Challenges and Possibilities for Safe and Secure
ASN.1 Encoders and Decoders

Mark Tullsen

Galois, Inc.

LangSec, May 2018
Outline

1. Introduction
2. Five Things to Like About ASN.1
3. Obstacles to Secure ASN.1
4. Approaches to Secure ASN.1 Encoders/Decoders
5. Galois’ Current & Future Work
Introduction

Five Things to Like About ASN.1

Obstacles to Secure ASN.1

Approaches to Secure ASN.1 Encoders/Decoders

Galois’ Current & Future Work
Primarily a "position paper": Secure ASN.1 IMHO

For technical details, see our CAV 2018 paper
  - Formal Verification of a Vehicle-to-Vehicle (V2V) Messaging System
What is ASN.1?

- ASN.1 is *not*
  - a format
  - a single specification
  - a library (that we can implement once)

- ASN.1 is
  - a language by which we define hundreds of protocols and data-formats!
The Importance (& Risks) of ASN.1

- ASN.1 pervasive
- Decoding ASN.1-defined messages: definitely on the *attack surface*
Introduction

Five Things to Like About ASN.1

Obstacles to Secure ASN.1

Approaches to Secure ASN.1 Encoders/Decoders

Galois’ Current & Future Work
ASN.1 is Abstract

ASN.1 does not limit us to

- a single implementation language
- nor to a single de facto representation of the "abstract" values

\[
T ::= \text{SEQUENCE (SIZE}(1..4)\text{)} \ \text{OF INTEGER}
\]

One may have different *concrete* values/representations:

```c
typedef int T[4]; // probably not if SIZE(1..10000)
typedef int *T; // length elsewhere, or special encode of "last"
typedef struct node *T; // i.e., a linked-list
```
ASN.1 is Highly expressive (for describing data)

- ASN.1 provides an extensive and powerful set of types and constraints for *describing data* (beyond typical programming languages!)
ASN.1 provides an extensive and powerful set of types and constraints for describing data (beyond typical programming languages!)

Address ::= SEQUENCE {
  street  VisibleString (SIZE (5 .. 50)) OPTIONAL,
  city    VisibleString (SIZE (2..30)),
  state   VisibleString (SIZE(2) ^ FROM ("A"..'Z")),
  zipCode NumericString (SIZE(5 | 9))
}

ListOfItems ::= SEQUENCE (SIZE (1..100)) OF Item

Item ::= SEQUENCE {
  itemCode INTEGER (1..99999),
  power    INTEGER (110 | 220),
  deliveryTime INTEGER (8..12 | 14..19),
  isTaxable BOOLEAN
}
ASN.1 is Compositional

I.e., we can compose small data definitions to create larger ones.

- Has a real module system.
- Can refer to modules outside the current system.
- Can embed data of undetermined types, safely and sanely.
  - With Information Objects . . . even more elegantly
ASN.1 is Versatile: Multiple Encoding Schemes

- Allows for a variety of encoding methods, each specified separately from the definition of abstract values.
ASN.1 is Versatile: Multiple Encoding Schemes

- Allows for a variety of encoding methods, each specified separately from the definition of abstract values.

- Self-describing: BER, DER, etc.
- Highly bit-efficient: *OER, *PER (must know ASN.1 type to decode)
- Canonical signifies that a given value has only a single valid encoding
  - E.g., for X.509, DER is used (canonical)
ASN.1 is Extensible: Features for Protocol Evolution

- Provides mechanisms for extensibility to allow protocols and formats to evolve gracefully.

MyCoffeeShopMenu ::= SEQUENCE {
    coffee Price, 
    tea Price, 
    ...!1, -- extensible, exception-marker!
    [[ sandwich Price, -- Version2 
      dessert Price ]]
}

This data specification marked as extensible has evolved. The bits on the wire indicate if the data is in the base or extension. Thus, Version1 encoders/decoders work on Version2 data.
ASN.1 is Extensible: Features for Protocol Evolution

- Provides mechanisms for extensibility to allow protocols and formats to evolve gracefully.

\[
\text{MyCoffeeShopMenu ::= SEQUENCE \{ coffee Price, tea Price, ...!1, -- extensible, exception-marker! [ [ sandwich Price, -- Version2 dessert Price ] ] }
\]

- This data specification
  - marked as extensible
  - has evolved

- The bits on the wire indicate if the data is in the base or extension
  - Thus, Version1 encoders/decoders work on Version2 data.
Introduction

Five Things to Like About ASN.1

Obstacles to Secure ASN.1

Approaches to Secure ASN.1 Encoders/Decoders

Galois’ Current & Future Work
Obstacles to Secure ASN.1

Following Frederick Brooks—and Aristotle—in the use of these terms:

- Obstacles, *Essential* (intrinsic, "of the essence")
- Obstacles, *Accidental* (historical, poor design, etc.)
Essential Obstacles

- The *Five Things to Like* each creates some degree of complexity in the ASN.1 definition language itself
- non-trivial learning curve for users and implementers
- complexity for an ASN.1 compiler
- complexity in (and multiplicity of) the encode/decode routines
- etc.

Thus, a large effort to fully support the language

- Unsurprisingly, fully compliant compilers are expensive & proprietary

There *will* be a loss of abstraction in the concrete types: room for errors in the interfacing code. (The cost of *Things to Like* 1, 2, 5.)

With new & improved (bit-efficient) encoding schemes (PER, UPER, OER, etc.),

- Previous—and simpler—library-based solutions inadequate
- Compiler that uses global knowledge and "constraint solving": recommended!
Accidental Obstacles

- Evolutionary artifacts
  - Complicate the language
  - Force us to support old features and old specs
  - E.g.,
    - explicit TAGs / AUTOMATIC TAGS
    - newer encoding schemes that fix old ones
    - methods to encode unknowns or parameters

- Over-complexity
  - Information Objects
  - Encoding scheme details
    - e.g., long tags vs. long lengths vs. long values
    - Many closely related encoding schemes (is there a canonical encoding or not)
  - The ASN.1 Language itself
    - Literally thousands of grammar rules to parse the language.
The Unfortunate Outcome

- ASN.1’s complex and evolving language features, as well as its complex and evolving encoding schemes, together
  - hinder adoption
  - increase the complexity of tools & compilers,
  - and necessitate large and complex encoder/decoder implementations.
ASN.1’s complex and evolving language features, as well as its complex and evolving encoding schemes, together
- hinder adoption
- increase the complexity of tools & compilers,
- and necessitate large and complex encoder/decoder implementations.

None of which bode well for the task of creating robust & secure implementations.
Introduction

Five Things to Like About ASN.1

Obstacles to Secure ASN.1

Approaches to Secure ASN.1 Encoders/Decoders

Galois’ Current & Future Work
The Goal

... creating robust & secure\(^1\) encoder/decoder implementations.

\(^1\)secure: absence of software flaws
The Goal

... creating robust & secure\(^1\) encoder/decoder implementations.

With very high assurance.

\(^1\)secure: absence of software flaws
The Goal

... creating robust & secure\textsuperscript{1} encoder/decoder implementations.

With \textit{very high assurance}.

Provably secure?

\footnotesize\textsuperscript{1}secure: absence of software flaws
The Goal

... creating robust & secure\(^1\) encoder/decoder implementations.

With very high assurance.

Provably secure?

With reasonable effort?

\(^1\) secure: absence of software flaws
Generating Good Code?

- Roll your own ASN.1 encoder/decoder
  - Possible!
  - Isn’t ASN.1 simple stuff: *just* serializing/deserializing?
- Open source compilers
  - Limited in features and support.
  - E.g., code *definitely* not optimized for clarity/verification
- Commercial compilers
  - Objective Systems compiler "bad memory bug"
  - Library flaws
    - Exploitable flaw in MS Windows library for years
Generating Good Code?

- Roll your own ASN.1 encoder/decoder
  - Possible!
  - Isn’t ASN.1 simple stuff: *just* serializing/deserializing?
- Open source compilers
  - Limited in features and support.
  - E.g., code *definitely* not optimized for clarity/verification
- Commercial compilers
  - Objective Systems compiler "bad memory bug"
  - Library flaws
    - Exploitable flaw in MS Windows library for years

None of these can *guarantee* security.
Secure Code via Testing?

- Extensive testing
  - incomplete!
  - intelligent, expert, human testing: good idea

"Program testing can be used to show the presence of bugs, but never to show their absence!" - Dijkstra
Secure Code via Testing?

- Extensive testing
  - incomplete!
  - intelligent, expert, human testing: good idea

"Program testing can be used to show the presence of bugs, but never to show their absence!" - Dijkstra

"Exhaustive Testing"?

- Given this decode function with execution time of 1ms:
  ```c
  bool decode( uint64_t x, uint64_t y, uint64_t z) {
      //..
  }
  ```
- Time to exhaustively test: $2 \times 10^{47}$ years
Secure Code by Construction

- Develop a fully-compliant (open-source) ASN.1 compiler.
  - Make it a verified compiler
  - Remember to verify the libraries, ;-)
  - Make it certifying (generating proof of correctness/safety)
Secure Code by Construction

- Develop a fully-compliant (open-source) ASN.1 compiler.
  - Make it a verified compiler
  - Remember to verify the libraries, ;-)  
  - Make it certifying (generating proof of correctness/safety)

- Immensely Large Effort
In order to reduce trusted computing base...

- Full ASN.1 to ASN.1-lite transformation tool.
  - Possibly remove 80% of the complexity of ASN.1
  - Allow for use of simpler, open-source ASN.1 tools/compilers
  - Much of the complexity of ASN.1 "goes away" in compilation.
Another approach to avoiding giant monolithic proofs ...

- Use an Intermediate Language For Serialization (ILS)?
  - Just powerful enough to describe BER, DER, UPER, OER
  - TODO: What are the key primitives and combinators needed?

```
VALUES    spec(uper).ils    BITS
    ↓                        ↓
  spec.asn                  spec.c
```

- Assurance & Verification?
  - can verify the parts separately!
Secure Code by Post-Hoc Verification

- For a given encode/decode pair on a given spec:
  - Formally verify encoder/decoder code
- However
  - Expertise? Effort?!
  - Every spec or code update requires redoing verification.
- What might we verify?
  - Full verification WRT ASN.1 semantics: highly infeasible
  - Verify encode/decode consistency (round trip property): feasible
    - have proof of memory safety as a consequence
For a given encode/decode pair on a given spec:
- Formally verify encoder/decoder code

However
- Expertise? Effort?!
- Every spec or code update requires redoing verification.

What might we verify?
- Full verification WRT ASN.1 semantics: highly infeasible
- Verify encode/decode consistency (round trip property): feasible
  - have proof of memory safety as a consequence

We’ve done this: A *formal* verification of 1K+ lines of C
- Basic Safety Message in V2V (part one, a subset of ASN.1)
- Refer to our CAV 2018 paper
Question: Are implementations A, B, C equivalent?

Method

1. Create intelligent test-generation capabilities
   - legal *and* illegal values
   - quickcheck/smallcheck like test generation (not just generating every number in $\text{INTEGER}(1..1000000)$)

2. Create the glue code to run tests on each implementation
   - Non-trivial: Every compiler (or library) has a different API for creating values, encoding, and decoding!

3. For each test, verify all implementations match.
1 Introduction

2 Five Things to Like About ASN.1

3 Obstacles to Secure ASN.1

4 Approaches to Secure ASN.1 Encoders/Decoders

5 Galois’ Current & Future Work
Done
- Accomplished verification of code for V2V Basic Safety Message (J2735.asn)
- The automatic verification needed a bit of expert assistance
  - i.e., providing lemmas to allow SAW to terminate
- Doesn’t support all ASN.1 constructs

Next Steps
- Get the verification "automated"
  - generate lemmas/hints for SAW when we generate the C code
- Support more ASN.1 features

Future Goal
- Given any ASN.1 specification, generate valid C code and sufficient lemmas/hints to ensure SAW will terminate.
Secure Code by Intelligent Consensus Testing

(New work the DOT is funding us to do.)

- **Question:** Are implementations A, B, C equivalent?
- **Method**
  1. Create intelligent test-generation capabilities
  2. Create the glue code to run tests on each implementation
     - Research: capture the commonality in a DSL
  3. For each test, verify all implementations match.
- **Status**
  - In progress.
  - Goal: consensus among 2-3, eventually . . .
  - Developing something like ILS so as to get results that apply to multiple encoding schemes.
Approach One: Abandon Ship?

- Forgo ASN.1: roll your own (or pick some trendy) data serializer/deserializer
  - E.g., protocol buffers, json, avro(json), xml(..), etc, etc.
- Nice
  - Could reduce the complexity issues
  - Relevant for a new, not-wanting-to-play-with-others format
Approach One: Abandon Ship?

- Forgo ASN.1: roll your own (or pick some trendy) data serializer/deserializer
  - E.g., protocol buffers, json, avro(json), xml(..), etc, etc.
- Nice
  - Could reduce the complexity issues
  - Relevant for a new, not-wanting-to-play-with-others format
- But generally these
  - lack most of our *Five Things to Like*
  - have no separable "specification" of the data
    - are not primarily a data description language
  - are tied to implementation(s)
    - is there a *specification* for the encoding?
- And
  - we still have the verification effort,
Approach One: Abandon Ship?

- Forgo ASN.1: roll your own (or pick some trendy) data serializer/deserializer
  - E.g., protocol buffers, json, avro(json), xml(..), etc, etc.
- Nice
  - Could reduce the complexity issues
  - Relevant for a new, not-wanting-to-play-with-others format
- But generally these
  - lack most of our Five Things to Like
  - have no separable "specification" of the data
    - are not primarily a data description language
  - are tied to implementation(s)
    - is there a specification for the encoding?
- And
  - we still have the verification effort,
  - can we just stop re-inventing the wheel!
    - we could be using and contributing to the library of ASN.1 defined formats