Research Report: Formally-Verified ASN.1 Protocol C-language Stack

Nika Pona
Digamma.ai
npona@digamma.ai

Vadim Zaliva
Carnegie Mellon University
Department of Electrical and Computer Engineering
vzaliva@cmu.edu

***Abstract*—We describe our approach and progress in verification of a mature open-source ASN.1 compiler, *ASN1C*, using the Coq proof assistant. Once completed, our project will provide state-of-the-art high assurance suitable for mission-critical systems. Furthermore, since formal verification will be layered atop a well-tested ASN.1 stack, it will combine the benefits of high-performance portable stack implementation with formal correctness guarantees. As an essential step in our approach, the project will also provide a formalization of a key part of the ASN.1 standard. Such formal specification could subsequently be used by others to analyze ASN.1 properties and validate other implementations.**

I. Introduction

A. Background

The ASN.1 (Abstract Syntax Notation One) [1] joint standard of the International Telecommunication Union (ITU-T) and the International Organization for Standardization (ISO/IEC) provides an essential interface description language for defining data structures for serialization and deserialization in cross-platform data exchange. ASN.1 is vitally relied upon by core aspects of the Internet infrastructure and Internet applications, such as telephony, enterprise computing, utilities, finance, military, security, digitally-controlled infrastructure, transportation, medical systems, and commercial cloud computing.

Using ASN.1 language, one can define data structures that use ASN.1 *primitive types*, such as *INTEGER*, *BOOLEAN*, *OBJECT IDENTIFIER*, and *constructed types*, such as *SEQUENCE (OF)*, *SET (OF)*, and *CHOICE (OF)*. An example ASN.1 module that describes an X.509-like public key certificate is shown in Listing 1 (revised for brevity). Such certificates, for example, are used by every browser to access HTTPS web sites. The unabridged version of the original ASN.1 definition for the X.509 standard takes about 1,000 lines of ASN.1.

Listing 1: ASN.1 example of X.509 certificates

```
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
```

A typical ASN.1 stack consists of a *compiler* which parses ASN.1 syntax definitions, as shown in the Listing 1, and produces either a source code of a specialized protocol encoder/decoder for this data or a run-time data for a parametric protocol encoder/decoder.

B. Motivation

These days, embedded and user computing devices implement increasingly vast numbers of essential functions and applications, many of which exchange data using ASN.1. The ASN.1 standard is large and complex: it currently comprises twelve sub-standards spanning 862 pages supplemented by additional pages of corrigenda. This opens a door for many potential software bugs and malicious exploits. As a result, the interconnected communicating systems are becoming less stable and less reliable, posing greater risks and potential dangers. Disruption of the ASN.1 based communications could threaten the functioning of the entire critical infrastructure our society relies upon.

The Computer Vulnerabilities and Exposures (CVE) database [2] lists critical ASN.1-related bugs that are found each year in the existing systems. Noteworthy exposures have

already been discovered [3] that clearly illuminate potential vast risk and exposure. We analyzed the last four years of ASN.1-related issues reported in the CVE database[1]. Among the vulnerabilities studied are CVEs for various software and hardware products and vendors, including *Apple*, *axTLS*, *Botan*, *Bounty Castle*, *librcrypto++*, *libtasn*, *LibTomCrypt*, *Linux Kernel*, *MatrixSSL*, *Mozilla NSS (Firefox)*, *Objective Systems*, *OpenSSL*, *PolarSSL*, *RSA BSAFE*, *Samba*, *Samsung*, *Snapdragon*, *strongSwan*, and *Wireshark*. Out of 52 problems analyzed, 39 are related to memory safety, 6 related to stack and heap bounds checking, and 3 are related to issues caused by applications accepting poorly formed ASN.1 input. Proving just following six formal properties can prevent 49 out of 52 vulnerabilities, which is more than 90% of reported vulnerabilities:

1) Absence of read memory access violations (reading from memory locations outside of a permitted range).
2) Absence of write memory access violations (writing to memory locations outside of a permitted range).
3) Strict standard compliance (detecting not well-formed input)
4) Guaranteed parser program termination
5) Guaranteed bounds on stack size
6) Guaranteed bounds on heap size

C. Our approach

To date, the relevant systems and methods are either (a) automatically tested but not formally verified, (b) use verification approaches which rely on automatic extraction from executable specifications (for example involving network stack synthesis [4], optimizing compilers [5], cryptographic libraries [6], and encoder/decoders [7]), or (c) apply a form of formal verification which only proves partial correctness properties (partial verification of NAT stack proving that only parts of DPDK are specification compliant [8], partial verification of Linux kernel TCP implementation with 55% line coverage and 92% protocol coverage [9]). Consequently, (a) and (c) do not provide sufficient correctness guarantees, while (b) is often impractical due to poor performance and compatibility limitations. In contrast, we pursue a far deeper and comprehensive verification approach to performance and portability and seek to prove actual industrial-level C-code implementation.

As shown in Figure 1, the project begins with ITU-T standard document in the form of human-readable text. We manually convert it into formal specification (H.spec). This is high-level specification (in Coq) which describes the correspondence between data types and packet octets data layout. This specification is one of the outputs of this project and has a value of its own.

The next level of refinement is executable specification (E.spec). Also written in Coq, it declares "encoder" and "decoder" for each type as a pair of pure functions and basically abstracts the encoding/decoding algorithm used in the C function and operates on abstract types such as Coq integers and lists of bytes. We prove that the E.spec encodes and decodes bytes in conformance with the high-level specification. Moreover, we prove the roundtrip property for each encoder/decoder pair.

[1]See Appendix B for more details.

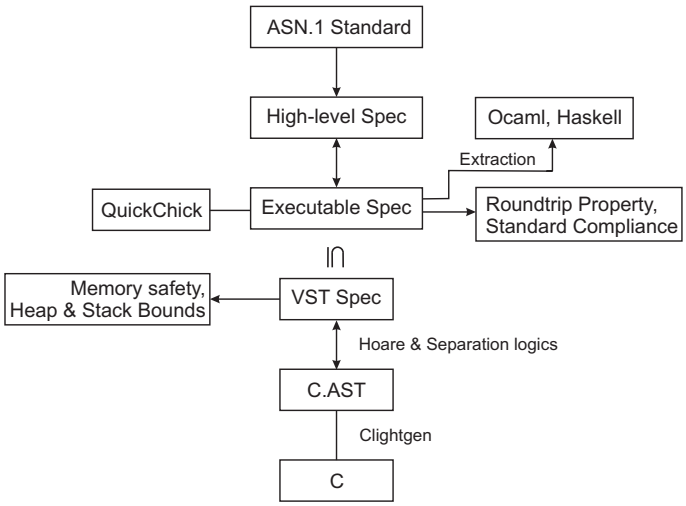

Fig. 1: Verification Architecture

From E.spec we can *extract* a fully functional encoder/decoder program in languages supported by the Coq extraction mechanism (e.g. OCaml or Haskell) [10]. Such a possibility is shown in the box on the right in Figure 1, although we do not plan to pursue it yet. However it could be explored in the future, for example, in case we need to generate an ASN.1 stack for special platforms where OCaml code opens some additional opportunities for parallelization, optimization, or integration. One example of such platforms could be MirageOS [11], which is an OCaml-based unikernel.

The box at the left, marked *QuickChick*, represents another possibility we might not pursue immediately. It uses randomized property-based automated testing based on an executable specification to further verify its correctness. This work could be done using QuickChick [12] in Coq. It could also be used to generate a test-suite for the extracted code.

The next level is VST specification (VST.spec). VST stands for Verified Software Toolchain [13], a tool that formalizes separation logic in Coq using operational C semantics from CompCert. It provides uniform syntax for stating pre- and postconditions of imperative programs execution, as well as tactics that automatize machine arithmetic and separation logic proofs. VST.spec states that the C program terminates and returns the machine representation of the value mandated by E.spec and that it satisfies the memory specification consisting of pre- and post-conditions written in separation logic. Correspondence to the memory specification guarantees memory safety. Further, we can add post-conditions about heap and stack bounds.

At the bottom of Figure 1 is a box labeled "C" which represents the existing C code of the *ASN1C* compiler. Using an automated conversion provided by the `clightgen` tool within the CompCert C compiler [5], we convert this to

Abstract Syntax Trees (AST) (labeled "C.AST" in Figure 1) rendered from a subset of C-language, called Clight [14], which is recognized by Coq. More specifically, CompCert converts C "concrete syntax" to "abstract" Clight syntax. The resulting Clight program is just a reification of the original C program in Coq, retaining its overall structure; Clight is further assigned formal semantics by CompCert that can be used to further reason about the behaviour of programs within Coq.

Finally, as the most critical (and laborious) step of the project, we prove that semantics of Clight translation for each given function correspond to its VST specification. This guarantees that the C program terminates, is memory safe and behaves exactly like the executable and hence the high-level specification.

We perform all the proofs using the Coq Proof Assistant [15]. From the proofs Coq can generate a "certificate," which is a program in a formal language based on the Calculus of Inductive Constructions ("CIC") [16]. The certificate allows 3rd party validation of proofs which have been developed in Coq. The CIC is a small and mathematically well-defined formal language that serves as the underlying formal system of Coq. The generated certificate is automatically validated by the relatively small Coq kernel. Although a fraudulent certificate could hypothetically be created, it would not pass validation when submitted to the Coq kernel. The resulting outcome limits the trust need to rely only on the small CIC formal language and the small Coq kernel.

Additionally, our (modified) version of *ASN1C* could be compiled with the certified CompCert compiler [5] to extend correctness guarantees through all levels down to machine code. This will provide state-of-the-art high assurance suitable for mission-critical systems. Further, since formal verification will be layered atop of a widely used ASN.1 stack, it could be offered to current users immediately. This makes it attractive to new users who require higher assurance levels than current non-verified implementations provide.

II. Preliminary work

Before committing ourselves to this project, we performed some initial exploratory work to try different validation approaches, evaluate tools and libraries, and estimate effort required. Briefly summarized below are the results of some of the work we performed.

A. Verifying floating-point numbers encoding

As a first estimate of the difficulties associated with verifying ASN.1-related programs, we wrote formal specification of an encoder-decoder pair for a small but particularly error-prone subset of the standard – *floating-point numbers*. The development is available on github [17]. Our first approach was relatively straightforward: define types representing ASN.1-encoded data in Coq, provide functions for converting between representations, and prove that they operate correctly. In particular, we proved the roundtrip property for the defined encoders/decoders (i.e. the decoder is the inverse of the encoder).

Although providing guarantees of correctness, this technique has major disadvantages. First of all, our definitions, being written in pure Coq, have only one connection to the real world - through automatic code extraction. This immediately creates a set of problems:

- Automatic code extraction from Coq is not formally verified.
- Extracted code generally runs much slower than its counterparts implemented in other languages[2].
- Extracted code might not be compatible with other code or viable in some real-world use-case scenarios.

However, this exploration allowed us to home in on what approaches to verification are viable and improve our effort estimates. Moreover, we tested specification and proving techniques which will be useful for future experimentation.

B. Verifying simple ASN1C function

To estimate the effort required to formally verify C code and to experiment with various verification strategies, we decided to try to verify a small function from an existing ASN.1 compiler. We chose function `asn_strtoimax_lim` from the *ASN1C* compiler. *XER* decoding functions for *INTEGER*, *OBJECT-IDENTIFIER*, and *RELATIVE-OID* types (and hence all constructed types that use these primitive types) critically depend on this function. The function is relatively simple but at the same time, it uses many features of C that make verifying imperative programs challenging.

The only specification was the following comment in the source code:

> *Parse the number in the given string until the given *end position, returning the position after the last parsed character using the same (*end) pointer. WARNING: This behavior is different from the standard strtol/strtoimax(3).*

Additional specification details must be inferred from the source code and usage examples. Full source code of the function is included in Appendix A.

1) Problems discovered: Despite that fact that this function lineage could be traced back 15 years and that it is part of mature, well-tested ASN.1 compiler presently used in many production systems, we found three bugs or problems in the current implementation. These had never been reported before and had passed all human code reviews as well as unit and fuzzying tests.

a) Negative range bug: When we go beyond the allowed *integer* range, an incorrect result is given for some inputs. For example, assuming we are working on an 8-bit system and the maximum signed integer (`MAX_INT`) value is 127, parsing the input string "`-1281`" is successful and returns the value -127 instead of the expected range error message. This happens whenever the input string represents a number smaller than `MIN_INT`, due to the fact that its absolute value is greater than `MAX_INT`, thus the negative number cannot be treated

[2]The extracted code performed 15 times slower than the *ASN1C*.

as a value × sign when the value is represented as *integer*. This bug was reported and promptly fixed by developers.

b) Problem with pointer aliasing: Another bug we discovered was related to potentially overlapping memory areas pointed by argument pointers. Under some circumstances, the value of the `end` pointer parameter is treated as a part of the input data, and the resulting error value could be incorrect. This bug would never occur if the function is always called with non-overlapping pointer arguments. However this may be viewed as an implicit pre-condition which should be part of the function's specification.

c) Specification ambiguity: After addressing the two bugs we discovered, we were able to successfully verify that the function finally corresponds to the specification we wrote for it. However, the following behavior was noticed; for input "`a`", it records the value 0 (the same behaviour as for input "`0`") and returns the `EXTRA_DATA` error message (the same behaviour as for input "`0a`"), which was probably unintended.

2) Direct operational semantics proof: First we formulated functional correctness and proved it using big-step operational semantics of *C light*, defined in CompCert. In this proof, we used pure-function re-implementation in Coq as an intermediate specification.

This function took addresses as inputs and operated on memory using `load` and `store` operations from *CompCert's* memory model, while calculating the resulting machine integer value. The proof went by induction on the distance between input pointers and the main difficulty apart from trying to prove a faulty program (that's when we discovered two bugs) was operational semantics control flow minutiae and machine arithmetic proofs. However, only a couple of lemmas about the specification were needed, and proofs of different cases were very repetitive. Since functional and memory specification were intertangled, it was more difficult to read the specification and make sure it was correct. Hence, we missed the specification "bug" mentioned above.

3) Proof using VST: The *Verified Software Toolchain* offers solutions to problems we encountered while doing direct operational semantics proof; it has good automation of control flow, some automation for machine arithmetic, and clear separation of functional and memory-related parts of the specification. It also provides a uniform way of stating functional correctness and memory safety, which reduces the chances of having the wrong specification. Proofs here are done using *separation* logic implemented in Coq, which are proven to be sound with respect to operational semantics of CompCert. Similarly as before, we rely on C light syntax and semantics. VST has tactics that can solve simple entailments in these logics. However, they are not powerful enough to significantly reduce the overall proof effort. In fact, with respect to memory safety specifications, direct operational semantics proof were shorter and more straightforward. However, this problem can be solved by improving the existing tactics and fine-tuning the specification style, so in the end we find this approach more viable for a large project.

In this simple example, we test our architecture from Section I-C. First, we write a high-level specification of this function in declarative, relational style (H.spec). Each constructor corresponds to a return message (state) and stores a value and the number of iterations of the function (used to store the result in memory). Such specification can be easily examined and refined.

```
(* Relation between input string, value,
   index an asn_strtox_result_e error message *)
 Inductive asn_strtoimax_lim : list byte → Z →
Z → asn_strtox_result_e → Prop :=
  (* Invalid data encountered *)
  | ERROR_INVAL:
      asn_strtoimax_lim nil 0 0 ERROR_INVAL
  (* More data expected (e.g. "+") *)
  | EXPECT_MORE : ∀ ls c,
      ls = [c]  →
      is_sign c = true →
      asn_strtoimax_lim ls 0 1 EXPECT_MORE
 (* Non-digit encountered *)
  | EXTRA_DATA : ∀ c ls z i,
      asn_strtoimax_lim ls z i OK →
      is_digit c = false →
      asn_strtoimax_lim (ls ++[c]) z i EXTRA_DATA
    ...
```

The next level is the *executable specification* `Z_of_string` (E.spec), which we prove to be equivalent to the relational specification. However, it could be extracted to Coq and is easier to use in future proofs of semantic equivalence with C code. Here, we experiment with two approaches. The function `Z_of_string` can serve as a functional specification for `asn_strtoimax_lim` and differs little from the relational specification.

```
Fixpoint Z_of_string_loop (s : list byte)
        (val i : Z) (b : 𝔹) :=
    match s with
    | [] ⇒ {| OK; val; i |}
    | c :: tl ⇒
      if is_digit c
      then let val' := app_char b val c in
           if bounded val'
           then Z_of_string_loop tl val' (i + 1) b
           else {| ERROR_RANGE; val'; i; |}
      else {| EXTRA_DATA; val; i; |}
    end.

Definition app_char (b : 𝔹) v c :=
  if b then v * 10 + (Z_of_char c)
       else v * 10 − (Z_of_char c).
```

Since `Z_of_string` has a different structure from the C implementation, the proof requires many lemmas to connect functional specification to the C code and complicate the loop invariant. It is more effective to separate the functional aspect of the proof from the C-proof as much as possible to allow for more automation in the proof of C function correctness. Thus, we define another intermediate specification `Z_of_string_C`, which is basically functional re-implementation of the C function, like the one used in the

operational semantics proof but without mention of memory or machine integers. Then, we prove that `Z_of_string_C` is equivalent to `Z_of_string`, which is a simple functional correctness proof that allows eliminating the need for additional lemmas in the C proof that connect the abstract specification with the C code structure.

```
Fixpoint Z_of_string_loop_C (s : list byte)
         (val i : Z) (b : 𝔹) :=
    match s with
    | [] ⇒ {| OK; val; i |}
    | c :: tl ⇒
      if is_digit c
      then let d := (Z_of_char c) in
           let val' := val*10 + d in
           if v <? upper_boundary
           then Z_of_string_loop_C tl val' (i + 1) b
           else if (v =? upper_boundary)
                   &&(d <=? (last_digit_max b))
                then match tl with
                     | [] ⇒ {| OK; val'; (i + 1) |}
                     | c :: tl ⇒
                       if is_digit c
                       then {| ERROR_RANGE;
                               app_char b val' c;
                               (i + 1) |}
                       else {| EXTRA_DATA;
                               val';
                               (i + 1) |}
                     end
                else {| ERROR_RANGE; val'; i |}
      else {| EXTRA_DATA ; val; i |}
    end.
```

We then wrote two VST specifications: one that uses the high-level specification and one that uses executable specification to state functional correctness. Memory specification for the function is expressed in terms of special memory or *spatial* predicates. For instance, the predicate `(data_at t ls p)` states that at the address `p` there is content `ls` of type `t`. One can combine such predicates with usual propositional connectives or with *separation conjunction* (here written as `;`). Intuitively, if `P` and `Q` are spatial predicates, `p` and `q` are pointers, then `(P(p) ; Q(q))` means that `P(p)` and `Q(q)` are true in separate sub-heaps of the memory or, in other words, that the pointers `p` and `q` don't overlap. Since we compare pointers between `str` and `*end` in the body of the function, we also have to ensure that they are valid pointers according to the C standard, are comparable (i.e. point within the same object), and non-overlapping. For instance, the memory precondition for the function will now have the following form:

```
PRE ((* str and *end are valid pointers *)
     valid_pointer *end; valid_pointer str ;
     (* str points to contents
     ls of type char array *)
     data_at (array char) ls str ;
     (* end points to end' *)
     data_at (ptr char) *end end;
     (* intp points to some value v *)
     data_at long v intp)
```

Moreover, the post-condition will record the changes in memory. We then prove that, given the precondition and after the execution of `asn_strtoimax_lim`, the post-condition will hold.

```
POST((* the fist four lines of the precondition
        didn't change after execution *)
     ...
     let r := result (Z_of_string ls) in
      (* in 3 cases intp stays unchanged,
        otherwise store the end value
         of Z_of_string *)
      match r with
        | ERROR_RANGE
        | ERROR_INVAL
        | EXPECT_MORE ⇒
          data_at long v intp
        | _ ⇒ data_at long
                    (value (Z_of_string ls))
                     intp
      end ;
     (* if str >= end, end doesn't change,
        otherwise store the address of the last
         char read (before going out of range,
          reading extra data
        or success) *)
      let i := index (Z_of_string ls) in
      if str <? *end
      then data_at (ptr char) (str + i) *end
      else data_at (ptr char) *end end).
```

Given experiments on this example, we delineated our strategy for proving *ASN1C* correctness as described in Section I-C.

III. Project Scope and Plan

The International Telecommunications Union X.509 standard [18] defines the format of public key certificates used in many cryptographic Internet protocols (including TLS/SSL, the basis for HTTPS protocol for secure browsing the web), certificate revocation lists, certification path validation, electronic signatures, and many other essential applications. X.509 is based on ASN.1, using an important subset (Distinguished Encoding Rules, "DER") of ASN.1 [19] [20].

To limit the scope of the initial stage of the project while exercising and showcasing all features of the complete ASN.1 standard, we focus the detailed verification on the DER part, which can immediately be used to implement verified X.509 stacks for use in production applications.

The *ASN1C* compiler supports BER, DER, CER, XER, PER and OER encoding/decoding rules. The total backend code is around 21,000 lines of code. We estimated X.509 related part to be around 3,000 lines of code. To compare, it took 18 person-years to write and verify 8,700 lines of code of *sel4*, a formally verified operating-system kernel [4]. This gives us a conservative estimate of 6 person-years for this project. However, in the aforementioned project most of the time was used to develop technologies that are already available in the Coq community (formal semantics of C, some automation for machine integer arithmetic and separation logic proofs in VST).

The X.509 focus provides a setting to answer after completion of the 1st stage essentially all questions to determine the technical feasibility of the proposed concept of full ASN.1 stack verification. These questions include the following points:

1) How *ASN1C* source code needs to be refactored to make it suitable for verification. In particular:
   a) Which features of general C-language unsupported by CompCert's C language semantics need to be avoided?
   b) Are there any code organization changes which will aid structuring proofs (for example to make it easier to relate each function to a corresponding lemma)?
   c) Are there any simplifications which could be done to *ASN1C* code by removing rarely used or obscure features that will make it easier to verify?
   d) Since the *ASN1C* code base dates back more than a decade, are there are any low-level manual optimizations that could be eliminated by a combination of modern compilers and hardware that can handle them as well?
2) What high-level properties can be proven which will translate into additional code safety guarantees?
3) How should formalization of ASN.1 standard in Coq be written? - i.e. How to balance clarity, readability, and comprehensiveness?
4) How many of the proving steps could be automated and how can these automations speed up the proof process? Can they be used to estimate the effort required to prove the remainder of the ASN.1 stack?
5) How many of the proving methodologies and tools developed during the first phase could be re-used to prove similar software and other protocols?
6) Are executable specifications of encoders/decoders completely sufficient to "extract" a working skeleton of the ASN.1 stack in OCaml? How much of additional "glue" code needs to be written to transform the result into a working product? How do code size and the performance of such extracted stack compare to the original C-language implementation from *ASN1C*?

To address these questions determining the technical feasibility of our proposed concept, we defined key tasks and objectives:

Task 1 Formalization of X.509-Relevant Parts of ASN.1 Specification
Task 2 Refactoring of the ASN1C C Code Implementation of ASN.1
Task 3 Primitive and Constructed Type Verification
Task 4 Proof of Additional High-Level Properties
Task 5 Code Extraction from Execution Specification
Task 6 Final Code, Associated Documentation, Estimates of Additional Work

IV. Project Status

Now, we are working in parallel on the high-level specification of the standard and executable specification of the *ASN1C* encoders/decoders, while recording changes that should be made to the C code.

We start by formalizing BER and DER encoding rules for several primitive and constructed types in relational style. Each premise of a constructor formalizes (part of) a paragraph in the standard. E.g., for boolean type:

```
Inductive DER_Bool : 𝔹→ list byte → Prop :=
| False_Bool_DER : (* 8.2.2 *)
    DER_Bool false [all_zero]
| True_Bool_DER : (* 11.1 *)
    DER_Bool true [all_one].
```

Then we use our primitive type specifications to formalize DER relation for each type using predicates about length and tags. The formalization is first-order, thus it can be easily translated to other languages than Coq. Moreover, the relation to the text is straightforward, which will ease its inspection and validation.

```
Inductive DER : asn_value → list byte → Prop :=
| Bool_DER b t v:
    PrimitiveTag t → (* 8.2.1 *)
    DER_Bool b v →
    DER (BOOLEAN b) (t ++[Byte.one] ++v)

| Integer_short_DER t v z :
    PrimitiveTag t → (* 8.3.1 *)
    length v = 1 → (* 8.3.2, case 1 *)
    DER_Integer z v →
    DER (INTEGER z) (t ++[Byte.one] ++v)

| Integer_long_DER t l v z:
    PrimitiveTag t → (* 8.3.1 *)
    DefiniteLength (length v) l → (* 10.1 *)
    1 < length v → (* 8.3.2, case 2 *)
    v[1] != all_one → (* 8.3.2, (a) *)
    get_bit 7 v[1] = 1 → (* 8.3.2, (b) *)
    DER_Integer z v →
    DER (INTEGER z) (t ++l ++v)
     ...
```

We write executable specification that is close to the actual implementation, since we have to keep in mind the future proof effort, especially with respect to the C code. This involves devising structures and functions that are easily translatable to their C counterparts.

The *ASN1C* compiler has a modular structure, so we can proceed with verification in a modular way by exploiting this structure. From a high-level view, the *ASN1C* compiler consists of the library of primitive type decoders (*INTEGER*, *BOOLEAN*, *FLOATING POINT*, etc.). Each of these can be verified with respect to its ASN.1 specification. Our previous work on floating point numbers is an example of this. We start with the simplest types; we write the executable specification that corresponds to the C function; and then check it with respect to the specification. Even for the simplest boolean type, we find that the decoder was not behaving according to the standard; in particular, several encodings for boolean values were possible, which clearly violates the ASN.1 standard for DER encoding/decoding.

Furthermore, the compiler contains functions that decode/encode constructed types (such as *SEQUENCE*, *CHOICE*, *SET*). Then, from a given ASN.1 type definition, the *ASN1C* compiler produces an internal representation of that type, specifying which existing decoders/encoders to apply and in which order, as well as the output/input C structures for the functions. In Coq, these internal representations correspond to trees with nodes labelled by the type's tags and decoder/encoder types.

```
Inductive decoder_type := BOOLEAN | INTEGER ...

Inductive TYPE_descriptor :=
  DEF { tags : list Z;
        decoder : decoder_type;
        elements : list TYPE_descriptor
      }.
```

Then, a constructed type decoder has type `TYPE_descriptor → list byte → option asn_value` and traverses the tree and applies the respective primitive decoders (this is a simplified definition; in the actual implementation, we use stream monad for input and monadic error handling). In *ASN1C*, the decoder is implemented as a recursive function that readily translates into a nested or mutually recursive function in Coq.

Given a particular ASN.1 type definition that translates into a `TYPE_descriptor`, the `decoder` function for that particular type corresponds to a transition system with states where primitive decoders are called. Hence, one of the ways we can formalize the compiler is as a function that creates a transition system from a given ASN.1 type definition or `TYPE_descriptor` tree. Since the run of the recursive `decoder` function corresponds to the transition system, we can use this model in the proof of implementation correctness.

The `decoder` function could also be viewed in terms of a sequence parser combinator, known from the monadic parser combinators approach [21]. Given decoders `p1, p2 : decoder asn_value` for primitive functions, it is possible to add an operator `seq` that takes two decoders and applies them sequentially `seq p1 p2`. Then the compiler can be seen as producing the concrete combination of parsers given the `TYPE_descriptor`. Currently, we are experimenting with these approaches.

V. Related Work

Microsoft's EverParse project [22] is the closest to what we do. However, they don't verify the existing protocol stack but build their own. They define their own input language which accepts C-like type definitions, which automatically generate specifications, their implementations, and proofs of correctness. This is possible because the produced decoders are combinations of existing parsers that are proved correct in F*. They follow the tradition of parser combinators. Starting with primitive parser combinators `fail`, `return[x]`, `read_byte`, and monadic composition of two parsers `and_then`, one can define combinators for parsing of pairs, mapping of functions on parser results, filtering of parser results etc. For each combinator there are two implementations: functional and C-like. C-like implementation is written in a subset of F* that models imperative language, called Low*. Functions written in Low* operate on memory addresses and use machine integer formalization. They rely on automatic extraction from Low* to C using the tool Kremlin.

The project Narcissus [7] also constructs correct binary parsers from a verified library of combinators written in Coq, but it generates only functional parsers and has the same disadvantages as our preliminary work on floating-point parsing. Built on Narcissus, there is a verified compiler in Coq for parsers and formatters specified by Protocol Buffers.

Galois, Inc. did some work on ASN.1 verification in the past (circa 2012) [23]. It appears that they abandoned the goal of full ASN.1 verification [24] that we pursue in our project with our more pragmatic approach; Galois is now only exploring a limited subset ASN.1 verification adequate for the "vehicle-to-vehicle" (V2V) market [25], but that particular subset has limited applicability, and their effort is limited by constraints of their chosen approach.

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

APPENDIX

A. asn_strtoimax_lim *source*

```
enum asn_strtox_result_e
asn_strtoimax_lim(const char *str, const char **end, intmax_t *intp) {
    int sign = 1;
    intmax_t value;

#define ASN1_INTMAX_MAX ((~(uintmax_t)0) >> 1)
    const intmax_t upper_boundary = ASN1_INTMAX_MAX / 10;
          intmax_t last_digit_max = ASN1_INTMAX_MAX % 10;
#undef ASN1_INTMAX_MAX

    if(str >= *end) return ASN_STRTOX_ERROR_INVAL;

    switch(*str) {
    case '-':
      last_digit_max++;
      sign = -1;
      /* FALL THROUGH */
    case '+':
      str++;
      if(str >= *end) {
        *end = str;
        return ASN_STRTOX_EXPECT_MORE;
      }
    }

    for(value = 0; str < (*end); str++) {
        switch(*str) {
        case 0x30: case 0x31: case 0x32: case 0x33: case 0x34:
        case 0x35: case 0x36: case 0x37: case 0x38: case 0x39: {
            int d = *str - '0';
            if(value < upper_boundary) {
                value = value * 10 + d;
            } else if(value == upper_boundary) {
                if(d <= last_digit_max) {
                    if(sign > 0) {
                      value = value * 10 + d;
                    } else {
                      sign = 1;
                      value = -value * 10 - d;
                    }
                } else {
                  *end = str;
                  return ASN_STRTOX_ERROR_RANGE;
                }
            } else {
              *end = str;
              return ASN_STRTOX_ERROR_RANGE;
            }
        }
            continue;
        default:
            *end = str;
            *intp = sign * value;
            return ASN_STRTOX_EXTRA_DATA;
        }
    }
    *end = str;
    *intp = sign * value;
    return ASN_STRTOX_OK;
}
```

B. ASN.1 Vulnerabilities Analysis

CVE	Product	Property violated	Description
2015-7061	Apple *OS	Memory safety (write)	remote attackers to execute arbitrary code or cause a denial of service (memory corruption) via a crafted certificate
2015-7060	Apple *OS	Memory safety (write)	denial of service (memory corruption) via a crafted certificate
2015-7059	Apple *OS	Memory safety (write)	denial of service (memory corruption) via a crafted certificate
2018-16253	axTLS	Parsing only well-formed input	X.509 metadata verification
2018-16149	axTLS	Memory safety (read)	blindly trusts the declared lengths in the ASN.1 structure
2017-1000416	axTLS		coding error in the ASN.1 parser
2016-9132	Botan	Memory safety (write)	decoding BER data an integer overflow; memory corruption
2015-5726	Botan	Memory safety (read)	remote attackers to cause a denial of service (application crash)
2016-1000342	Bouncy Castle	Parsing only well-formed input	introduction of 'invisible' data into a signed structure
2016-1000338	Bouncy Castle	Parsing only well-formed input	introduction of 'invisible' data into a signed structure
2017-11496	Gemalto ACC	Memory safety (write)	remote attackers to execute arbitrary code
2016-9939	libcrypto++	Bounds on heap size	If there is not enough content octets in the ASN.1 object, then the function will fail and the memory block will be zeroed even if its unused. There is a noticeable delay during the wipe for a large allocation
2015-2806	libtasn1	Memory safety (write)	remote attackers to have unspecified impact via unknown vectors
2016-6129	LibTomCrypt	Memory safety (read)	does not validate that the message length is equal to the ASN.1 encoded data length
2019-9162	Linux Kernel	Memory safety (read/write)	out-of-bounds read and write operations possible
2016-2053	Linux Kernel	Parsing only well-formed input	Skipping non-optional fields after optional
2016-0758	Linux Kernel	Memory safety (read)	local users to gain privileges via crafted ASN.1 data
2019-13470	MatrixSSL	Memory safety (read)	out-of-bounds read
2016-6891	MatrixSSL	Memory safety (read)	(out-of-bounds read) via a crafted ASN.1 Bit Field primitive
2016-1950	Mozilla NSS	Memory safety (write)	remote attackers to execute arbitrary code via crafted ASN.1 data in an X.509 certificate
2015-7182	Mozilla NSS	Memory safety (write)	denial of service (application crash) or possibly execute arbitrary code
2015-7181	Mozilla NSS	Memory safety (write)	improperly restricts access to an unspecified data structure, which allows remote attackers to cause a denial of service (application crash) or possibly execute arbitrary code
2016-5080	Objective Systems ASN1C	Memory safety (write)	execute arbitrary code or cause a denial of service
2018-0739	OpenSSL	Bounds on stack size	exceed the stack given malicious input with excessive recursion
2016-7053	OpenSSL	Memory safety (read)	crash with a NULL pointer dereference
2016-2842	OpenSSL	Memory safety (write)	remote attackers to cause a denial of service (out-of-bounds write or memory consumption)
2016-2176	OpenSSL	Memory safety (read)	remote attackers to obtain sensitive information from process stack memory or cause a denial of service (buffer over-read)
2016-2109	OpenSSL	Bounds on heap size	denial of service (memory consumption)
2016-2108	OpenSSL	Memory safety (write)	(buffer underflow and memory corruption)
2016-0799	OpenSSL	Memory safety (read/write)	remote attackers to cause a denial of service (overflow and out-of-bounds read)
2015-3195	OpenSSL	Memory safety (read)	obtain sensitive information from process memory by triggering a decoding failure in a PKCS#7
2015-3194	OpenSSL	Memory safety (read)	denial of service (NULL pointer dereference and application crash) via an RSA PSS ASN.1 signature
2015-1790	OpenSSL	Memory safety (read)	remote attackers to cause a denial of service (NULL pointer dereference and application crash)
2015-0289	OpenSSL	Memory safety (read)	attackers to cause a denial of service (NULL pointer dereference and application crash)
2015-0287	OpenSSL	Memory safety (write)	denial of service (invalid write operation and memory corruption)
2015-0286	OpenSSL	Memory safety (read)	denial of service (invalid read operation and application crash)
2015-0208	OpenSSL		denial of service (NULL pointer dereference and application crash)
2015-1182	PolarSSL	Memory safety (write)	remote attackers to cause a denial of service (crash) or possibly execute arbitrary code
2018-11058	RSA BSAFE	Memory safety (read)	Buffer Over-Read vulnerability when parsing ASN.1 data
2018-11056	RSA BSAFE	Bounds on stack size	remote attacker could use maliciously constructed ASN.1 data that would exhaust the stack
2018-11054	RSA BSAFE	Native integer overflows	integer overflow vulnerability
2015-7540	Samba	Bounds on heap size	remote attackers to cause a denial of service (memory consumption and daemon crash)
2019-6740	Samsung S9	Memory safety (write)	does not properly validate the length of user-supplied data prior to copying it to a fixed-length heap-based buffer
2017-18315	Snapdragon SOC	Memory safety (read)	Buffer over-read vulnerabilities
2017-9023	strongSwan	Termination	infinite loop in CHOICE parsing
2019-9209	Wireshark	Memory safety (write)	buffer overflow associated with excessive digits in time values
2019-5718	Wireshark	Memory safety (read)	boundary check to make sure we don't go past the end of "ptr"
2019-13619	Wireshark	Memory safety (read)	properly restricting buffer increments
2018-14343	Wireshark		integer overflow in BER lengths
2016-4421	Wireshark	Bounds on stack size	remote attackers to cause a denial of service (deep recursion, stack consumption, and application crash) via a packet that specifies deeply nested data
2016-4418	Wireshark	Memory safety (read)	remote attackers to cause a denial of service (buffer over-read and application crash)
2016-2522	Wireshark	Memory safety (read)	remote attackers to cause a denial of service (out-of-bounds read and application crash)

References

[1] ISO/IEC 8824-1, "Abstract Syntax Notation One (ASN.1): Specification of basic notation," 2015, accessed 30-January-2020. [Online]. Available: http://handle.itu.int/11.1002/1000/12479

[2] R. A. Martin, "Managing vulnerabilities in networked systems," *Computer*, vol. 34, no. 11, pp. 32–38, Nov 2001.

[3] R. Chirgwin, "Guilt by ASN: Compiler's bad memory bug could sting mobes, cell towers telco, embedded systems may inherit remote vulns," 2016. [Online]. Available: https://www.theregister.co.uk/2016/07/19/asn_objective_systems_asn_compiler_memory_bug/

[4] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *ACM Transactions on Computer Systems*, vol. 32, no. 1, pp. 2:1–2:70, Feb. 2014.

[5] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009. [Online]. Available: https://doi.org/10.1145/1538788.1538814

[6] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A Verified Modern Cryptographic Library," in *ACM Conference on Computer and Communications Security (CCS)*, Dallas, United States, Oct. 2017. [Online]. Available: https://hal.inria.fr/hal-01588421

[7] B. Delaware, S. Suriyakarn, C. Pit-Claudel, Q. Ye, and A. Chlipala, "Narcissus: Correct-by-construction derivation of decoders and encoders from binary formats," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3341686

[8] S. Pirelli, A. Zaostrovnykh, and G. Candea, "A formally verified nat stack," in *Proceedings of the 2018 Afternoon Workshop on Kernel Bypassing Networks*, 2018, pp. 8–14.

[9] M. Musuvathi, D. R. Engler *et al.*, "Model checking large network protocol implementations." in *NSDI*, vol. 4, 2004, pp. 12–12.

[10] J.-C. Filliâtre and P. Letouzey, "Extraction of programs in ocaml and haskell," [Accessed 30-January-2020]. [Online]. Available: https://coq.inria.fr/refman/addendum/extraction.html

[11] T. Imada, "Mirageos unikernel with network acceleration for iot cloud environments," in *Proceedings of the 2018 2nd International Conference on Cloud and Big Data Computing*, 2018, pp. 1–5.

[12] M. Dénès, C. Hritcu, L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce, "Quickchick: Property-based testing for coq," in *The Coq Workshop*, 2014.

[13] A. W. Appel, *Program logics for certified compilers*. Cambridge University Press, 2014.

[14] S. Blazy and X. Leroy, "Mechanized semantics for the clight subset of the c language," *Journal of Automated Reasoning*, vol. 43, no. 3, pp. 263–288, 2009.

[15] T. Coq Development Team, *The Coq Proof Assistant Reference Manual, version 8.10.1*, Oct. 2017. [Online]. Available: http://coq.inria.fr

[16] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[17] I. Zaichuk, "Formal verification of floating-point number conversion between ASN.1 BER and IEEE 754 binary encodings," 2019. [Online]. Available: https://github.com/digamma-ai/asn1fpcoq/blob/master/doc/paper/paper.pdf

[18] ISO/IEC 9594-8, "X.590 (10/2019): Public-key and attribute certificate frameworks," http://handle.itu.int/11.1002/1000/14033, 2019, [Online; accessed 30-January-2020].

[19] ISO/IEC 8825-1, "ASN.1 encoding rules: Specification of basic encoding rules (BER), canonical encoding rules (CER) and distinguished encoding rules (DER)," 2015, [accessed 30-January-2020]. [Online]. Available: http://handle.itu.int/11.1002/1000/12483

[20] B. S. Kaliski Jr and C. Redwood City, "A layman's guide to a subset of ASN.1, BER, and der," 1993.

[21] G. Hutton and E. Meijer, "Monadic Parser Combinators," Department of Computer Science, University of Nottingham, Technical Report NOTTCS-TR-96-4, 1996.

[22] T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. Kobeissi, and J. Protzenko, "Everparse: Verified secure zero-copy parsers for authenticated message formats," in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds. USENIX Association, 2019, pp. 1465–1482. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/delignat-lavaud

[23] P. Steckler, "A formal semantics for ASN.1," in *2019 High Confidence Software and Systems Conference, Annapolis, Maryland*, 2019. [Online]. Available: https://cps-vo.org/node/2313

[24] M. Tullsen, "Challenges and possibilities for safe and secure ASN.1 encoders and decoders," in *Fifth Workshop on Language-Theoretic Security (LangSec) Conference*, 2018. [Online]. Available: http://spw18.langsec.org/slides/Tullsen-Safe-ASN1.pdf

[25] M. Tullsen, L. Pike, N. Collins, and A. Tomb, "Formal verification of a vehicle-to-vehicle (V2V) messaging system," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 413–429.