Parsing BER Using PEG

December 4, 2021

Kees-Jan Hermans (kees.jan.hermans@gmail.com)

Abstract

Because of its complexity - nested binary TLV encoding and various compression schemes - parsing BER/DER formatted messages is considered problematic. To the knowledge of the author, formal parsing of BER/DER is not addressed by available parsing solutions. Instead, custom, hand coded solutions or generated code prevail. This is a security risk from a point of view of bugs or code proliferation. The changes proposed in the paper extend Parsing Expression Grammars (PEG), enabling it to parse these formats, while retaining a simple grammar definition, and requiring only minimal changes to the resulting assembly and bytecode interpreter. This paper also shows that, having implemented those changes, and using a grammar that closely mimics its ASN.1 counterpart definition, it becomes possible to split up X.509 and SNMPv3 messages in their underlying parts. The most notable findings are that this method allows DER well-formedness checks, as well as formal structures checks, and allows formal structure content, such as digital signatures, to be isolated (captured) from the input. It shows how OIDs can be broken up in their composing parts, and the way in which text fields of binary formats can still be text-parsed (eg email address validation). However further study is required with respect to machine implementation of the Engine, and the representation of certain binary fields (eg INTEGERs, OIDs) from the input capture list.

Version 0.8.4 **Unclassified**

Contents

1 Research Goal

1.1 Problem Statement

Underlying the process of parsing certain application message formats, most notably SNMP [9] (which is typically used for managing network devices) and PKI related messages (X.509 certificates, CRL's, etc [11]) is a grammar specification (Abstract Syntax Notation (ASN.1) [7]) and its binary encodings (Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER) [8], etc, from now on collectively referred to as 'BER').

Because these messages are Internet-exchanged, BER encoded messages must be parseable by a recipient that cannot establish their origin, potentially exposing itself to mistakes and malice. To have a BER parser that does not fail, is crucial for the security of logic that depends on it. Ideally therefore, one would like to have a parser that a) is correct, b) retains logical integrity (ie preferably is hardware-based), and c) can traverse deeply into the message (ie may not just perform a 'well formedness' check).

BER formatting and the standards that use it, pose a problem to hardware parsing execution for several reasons: the fact that BER is a nested binary Type-Length-Value (TLV) format, and that it has several different compression schemes, most notably of numbers. Also often, at least in the case of SNMP and X.509, the messages may contain a cryptographic signature (which would allow you to verify the integrity of the message and the authenticity of the sender), but these are positioned in the format in a place that already requires parsing, defying the purpose of the security measure. Lastly, the ASN.1 grammar and standards texts describing these formats are sometimes less than formal (ie they require human interpretation). This paper proposes several methods to address these issues.

1.1.1 Assumptions

This paper assumes that the reader is familiar with ASN.1 and its binary encodings BER and DER, message formats that use this encoding such as SNMP and X.509, and parsing concepts enabled by grammars, both of the more general kind (eg Backus-Naur [3], regular expressions [2], Lex and Yacc [4]) and the more specific – in this case: Parsing Expression Grammar, or PEG [1], or, even more specific: its Lua implementation: LPEG [5].

1.1.2 High Level Goals

Aside from proving that BER can be made parseable using PEG-like parsers, the following high level goals are also kept in mind:

Minimal extension footprint. The amount of bugs in code is directly proportional to its size [14]. To extend any platform with more code therefore, one always needs a thorough rationale. Most notably the PEG bytecode Engine would theoretically suffer the most from any extension of its code base, both from a security and an efficiency standpoint. Therefore, extensions to the Engine must be kept to a minimum.

Retaining grammar readability. The weakness of any programmable system is mostly contained in the human doing the programming. Readable grammar is an important step in between the intention of the user, and the bytecode execution. Extensions to the existing grammar structure must be kept minimal and, where they occur, must fall within the existing language paradigm.

1.2 BER Parsing Aspects

1.2.1 Nesting

Like many object serialization formats (such as XML, JSON, YAML, etc), BER seldom encodes just a single scalar: instead it usually requires you to start off with a compound type, containing multiple members which in turn can be compound types etc, effectively serializing a complex tree of data, the end nodes of which are scalars (or empty compound types). Certainly the more specific examples of SNMP payloads or X.509 objects all abide by this rule.

The difference between BER and text-based serialization formats such as XML, lies in the way nested sections are delimited by the embedding element: BER tells the parser beforehand, which section of the input, delimited by a byte offset, is nested. XML and the like, on the other hand, leave the parser in the dark about the length of the nested section, until a closing token is encountered. There is a nuance here, as BER also has indefinite-length encoding (using a Length signifier 0x80 and a closing token in the form of 0x00 0x00). This case would be easily handled by existing tokenizing parsers, and is therefore not in scope of this paper.

Below is an example of a BER encoded pair of scalars (the bottom row hexadecimal string): an OID (type 0x06), and an IPv4 address (type 0x40), nested inside of a SEQUENCE (type 0x30).

```
Type
|  Length
|  |  Value (0x18, runs until here-----------------------------------------|)
|  |  |                                                                    |
|  |  Subtype                                               Subtype2       |
|  |  |  Sublength                                          |  Sublength2  |
|  |  |  |  Subvalue (0x10 runs until here---------------|) |  |  Subvalue2|
|  |  |  |  |                                            |  |  |  |        |
30 18 06 10 2B 06 01 04 01 81 E0 6B 02 02 06 01 06 03 01 01 40 04 C0 A8 50 01
```

Whereas, for example, in JSON [15], such a message might be encoded as follows:

```
[ "1.3.6.1.4.1.28777.2.2.6.1.6.3.1.1", "192.168.80.1" ]
```

(You can have a discussion about the semantics here, as the intention of the BER encoding is to format a name/value pair).

1.2.2 BER Length Encoding

BER is encoded as a binary Type-Length-Value (TLV) (and, in nested compound values, as sequences of TLV's – as in the Subvalues above). The Type or ('Tag') field of the TLV is a single byte that poses no parsing problems whatsoever, it can simply be matched and used as a discriminant for any further parsing action.

The Value field of the TLV is simply as long (in bytes) as the Length field denotes (so the lengths of the Type and Length fields of the TLV are considered implicit (Type is always one byte long) or self-descriptive (Length) and therefore ignored.

The Length field of the TLV is the problematic one, from a parsing perspective. To encode it, there are two forms: short (for lengths between 0 and 127), and long definite (for lengths between 0 and 21008 -1).

Short form. One octet. Bit 8 has value "0" and bits 7-1 give the length. Long form. Two to 127 octets. Bit 8 of first octet has value "1" and bits 7-1 give the number of additional length octets. Second and following octets give the length, base 256, most significant digit first. [10] Note that lengths 0-127 bytes are implicit in both forms (ie a length of decimal 32 can be encoded as 0x20 and as 0x81 0x20) and that the second form has many, many other possibilities of encoding the same (for example as 0x84 0x00 0x00 0x00 0x20)). Also note that certain derivatives of BER, most notably DER, constrain this behavior (X.690 §10.1 [10]):

"The definite form of length encoding shall be used, encoded in the minimum number of octets"

1.2.3 OID Encoding

OIDs (object identifiers) are crucial to the whole idea of SNMP and X.509. The function as the globally unique identifiers, whose values we're interested in. They can be thought of as a leaf node from a global tree of numbers. Their unique description is formed by the string of numbers (called 'subidentifiers') that represents the path one must traverse from the top of the tree, through this tree, to reach this node. To be able to present OIDs, safely, to a user, from this parser, is probably desirable. How this is done, whether in binary or in human readable form, is depends on the situation. In human readable form, OIDs are denoted in as a string of subidentifiers with dots in between, for example:

```
1.3.6.1.4.1.2681.1.2.102
```

BER encodes OIDs as follows:

- The first two subidentifiers are absorbed in the first octet, as follows:
  - The first subidentifier is multiplied by forty (decimal 40).
  - The second subidentifier is added up to this number.
  - The resultant number is encoded as a single byte (note the implication that the first subidentifier cannot be $>$5 and the second cannot be $>$39).
- Any following subidentifier $<$127 is encoded as a single byte (most relevant bit set to zero).
- Any following subidentifier $>=$127 is encoded 'a bit like UTF-8', that is: the minimal length of encoding the number in seven-bit portions is calculated, and for those numbers, minus one, the amount of bytes is produced, with the most relevant bit set to one, plus the remaining portion of seven bits, with the most relevant bit set to zero.

Note that OIDs are always encoded in a fashion that is as sparse as possible (and therefore deterministic). X.690 [10] stipulates (in §8.19.2) that

"The subidentifier shall be encoded in the fewest possible octets, that is, the leading octet of the subidentifier shall not have the value 80_{16}*."*

Below is an example of the OID above, BER encoded and represented as a hexadecimal string:

```
2B 06 01 04 01 94 79 01 02 66
```

1.2.4 INTEGER Encoding

The ASN.1 INTEGER type is encoded in BER by foregoing any leading zeroes, but keeping the signed minus bit: Contents octets give the value of the integer, base 256, in two's complement form, most significant digit first, with the minimum number of octets. The value 0 is encoded as a single 00 octet. Some example BER encodings (which also happen to be DER encodings) are given in the table below (from [8]):

Integer value	**BER encoding**
0	02 01 00
127	02 01 7F
128	02 02 00 80
256	02 02 01 00
-128	02 01 80
-129	02 02 FF 7F

1.2.5 Binary Encoding Containing Parseable Text

In certain cases 'normal' text is embedded inside a binary structure. The text in question may even have a structure that must be parsed further. The premise of this paper is, that this can still be done using normal grammar constructs (ie grammar rule definitions). An email address value of an X.509 certificate for example, can still fail a certificate policy or, on success, be served up in its individual parts in the capture list.

1.2.6 Digital Signatures

The topmost structure of an X.509 certificate (RFC 5280 [12]), which is a cryptographically signed DER encoded structure, is defined as follows:

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

TBSCertificate  ::=  SEQUENCE  {
        version         [0]  EXPLICIT Version DEFAULT v1,
        serialNumber         CertificateSerialNumber,
        signature            AlgorithmIdentifier,
        issuer               Name,
        validity             Validity,
        subject              Name,
        subjectPublicKeyInfo SubjectPublicKeyInfo,
        issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                             -- If present, version MUST be v2 or v3
}
```

This would require, of a parser, to get to the signature and therefore at least determine that the integrity of the message and the authenticity of the sender, the following, namely to parse:

- A SEQUENCE, containing
  - A SEQUENCE, followed by

- A SEQUENCE, containing
  - An OID that must be understood, since it contains the signature algorithm, followed by
  - Potentially, a parameter (of type ANY) to that OID (however, in practice, always NULL), followed by
- A BIT STRING containing the signature value (also depending on convention). For further elaboration on this subject, see [C].

1.2.7 ASN.1 Compilation

1.2.7.1 'Your Blob Goes Here'

In many places in standardized ASN.1 definitions, the definition of a type is not formal, but instead has be interpreted by humans. Examples are RFC5280 [12], line 976:

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

Line 1097:

```
AttributeValue ::= ANY -- DEFINED BY AttributeType
```

Line 2104:

```
OtherName ::= SEQUENCE {
        type-id    OBJECT IDENTIFIER,
        value      [0] EXPLICIT ANY DEFINED BY type-id }
```

The 'DEFINED BY' clause in ASN.1 does not have any formal meaning [???] (as can be seen from the comment '–' introduction in the definition, which can leave it in or out without consequence).

1.2.7.2 Type Ambiguity

To have the ASN.1 definition of a structure, however informal it is in certain places, always remains necessary. The BER encoding of ASN.1 only carries over with it the base type of the value (a SEQUENCE, an INTEGER, a BIT STRING, etc). It's not possible to derive a value's proper type (and therefore, its meaning) merely from the BER encoding.

1.2.8 In Summary / Sub Goals

A usable BER parser, one that is described in this paper, provides:

- Acceptable security of the parsing process, ie including the intrinsic problems of the format (TLV Length fields etc).
- Fine grained access to captured binary fields (eg OIDs, numbers, etc). Preferably in form that is natural to the parsing mechanism user (ie a form that does not require further processing).
- Fine grained access to captured embedded text fields (eg the components of X.509 DN elements, such as email addresses).

- The possibility of descending into otherwise undefined fields (sub parsing of ASN.1 field definitions that have been made in human readable standards text only).
- The possibility of applying digital signatures checks correctly and securely.

1.3 Existing / Preceding Work

The work in this paper is based upon the following existing and preceding work: LPEG and Naigama. This paper assumes that the reader is familiar with the former, but will expand on any details that stem from the latter (since it’s a project by the author).

1.3.1 LPEG

LPEG, or Lua Parsing Expression Grammars, is a PEG implementation within the scripting language Lua [6] [5]. It defines a grammar for specifying grammar rules, and an engine that processes bytecode, which is produced by the grammar compiler. On successfully processing an input, the LPEG user can use a capture list for further processing.

1.3.2 Naigama

Naigama [13] implements the LPEG idea functionally, in a way that’s modular and not associated with Lua. It has both grammar, bytecode and an execution engine. Just like LPEG, it allows the user to extract capture regions on success. It is a project maintained by the author, and can be found here: https://github.com/kjhermans/naigama.

Naigama extends LPEG (in a non-compatible manner), and implements among others the following relevant, extra functionality:

It provides an assembly language as an extra programming artifact in between the grammar and bytecode stages. It allows you to program this assembly directly and feed it to the assembler without the need to use the grammar compiler.

It provides bitwise matching. Note: This paper will use Naigama when grammar and assembly examples are given, and provide the reader with explanation when this is different from LPEG.

1.3.2.1 Bitwise Matching

Naigama introduces bitwise matching. This is much like character matching, but then only for a portion of the byte. A mask, which is applied in a logical ‘and’-fashion, and the expected resultant octet value, are given. On success, just like with the ‘char’ instruction, the input pointer is increased by one, and the next instruction is executed. Grammatically, it is defined as follows:

```
IPv4 <- |40|f0|
```

(This would match the first four bits of a byte, when they contain the value 4, or binary 0100). Assembly instruction definition:

```
MASKEDCHARINSTR    <- { ’maskedchar’ } S HEXBYTE S HEXBYTE
S                  <- %s+
HEXBYTE            <- [a-fA-F0-9]^2
```

A rule such as the one above, would be compiled as follows:

```
IPv4:
  maskedchar 40 f0
  ret
```

1.4 Work Approach

The work described in this paper has the following build-up:

- Analysis of missing function. The gap between the existing technologies and the problem we're trying to solve with them, must be made clear in detail.
- The proposal will be run 'bottom-up', that is to say: any changes are first weighed in terms of the impact they may have on the functioning and the size of the code base of the Engine.
- These changes then have their consequences worked out in the assembly and grammar spec.
- An implementation is made, the changes are tested against a small set, and finally the results are reported on, and any conclusions are drawn.

1.5 Expected Results

When all the issues can be addressed, then ideally, will be achieved the following:

A minimal extension to existing specification. That is to say:

- The amount of (bytecode / assembly) instructions must remain small.
- The amount of Engine implementation changes should be minimal.
- The amount of added Engine execution primitives and artifacts should remain small.

Capture relevant fields in relevant representation. Binary and human readable representation are opposite concepts in many types such as integers, where in strings, they are the same (save for discussions about terminating zeroes). The expected result of this research is to see how far the concept of acceptable representation of all types to all users can be taken.

Allow deep parsing, also of embedded text. So that, for example, domain name or email address parsing can be incorporated into the machine language.

Allow (partial) ASN.1 compilation. A complete treatment of the ASN.1 compilation (ie to PEG grammar), is beyond the scope of this document (although it is probably a lot of, but not very difficult, work). However, I will touch on the following: 1) ASN.1 compilation patterns, 2) the conversion of OIDs (in human readable representation) and INTEGERs to their binary representation for matching purposes in grammar (as it would be a necessary utility for the proposition in this paper to work), and 3) the less formal parts of ASN.1, and how to deal with them generically.

2 Work

2.1 Analysis of Missing Function

Parsers, more precisely tokenizers, can do many things that are required for binary parsing already. For example, LPEG can split up (most of) the IPv4 header just fine:

```
IPV4HDR  <- VRSIHL TOS TOTLEN ID FRAGWORD TTL PROTO CHK SRC DST
VRSIHL   <- { . }
TOS      <- { . }
TOTLEN   <- { .. }
ID       <- { .. }
FRAGWORD <- { .. }
TTL      <- { . }
PROTO    <- { . }
CHK      <- { .. }
SRC      <- { .... }
DST      <- { .... }
```

Note that in this example LPEG already comes up short where fields are split up bitwise, which is the case for the VRSIHL, TOS, and FRAGWORD field. However also note that, using binary but whole-byte matching, one could replace the VRSIHL rule with the following rule (and capture 99% of IPv4 packets, where options aren't defined):

```
VRSIHL <- { 0x45 }
```

Giving you a discriminant on your input ('you are indeed parsing an IPv4 header'). One could even list all the options for the first byte of an IPv4 header (and capture all packet headers), like so:

```
VRSIHL <- { 0x45 / 0x46 / 0x47 / 0x48 / 0x49 / 0x4a /
            0x4b / 0x4c / 0x4d / 0x4e / 0x4f }
```

You can even go so far as to have each discriminant fetch its own header size, like so:

```
IPV4HDR <- 0x45 HDRFIELDS / 0x46 HDRFIELDS { .... } /
           0x47 HDRFIELDS { ........ } / -- etc
```

Defining these 'discriminant-vs-length' rules provides you with rudimentary 'if-then' functionality in a grammar. However, a normal, text-token based tokenizer/parser (like Lex/Yacc, LPEG) cannot go any further and therefore cannot parse BER. For the following reasons:

- It cannot perform less-than-a-byte (bitwise) matching. To be able to do this is necessary, because different (groups of) bits of single byte can convey a different meaning in BER. This problem is most prevalent in TLV Length values, but also resurfaces in the representation of captured ASN.1 INTEGER types and OIDs.
- Text parsing tools generally look for delimiters in tokens in the text itself. It has to encounter tokens, not bits or lengths as a way of moving on (to the next token or fail).

- Once the parsing process has started, it runs formally along the lines of the structure definition: it cannot interpret data from the input itself to use in, or steer, the parsing process.

All of these issues pertain to compression done at the bit level when representing numbers, most notably in parsing TLV Lengths. What follows is a discussion of this topic, and the other number compression schemes in BER.

2.2 Parsing TLV Lengths

A TLV Length field delimits the length of input of a nested section (the Value field) to be parsed. This is very different from text parsing, where the delimitation comes in the form of tokens (that are only encountered when the nested part has already been processed). What is needed is the possibility to isolate areas of input for nested parsing, based beforehand on the amount of bytes that this area is supposed to be in size. The problem breaks down into the following underlying ones: to read the Length field, to redirect this information into a context usable by the Engine, and to have the Engine features to support this and finally, the assembly instructions and the grammar syntax to describe this process.

2.2.1 Limiting the End-of-Input Temporarily

To accommodate length values being given before the to-be-parsed input is encountered, and as its only delimiter (ie without a closing token), we need to introduce a method to limit the input, so that grammar ’!.’ means end-of-input at a point in the input that lies at the end of the sub-section, ie before the real end-of-input (and also, for example, grammar ’.*’ runs to this point and no further).

The proposal is to do this in a calling context. So, any grammar ’FOO <- BAR’ rule will, when BAR is called, limit the input for the context in which BAR is executed: either to the current end-of-input (the default situation), or to a point before the current end-of-input. Returning from this calling context, through RET or FAIL will restore the original end-of-input.

2.2.2 Parsing the End-of-Input Value from the Engine

The end-of-input value is given, in BER, as the Length field in the TLV currently under scrutiny by the Engine. Naigama is capable of parsing the BER TLV Length part through bitwise matching or a range (LPEG would have to stick to a range), both in grammar and in assembly. As follows (a bit like the extensive IPv4 header example), using the following grammar:

```
BERLENGTH <- & |00|80| { . } /
             0x81 { . } / 0x82 { .. } / 0x83 { ... } / 0x84 { .... }
```

Note that, in this example, we're only willing to accept four-byte length encodings. When you're operating on a 64-bit platform (and you think it's reasonable to be processing values with lengths of over 4 gigabyte), then you can simple extend the above pattern to include 0x85 up to 0x88 as well.

The assembly of the grammar above (generated by the Naigama compiler):

```
__RULE_BERLENGTH:
  catch __ALT_2
  catch __SCANNER_3
  maskedchar 00 80
  backcommit __SCANNER_3_OUT
__SCANNER_3:
  fail
__SCANNER_3_OUT:
  opencapture 0
  any
__SUCCESS_4:
  closecapture 0
  commit __SUCCESS_1
__ALT_2:
  catch __ALT_5
  char 81
  opencapture 1
  any
__SUCCESS_6:
  closecapture 1
  commit __SUCCESS_1
__ALT_5:
  catch __ALT_7
  char 82
  opencapture 2
  any
  any
__SUCCESS_8:
  closecapture 2
  commit __SUCCESS_1
__ALT_7:
  catch __ALT_9
  char 83
  opencapture 3
  any
  any
  any
__SUCCESS_10:
  closecapture 3
  commit __SUCCESS_1
__ALT_9:
  char 84
  opencapture 4
  any
  any
  any
  any
__SUCCESS_11:
  closecapture 4
__SUCCESS_1:
  ret -- BERLENGTH
```

The alternative to this approach, is to create engine-intrinsic methods to parse, consume and interpret the BER TLV Length field (perhaps to create support in the LPEG engine). This is however not the approach taken in this paper.

2.2.3 Changes to the Engine

2.2.3.1 To Implement Temporary Input Length Delimitation

The following are the required changes to the Engine, in order for it to implement temporary input length delimitation:

- The Engine shall contain, next to its normal, ultimately delimiting input length value (*li*) (this value is now used to verify the validity of the current input offset value), one more register: the new limit to input (*li'*), and also a bit, indication whether or not *li'* is set (*sli'*).
- Each existing instruction may set *sli'* to zero. This is a safety feature; any instruction that sets *sli'* must be followed by a 'call' instruction. An alternative to this approach is code inspection (to verfify that, indeed, every instruction setting *sli'* is indeed directly followed by a 'call' instruction).
- The 'call' instruction however, shall first check to see if *sli'* is set, and if it is, push li into the calling context on the stack, and assign *li'* to *li*.
- The 'ret' instruction shall restore its calling context's copy of *li*.
- The FAIL condition, cleaning up a calling context from the stack, shall also restores its copy of *li*.

The following conditions then shall be applied:

- Given input offset oi, each matching instruction shall check that $oi < li$.
- At each setting of *li'* an instruction shall check that $li' <= li$.

2.2.3.2 To Fill the Limiting Register

This leaves the question of how the *li'* register is filled (and the *sli'* bit is set). This paper proposes the introduction of a new instruction. As follows:

- The 'intrpcapture' ('interpret capture') instruction shall be introduced, which translates the contents of a capture region to the *li'* register, and sets *sli'*:
  - Has defined as its first parameter, a 'mode' which, for now, only has one possible value: to interpret the capture as a right-aligned, 32-bit, unsigned integer.
  - Has optionally defined as its second parameter, the slot value of the capture region. When this is set, the capture list will be examined, top to bottom, for the first occurrence of a capture region with the matching slot number. If unset, the topmost capture will be taken.

2.2.4 Changes to the Assembly

The assembly only has to be changed by introducing the 'intrpcapture' instruction. 2.2.4.1 The 'intrpcapture' Instruction Grammatically, this addition to the assembly shall be defined as follows:

```
INTRPCAPTUREINSTR <- ‘intrpcapture’ S MODE S SLOTNUMBER
MODE              <- ‘ruint32’
SLOTNUMBER        <- UNSIGNED / ‘default’
```

Which, in practice, will probably look like this in assembly:

```
intrpcapture ruint32 default
```

2.2.5 Changes to the Grammar

In order for the compiler to correctly emit the ‘intrpcapture’ instruction mnemonic and parameters, it must have the semantic tools to do so. To this purpose, a special calling context is created: normally in PEG, what is compiled as a rule-call is in grammar simply denoted as the identifier of the rule. For this purpose however, the rule identifier is dressed up a little, and is bound together with the capture and the type conversion of the capture required, using opening and closing double angle brackets.

Grammatically, the grammar syntax involved, is defined as follows:

```
LIMITEDCALL <- LCALLOPEN METHOD COLON VARREF COLON IDENTIFIER LCALLCLOSE
LCALLOPEN   <- ‘<<’
LCALLCLOSE  <- ‘>>’
METHOD      <- ‘ruint32’
-- existing definitions of COLON, VARREF and IDENTIFIER
```

Resulting in the following example grammar; note the special denotation of the LISTCONTENT rule call:

```
LIST        <- 0x30 DERLENGTH <<ruint32:$_:LISTCONTENT>>
DERLENGTH   <- & |00|80| { . } /
               0x81 { . } / 0x82 { .. } / 0x83 { ... } / 0x84 { .... }
LISTCONTENT <- .*
```

This introduces another concept: the default capture. In Naigama, references (‘variables’) can be made to items in the capture list, to use those for matching input (this is also true for LPEG, which uses another grammar convention (with ‘=’)). This is extremely convenient, for example, for matching closing tag names to opening ones in XML. Variables can be named, and in Naigama, also numbered (referring to slot number). The ‘_’ variable refers to the topmost capture of the capture list.

2.3 OIDs

2.3.1 Matching any OID

This section treats BER capture completely, using the concepts from the preceding sections. An implementation of these were made in Naigama, and subsequently executed.

2.3.1.1 Input Example of an OID encoded as DER TLV:

```
06 10 2B 06 01 04 01 81 E0 6B 02 02 06 01 06 03 01 01
```

The above example input can be broken down as follows:

- Type (0x06), followed by:
- Length (single byte encoding 0x10 / decimal 16), followed by:
- 16 bytes of value payload, consisting both of capturable single byte subidentifiers, as well as those that have been encoded using multiple bytes.

2.3.1.2 Grammar

```
OID        <- 0x06 BERLENGTH <<ruint32:$_:OIDVALUE>>
OIDVALUE   <- { { . } { |80|80|* |00|80| }* }
BERLENGTH  <- & |00|80| { . } /
              0x81 { . } / 0x82 { .. } / 0x83 { ... } / 0x84 { .... }
```

Note that the BERLENGTH rule has been treated in [2.2.2].

2.3.1.2.1 An Alternative Grammar

When using an indiscriminate-length quantifier, such as in the example above, is problematic from a resource perspective (*ie* you want the parser to fail when an OID contains too many elements, not the parser-using logic), then, in Naigama, you can also formulate the grammar as follows (note the '^-*n*' quantifiers, which specify that an OID subidentifer cannot exceed 4 bytes in denotation length (protecting your CPU from integer overflow) and the an OID cannot have more than 16 subidentifiers):

```
OID        <- 0x06 BERLENGTH <<ruint32:$_:OIDVALUE>>
OIDVALUE   <- { { . } { |80|80|^-4 |00|80| }^-16 }
BERLENGTH  <- ...
```

2.3.1.3 Assembly

The grammar above (the quantifier-less variety) produces the following assembly (note the 'intrpcapture' instruction).

```
  call OID
  end 0

__RULE_OID:
  char 06
  call BERLENGTH
  intrpcapture ruint32 default
  call OIDVALUE
__SUCCESS_1:
  ret -- OID

__RULE_OIDVALUE:
  opencapture 0
  opencapture 1
  any
__SUCCESS_4:
  closecapture 1
  catch __FORGIVE_5
__FOREVER_6:
  opencapture 2
  catch __FORGIVE_8
__FOREVER_9:
  maskedchar 80 80
  partialcommit __FOREVER_9
__FORGIVE_8:
  maskedchar 00 80
__SUCCESS_7:
  closecapture 2
  partialcommit __FOREVER_6
__FORGIVE_5:
```

```
__SUCCESS_3:
  closecapture 0
__SUCCESS_2:
  ret -- OIDVALUE

__RULE_BERLENGTH:
  catch __ALT_11
  catch __SCANNER_12
  maskedchar 00 80
  backcommit __SCANNER_12_OUT
__SCANNER_12:
  fail
__SCANNER_12_OUT:
  opencapture 3
  any
__SUCCESS_13:
  closecapture 3
  commit __SUCCESS_10
__ALT_11:
  catch __ALT_14
  char 81
  opencapture 4
  any
__SUCCESS_15:
  closecapture 4
  commit __SUCCESS_10
__ALT_14:
  catch __ALT_16
  char 82
  opencapture 5
  any
  any
__SUCCESS_17:
  closecapture 5
  commit __SUCCESS_10
__ALT_16:
  catch __ALT_18
  char 83
  opencapture 6
  any
  any
  any
__SUCCESS_19:
  closecapture 6
  commit __SUCCESS_10
__ALT_18:
  char 84
  opencapture 7
  any
  any
  any
  any
__SUCCESS_20:
  closecapture 7
__SUCCESS_10:
  ret -- BERLENGTH
```

2.3.1.4 Engine Execution

For complete engine execution output and states, refer to [A]. The abbreviated output of the engine, based on the quantifier-less grammar:

```
End code: 0
16 actions total
Action #0: capture slot 3, 1->1 "\x10"
Action #1: capture slot 0, 2->16
"+\x06\x01\x04\x01\x81\xe0k\x02\x02\x06\x01\x06\x03\x01\x01"
Action #2: capture slot 1, 2->1 "+"
Action #3: capture slot 2, 3->1 "\x06"
Action #4: capture slot 2, 4->1 "\x01"
Action #5: capture slot 2, 5->1 "\x04"
Action #6: capture slot 2, 6->1 "\x01"
Action #7: capture slot 2, 7->3 "\x81\xe0k"
Action #8: capture slot 2, 10->1 "\x02"
Action #9: capture slot 2, 11->1 "\x02"
Action #10: capture slot 2, 12->1 "\x06"
Action #11: capture slot 2, 13->1 "\x01"
Action #12: capture slot 2, 14->1 "\x06"
Action #13: capture slot 2, 15->1 "\x03"
Action #14: capture slot 2, 16->1 "\x01"
Action #15: capture slot 2, 17->1 "\x01"
Number of instructions: 109
Max stack depth: 4
```

Conclusion: Naigama is capable of parsing BER encoded OID TLV formatting, as well as splitting up the input in its subidentifier parts. It does not however split up the first capture (Action #1) which, semantically, consists of two OID subidentifiers, and it does also not bitshift a capture (Action #7) which has a value $>$ 127.

2.3.2 Matching a Known OID

Since OID encoding is deterministic, known OID parsing does not require any special tricks, only that your grammar can define non-text (binary) character matching rules. For example, as follows:

```
DN_EMAIL <- 0x06 0x09 0x2a 0x86 0x48 0x86 0xf7 0x0d 0x01 0x09 0x01
```

2.3.3 Presenting any OID

Although OIDs cannot be represented textually from a capture to a user in a simple manner (for reasons given above: the fact that the first two OID subidentifiers are joined in one binary octet, and that subidentifiers $>$ 127 require a different encoding, including bits that are not usable in binary number representation), the reverse should be relatively easy. An ASN.1-to-PEG compiler can precompile OIDs to their binary representation and, as such, use them for matching.

Admittedly, this solves only half the problem. The 'here in the input should be any OID and I would like to know what it is' problem isn't addressed by this method; that still has to be parsed out of the capture by the user.

2.4 INTEGERs

2.4.1 Matching any INTEGER

It makes sense to protect your machine intrinsic types by not allowing infinitely long integers (much like the BERLENGTH rule does not allow for infinitely long TLV length definitions). For example, by creating the following grammar definition (0x02 is the BER INTEGER type specific tag):

```
ACCEPTABLEINTEGER <- 0x02 (
                       0x00 / 0x01 { . } / 0x02 { .. } / 0x03 { ... } / 0x04 { .... }
                     )
```

This definition (which, btw is vanilla LPEG) should take care that no integer encoded as BER will ever overflow your 32-bit system. Because if it does, the Engine will not match the input pattern (*ie* raise the FAIL condition). Which is how it should be.

The problem is with the people who create standards. X.509, for example, defines the second field of the TBSCertificate [1.2.6] compound type, the 'serialNumber' field, as a 20-byte INTEGER. Of course, no one will ever do arithmetic with this number, so it makes more sense to treat it as a string. And in this specific case, presumably, one could make a specific definition for it, like thus:

```
SERIALNUMBER <- 0x02 BERLENGTH <<ruint32:$_:SERNUMCONTENT>>
SERNUMCONTENT <- { .* }
```

However, now you may end up with problems when you do a more generic check of your X.509 certificate (a 'well formedness' check, which tells you that the input is properly encoded BER). You can now no longer distinguish between integers that you want to use intrinsically as integers, and those that are actually strings.

Then again, a pure well-formedness check is not supposed to yield a usable capture list, just a binary answer to the question 'is my input well formed?'.

2.4.2 Matching a Known INTEGER

BER and DER are required to encode INTEGERs in the shortest way possible. X.690 [10] (in §8.3.2) states:

If the contents octets of an integer value encoding consist of more than one octet, then the bits of the first octet and bit 8 of the second octet:

a) shall not all be ones; and

b) shall not all be zero.

NOTE – These rules ensure that an integer value is always encoded in the smallest possible number of octets.

The above makes INTEGER encoding deterministic and therefore, matching a known INTEGER is as simple as formatting it in your grammar as byte literals.

2.4.3 Presenting any INTEGER

Presenting any captured INTEGER is problematic in the same way, more or less, OID subidentifiers are: INTEGER captures will be in binary, but may not be intrinsically usable on your machine. For that, the capture needs to be right shifted to fill up the machine intrinsic integer type, interpreted as network ordered and, if the most significant bit of the capture is set, be interpreted as a negative value. This paper provides no further solution should this be an issue.

2.5 Sub Parsing of Text

Limiting the end-of-input temporarily, has the effect that any sub-rule can simply switch to text parsing from that point onwards. The following (for email parsing purposes extremely simplified) grammar illustrates this (notice that, for clarity, the capture regions of the individual OID subidentifiers have been removed in this example):

```
SEQUENCE       <- SEQUENCETYPE BERLENGTH <<ruint32:$_:SEQUENCEVALUE>>
SEQUENCEVALUE  <- OID EMAIL
BERLENGTH      <- & |00|80| { . } /
                  0x81 { . } / 0x82 { .. } / 0x83 { ... } / 0x84 { .... }

OID            <- OIDTYPE BERLENGTH <<ruint32:$_:OIDVALUE>>
OIDVALUE       <- { ( . ) ( |80|80|* |00|80| )* }

EMAIL          <- IASTRING BERLENGTH <<ruint32:$_:EMAILVALUE>>
EMAILVALUE     <- { USERNAME ’@’ FQDN }
USERNAME       <- { [a-zA-Z0-9.]+ }
FQDN           <- { [a-zA-Z0-9.]+ }

SEQUENCETYPE   <- 0x30
OIDTYPE        <- 0x06
IASTRING       <- 0x16
```

This is fed by the following piece of DER (a list containing an OID and an IA5STRING – taken from a certificate):

```
30 27
   06 09 2a 86 48 86 f7 0d 01 09 01
   16 1a 6b 65 65 73 2e 6a 61 6e 2e 68 65 72 6d 61 6e 73 40 67 6d 61 69 6c 2e 63 6f 6d
```

The code then executes, and captures the email address, as well as its composing portions, as can be seen in the resultant capture list below:

```
End code: 0
7 actions total
Action #0: capture slot 0, 1->1 "’"
Action #1: capture slot 0, 3->1 "\x09"
Action #2: capture slot 5, 4->9 "*\x86H\x86\xf7\x0d\x01\x09\x01"
Action #3: capture slot 0, 14->1 "\x1a"
Action #4: capture slot 6, 15->26 "kees.jan.hermans@gmail.com"
Action #5: capture slot 7, 15->16 "kees.jan.hermans"
Action #6: capture slot 8, 32->9 "gmail.com"
Number of instructions: 155
Max stack depth: 6
```

2.6 ASN.1 Compilation

ASN.1 compilation (to PEG grammar) is mostly out of scope for this paper, but for the following aspects:

- The similarity between the definitions / resulting grammar patterns.
- Pre compiling literals.
- The 'human interpretation' aspect.

2.6.1 ASN.1 Compilation Patterns

PEG grammar patterns can be made pretty similar to the ASN.1 definitions they represent. Given the following ASN.1 example definition:

```
SomeType ::= SEQUENCE {
        member1 SomeSubType,
        member2 SomeOtherSubType
}
```

One can create a PEG grammar that follows it, namespace-wise, as well as structurally, like so (using an imaginary TLV Tag number 0xaa):

```
SOMETYPE_TLV         <- SOMETYPE_TYPE BERLENGTH <<ruint32:$_:SOMETYPE_VALUE>>
SOMETYPE_TYPE        <- 0x30
SOMETYPE_VALUE       <- SOMESUBTYPE_TLV SOMEOTHERSUBTYPE_TLV

SOMESUBTYPE_TLV      <- SOMESUBTYPE_TYPE BERLENGTH <<ruint32:$_:SOMESUBTYPE_VALUE>>
SOMESUBTYPE_TYPE     <- 0xaa
SOMESUBTYPE_VALUE    <- ...

SOMEOTHERSUBTYPE_TLV <- ...
```

It should be easy enough to write a compiler that makes this transformation a generic feature.

2.6.2 Pre Compiling Literals

As noted, when one specifically searches for INTEGERs or OIDs to match – they are encoded deterministically, and can therefore be pre-compiled into their binary form. See [2.3.2] and [2.4.2].

2.6.3 Interpreting less-then-Formal Definitions

Where ASN.1 specifies an 'ANY' type, or where a Tag (TLV type) has been specified for example as a 'context specific class' (0xa0 or 0xa3), and one has no immediate idea what these contain and/or one wants to leave it to the capture processing code to deal with this region (ie the region's layout can have a different structure depending on some condition elsewhere in the input), it's possible to define a 'generic BER grammar' to descend into this region. This is more or less the same as a well-formedness check, but then with captures. For example, like so (obviously incomplete, just note the 'ANY' definition):

```
ANY            <- GENERICLIST / OID / INTEGER / IPV4 / NULL /
                  BSTRING / PSTRING / ISTRING / USTRING / OSTRING /
                  GENERICSET / GCTXSPCLASS / TIMESTAMP /
                  BOOLEAN

GENERICLIST    <- SEQUENCE BERLENGTH <<ruint32:$_:LISTCONTENT>>
GENERICSET     <- SET BERLENGTH <<ruint32:$_:LISTCONTENT>>
GCTXSPCLASS    <- CTXSPCLASS BERLENGTH <<ruint32:$_:LISTCONTENT>>
LISTCONTENT    <- { ANY }* !.

SEQUENCE       <- 0x30
SET            <- 0x31
CTXSPCLASS     <- 0xa3
INTEGER        <- INTEGERTYPE BERLENGTH <<ruint32:$_:INTEGERVALUE>>
INTEGERTYPE    <- 0x02 / 0xa0
INTEGERVALUE   <- { .* }
IPV4           <- 0x40 0x04 { .... }
NULL           <- 0x05 0x00
BITSTRING      <- 0x03
TIMESTAMP      <- 0x17 BERLENGTH <<ruint32:$_:TIMECONTENT>>
TIMECONTENT    <- { .* }
BOOLEAN        <- 0x01 0x01 { . }

PRINTABLESTRING <- 0x13
IASTRING        <- 0x16
UTF8STRING      <- 0x0c
OCTETSTRING     <- 0x04

BSTRING         <- BITSTRING BERLENGTH <<ruint32:$_:STRINGCNT>>
PSTRING         <- PRINTABLESTRING BERLENGTH <<ruint32:$_:STRINGCNT>>
ISTRING         <- IASTRING BERLENGTH <<ruint32:$_:STRINGCNT>>
USTRING         <- UTF8STRING BERLENGTH <<ruint32:$_:STRINGCNT>>
OSTRING         <- OCTETSTRING BERLENGTH <<ruint32:$_:STRINGCNT>>
STRINGCNT       <- { .* }

OID            <- 0x06 BERLENGTH <<ruint32:$_:OIDVALUE>>
OIDVALUE       <- { { . } { |80|80|* |00|80| }* }
```

Bear in mind that using generic parsing as exemplified above, does expose one to the risks of captures that exceed machine intrinsic type sizes (as in the case of INTEGERs, for example, see [2.4.3]).

3 Overview of Changes

3.1 Engine

3.1.1 Bytecode

The proposed bytecode contains one more instruction: the binary representation of ‘intrpcapture’, plus its two parameters: mode and capture slot.

3.1.2 Input

The input is allowed to be BER.

3.1.3 Stack

The stack ‘call’ elements will contain an extra, restorable input length field.

3.1.4 Capture List

The capture list may contain extra entries to hold the length field captures.

3.1.5 Input Length Delimitation Register

A register is introduced, that contains the temporarily delimited input length value.

3.2 Assembly

3.2.1 Maskedchar

The ‘maskedchar’ instruction functions like the ‘char’ instruction, but only for a part of the byte.

3.2.2 Intrpcapture

The ‘intrpcapture’ instruction is introduced. It takes two parameters: mode (only ‘ruint32’ for now) and capture slot (only ‘$_’) for now.

3.3 Grammar

3.3.1 Bitwise Matching

Naigama introduces the concept of bitwise matching, using a special instruction that is parameterized by both the mask and the result of the masking operation.

3.3.2 Scoped Calling

This paper proposes a grammatical grammar construct that parameterizes a call to a rule symbol using a method for converting a capture into an input length limit.

4 Conclusions

Parsing BER formats and presenting the resulting captures to the user requires relatively little effort, namely:

- Addition of bitwise matching by introducing a superset to the normal 'char' instruction, one that has been extended with a bitmask.
- The construction of BER TLV Length field interpretation and consumption as a discrete Engine function.
- Input buffer isolation ('temporary input shortening') of PEG calling contexts, both in assembly instructions (through the addition of the reg_derlength instruction), in grammar definitions (through the introduction of scoped calling), and stack use by the Engine.
- For extra safety, the addition (to PEG) of capture range definitions (optional).

What this provides:

- A way to establish the well-formedness of a BER formatted input.
- A way to capture relevant regions from said input, especially if they can be interpreted as-is (strings, signatures).
- A way to do text sub-parsing in isolated regions of the input.

What is missing / is confined to further study:

- OID subidentifiers and INTEGER values are represented in captures as they are found in the input, requiring bitshifting functions and conditionals to present these as intrinsic or human readable to the user.

References

[1] A Text Pattern-Matching Tool based on Parsing Expression Grammars https://www.inf.puc-rio.br/˜roberto/docs/peg.pdf

[2] Regular Expressions https://en.wikipedia.org/wiki/Regular_expression

[3] Backus Naur Form https://en.wikipedia.org/wiki/Backus-Naur_form

[4] Yacc Yet Another Compiler Compiler https://en.wikipedia.org/wiki/Yacc

[5] Lua PEG http://www.inf.puc-rio.br/˜roberto/lpeg/

[6] Lua Programming Language https://www.lua.org/

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

[8] A Layman's Guide to a Subset of ASN.1, BER, and DER https://luca.ntop.org/Teaching/Appunti/asn1.html

[9] SNMP https://datatracker.ietf.org/doc/html/rfc3413

[10] X.690 https://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf

[11] X.509 https://www.itu.int/rec/T-REC-X.509

[12] Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile https://datatracker.ietf.org/doc/html/rfc5280

[13] https://github.com/kjhermans/naigama

[14] https://www.researchgate.net/publication/316922118_An_Investigation_of_the_Relationships_between_Lines_of_Code_and_Defects

[15] JSON, JavaScript Object Notation https://www.json.org/

Appendices

A Matching an OID Name Value Pair

This section gives a complete overview of the compilation and running of grammar on inputs using the methods described in this paper. The following grammar is defined to match the example in the first chapter of this paper: a SEQUENCE containing an OID and an IPV4ADDRESS:

```
TOP        <- LIST
BERLENGTH  <- & |00|80| { . } /
              0x81 { . } / 0x82 { .. } / 0x83 { ... } / 0x84 { .... }
LIST       <- 0x30 BERLENGTH <<ruint32:$_:OBJECTS>>
OBJECTS    <- OID IPV4
OID        <- 0x06 BERLENGTH <<ruint32:$_:OIDVALUE>>
IPV4       <- 0x40 0x04 { .... }
OIDVALUE   <- { { . } { |80|80|* |00|80| }* }
```

This results in the following assembly:

```
  call TOP
  end 0

__RULE_TOP:
  call LIST
__SUCCESS_1:
  ret -- TOP

__RULE_BERLENGTH:
  catch __ALT_3
  catch __SCANNER_4
  maskedchar 00 80
  backcommit __SCANNER_4_OUT
__SCANNER_4:
  fail
__SCANNER_4_OUT:
  opencapture 0
  any
__SUCCESS_5:
  closecapture 0
  commit __SUCCESS_2
__ALT_3:
  catch __ALT_6
  char 81
  opencapture 1
  any
__SUCCESS_7:
  closecapture 1
  commit __SUCCESS_2
__ALT_6:
  catch __ALT_8
  char 82
  opencapture 2
  any
  any
__SUCCESS_9:
  closecapture 2
  commit __SUCCESS_2
__ALT_8:
  catch __ALT_10
  char 83
  opencapture 3
  any
  any
  any
__SUCCESS_11:
  closecapture 3
  commit __SUCCESS_2
__ALT_10:
  char 84
  opencapture 4
  any
  any
  any
  any
__SUCCESS_12:
  closecapture 4
__SUCCESS_2:
  ret -- BERLENGTH

__RULE_LIST:
  char 30
  call BERLENGTH
  intrpcapture ruint32 default
  call OBJECTS
__SUCCESS_13:
  ret -- LIST

__RULE_OBJECTS:
  call OID
  call IPV4
__SUCCESS_14:
  ret -- OBJECTS

__RULE_OID:
  char 06
  call BERLENGTH
  intrpcapture ruint32 default
  call OIDVALUE
__SUCCESS_15:
  ret -- OID

__RULE_IPV4:
  char 40
  char 04
  opencapture 5
  any
  any
  any
  any
__SUCCESS_17:
  closecapture 5
__SUCCESS_16:
  ret -- IPV4

__RULE_OIDVALUE:
  opencapture 6
  opencapture 7
  any
__SUCCESS_20:
  closecapture 7
  catch __FORGIVE_21
__FOREVER_22:
  opencapture 8
  catch __FORGIVE_24
__FOREVER_25:
  maskedchar 80 80
  partialcommit __FOREVER_25
__FORGIVE_24:
  maskedchar 00 80
__SUCCESS_23:
  closecapture 8
  partialcommit __FOREVER_22
__FORGIVE_21:
__SUCCESS_19:
  closecapture 6
__SUCCESS_18:
  ret -- OIDVALUE

  end 0
```

Given the following input (in hexadecimal):

```
30 18 06 10 2B 06 01 04 01 81 E0 6B 02 02 06 01 06 03 01 01 40 04 C0 A8 50 01
```

The following is the output of the engine in debug mode (reduced font size to fit page horizontally):

```
Processing 26 bytes of input
000001          CALL bc 000 in 00 301806102b060104 st (000 prec.)
            __RULE_TOP:
000002          CALL bc 016 in 00 301806102b060104 st (000 prec.) CLL:8
            __RULE_LIST:
000003          CHAR bc 284 in 00 301806102b060104 st (000 prec.) CLL:8 CLL:24
000004          CALL bc 292 in 01 1806102b06010401 st (000 prec.) CLL:8 CLL:24
            __RULE_BERLENGTH:
000005         CATCH bc 028 in 01 1806102b06010401 st (000 prec.) CLL:8 CLL:24 CLL:300
000006         CATCH bc 036 in 01 1806102b06010401 st (003 prec.) ALT:96
000007    MASKEDCHAR bc 044 in 01 1806102b06010401 st (004 prec.) ALT:64
000008    BACKCOMMIT bc 056 in 02 06102b0601040181 st (004 prec.) ALT:64
            __SCANNER_4_OUT:
000009   OPENCAPTURE bc 068 in 01 1806102b06010401 st (003 prec.) ALT:96
000010           ANY bc 076 in 01 1806102b06010401 st (003 prec.) ALT:96
            __SUCCESS_5:
000011  CLOSECAPTURE bc 080 in 02 06102b0601040181 st (003 prec.) ALT:96
000012        COMMIT bc 088 in 02 06102b0601040181 st (003 prec.) ALT:96
            __SUCCESS_2:
000013           RET bc 280 in 02 06102b0601040181 st (000 prec.) CLL:8 CLL:24 CLL:300
000014  INTRPCAPTURE bc 300 in 02 06102b0601040181 st (000 prec.) CLL:8 CLL:24
000015          CALL bc 312 in 02 06102b0601040181 st (000 prec.) CLL:8 CLL:24
            __RULE_OBJECTS:
000016          CALL bc 324 in 02 06102b0601040181 st (000 prec.) CLL:8 CLL:24 CLL:320
```

```
            __RULE_OID:
000017          CHAR bc 344 in 02 06102b0601040181 st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:332
000018          CALL bc 352 in 03 102b0601040181e0 st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:332
            __RULE_BERLENGTH:
000019         CATCH bc 028 in 03 102b0601040181e0 st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:332 CLL:360
000020         CATCH bc 036 in 03 102b0601040181e0 st (005 prec.) ALT:96
000021    MASKEDCHAR bc 044 in 03 102b0601040181e0 st (006 prec.) ALT:64
000022    BACKCOMMIT bc 056 in 04 2b0601040181e06b st (006 prec.) ALT:64
            __SCANNER_4_OUT:
000023   OPENCAPTURE bc 068 in 03 102b0601040181e0 st (005 prec.) ALT:96
000024           ANY bc 076 in 03 102b0601040181e0 st (005 prec.) ALT:96
            __SUCCESS_5:
000025  CLOSECAPTURE bc 080 in 04 2b0601040181e06b st (005 prec.) ALT:96
000026        COMMIT bc 088 in 04 2b0601040181e06b st (005 prec.) ALT:96
            __SUCCESS_2:
000027           RET bc 280 in 04 2b0601040181e06b st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:332 CLL:360
000028  INTRPCAPTURE bc 360 in 04 2b0601040181e06b st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:332
000029          CALL bc 372 in 04 2b0601040181e06b st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:332
            __RULE_OIDVALUE:
000030   OPENCAPTURE bc 436 in 04 2b0601040181e06b st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:332 CLL:380
000031   OPENCAPTURE bc 444 in 04 2b0601040181e06b st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:332 CLL:380
000032           ANY bc 452 in 04 2b0601040181e06b st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:332 CLL:380
            __SUCCESS_20:
000033  CLOSECAPTURE bc 456 in 05 0601040181e06b02 st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:332 CLL:380
000034         CATCH bc 464 in 05 0601040181e06b02 st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:332 CLL:380
            __FOREVER_22:
000035   OPENCAPTURE bc 472 in 05 0601040181e06b02 st (005 prec.) ALT:536
000036         CATCH bc 480 in 05 0601040181e06b02 st (005 prec.) ALT:536
            __FOREVER_25:
000037    MASKEDCHAR bc 488 in 05 0601040181e06b02 st (006 prec.) ALT:508
======== FAIL
            __FORGIVE_24:
000038    MASKEDCHAR bc 508 in 05 0601040181e06b02 st (005 prec.) ALT:536
            __SUCCESS_23:
000039  CLOSECAPTURE bc 520 in 06 01040181e06b0202 st (005 prec.) ALT:536
000040 PARTIALCOMMIT bc 528 in 06 01040181e06b0202 st (005 prec.) ALT:536
            __FOREVER_22:
000041   OPENCAPTURE bc 472 in 06 01040181e06b0202 st (005 prec.) ALT:536
000042         CATCH bc 480 in 06 01040181e06b0202 st (005 prec.) ALT:536
            __FOREVER_25:
000043    MASKEDCHAR bc 488 in 06 01040181e06b0202 st (006 prec.) ALT:508
======== FAIL
            __FORGIVE_24:
000044    MASKEDCHAR bc 508 in 06 01040181e06b0202 st (005 prec.) ALT:536
            __SUCCESS_23:
000045  CLOSECAPTURE bc 520 in 07 040181e06b020206 st (005 prec.) ALT:536
000046 PARTIALCOMMIT bc 528 in 07 040181e06b020206 st (005 prec.) ALT:536
            __FOREVER_22:
000047   OPENCAPTURE bc 472 in 07 040181e06b020206 st (005 prec.) ALT:536
000048         CATCH bc 480 in 07 040181e06b020206 st (005 prec.) ALT:536
            __FOREVER_25:
000049    MASKEDCHAR bc 488 in 07 040181e06b020206 st (006 prec.) ALT:508
======== FAIL
            __FORGIVE_24:
000050    MASKEDCHAR bc 508 in 07 040181e06b020206 st (005 prec.) ALT:536
            __SUCCESS_23:
000051  CLOSECAPTURE bc 520 in 08 0181e06b02020601 st (005 prec.) ALT:536
000052 PARTIALCOMMIT bc 528 in 08 0181e06b02020601 st (005 prec.) ALT:536
            __FOREVER_22:
000053   OPENCAPTURE bc 472 in 08 0181e06b02020601 st (005 prec.) ALT:536
000054         CATCH bc 480 in 08 0181e06b02020601 st (005 prec.) ALT:536
            __FOREVER_25:
000055    MASKEDCHAR bc 488 in 08 0181e06b02020601 st (006 prec.) ALT:508
======== FAIL
            __FORGIVE_24:
000056    MASKEDCHAR bc 508 in 08 0181e06b02020601 st (005 prec.) ALT:536
            __SUCCESS_23:
000057  CLOSECAPTURE bc 520 in 09 81e06b0202060106 st (005 prec.) ALT:536
000058 PARTIALCOMMIT bc 528 in 09 81e06b0202060106 st (005 prec.) ALT:536
            __FOREVER_22:
000059   OPENCAPTURE bc 472 in 09 81e06b0202060106 st (005 prec.) ALT:536
000060         CATCH bc 480 in 09 81e06b0202060106 st (005 prec.) ALT:536
            __FOREVER_25:
000061    MASKEDCHAR bc 488 in 09 81e06b0202060106 st (006 prec.) ALT:508
000062 PARTIALCOMMIT bc 500 in 10 e06b020206010603 st (006 prec.) ALT:508
            __FOREVER_25:
000063    MASKEDCHAR bc 488 in 10 e06b020206010603 st (006 prec.) ALT:508
000064 PARTIALCOMMIT bc 500 in 11 6b02020601060301 st (006 prec.) ALT:508
            __FOREVER_25:
000065    MASKEDCHAR bc 488 in 11 6b02020601060301 st (006 prec.) ALT:508
======== FAIL
            __FORGIVE_24:
```

```
000066    MASKEDCHAR bc 508 in 11 6b02020601060301 st (005 prec.) ALT:536
            __SUCCESS_23:
000067  CLOSECAPTURE bc 520 in 12 0202060106030101 st (005 prec.) ALT:536
000068 PARTIALCOMMIT bc 528 in 12 0202060106030101 st (005 prec.) ALT:536
            __FOREVER_22:
000069   OPENCAPTURE bc 472 in 12 0202060106030101 st (005 prec.) ALT:536
000070         CATCH bc 480 in 12 0202060106030101 st (005 prec.) ALT:536
            __FOREVER_25:
000071    MASKEDCHAR bc 488 in 12 0202060106030101 st (006 prec.) ALT:508
======== FAIL
            __FORGIVE_24:
000072    MASKEDCHAR bc 508 in 12 0202060106030101 st (005 prec.) ALT:536
            __SUCCESS_23:
000073  CLOSECAPTURE bc 520 in 13 02060106030101__ st (005 prec.) ALT:536
000074 PARTIALCOMMIT bc 528 in 13 02060106030101__ st (005 prec.) ALT:536
            __FOREVER_22:
000075   OPENCAPTURE bc 472 in 13 02060106030101__ st (005 prec.) ALT:536
000076         CATCH bc 480 in 13 02060106030101__ st (005 prec.) ALT:536
            __FOREVER_25:
000077    MASKEDCHAR bc 488 in 13 02060106030101__ st (006 prec.) ALT:508
======== FAIL
            __FORGIVE_24:
000078    MASKEDCHAR bc 508 in 13 02060106030101__ st (005 prec.) ALT:536
            __SUCCESS_23:
000079  CLOSECAPTURE bc 520 in 14 060106030101____ st (005 prec.) ALT:536
000080 PARTIALCOMMIT bc 528 in 14 060106030101____ st (005 prec.) ALT:536
            __FOREVER_22:
000081   OPENCAPTURE bc 472 in 14 060106030101____ st (005 prec.) ALT:536
000082         CATCH bc 480 in 14 060106030101____ st (005 prec.) ALT:536
            __FOREVER_25:
000083    MASKEDCHAR bc 488 in 14 060106030101____ st (006 prec.) ALT:508
======== FAIL
            __FORGIVE_24:
000084    MASKEDCHAR bc 508 in 14 060106030101____ st (005 prec.) ALT:536
            __SUCCESS_23:
000085  CLOSECAPTURE bc 520 in 15 0106030101______ st (005 prec.) ALT:536
000086 PARTIALCOMMIT bc 528 in 15 0106030101______ st (005 prec.) ALT:536
            __FOREVER_22:
000087   OPENCAPTURE bc 472 in 15 0106030101______ st (005 prec.) ALT:536
000088         CATCH bc 480 in 15 0106030101______ st (005 prec.) ALT:536
            __FOREVER_25:
000089    MASKEDCHAR bc 488 in 15 0106030101______ st (006 prec.) ALT:508
======== FAIL
            __FORGIVE_24:
000090    MASKEDCHAR bc 508 in 15 0106030101______ st (005 prec.) ALT:536
            __SUCCESS_23:
000091  CLOSECAPTURE bc 520 in 16 06030101________ st (005 prec.) ALT:536
000092 PARTIALCOMMIT bc 528 in 16 06030101________ st (005 prec.) ALT:536
            __FOREVER_22:
000093   OPENCAPTURE bc 472 in 16 06030101________ st (005 prec.) ALT:536
000094         CATCH bc 480 in 16 06030101________ st (005 prec.) ALT:536
            __FOREVER_25:
000095    MASKEDCHAR bc 488 in 16 06030101________ st (006 prec.) ALT:508
======== FAIL
            __FORGIVE_24:
000096    MASKEDCHAR bc 508 in 16 06030101________ st (005 prec.) ALT:536
            __SUCCESS_23:
000097  CLOSECAPTURE bc 520 in 17 030101__________ st (005 prec.) ALT:536
000098 PARTIALCOMMIT bc 528 in 17 030101__________ st (005 prec.) ALT:536
            __FOREVER_22:
000099   OPENCAPTURE bc 472 in 17 030101__________ st (005 prec.) ALT:536
000100         CATCH bc 480 in 17 030101__________ st (005 prec.) ALT:536
            __FOREVER_25:
000101    MASKEDCHAR bc 488 in 17 030101__________ st (006 prec.) ALT:508
======== FAIL
            __FORGIVE_24:
000102    MASKEDCHAR bc 508 in 17 030101__________ st (005 prec.) ALT:536
            __SUCCESS_23:
000103  CLOSECAPTURE bc 520 in 18 0101____________ st (005 prec.) ALT:536
000104 PARTIALCOMMIT bc 528 in 18 0101____________ st (005 prec.) ALT:536
            __FOREVER_22:
000105   OPENCAPTURE bc 472 in 18 0101____________ st (005 prec.) ALT:536
000106         CATCH bc 480 in 18 0101____________ st (005 prec.) ALT:536
            __FOREVER_25:
000107    MASKEDCHAR bc 488 in 18 0101____________ st (006 prec.) ALT:508
======== FAIL
            __FORGIVE_24:
000108    MASKEDCHAR bc 508 in 18 0101____________ st (005 prec.) ALT:536
            __SUCCESS_23:
000109  CLOSECAPTURE bc 520 in 19 01______________ st (005 prec.) ALT:536
000110 PARTIALCOMMIT bc 528 in 19 01______________ st (005 prec.) ALT:536
            __FOREVER_22:
```

```
000111   OPENCAPTURE bc 472 in 19 01______________ st (005 prec.) ALT:536
000112         CATCH bc 480 in 19 01______________ st (005 prec.) ALT:536
            __FOREVER_25:
000113    MASKEDCHAR bc 488 in 19 01______________ st (006 prec.) ALT:508
======== FAIL
            __FORGIVE_24:
000114    MASKEDCHAR bc 508 in 19 01______________ st (005 prec.) ALT:536
            __SUCCESS_23:
000115  CLOSECAPTURE bc 520 in 20 ________________ st (005 prec.) ALT:536
000116 PARTIALCOMMIT bc 528 in 20 ________________ st (005 prec.) ALT:536
            __FOREVER_22:
000117   OPENCAPTURE bc 472 in 20 ________________ st (005 prec.) ALT:536
000118         CATCH bc 480 in 20 ________________ st (005 prec.) ALT:536
            __FOREVER_25:
000119    MASKEDCHAR bc 488 in 20 ________________ st (006 prec.) ALT:508
======== FAIL
            __FORGIVE_24:
000120    MASKEDCHAR bc 508 in 20 ________________ st (005 prec.) ALT:536
======== FAIL
            __FORGIVE_21:
            __SUCCESS_19:
000121  CLOSECAPTURE bc 536 in 20 ________________ st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:332 CLL:380
            __SUCCESS_18:
000122           RET bc 544 in 20 ________________ st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:332 CLL:380
            __SUCCESS_15:
000123           RET bc 380 in 20 4004c0a85001____ st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:332
000124          CALL bc 332 in 20 4004c0a85001____ st (000 prec.) CLL:8 CLL:24 CLL:320
            __RULE_IPV4:
000125          CHAR bc 384 in 20 4004c0a85001____ st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:340
000126          CHAR bc 392 in 21 04c0a85001______ st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:340
000127   OPENCAPTURE bc 400 in 22 c0a85001________ st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:340
000128           ANY bc 408 in 22 c0a85001________ st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:340
000129           ANY bc 412 in 23 a85001__________ st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:340
000130           ANY bc 416 in 24 5001____________ st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:340
000131           ANY bc 420 in 25 01______________ st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:340
            __SUCCESS_17:
000132  CLOSECAPTURE bc 424 in 26 ________________ st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:340
            __SUCCESS_16:
000133           RET bc 432 in 26 ________________ st (000 prec.) CLL:8 CLL:24 CLL:320 CLL:340
            __SUCCESS_14:
000134           RET bc 340 in 26 ________________ st (000 prec.) CLL:8 CLL:24 CLL:320
            __SUCCESS_13:
000135           RET bc 320 in 26 ________________ st (000 prec.) CLL:8 CLL:24
            __SUCCESS_1:
000136           RET bc 024 in 26 ________________ st (000 prec.) CLL:8
000137           END bc 008 in 26 ________________ st (000 prec.)
End code: 0
18 actions total
Action #0: capture slot 0, 1->1 "\x18"
Action #1: capture slot 0, 3->1 "\x10"
Action #2: capture slot 6, 4->16 "+\x06\x01\x04\x01\x81\xe0k\x02\x02\x06\x01\x06\x03\x01\x01"
Action #3: capture slot 7, 4->1 "+"
Action #4: capture slot 8, 5->1 "\x06"
Action #5: capture slot 8, 6->1 "\x01"
Action #6: capture slot 8, 7->1 "\x04"
Action #7: capture slot 8, 8->1 "\x01"
Action #8: capture slot 8, 9->3 "\x81\xe0k"
Action #9: capture slot 8, 12->1 "\x02"
Action #10: capture slot 8, 13->1 "\x02"
Action #11: capture slot 8, 14->1 "\x06"
Action #12: capture slot 8, 15->1 "\x01"
Action #13: capture slot 8, 16->1 "\x06"
Action #14: capture slot 8, 17->1 "\x03"
Action #15: capture slot 8, 18->1 "\x01"
Action #16: capture slot 8, 19->1 "\x01"
Action #17: capture slot 5, 22->4 "\xc0\xa8P\x01"
Number of instructions: 137
Max stack depth: 7
```

B SNMPv3 DER Example

Split out, our hexadecimal SNMPv3 payload example looks like this:

```
30 81 E2
  02 01 03
  30 12
    02 04 58 08 EE 29
    02 04 7F FF FF FF
    04 01 04
    02 01 03
  04 25
    30 23
      04 10 80 00 70 7F 40 53 6B 79 54 61 6C 65 00 00 04 D2
      02 01 00
      02 03 12 41 29
      04 03 66 6F 6F
      04 00
      04 00
  30 81 A1
    04 10 80 00 70 7F 40 53 6B 79 54 61 6C 65 00 00 04 D2
    04 00
    A3 81 8A
      02 04 23 A8 57 69
      02 01 00
      02 01 00
      30 7C
        30 18
          06 10 2B 06 01 04 01 81 E0 6B 02 02 06 01 06 03 01 01
          40 04 C0 A8 50 01
        30 18
          06 10 2B 06 01 04 01 81 E0 6B 02 02 06 01 06 03 01 02
          40 04 C0 A8 50 00
        30 18
          06 10 2B 06 01 04 01 81 E0 6B 02 02 06 01 06 03 01 03
          40 04 FF FF FF 00
        30 15
          06 10 2B 06 01 04 01 81 E0 6B 02 02 06 01 06 03 01 04
          42 01 64
        30 15
          06 10 2B 06 01 04 01 81 E0 6B 02 02 06 01 06 03 01 05
          02 01 04
```

We try to parse this, using a shorter, generic BER grammar definition:

```
SNMPV3         <- GENERICLIST

BERLENGTH      <- & |00|80| { . } /
                  0x81 { . } / 0x82 { .. } / 0x83 { ...  } / 0x84 { .... }

ANY            <- GENERICLIST / OID / INTEGER / IPV4 / NULL /
                  BSTRING / PSTRING / ISTRING / USTRING / OSTRING /
                  GENERICSET / GCTXSPCLASS / TIMESTAMP /
                  BOOLEAN / GINTEGER

GENERICLIST    <- SEQUENCE BERLENGTH <<ruint32:$_:LISTCONTENT>>
```

```
GENERICSET     <- SET BERLENGTH <<ruint32:$_:LISTCONTENT>>
GCTXSPCLASS    <- CTXSPCLASS BERLENGTH <<ruint32:$_:LISTCONTENT>>
LISTCONTENT    <- { ANY }* !.

SEQUENCE       <- 0x30
SET            <- 0x31
CTXSPCLASS     <- 0xa3
INTEGER        <- INTEGERTYPE BERLENGTH <<ruint32:$_:INTEGERVALUE>>
INTEGERTYPE    <- 0x02 / 0xa0
INTEGERVALUE   <- { .* }
IPV4           <- 0x40 0x04 { .... } !.
NULL           <- 0x05 0x00
BITSTRING      <- 0x03
TIMESTAMP      <- 0x17 BERLENGTH <<ruint32:$_:TIMECONTENT>>
TIMECONTENT    <- { .* }
BOOLEAN        <- 0x01 0x01 { . } !.
GAUGE32        <- 0x42
GINTEGER       <- GAUGE32 BERLENGTH <<ruint32:$_:INTEGERVALUE>>

PRINTABLESTRING <- 0x13
IASTRING        <- 0x16
UTF8STRING      <- 0x0c
OCTETSTRING     <- 0x04

BSTRING         <- BITSTRING BERLENGTH <<ruint32:$_:STRINGCNT>>
PSTRING         <- PRINTABLESTRING BERLENGTH <<ruint32:$_:STRINGCNT>>
ISTRING         <- IASTRING BERLENGTH <<ruint32:$_:STRINGCNT>>
USTRING         <- UTF8STRING BERLENGTH <<ruint32:$_:STRINGCNT>>
OSTRING         <- OCTETSTRING BERLENGTH <<ruint32:$_:STRINGCNT>>
STRINGCNT       <- { .* }

OID            <- 0x06 BERLENGTH <<ruint32:$_:OIDVALUE>>
OIDVALUE       <- { { . } { |80|80|* |00|80| }* } !.
```

Without having to list all the captures, it suffices to say that Naigama successfully parses this input, using 2390 instructions, and capturing 149 regions from the input.

C Parsing a Certificate

I created a small self-signed certificate, by issueing:

```
$ openssl genrsa  -out myCA.key 1024
$ openssl req -x509 -new -nodes -key myCA.key -sha256 -days 1825 -out
myCA.pem
```

It contained the following ASCII text:

```
-----BEGIN CERTIFICATE-----
MIIDDDCCAnWgAwIBAgIUKn7OCa82Nnjj0fp4iah8zHtxxpgwDQYJKoZIhvcNAQEL
BQAwgZcxCzAJBgNVBAYTAk5MMQswCQYDVQQIDAJVVDEQMA4GA1UEBwwHTGVlcmRh
bTEOMAwGA1UECgwFTXlvcmcxEzARBgNVBAsMClRoZXNlY3Rpb24xGTAXBgNVBAMM
EEtlZXMtSmFuIEhlcm1hbnMxKTAnBgkqhkiG9w0BCQEWGmtlZXMuamFuLmhlcm1h
bnNAZ21haWwuY29tMB4XDTIxMDkwNTA5MDIwMVoXDTI2MDkwNDA5MDIwMVowgZcx
CzAJBgNVBAYTAk5MMQswCQYDVQQIDAJVVDEQMA4GA1UEBwwHTGVlcmRhbTEOMAwG
A1UECgwFTXlvcmcxEzARBgNVBAsMClRoZXNlY3Rpb24xGTAXBgNVBAMMEEtlZXMt
SmFuIEhlcm1hbnMxKTAnBgkqhkiG9w0BCQEWGmtlZXMuamFuLmhlcm1hbnNAZ21h
aWwuY29tMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC28xfleEOTI3grHidy
Jm1054Oa8fNCP6FnhCHVTn+Z7fQSaD2KAJj7w6hGIsFN9F0pPxAQWf3qwQfWjzHO
HnHfgJlQ2tFdqTNrTZM+jKtEaDQonSNKfG73qIoji0BrgxrivkrVidd8/hI5WLL+
NZ53hqvzrvJfUxcMIk49PXeLfwIDAQABo1MwUTAdBgNVHQ4EFgQUEFQ9hHX7QGyp
+azLvXfpqc/9PH4wHwYDVR0jBBgwFoAUEFQ9hHX7QGyp+azLvXfpqc/9PH4wDwYD
VR0TAQH/BAUwAwEB/zANBgkqhkiG9w0BAQsFAAOBgQBFg9sfb6POW+ecQEE7JoUx
4njchahJf+5ofRHHQusiIz3/Omb3lcHJUsOVa1VzbFwKyYN5iTQ/Doa8FDhSuel+
trPg3HASgvqHzjgpQKL7IaUQdYqhbWcI2trqX4OnNyl7m1G+PGgvrjJ3ZmdlMPY2
KUmgO3iaaX1JNRh96N87bg==
-----END CERTIFICATE-----
```

To get the hexadecimal content of the file, I issued:

```
$ cat myCA.pem | tail -n +2 | head -n -1 | base64 --decode | xxd -p
```

This results in hexadecimal:

```
3082030c30820275a00302010202142a7ece09af363678e3d1fa7889a87c
cc7b71c698300d06092a864886f70d01010b0500308197310b3009060355
040613024e4c310b300906035504080c0255543110300e06035504070c07
4c65657264616d310e300c060355040a0c054d796f726731133011060355
040b0c0a54686573656374696f6e311930170603550403 0c104b6565732d
4a616e204865726d616e733129302706092a864886f70d010901161a6b65
65732e6a616e2e6865726d616e7340676d61696c2e636f6d301e170d3231
303930353039303230315a170d3236303930343039303230315a30819731
0b3009060355040613024e4c310b300906035504080c0255543110300e06
035504070c074c65657264616d310e300c060355040a0c054d796f726731
133011060355040b0c0a54686573656374696f6e31193017060355040 30c
104b6565732d4a616e204865726d616e733129302706092a864886f70d01
0901161a6b6565732e6a616e2e6865726d616e7340676d61696c2e636f6d
30819f300d06092a864886f70d010101050003818d0030818902818100b6
f317e578439323782b1e2772266d74e7839af1f3423fa1678421d54e7f99
edf412683d8a0098fbc3a84622c14df45d293f101059fdeac107d68f31ce
1e71df809950dad15da9336b4d933e8cab446834289d234a7c6ef7a88a23
8b406b831ae2be4ad589d77cfe123958b2fe359e7786abf3aef25f53170c
224e3d3d778b7f0203010001a3533051301d0603551d0e041604141054 3d
8475fb406ca9f9accbbd77e9a9cffd3c7e301f0603551d23041830168014
```

```
10543d8475fb406ca9f9accbbd77e9a9cffd3c7e300f0603551d130101ff
040530030101ff300d06092a864886f70d01010b0500038181004583db1f
6fa3ce5be79c40413b268531e278dc85a8497fee687d11c742eb22233dff
3a66f795c1c952c3956b55736c5c0ac9837989343f0e86bc143852b9e97e
b6b3e0dc701282fa87ce382940a2fb21a510758aa16d6708dadaea5f83a7
37297b9b51be3c682fae327766676530f6362949a03b789a697d4935187d
e8df3b6e
```

Which can be split out as follows:

```
30 82 03 0c -- Certificate ::= SEQUENCE
  30 82 02 75 -- tbsCertificate TBSCertificate ::= SEQUENCE
    a0 03 02 01 02 -- version
    02 14 2a 7e ce 09 af 36 36 78 e3 d1 fa 78 89 a8 7c cc 7b 71 c6 98 -- serial#
    30 0d -- signature
      06 09 2a 86 48 86 f7 0d 01 01 0b
      05 00
    30 81 97 -- issuer
      31 0b
        30 09
          06 03 55 04 06
          13 02 4e 4c
      31 0b
        30 09
          06 03 55 04 08
          0c 02 55 54
      31 10
        30 0e
          06 03 55 04 07
          0c 07 4c 65 65 72 64 61 6d
      31 0e
        30 0c
          06 03 55 04 0a
          0c 05 4d 79 6f 72 67
      31 13
        30 11
          06 03 55 04 0b
          0c 0a 54 68 65 73 65 63 74 69 6f 6e
      31 19
        30 17
          06 03 55 04 03
          0c 10 4b 65 65 73 2d 4a 61 6e 20 48 65 72 6d 61 6e 73
      31 29
        30 27
          06 09 2a 86 48 86 f7 0d 01 09 01
          16 1a 6b 65 65 73 2e 6a 61 6e 2e 68 65 72 6d 61 6e 73 40 67 6d 61
                69 6c 2e 63 6f 6d
    30 1e -- validity
      17 0d 32 31 30 39 30 35 30 39 30 32 30 31 5a
      17 0d 32 36 30 39 30 34 30 39 30 32 30 31 5a
    30 81 97 -- subject
      31 0b
        30 09
          06 03 55 04 06
          13 02 4e 4c
      31 0b
        30 09
          06 03 55 04 08
          0c 02 55 54
      31 10
        30 0e
          06 03 55 04 07
          0c 07 4c 65 65 72 64 61 6d
      31 0e
        30 0c
          06 03 55 04 0a
          0c 05 4d 79 6f 72 67
      31 13
        30 11
          06 03 55 04 0b
          0c 0a 54 68 65 73 65 63 74 69 6f 6e
      31 19
        30 17
          06 03 55 04 03
          0c 10 4b 65 65 73 2d 4a 61 6e 20 48 65 72 6d 61 6e 73
      31 29
        30 27
          06 09 2a 86 48 86 f7 0d 01 09 01
          16 1a 6b 65 65 73 2e 6a 61 6e 2e 68 65 72 6d 61 6e 73 40 67 6d 61
                69 6c 2e 63 6f 6d
    30 81 9f -- subjectPublicKeyInfo
      30 0d
        06 09 2a 86 48 86 f7 0d 01 01 01
        05 00
      03 81 8d -- bit string BER encoding two INTEGERs
            0030818902818100b6f317e578439323
            782b1e2772266d74e7839af1f3423fa1
            678421d54e7f99edf412683d8a0098fb
            c3a84622c14df45d293f101059fdeac1
            07d68f31ce1e71df809950dad15da933
            6b4d933e8cab446834289d234a7c6ef7
            a88a238b406b831ae2be4ad589d77cfe
            123958b2fe359e7786abf3aef25f5317
            0c224e3d3d778b7f0203010001
    a3 53 -- context specific class / issuerUniqueID
      30 51
        30 1d
          06 03 55 1d 0e
          04 16 04 14 10 54 3d 84 75 fb 40 6c a9 f9 ac cb bd 77 e9 a9 cf
                fd 3c 7e
        30 1f
          06 03 55 1d 23
          04 18 30 16 80 14 10 54 3d 84 75 fb 40 6c a9 f9 ac cb bd 77 e9
                a9 cf fd 3c 7e
        30 0f
          06 03 55 1d 13
          01 01 ff
          04 05 30 03 01 01 ff
  30 0d -- signatureAlgorithm
    06 09 2a 86 48 86 f7 0d 01 01 0b
    05 00
  03 81 81 -- signatureValue (bit string)
        004583db1f6fa3ce5be79c40413b2685
        31e278dc85a8497fee687d11c742eb22
        233dff3a66f795c1c952c3956b55736c
        5c0ac9837989343f0e86bc143852b9e9
        7eb6b3e0dc701282fa87ce382940a2fb
        21a510758aa16d6708dadaea5f83a737
        297b9b51be3c682fae327766676530f6
        362949a03b789a697d4935187de8df3b
        6e
```

I then created the following Naigama PEG grammar:

```
CERTIFICATE    <- SEQUENCE BERLENGTH <<ruint32:$_:CERTCONTENT>>

BERLENGTH      <- & |00|80| { . } /
                  0x81 { . } / 0x82 { .. } / 0x83 { ...  } / 0x84 { .... }

CERTCONTENT    <- TBSCERTIFICATE
                  SIGNATUREALGORITHM
                  SIGNATUREVALUE
TBSCERTIFICATE <- SEQUENCE BERLENGTH <<ruint32:$_:TBSCERTCONTENT>>
TBSCERTCONTENT <- VERSION
                  SERIALNUMBER
                  SIGNATURE
                  ISSUER
                  VALIDITY
                  SUBJECT
                  SUBJECTPUBKEYINFO
                  ISSUERUNIQUEID ?
VERSION        <- INTEGER
SERIALNUMBER   <- INTEGER
SIGNATURE      <- SEQUENCE BERLENGTH <<ruint32:$_:ALGIDENTCONT>>
ALGIDENTCONT   <- ALGORITHM
                  PARAMETERS ?
ALGORITHM      <- { OID }
PARAMETERS     <- ANY
ISSUER         <- SEQUENCE BERLENGTH <<ruint32:$_:ISSUERCONTENT>>
ISSUERCONTENT  <- { ISSUERNV }*
ISSUERNV       <- SET BERLENGTH <<ruint32:$_:ISSUERNV_>>
ISSUERNV_      <- SEQUENCE BERLENGTH <<ruint32:$_:ISSUERNV__>>
ISSUERNV__     <- ISSUERNAME ISSUERVALUE
ISSUERNAME     <- { OID }
ISSUERVALUE    <- { ANY }
SIGNATUREALGORITHM <- SEQUENCE BERLENGTH <<ruint32:$_:SIGALGCONTENT>>
SIGALGCONTENT  <- OID ANY ?
SIGNATUREVALUE <- BITSTRING BERLENGTH <<ruint32:$_:SIGVALCONTENT>>
SIGVALCONTENT  <- { .* }
VALIDITY       <- SEQUENCE BERLENGTH <<ruint32:$_:VALIDITYCONTENT>>
VALIDITYCONTENT <- VALIDFROM VALIDUNTIL
VALIDFROM      <- TIMESTAMP
VALIDUNTIL     <- TIMESTAMP
SUBJECT        <- SEQUENCE BERLENGTH <<ruint32:$_:SUBJECTCONTENT>>
SUBJECTCONTENT <- SUBJENTRY*
```

```
SUBJENTRY      <- SET BERLENGTH <<ruint32:$_:SUBJENTRYNV_>>
SUBJENTRYNV_   <- SEQUENCE BERLENGTH <<ruint32:$_:SUBJENTRYNV__>>
SUBJENTRYNV__  <- SUBJENTRYNAME SUBJENTRYVALUE
SUBJENTRYNAME  <- { OID }
SUBJENTRYVALUE <- { ANY }
SUBJECTPUBKEYINFO <- SEQUENCE BERLENGTH <<ruint32:$_:SPKICONTENT>>
SPKICONTENT    <- { ANY }*
ISSUERUNIQUEID <- CTXSPCLASS BERLENGTH <<ruint32:$_:ISSUERUIDCONTENT>>
ISSUERUIDCONTENT <- { ANY }*

ANY            <- GENERICLIST / OID / INTEGER / IPV4 / NULL /
                  BSTRING / PSTRING / ISTRING / USTRING / OSTRING /
                  GENERICSET / GCTXSPCLASS / TIMESTAMP /
                  BOOLEAN

GENERICLIST    <- SEQUENCE BERLENGTH <<ruint32:$_:LISTCONTENT>>
GENERICSET     <- SET BERLENGTH <<ruint32:$_:LISTCONTENT>>
GCTXSPCLASS    <- CTXSPCLASS BERLENGTH <<ruint32:$_:LISTCONTENT>>
LISTCONTENT    <- { ANY }*

SEQUENCE       <- 0x30
SET            <- 0x31
CTXSPCLASS     <- 0xa3
INTEGER        <- INTEGERTYPE BERLENGTH <<ruint32:$_:INTEGERVALUE>>
INTEGERTYPE    <- 0x02 / 0xa0
INTEGERVALUE   <- { .* }
IPV4           <- 0x40 0x04 { .... }
NULL           <- 0x05 0x00
BITSTRING      <- 0x03
TIMESTAMP      <- 0x17 BERLENGTH <<ruint32:$_:TIMECONTENT>>
TIMECONTENT    <- { .* }
BOOLEAN        <- 0x01 0x01 { . }

PRINTABLESTRING <- 0x13
IASTRING        <- 0x16
UTF8STRING      <- 0x0c
OCTETSTRING     <- 0x04

BSTRING         <- BITSTRING BERLENGTH <<ruint32:$_:STRINGCNT>>
PSTRING         <- PRINTABLESTRING BERLENGTH <<ruint32:$_:STRINGCNT>>
ISTRING         <- IASTRING BERLENGTH <<ruint32:$_:STRINGCNT>>
USTRING         <- UTF8STRING BERLENGTH <<ruint32:$_:STRINGCNT>>
OSTRING         <- OCTETSTRING BERLENGTH <<ruint32:$_:STRINGCNT>>
STRINGCNT       <- { .* }

OID            <- 0x06 BERLENGTH <<ruint32:$_:OIDVALUE>>
OIDVALUE       <- { { . } { |80|80|* |00|80| }* }
```

This is, other than the SNMPv3 example, which used a generic BER 'well formedness' check including captures, a specific, ASN.1 definition mirroring grammar. The result, in Naigama, is a successful match, having executed 4533 instructions and resulting in 255 capture regions in 23 uniquely typed slots.

D Parsing a Certificate for its Signature

The grammar below will extract the signature value, along with the OID that specifies the signature type (including its parameter) from a certificate.

```
CERTIFICATE    <- SEQUENCE BERLENGTH <<ruint32:$_:CERTCONTENT>>

BERLENGTH      <- & |00|80| { . } /
                  0x81 { . } / 0x82 { .. } / 0x83 { ...  } / 0x84 { .... }

CERTCONTENT    <- TBSCERTIFICATE
                  SIGNATUREALGORITHM
                  SIGNATUREVALUE
TBSCERTIFICATE <- SEQUENCE BERLENGTH <<ruint32:$_:TBSCERTCONTENT>>
TBSCERTCONTENT <- .*

SIGNATUREALGORITHM <- SEQUENCE BERLENGTH <<ruint32:$_:SIGALGCONTENT>>
SIGALGCONTENT  <- OID ANY?

ANY            <- . BERLENGTH <<ruint32:$_:ANYCONTENT>>
ANYCONTENT     <- { .* }

SEQUENCE       <- 0x30
BITSTRING      <- 0x03

OID            <- 0x06 BERLENGTH <<ruint32:$_:OIDVALUE>>
OIDVALUE       <- { . ( |80|80|* |00|80| )* }

SIGNATUREVALUE <- BITSTRING BERLENGTH <<ruint32:$_:SIGVALCONTENT>>
SIGVALCONTENT  <- { .* }
```

This grammar skips all the content, allowing you to quickly perform a cryptographic verification on the certificate. The output of the parsing process:

```
End code: 0
9 actions total
Action #0: capture slot 2, 2->2 "\x03^L"
Action #1: capture slot 2, 6->2 "\x02u"
Action #2: capture slot 0, 638->1 "^M"
Action #3: capture slot 0, 640->1 "     "
Action #4: capture slot 6, 641->9 "*\x86H\x86\xf7^M\x01\x01^K"
Action #5: capture slot 0, 651->1 "\x00"
Action #6: capture slot 5, 652->0 ""
Action #7: capture slot 1, 654->1 "\x81"
Action #8: capture slot 7, 655->129
"\x00E\x83\xdb\x1fo\xa3\xce[\xe7\x9c@A;&\x851\xe2x\xdc\x85\xa8I\x7f\xeeh}
\x11\xc7B\xeb\"#=\xff:f\xf7\x95\xc1\xc9R\xc3\x95kUsl\\\xc9\x83y\x894?\x0e
\x86\xbc\x148R\xb9\xe9~\xb6\xb3\xe0\xdcp\x12\x82\xfa\x87\xce8)@\xa2\xfb!
\xa5\x10u\x8a\xa1mg\x08\xda\xda\xea\_\x83\xa77){\x9bQ\xbe<h/\xae2wfge0\xf6
6)I\xa0;x\x9ai}I5\x18}\xe8\xdf;n"
Number of instructions: 1676
Max stack depth: 8
```

The artifacts from this process are also few in number. The slots 0-4 can be ignored, as they are used by the BERLENGTH rule only, and consumed for the purpose of limiting the input during certain rule calls. Slot 5, 6 and 7 will contain the capture regions you're interested in.