Formally-Verified ASN.1 Protocol C-language Stack

Vadim Zaliva [1]   Nika Pona [2]

[1]Carnegie Mellon University

[2]Digamma.ai

What is ASN.1?

- At Digamma.ai we are verifying a compiler for ASN.1
- The ASN.1 is a language for defining data structures and rules for serialization and de-serialization.
- Initially we focus on a subset of ASN.1 used in the X.509 standard which defines the format of public key certificates.
- We formalize Basic Encoding Rules (BER) and Distinguished Encoding Rules (DER)

ASN.1 example of an X.509-like certificate

```
X509 DEFINITIONS ::= BEGIN

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

  TBSCertificate  ::=  SEQUENCE  {
     version         [0]  INTEGER,
     serialNumber         INTEGER,
     signature            AlgorithmIdentifier,
     issuer               Name,
     subject              Name,
     subjectPublicKeyInfo SubjectPublicKeyInfo,
  }

  SubjectPubicKeyInfo ::= SEQUENCE {
    algorithm            AlgorithmIdentifier,
    subjectPublicKey     BIT STRING
  }

  AlgorithmIdentifier ::= SEQUENCE {
    algorithm            OBJECT IDENTIFIER
  }

  Name ::= SEQUENCE OF SET OF SEQUENCE {
    type                 OBJECT IDENTIFIER,
    value                ANY DEFINED BY type
  }

END
```

ASN1C compiler

An ASN.1 compiler parses ASN.1 syntax definitions and produces either a source code of a specialized protocol encoder/decoder for this data type or a run-time data for a parametric encoder/decoder.

We are verifying a mature open-source ASN.1 compiler, ASN1C (*https://github.com/vlm/asn1c*). It is well-tested and widely used. We do the verification in Coq proof assistant.

What Coq does?

In Coq you can:

- define functions and predicates
- state mathematical theorems and software specifications
- interactively develop formal proofs of theorems
- machine-check these proofs by a relatively small trusted kernel based on the Calculus of Inductive Constructions
- compile certified programs to languages like OCaml, Haskell or Scheme.

Preliminary work: traditional approach

First, we tried the traditional approach on an error-prone part of ASN.1: floating-point numbers encoding/decoding (*`https://github.com/digamma-ai/asn1fpcoq`*). We wrote the encoders/decoders in Coq, proved their correctness and extracted to OCaml. This approach is not very practical since the generated code is not as efficient and usable as the C code.

Therefore we decided to try out a different approach: verify the C code directly.

Working with C semantics

We rely on the work previously done for the CompCert project (`http://compcert.inria.fr/`). CompCert is a verified compiler for C, written in Coq and proved to work correctly

- We parse C code into a Coq abstract syntax tree using CompCert
- Write a specification in Coq
- Prove that the generated AST behaves according to the specification, according to semantics of C defined in CompCert

Preliminary experiments

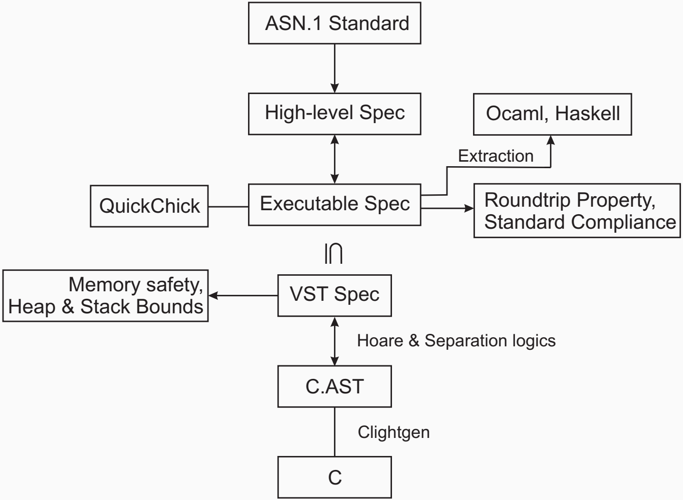
First we took a relatively simple but representative function *strtoimax* (string to integer conversion with bounds checking) from ASN1C and proved it correct using two approaches:

- proof using operational semantics defined in CompCert
- proof using separation logic defined on top of CompCert's operational semantics using Verified Software Toolchain (VST, *https://github.com/PrincetonUniversity/VST*)

During this experiment we found three bugs in this function (integer overflow, wrong memory write, semantically unintended behaviour). We saw that using VST is more practical.

Verification Architecture

We ended up with following verification architecture:



Verification Architecture explained: BOOLEAN encoder/decoder

Now we explain the verification architecture on example of the boolean decoder. We focus on Basic Encoding Rules (BER).

The ASN.1 Standard says:

> §8.2.1. *The contents octets shall consist of a single octet.*
> §8.2.2. *If the boolean value is FALSE the octet shall be zero. If the boolean value is TRUE the octet shall have any non-zero value, as a sender's option.*

High-level spec (BOOLEAN)

```
Inductive BER_Bool : 𝔹 → list byte → Prop :=
| False_Bool_BER: BER_Bool false [0]
| True_Bool_BER b : b <> 0 → BER_Bool true [b].
```

BER_Bool is a relation between booleans and lists of bytes (octets) with two rules that define this relation and formalize (part of) a paragraph in the actual standard. This relation defines how a value is encoded. Then *BER* relation (next slide) defines how the whole packet (tag-length-value) is encoded.

High-level spec for other types

```
Inductive BER : asn_value → list byte → Prop :=
| Bool_BER b t v:
    PrimitiveTag t → (* § 8.2.1 *)
    BER_Bool b v →
    BER (BOOLEAN b) (t ++[1] ++v)

| Integer_long_BER t l v z:
    PrimitiveTag t → (* 8.3.1 *)
    Length (length v) l → (* 10.1 *)
    1 < length v → (* 8.3.2, case 2 *)
    (v[0] = 255 → get_bit 0 v[1] = 0
    ∧ v[0] = 0 → get_bit 0 v[1] = 1) → (* 8.3.2, (a) and (b) *)
    BER_Integer z v →
    BER (INTEGER z) (t ++l ++v)
  ...

  | Sequence_BER t l ls vs:
    let v := flatten vs in
    ConstructedTag t → (* 8.9.1 *)
    Length (length v) l → (* 10.1 *)
    (∀  n, n < length ls → BER ls[n] vs[n]) → (* 8.9.2 *)
    BER (SEQUENCE ls) (t ++l ++v)
```

Decoder C implementation

```c
asn_dec_rval_t
BOOLEAN_decode_ber(const asn_codec_ctx_t *opt_codec_ctx,
                   const asn_TYPE_descriptor_t *td, void **bool_value,
                   const void *buf_ptr, size_t size, int tag_mode) {
    BOOLEAN_t *st = (BOOLEAN_t *)*bool_value;
        asn_dec_rval_t rval;
        ber_tlv_len_t length;

        if(st == NULL) {
                st = (BOOLEAN_t *)(*bool_value = CALLOC(1, sizeof(*st)));
                if(st == NULL) {
                        rval.code = RC_FAIL;
                        rval.consumed = 0;
                        return rval;
                }
        }
        rval = ber_check_tags(opt_codec_ctx, td, 0, buf_ptr, size,
                tag_mode, 0, &length, 0);
        if(rval.code != RC_OK)
                return rval;

        buf_ptr = ((const char *)buf_ptr) + rval.consumed;
        size -= rval.consumed;
        if(length > (ber_tlv_len_t)size || length != 1) {
                ASN__DECODE_FAILED;
        }

        *st = *((const uint8_t *)buf_ptr);

        rval.code = RC_OK;
        rval.consumed += length;

        return rval;
}
```

Executable spec

Executable specification is an abstraction of the C implementation of the decoder.

```
Definition bool_decoder (td : TYPE_descriptor) (ls : list byte)
    : error (byte * Z) :=
    match ls with
    | [] => inl FAIL
    | _ => (consumed, expected) <- ber_check_tags td ls ;
            if Zlength ls - consumed < expected || (expected != 1)
            then inl FAIL
            else y <- hd (skipn consumed ls) ;;
            inr (y, consumed + 1)
    end.
```

Functional correctness and the “roundtrip” property

We show that decoder is inverse of encoder.

```
Theorem boolean_roundtrip : ∀ td ls b z,
    decoder_type td = BOOLEAN_t →
    bool_encoder td b = inr (z, ls) →
    bool_decoder td ls = inr (b, z).
```

We prove that the executable spec encodes and decodes bytes in conformance with the high-level specification.

```
Theorem bool_decoder_correctness : ∀ td ls b z,
    bool_decoder td ls = inr (b, z) ↔ BER (BOOLEAN b) (firstn z ls).
```

VST specification

To show C implementation correctness wrt the executable (and hence high-level spec) we prove a separation logic triple

$$P\{c\}Q$$

that given the precondition P, the execution of the C light function c terminates with the post-condition Q being true. The post-condition says that c returns the value according to the executable spec.

VST spec, encoder pre- and post-condition

The memory specification uses spatial predicates $v \leftarrow p$ ("at address p there is a value v").

We can combine the predicates using the separating conjuction $*$: each such conjunct is true on a separate sub-heap of the memory, thus guaranteeing non-overlapping of pointers.

The precondition relates the C types such as _asn_TYPE_descriptor_s, int, $*$char of BOOLEAN_decoder_ber to the abstract types of Coq *TYPE_descriptor*, $\mathbb{B}$, *list byte* etc.

In the post-condition, we use the executable specification to state that the correct result is written in memory.

VST spec, decoder pre- and post-condition

```
PRE [(td : TYPE_descriptor) ↦ td_p *
     (buf : list byte) ↦ buf_p ... *
     bool_value_p ↦ bool_value_pp *
     (res : code * Z) ↦ res_p *
     if bool_value_p == null then emp else _ ↦ bool_value_p]

POST [ (* Unchanged memory *)
       td ↦ td_p * buf ↦ buf_p ... *
        (* Changed memory *)
       EX v : val, EX ls : list val,
           v ↦ bool_value_pp *
             if v == null
             then res ↦ (RC_FAIL, 0)
             else match bool_decoder td buf with
                    | inr (r, c) ⇒ res ↦ (RC_OK, c) * v ↦ r
                    | inl FAIL ⇒ res ↦ (RC_FAIL, 0) * v ↦ ls
                 end).
```

VST proof

The proof is done using so-called *forward simulation*. To prove $P\{c\}Q$:

- start assuming the precondition P
- sequentially execute statements of the function c
- each statement generates a post-condition that follows form its execution
- after executing the last statement of c, prove that the post-condition Q holds.

VST provides tactics to do most of these steps automatically. One has to provide joint postconditions for if statements and loop invariant for the loop

Lessons learned and future work

- We have the basic infrastructure in place to prove the X.509 part of ASN1C
- The memory-related parts of the proof are uniform so can be reused
- We use a layered approach to decrease the creative effort in the VST proof
- A realistic subset of C code is supported by VST
- But extensions and more automation is needed for industrial scale projects

Join us!

The project is in active development right now, but given the ambitious scope a significant effort is required for it's completion. *Digamma.ai* is committed to sponsor the initial stage of the project and we are currently looking for industry and academic partners to join us in the full ASN.1 verification endeavor.

Contact:

- Vadim Zaliva, ✉ vzaliva@cmu.edu, @vzaliva
- Nika Pona, ✉ npona@digamma.ai