* × *tagged_type* × [‘OPTIONAL] *option*) *list*

The type *tagged_type* models the tagged types of core ASN.1, in which a type (*core_type*) can be preceded by a list of tags. Constructor names of type *core_type* are almost self-explanatory, except ‘TRef which denotes type references. The type *components* defines the components of `SET` and `SEQUENCE` core ASN.1 types: it is a triple made of a label, a tagged type and an optional `OPTIONAL` component's attribute. OCaml values of type *core_type* are noted T and *tagged_type* values $\overline{\mathrm{T}}$. The mapping of type *label* → *tagged_type*, which is the argument of ‘CHOICE, is noted F. Values of type *components* are lists noted Φ of components noted φ, *e.g.,* ‘SET $(\varphi :: \Phi)$. An ASN.1 module is modeled by a type environment which is modeled by a function Γ from type names to tagged types, since there are no more value references in core ASN.1. Values of type *tag* are noted ψ and lists of tags Ψ.

4 Coding and decoding

BER codes. The structure of a BER code is based on the triple (*tag*, *length*, *contents*). The *tag* field corresponds to the tag of the value type in ASN.1, the *length* is the length of the contents field and the *contents* field is either another code (in which case the code is said *constructed*) or the encoding of a primitive type (in which case the code is said *primitive*). A primitive type is an ASN.1 built-in type which is not defined in terms of other types, *e.g.,* the `INTEGER` type. If the contents length is unknown at the encoding-time, it is possible for the coder to provide a special dummy length and then close the code

with an *ending octet*, in which case the code is said to be in *indefinite form*, as opposed to *definite form*. Definite form requires that the sender computes the whole code before sending it (in order to be able to compute the contents length) and it allows the receiver to allocate a bounded amount of memory to store the incoming code. The indefinite form allows the sender to encode the value coming from the upper application as it comes throughout a buffer (i.e. faster encoding within a bounded space) but it requires the receiver to handle carefully the incoming stack size. Indeed, the BER codes have a recursive structure and one of the advertised vulnerabilities was due to a deeply embedded code in indefinite form which overflowed the receiver's stack because the *implementation* was mishandling the memory.

Abstract BER codes. A complete formalisation of the BER first requires a model of the codes *at the octet level*, by means of a context-free grammar for instance, and the proof of some relevant properties on it. For example, from a soundness point of view, it is important to prove that the grammar is not ambiguous, i.e. a given code cannot be described in more than one way (exactly one derivation tree); from the decoder's efficiency point of view, it is important to prove that the grammar can be recursively analysed without backtracking and with a small constant amount of look-ahead. Unfortunately, due to the limited room, we have to skip this interesting stage. We shall assume that we already deal with *abstract codes*, which correspond to the abstract syntax trees of the compilers: an abstract code does not model the octets, but rather the structure of the codes. As a consequence, the length field is not included in an abstract code since, conceptually, an abstract code is a tree, not a series as the original codes. Moreover, the concepts of definite and indefinite form are not relevant for abstract codes, since they apply to octet streams only. The abstract codes are thus modeled with an OCaml type since these types correspond to trees with user-defined nodes and leaves.

type *primitive_code* = Pint | Preal | Pminus_inf | Pplus_inf | Pstring
| Pbit_str | Pbool **of** *int* | Pnull

type *code* = (*tag_class* × *int*) × *contents*
and *contents* = Primitive **of** *primitive_code* | Constructed **of** *code list*

The type *primitive_code* captures the codes of the values from types `INTEGER`, `REAL`, `BIT STRING`, `OCTET STRING`, `BOOLEAN`, `NULL` and the numerous character string types. The abstract primitive codes carry little discriminative information for a given type; for example, *all* the `INTEGER` values are encoded into the same abstract code Pint, but codes of `REAL` values are still different (Preal). This way we abstract away octet-level details which would otherwise bring us too far. We nevertheless keep the `BOOLEAN` standard encoding: value `FALSE` is encoded as (Pbool 0) and `TRUE` is encoded as (Pbool n) for any $n > 0$. *This allows to maintain the non-determinism of the BER in the modeling.*

A *code* is a triple made of a *tag_class*, a tag number (*int*) and *contents*. The latter is either a primitive or a constructed code. A constructed code is a list of codes.

Inference rules. We define the encoding with a *system of inference rules*. These are logical implications $P_1 \wedge P_2 \wedge \ldots \wedge P_n \Rightarrow C$ graphically represented as

$$\frac{P_1 \qquad P_2 \qquad \ldots \qquad P_n}{C}$$

where the P_i are the *premises* and C is the *conclusion*. When there is no premise, C is an *axiom* and is simply noted C. An inference rule can be interpreted also from a computational point of view: in order to compute C, we need to compute the P_i first (order is not specified). The rules and axioms can contain unquantified variables (*free variables*). In this case they are implicitly universally quantified ($\forall$) at the beginning. For instance $\dfrac{P_1(x) \qquad P_2(y)}{P(x,y)}$ PROP actually denotes the property PROP which is $\forall x, y. P_1(x) \wedge P_2(y) \Rightarrow P(x,y)$. A system of inference rules is an unordered set of rules. A theorem is a judgement, i.e. a formal statement. A demonstration is a proof tree whose root (the conclusion) is the theorem, the inner nodes are the conclusions of its subtrees and the leaves are axioms.

Abstract BER. Let us note $\Gamma \vdash v : (\Psi, \mathrm{T}) \to c$ the judgement "In the environment Γ, the value v is encoded into the code c, following the type T with the tags Ψ." The environment models the module and is mandatory because recursive types are allowed, thus type references do exist. Given a type name x, the referred type is $\Gamma(x)$. Using a system of inference rules to define the encoding relation means that the successful encoding of a value matches a proof tree made with the following rules:

$$\frac{n > 0}{\Gamma \vdash \text{`}\mathsf{TRUE} : ([\tau, p], \text{`}\mathsf{BOOLEAN}) \to (\tau, \mathsf{Primitive}\,(\mathsf{Pbool}\,n))}\ \textsc{True}$$

$$\textsc{Ref}\ \frac{\Gamma \vdash v : \Gamma(x) \to c}{\Gamma \vdash v : ([\,], \text{`}\mathsf{TRef}\,(x)) \to c}$$

$$\frac{\pi \text{ is a permutation on } \textit{components} \qquad \Gamma \vdash v : (\Psi, \text{`}\mathsf{SEQUENCE}\,(\pi(\Phi))) \to c}{\Gamma \vdash v : (\Psi, \text{`}\mathsf{SET}\ \Phi) \to c}\ \textsc{Set}$$

$$\frac{\Gamma \vdash v : (\Psi, \mathrm{T}) \to c}{\Gamma \vdash v : ((\tau, \mathsf{EXPLICIT}) :: \Psi, \mathrm{T}) \to (\tau, \mathsf{Constructed}\,[c])}\ \textsc{Tags}$$

$$\frac{\begin{array}{c}\varphi = (l, \overline{\mathrm{T}}, \mathsf{Some}\,\text{`}\mathsf{OPTIONAL}) \\ \Gamma \vdash \text{`}\mathsf{Seq}\,M : ([\psi\,], \text{`}\mathsf{SEQUENCE}\;\Phi) \to c\end{array}}{\Gamma \vdash \text{`}\mathsf{Seq}\,((l, v) :: M) : ([\psi\,], \text{`}\mathsf{SEQUENCE}\,(\varphi :: \Phi)) \to c}\;\textsc{SeqOptOut}$$

$$\frac{\begin{array}{c}\varphi = (l, \overline{\mathrm{T}}, \mathsf{Some}\,\text{`}\mathsf{OPTIONAL}) \qquad \Gamma \vdash v : \overline{\mathrm{T}} \to c \\ \Gamma \vdash \text{`}\mathsf{Seq}\,M : ([\psi], \text{`}\mathsf{SEQUENCE}\;\Phi) \to (\tau, \mathsf{Constructed}\,C) \\ \overline{c} = (\tau, \mathsf{Constructed}\,(c :: C))\end{array}}{\Gamma \vdash \text{`}\mathsf{Seq}\,((l, v) :: M) : ([\psi], \text{`}\mathsf{SEQUENCE}\,(\varphi :: \Phi)) \to \overline{c}}\;\textsc{SeqOptIn}$$

Due to the lack of space, we only presented the more interesting rules, of which we shall comment the conclusions before the premises. Lists are noted between brackets and $a :: A$ is a list whose head is a and sub-list is A. A pair is either noted (a, b) or a, b. Rule True illustrates a primitive encodings which is non-deterministic (variable n is free). Pattern $[\tau, n]$ matches a list of a single element which is a pair whose first projection is named τ and the second is named n. Since we operate on core ASN.1, this tag is compulsorily the predefined `UNIVERSAL` and `IMPLICIT` tag of `INTEGER`. Rule Ref matches the encoding of a type reference $\text{`}\mathsf{TRef}(x)$ with no tags: we encode the referenced type $\Gamma(x)$. Rule Tags apply when an `EXPLICIT` tag occurs first. Note that $\Psi$ cannot be empty, i.e. $[\,]$, since an `IMPLICIT` tag only apply to a core type. Rule Set models the non-determinism of the BER with respect to the `SET` type: any permutation of the sub-codes is allowed.

Rules SeqOptOut and SeqOptIn model another non-deterministic behaviour: a component value whose type is `OPTIONAL` may not be encoded, as a sender's option. Hence these *two* rules have the same conclusion (it is the only case), contrary to rule Set in which non-determinism is modeled by a free variable (π). We did not model the encoding errors: at any time, given an environment Γ, a tagged type (Ψ, T) and a value v, if no conclusion $\Gamma \vdash v : (\Psi, \mathrm{T}) \to \star$ matches then it is a run-time error (we can build no code c in place of $\star$) and the implementation must handle properly this situation in an unspecified way. If the typing is statically done by the ASN.1 compiler, this should not happen, but since we decided not to model the typing, the typing is partly included in the encoding (i.e. at run-time).

Abstract decoding. As we said in section 2, the BER decoding process is not published, is up to the ASN.1 compiler implementors and can be modeled by a non-injective function. We propose the following equational definition we expect to be faithful. Let us note $\mathcal{D}(\Gamma, c, (\Psi, \mathrm{T}))$ the decoding of c in the environment Γ according to type T tagged Ψ.

$$\mathcal{D}(\Gamma, (((\mathsf{UNIVERSAL}, 1), \mathsf{Primitive}(\mathsf{Pbool}\,0))), ([\,], \text{`}\mathsf{BOOLEAN})) = \text{`}\mathsf{FALSE}$$
$$\mathcal{D}(\Gamma, (((\mathsf{UNIVERSAL}, 1), \mathsf{Primitive}(\mathsf{Pbool}\,n))), ([\,], \text{`}\mathsf{BOOLEAN})) = \text{`}\mathsf{TRUE} \quad \text{for all } n > 0$$
$$\mathcal{D}(\Gamma, c, ([\,], \text{`}\mathsf{TRef}\,(x))) = \mathcal{D}(\Gamma, c, \Gamma(x))$$
$$\mathcal{D}(\Gamma, (\tau, \mathsf{Constructed}\,[c]), ((\tau, \mathsf{EXPLICIT}) :: \Psi, \mathrm{T})) = \mathcal{D}(\Gamma, c, (\Psi, \mathrm{T}))$$
$$\mathcal{D}(\Gamma, (\tau, \kappa), ([\,], \text{`}\mathsf{CHOICE}\,F)) = \mathcal{D}(\Gamma, (\tau, \kappa), F(l)) \quad \text{where } F(l) = ((\tau, m) :: \Psi, \mathrm{T})$$

We do not provide the full definition for we lack of space and do not wish to drown the reader into too much technical details anyway.

5 Equivalences and soundness

Value equivalence. It is possible to present a complete definition of the value equivalence because we shaped core ASN.1 with this goal in mind. We note $A@B$ the catenation of lists A and B. We have

$$v \approx v \quad \textsc{Reflexivity} \qquad \frac{v_1 \approx v_2 \qquad \text{`}\mathsf{Seq}\,M_1 \approx \text{`}\mathsf{Seq}\,M_2}{\text{`}\mathsf{Seq}\,((l, v_1) :: M_1) \approx \text{`}\mathsf{Seq}\,((l, v_2) :: M_2)}\ \textsc{Seq}$$

$$\textsc{Transitivity}\ \frac{v_1 \approx v_2 \qquad v_2 \approx v_3}{v_1 \approx v_3} \qquad \frac{\exists l, v_2, M_2', M_2 . M = M_2' @ (l, v_2) :: M_2 \qquad v_1 \approx v_2 \qquad \text{`}\mathsf{Set}\,M_1 \approx \text{`}\mathsf{Set}\,M_2}{\text{`}\mathsf{Set}((l, v_1) :: M_1) \approx \text{`}\mathsf{Set}\,M}\ \textsc{Set}$$

$$\frac{v_1 \approx v_2}{v_2 \approx v_1}\ \textsc{Symmetry} \qquad \frac{v_1 \approx v_2}{\text{`}\mathsf{Chosen}\,(l, v_1) \approx \text{`}\mathsf{Chosen}\,(l, v_2)}\ \textsc{Choice}$$

$$\frac{v_1 \approx v_2 \qquad \text{`}\mathsf{SeqOf}\,V_1 \approx \text{`}\mathsf{SeqOf}\,V_2}{\text{`}\mathsf{SeqOf}\,(v_1 :: V_1) \approx \text{`}\mathsf{SeqOf}\,(v_2 :: V_2)}\ \textsc{SeqOf}$$

$$\frac{\exists v_2, V_2, V_2' . V = V_2' @ v_2 :: V_2 \qquad v_1 \approx v_2 \qquad \text{`}\mathsf{SetOf}\,V_1 \approx \text{`}\mathsf{SetOf}\,(V_2' @ V_2)}{\text{`}\mathsf{SetOf}\,(v_1 :: V_1) \approx \text{`}\mathsf{SetOf}\,V}\ \textsc{SetOf}$$

Our value equivalence amounts to a structural equality modulo permutations on sub-values of `SET` and `SET OF` types.

Code equivalence. The BER embed a lot of the type information into the codes through the use of tags and a structure isomorphic to types. This makes possible to define an equivalence relationship

between codes that relies on two codes only — no further context is needed.

$$\textsc{Reflexivity}\quad c \sim c \qquad \textsc{Symmetry}\quad \frac{c_1 \sim c_2}{c_2 \sim c_1} \qquad \textsc{Transitivity}\quad \frac{c_1 \sim c_2 \qquad c_2 \sim c_3}{c_1 \sim c_3}$$

$$\frac{m > 0 \qquad n > 0}{(\tau, \mathsf{Primitive}\,(\mathsf{Pbool}\,m)) \sim (\tau, \mathsf{Primitive}\,(\mathsf{Pbool}\,n))}\ \textsc{True}$$

$$\frac{\tau = (\mathsf{UNIVERSAL}, 16) \qquad c_1 \sim c_2 \qquad (\tau, \mathsf{Constructed}\,C_1) \sim (\tau, \mathsf{Constructed}\,C_2)}{(\tau, \mathsf{Constructed}\,(c_1 :: C_1)) \sim (\tau, \mathsf{Constructed}\,(c_2 :: C_2))}\ \textsc{Seq/SeqOf}$$

$$\frac{\tau = (\mathsf{UNIVERSAL}, 16) \qquad (\tau, \mathsf{Constructed}\,C_1) \sim (\tau, \mathsf{Constructed}\,C_2)}{(\tau, \mathsf{Constructed}\,(c_1 :: C_1)) \sim (\tau, \mathsf{Constructed}\,C_2)}\ \textsc{SeqOptOut}$$

Contrary to value equivalence, there are too many cases and hence we cannot present them all. Rule True defines the equivalence of two possibly different encodings of the value `TRUE`. Rule Seq/SetOf specifies when (and how, in fact) codes from values of types `SEQUENCE` and `SEQUENCE OF` are equivalent. By the way, note that the tags of these two types are identical, hence, in theory, this rule makes equivalent the encodings of, say, values of types `SEQUENCE {a INTEGER}` and `SEQUENCE OF INTEGER`, as soon as the integer value is the same. Rule SeqOptOut is dual to the homonym rule of the abstract BER where an optional value component is not encoded. Here, it is allowed to skip a sub-code when decoding. *We do not specify when a sub-code has to be skipped or in which code.* We leave this to a more refined specification and/or algorithm.

Equivalence properties. The properties we expect to hold in our BER model can now be restated in a formal way. First of all, proposition 1, which states that all the BER encodings of a given value, according to a given type, are equivalent, becomes through the use of formal notations:

Proposition 3. *If* $\Gamma \vdash v : \overline{\mathrm{T}} \to c_1$ *and* $\Gamma \vdash v : \overline{\mathrm{T}} \to c_2$ *then* $c_1 \sim c_2$.

Next, proposition 2 which states that the decoding of two equivalent codes lead to two equivalent values is now restated in the following way:

Proposition 4 (Equivalence entailment)**.** $c_1 \sim c_2 \Longrightarrow \mathcal{D}(\Gamma, c_1, \overline{\mathrm{T}}) \approx \mathcal{D}(\Gamma, c_2, \overline{\mathrm{T}})$

Finally, the soundness theorem 1, which says that the encoding and decoding of a core ASN.1 value v, following a core ASN.1 tagged type $\overline{\mathrm{T}}$, leads to a value which is equivalent to v, is now formally rephrased:

Theorem 3 (Soundness)**.** *If* $\Gamma \vdash v : \overline{\mathrm{T}} \to c$ *then* $v \approx \mathcal{D}(\Gamma, c, \overline{\mathrm{T}})$.

We have no room to show the proofs of these properties because they contain a great number of cases. One tricky aspect is the correct handling of sub-code permutations when dealing with `SET OF` and `SET` values: for a given unknown permutation on the sender's side, we must explicitly construct the reverse permutation on the receiver's side.

6 Conclusion

We presented a formal review design of the BER. On purpose, we abstracted away many low-level details in our model in order to understand, capture and formalise what are, according to us, the main characteristics of the BER. Therefore the further step would be to refine our model, by explicitly providing the coding and decoding functions for the primitive types, by reckoning with the various string types etc. Also we did not present evidences that the rewriting from the BER domain to its core ASN.1 subset conserves code equivalence, as pointed out in figure 2: this was a matter of room. We nevertheless think that our work dispels clouds of suspicion — if any — about the soundness of ASN.1 and the BER. More precisely, we mean that the composition of encoding and decoding yields a value which is equivalent to the original. The aim of our formal review design is to raise user's confidence on a solid ground and we doubt whether twenty more pages of formulæ would have been a stronger argument for the casual reader. Indeed, making explicit as many as possible assumptions and checking their consistence is inherently reassuring. The mere fact that we had to understand the rationale of the BER and put it into mathematical formalæ really brought to the fore a new understanding. Also the interest in choosing a system of inference rules to define our relationships is that this formalism closes the gap between specifications and algorithms. Besides, the suggested use of OCaml as an implementation language is motivated because, as a descendant of a logic meta-language, it is precisely suited to implement algorithms specified by means of inference rules. The way of deducing them consists mainly in providing a deterministic and constructive refinement which is sound and complete with respect to the initial specification. By constructive we mean for instance to replace existential quantifiers, the symmetry rule etc. by explicit procedures, and determinism means, in the context of this work, having no backtracking implied (*e.g.,* no overlapping conclusions).

References

[1] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Programmation d'applications avec Objective Caml*. O'Reilly France, 2000. 700 pp.

[2] Olivier Dubuisson. *ASN.1 — Communication Between Heterogeneous Systems*. Academic Press, 2000. ISBN 0-12-6333361-0.

[3] ITU-T Rec. X.680 (2002) or ISO/IEC 8824-1:2002. *Information technology — Abstract Syntax Notation One (ASN.1): Specification of basic notation*, 2002.

[4] ITU-T Rec. X.681 (2002) or ISO/IEC 8824-2:2002. *Information technology — Abstract Syntax Notation One (ASN.1): Information object specification*, 2002.

[5] ITU-T Rec. X.682 (2002) or ISO/IEC 8824-3:2002. *Information technology — Abstract Syntax Notation One (ASN.1): Constraint specification*, 2002.

[6] ITU-T Rec. X.683 (2002) or ISO/IEC 8824-4:2002. *Information technology — Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications*, 2002.

[7] ITU-T Rec. X.690 (2002) or ISO/IEC 8825-1:2002. *Information technology — ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*, 2002.

[8] ITU-T Rec. X.691 (2002) or ISO/IEC 8825-2:2002. *Information technology — ASN.1 Encoding Rules: Specification of Packed Encoding Rules (PER)*, 2002.

[9] Christian Rinderknecht. An Algorithm for Validating ASN.1 (X.680) Specifications using Set Constraints. *The Computer Journal*, 46(4), July 2003.