Formal Verification of a Vehicle-to-Vehicle (V2V) Messaging System

Mark Tullsen[1], Lee Pike[2], Nathan Collins[1], and Aaron Tomb[1]

[1] Galois, Inc., Portland, OR, USA {tullsen,conathan,atomb}@galois.com
[2] Groq, Inc. leepike@gmail.com **

Abstract. Vehicle-to-Vehicle (V2V) communications is a "connected vehicles" standard that will likely be mandated in the U.S. within the coming decade. V2V, in which automobiles broadcast to one another, promises improved safety by providing collision warnings, but it also poses a security risk. At the heart of V2V is the communication messaging system, specified in SAE J2735 using the Abstract Syntax Notation One (ASN.1) data-description language. Motivated by numerous previous ASN.1 related vulnerabilities, we present the formal verification of an ASN.1 encode/decode pair. We describe how we generate the implementation in C using our ASN.1 compiler. We define *self-consistency* for encode/decode pairs that approximates functional correctness without requiring a formal specification of ASN.1. We then verify self-consistency and memory safety using symbolic simulation via the *Software Analysis Workbench*.

Keywords: Automated Verification · ASN.1 · Vehicle-to-Vehicle · LLVM · Symbolic execution · SMT solver

1 Introduction

At one time, automobiles were mostly mechanical systems. Today, a modern automobile is a complex distributed computing system. A luxury car might contain tens of millions of lines of code executing on 50-70 microcontrollers, also known as *electronic control units* (ECUs). A midrange vehicle might contain at least 25 ECUs, and that number continues to grow. In addition, various radios such as Bluetooth, Wifi, and cellular provide remote interfaces to an automobile.

With all that code and remotely-accessible interfaces, it is no surprise that software vulnerabilities can be exploited to gain unauthorized access to a vehicle. Indeed, in a study by Checkoway *et al.* on a typical midrange vehicle, for every remote interface, they found some software vulnerability that provided an attacker access to the vehicle's internal systems [4]. Furthermore, in each case, once the interface is exploited, the attackers could parlay the exploit to make arbitrary modifications to other ECUs in the vehicle. Such modifications could include disabling lane assist, locking/unlocking doors, and disabling the brakes. Regardless of the interface exploited, full control can be gained.

** This work was performed while Dr. Pike was at Galois, Inc.

Meanwhile, the U.S. Government is proposing a new automotive standard for vehicle-to-vehicle (V2V) communications. The idea is for automobiles to have dedicated short-range radios that broadcast a *Basic Safety Message* (BSM)—e.g., vehicle velocity, trajectory, brake status, etc.—to other nearby vehicles (within approximately 300 meters). V2V is a crash prevention technology that can be used to warn drivers of unsafe situations—such as a stopped vehicle in the roadway. Other potential warning scenarios include left-turn warnings when line-of-sight is blocked, blind spot/lane change warnings, and do-not-pass warnings. In addition to warning drivers, such messages could have even more impact for autonomous or vehicle-assisted driving. The U.S. Government estimates that if applied to the full national fleet, approximately one-half million crashes and 1,000 deaths could be prevented annually [15]. We provide a more detailed overview of V2V in Section 2.

While V2V communications promise to make vehicles safer, they also provide an additional security threat vector by introducing an additional radio and more software on the vehicle.

This paper presents initial steps in ensuring that V2V communications are implemented securely. We mean "secure" in the sense of having no flaws that could be a vulnerability; confidentiality and authentication are provided in other software layers and are not in scope here. Specifically, we focus on the security of encoding and decoding the BSM. The BSM is defined using ASN.1, a data description language in widespread use. It is not an exaggeration to say that ASN.1 is the backbone of digital communications; ASN.1 is used to specify everything from the X.400 email protocol to voice over IP (VoIP) to cellular telephony. While ASN.1 is pervasive, it is a complex language that has been amended substantially over the past few decades. Over 100 security vulnerabilities have been reported for ASN.1 implementations in MITRE's Common Vulnerability Enumeration (CVE) [14]. We introduce ASN.1 and its security vulnerabilities in Section 3.

This paper presents the first work in formally verifying a subsystem of V2V. Moreover, despite the pervasiveness and security-critical nature of ASN.1, it is the first work we are aware of in which any ASN.1 encoder (that translate ASN.1 messages into a byte stream) and decoder (that recovers an ASN.1 message from a byte stream) has been formally verified. The only previous work in this direction is by Barlas *et al.*, who developed a translator from ASN.1 into CafeOBJ, an algebraic specification and verification system [1]. Their motivation was to allow reasoning about broader network properties, of which an ASN.1 specification may be one part, their work does not address ASN.1 encoding or decoding and appears to be preliminary.

The encode/decode pair is first generated by Galois' ASN.1 compiler, part of the *High-Assurance ASN.1 Workbench* (HAAW). The resulting encode/decode pair is verified using Galois' open source *Software Analysis Workbench* (SAW), a state-of-the-art symbolic analysis engine[6]. Both tools are further described in Section 4.

In Section 5 we state the properties verified: we introduce the notion of self-consistency for encode/decode verification, which approximates functional correctness without requiring a formal specification of ASN.1 itself. Then we describe our approach to verifying the self consistency and memory safety of the C implementation of the encode/decode pair in Section 6 using compositional symbolic simulation as implemented in SAW. In Section 7 we put our results into context.

2 Vehicle-to-Vehicle Communications

As noted in the introduction, V2V is a short-range broadcast technology with the purpose of making driving safer by providing early warnings. In the V2V system, the BSM is the key message broadcasted, up to a frequency of 10Hz (it can be perhaps lower due to congestion control). The BSM must be compatible between all vehicles, so it is standardized under SAE J2735 [7].

The BSM is divided into Part I and Part II, and both are defined with ASN.1. Part I is called the *BSM Core Data* and is part of every message broadcast. Part I includes positional data (latitude, longitude, and elevation), speed, heading, and acceleration. Additionally it includes various vehicle state information including transmission status (e.g., neutral, park, forward, reverse), the steering wheel angle, braking system status (e.g., Are the brakes applied? Are anti-lock brakes available/engaged?, etc.), and vehicle size. Our verification, described in Section 6, is over Part I.

Part II is optional and extensible. Part II could include, for example, regionally-relevant data. It can also include additional vehicle safety data, including, for example, which of the vehicle's exterior lights are on. It may include information about whether a vehicle is a special vehicle or performing a critical mission, such as a police car in an active pursuit or an ambulance with a critical patient. It can include weather data, and obstacle detection.

3 ASN.1

Abstract Syntax Notation One (ASN.1) is a standardized data description language in widespread usage. Our focus in this section is to give a sense of what ASN.1 is as well as its complexity. We particularly focus on aspects that have led to security vulnerabilities.

3.1 The ASN.1 Data Description Language and Encoding Schemes

ASN.1 was first standardized in 1984, with many revisions since. ASN.1 is a data description language for specifying messages; although it can express relations between request and response messages, it was not designed to specify stateful protocols. While ASN.1 is "just" a data description language, it is quite large and complex. Indeed, merely parsing ASN.1 specifications is difficult. Dubuisson notes that the grammar of ASN.1 (1997 standard) results in nearly 400 shift/reduce errors and over 1,300 reduce/reduce errors in a LALR(1) parser generator, while a LL(k) parser generator results in over 200 production rules beginning with the same lexical token [8]. There is a by-hand transformation of the grammar into an LL(1)-compliant grammar, albeit no formal proof of their equivalence [9].

Not only is the syntax of ASN.1 complex, but so is its semantics. ASN.1 contains a rich datatype language. There are at least 26 base types, including arbitrary integers, arbitrary-precision reals, and 13 kinds of string types). Compound datatypes include sum types (e.g., `CHOICE` and `SET`), records with subtyping (e.g., `SEQUENCE`), and recursive types. There is a complex constraint system (ranges, unions, intersections, etc.) on the types. Subsequent ASN.1 revisions support open types (providing a sort of dynamic typing), versioning to support forward/backward compatibility, user-defined constraints, parameterized specifications, and so-called *information objects* which provide an expressive way to describe relations between types.

So far, we have only highlighted the data description language itself. A set of *encoding rules* specify how the ASN.1 messages are serialized for transmission on the wire. Encoder and decoder pairs are always with respect to a specific schema and encoding rule. There are at least nine standardized ASN.1 encoding rules. Most rules describe 8-bit byte (octet) encodings, but three rule sets are dedicated to XML encoding. Common encoding rules include the Basic Encoding Rules (BER), Distinguished Encoding Rules (DER), and Packed Encoding Rules (PER). The encoding rules do not specify the transport layer protocol to use (or any lower-level protocols, such as the link or physical layer).

3.2 Example ASN.1 Specification

To get a concrete flavor of ASN.1, we present an example data *schema*. Let us assume we are defining messages that are sent (TX) and received (RX) in a query-response protocol.

```
MsgTx ::= SEQUENCE {
  txID  INTEGER(1..5),
  txTag UTF8STRING
}
MsgRx ::= SEQUENCE {
  rxID INTEGER(1..7),
  rxTag SEQUENCE(SIZE(0..10)) OF INTEGER
}
```

We have defined two top-level types, each a `SEQUENCE` type. A `SEQUENCE` is an named tuple of fields (like a C struct). The `MsgTx` sequence contains two fields: `txID` and `txTag`. These are typed with built-in ASN.1 types. In the definition of `MsgRx`, the second field, `rxTag`, is the `SEQUENCE OF` structured type; it is equivalent to an array of integers that can have a length between 0 and 10, inclusively. Note that the `txID` and `rxID` fields are *constrained* integers that fall into the given ranges.

ASN.1 allows us to write values of defined types. The following is a value of type MsgTx:

```
msgTx MsgTx ::= {
  txID  1,
  txTag "Some msg"
}
```

3.3 ASN.1 Security

There are currently over 100 vulnerabilities associated with ASN.1 in the MITRE Common Vulnerability Enumeration (CVE) database [14]. These vulnerabilities cover many vendor implementations as well as encoders and decoders embedded in other software libraries (e.g., OpenSSL, Firefox, Chrome, OS X, etc.). The vulnerabilities are often manifested as low-level programming vulnerabilities. A typical class of vulnerabilities are unallowed memory reads/writes, such as buffer overflows and over-reads and NULL-pointer dereferences. While generally arcane, ASN.1 was recently featured in the popular press when an ASN.1 vender flaw was found in telecom systems, ranging from cell tower radios to cellphone baseband chips [11]; an exploit could conceivably take down an entire mobile phone network.

Multiple aspects of ASN.1 combine to make ASN.1 implementations a rich source for security vulnerabilities. One reason is that many encode/decode pairs are hand-written and ad-hoc. There are a few reasons for using ad-hoc encoders/decoders. While ASN.1 compilers exist that can generate encoders and decoders (we describe one in Section 4.1), many tools ignore portions of the ASN.1 specification or do not support all encoding standards, given the complexity and breadth of the language. A particular protocol may depend on ASN.1 language features or encodings unsupported by most existing tools. Tools that support the full language are generally proprietary and expensive. Finally, generated encoders/decoders might be too large or incompatible with the larger system (e.g., a web browser), due to licensing or interface incompatibilities.

Even if an ASN.1 compiler is used, the compiler will include significant hand-written libraries that deal with, e.g., serializing or deserializing base types and memory allocation. For example, the unaligned packed encoding rules (UPER) require tedious bit operations to encode types into a compact bit-vector representation. Indeed, the recent vulnerability discovered in telecom systems is not in protocol-specific generated code, but in the associated libraries [11].

Finally, because ASN.1 is regularly used in embedded and performance-critical systems, encoders/decoders are regularly written in unsafe languages, like C. As noted above, many of the critical security vulnerabilities in ASN.1 encoders/decoders are memory safety vulnerabilities in C.

4 Our Tools for Generating and Verifying ASN.1 Code

We briefly introduce the two tools used in this work. First we introduce our ASN.1 compiler for generating the encode/decode pair, then we introduce the symbolic analysis engine used in the verification.

4.1 *High-Assurance ASN.1 Workbench* (HAAW)

Our *High-Assurance ASN.1 Workbench* (HAAW) is a suite of tools developed by Galois that supports each stage of the ASN.1 protocol development lifecycle: specification, design, development, and evaluation. It is composed of an interpreter, compiler, and validator, albeit with varying levels of maturity. HAAW is implemented in Haskell.

The HAAW compiler is built using semi-formal design techniques and is thoroughly tested to help ensure correctness. The implementation of the HAAW compiler is structured to be as manifestly correct as feasible. It effectively imports a (separately tested) ASN.1 interpreter which is then "partially-evaluated" on the fly to generate code. The passes are as follows: An input ASN.1 specification is "massaged" to a specification-like form which can be interpreted by a built-in ASN.1 interpreter. This specification-like form is combined with the interpreter code and is converted into a lambda-calculus representation; to this representation we apply multiple optimization rules; we finally "sequentialize" to a monadic lambda-calculus (where we are left with the lambda calculus, sequencing operators, and encoding/decoding primitives), this last representation is then transformed into C code. The generated code is linked with a library that encodes and decodes the basic ASN.1 types.

Moreover, while the HAAW compiler improves the quality of the code generated, we verify the generated code and libraries directly, so HAAW is not part of the trusted code-base.

4.2 The Software Analysis Workbench (SAW)

The *Software Analysis Workbench* (SAW)[3] is Galois' open-source, state-of-the-art symbolic analysis engine for multiple programming languages. Here we briefly introduce SAW, see Dockins *et al.* [6] for more details.

An essential goal of SAW is to generate semantic models of programs independent of a particular analysis task and to interface with existing automated reasoning tools. SAW is intended to be mostly automated but supports user-guidance to improve scalability.

The high-level architecture of SAW is shown in Figure 1. At the heart of SAW is *SAWCore*. SAWCore is SAW's intermediate representation (IR) of programs. SAWCore is a dependently-typed functional language, providing a functional representation of the semantics of a variety of imperative and functional languages.

[3] `saw.galois.com`

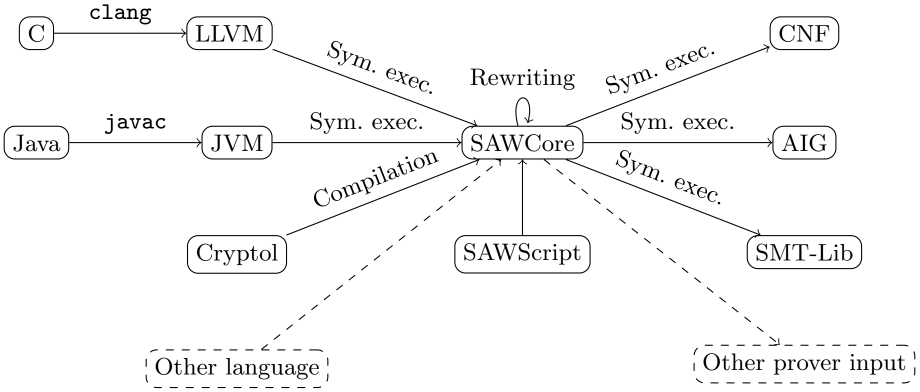

Fig. 1. SAW architecture, reproduced from [6].

SAWCore includes common built-in rewrite rules. Additionally, users can provide domain-specific rewrite rules, and because SAWCore is a dependently-typed language, rewrite rules can be given expressive types to prove their correctness.

SAW currently supports automated translation of both *low-level virtual machine* (LLVM) and *Java virtual machine* (JVM) into SAWCore. Thus, programming languages that can be compiled to these two targets are supported by SAW. Indeed, SAW can be used to prove the equivalence between programs written in C and Java.

SAWCore can also be generated from Cryptol. Cryptol is an open-source language[4] for the specification and formal verification of bit-precise algorithms [10], and we use it to specify portions of our code, as we describe in Section 6.

A particularly interesting feature of Cryptol is that it is a typed functional language, similar to Haskell, but includes a size-polymorphic type system that includes linear integer constraints. To give a feeling for the language, the concatenate operator (#) in Cryptol has the following type:

```
(#) : fst, snd, a (fin fst)
   => [fst]a -> [snd]a -> [fst + snd]a
```

It concatenates two sequences containing elements of type `a`, the first of length `fst`—which is constrained to be of finite (`fin`) length (infinite sequences are expressible in Cryptol)—and the second of length `snd`. The return type is a sequence of `a`'s of length `fst + snd`. Cryptol relies on *satisfiability modulo theories* (SMT) solving for type-checking.

SAWCore is typically exported to various formats supported by external third-party solvers. This includes SAT solver representations (and inverter graphs (AIG), conjunctive normal form (CNF), and ABC's format [3]), as well as SMT-Lib2 [2], supported by a range of SMT solvers.

[4] `https://cryptol.net/`

SAW allows bit-precise reasoning of programs, and has been used to prove optimized cryptographic software is correct [6]. SAW's bit-level reasoning is also useful for encode/decode verification, and in particular, ASN.1's UPER encoding includes substantial bit-level operations.

Finally, SAW includes *SAWScript*, a scripting language that drives SAW and connects specifications with code.

5 Properties: Encode/Decode Self Consistency

Ideally, we would prove full functional correctness for the encode/decode pair: that they correctly implement the ASN.1 UPER encoding/decoding rules for the ASN.1 types defined in SAE J2735. However, to develop a specification that would formalize all the required ASN.1 constructs, their semantics, and the proper UPER encoding rules would be an extremely large and tedious undertaking (decades of "man-years"?). Moreover, it is not clear how one would ensure the correctness of such a specification.

Instead of proving full functional correctness, we prove a weaker property by proving consistency between the encoder and decoder implementations. We call our internal consistency property *self-consistency*, which we define as the conjunction of two properties, *round-trip* and *rejection*. We show that self-consistency implies that decode is the inverse of encode, which is an intuitive property we want for an encode/decode pair.

The *round-trip property* states that a valid message that is encoded and then decoded results in the original message. This is a completeness property insofar as the decoder can decode all valid messages.

A less obvious property is the *rejection property*. The rejection property informally states that any invalid byte stream is rejected by the decoder. This is a soundness property insofar as the decoder *only* decodes valid messages.

In the context of general ASN.1 encoders/decoders, let us fix a schema S and an encoding rule. Let M_S be the set of all ASN.1 abstract messages that satisfy the schema. Let B the set of all finite byte streams. Let $enc_s : M_s \to B$ be an encoder, a total function on M_s. Let $error$ be a fixed constant such that $error \notin M_s$. Let the total function $dec_s : B \to (M_s \cup \{error\})$ be its corresponding decoder.

The round-trip and rejection properties can respectively be stated as follows:

Definition 1 (Round-trip).

$$\forall m \in M_s . dec_s(enc_s(m)) = m.$$

Definition 2 (Rejection).

$$\forall b \in B . dec_s(b) = error \vee enc_s(dec_s(b)) = b.$$

The two properties are independent: a decoder could properly decode valid byte streams while mapping invalid byte streams to valid messages. Such a decoder would be allowed by Round-trip but not by Rejection. An encode/decode pair that fails the Rejection property could mean that *dec* does not terminate normally on some inputs (note that *error* is a valid return value of *dec*). Clearly, undefined behavior in the decoder is a security risk.

Definition 3 (Self-consistency). *An encode/decode pair enc_S and dec_S is* self-consistent *if and only if it satisfies the round-trip and rejection properties.*

Self-consistency does not require any reference to a specification of ASN.1 encoding rules, simplifying the verification. Indeed, they are applicable to any encode/decode pair of functions.

However, as noted at the outset, self-consistency does not imply ful functional correctness. For example, for an encoder enc_S and decoder dec_S pair, suppose the messages $M_S = \{m_0,\ m_1\}$ and the byte streams B includes $\{b_0,\ b_1\} \subseteq B$. Suppose that according to the specification, it should be the case that $enc_S(m_0) = b_0$, $enc_S(m_1) = b_1$, $dec_s(b_0) = m_0$ and $dec(b_1) = m_1$, and for all $b \in B$ such that $b \neq b_0$ and $b \neq b_1$, $dec_S(b) = error$. However, suppose that in fact $enc_S(m_0) = b_1$, $enc_S(m_1) = b_0$, $dec_S(b_0) = m_1$ and $dec_S(b_1) = m_0$, and for all other $b \in B$, $dec(b) = error$. Then enc_S and dec_S satisfy both the round-trip and rejection properties, while being incorrect.

That said, if self-consistency holds, then correctness reduces to showing that either encoder or decoder matches its specification, but showing both hold is unnecessary.

In our work, we formally verify self-consistency and memory safety. We also give further, informal, evidence of correctness by both writing individual test vectors and by comparing our test vectors to that produced by other ASN.1 compilers.

6 Verification

Figure 2 summarizes the overall approach to generating and verifying the encode/decode pair, which we reference throughout this section.

6.1 First Steps

The given SAE J2735 ASN.1 specification (J2735.asn) is given as input to HAAW to generate C code for the encoder and decoder. A HAAW standard library is emitted (the dotted line from HAAW to `libHAAW.c` in Figure 2 denotes that the standard library is not specific to the SAE-J2735 specification and is not compiled from HAAW).

We wrote the round-trip and rejection properties (Section 5) as two C functions. For example, the round-trip property is encoded, approximately, as follows:

```
bool round_trip(BSM *msg_in) {
  unsigned char str[BUF_SIZE];
  enc(msg_in, str);
  BSM *msg_out;
```

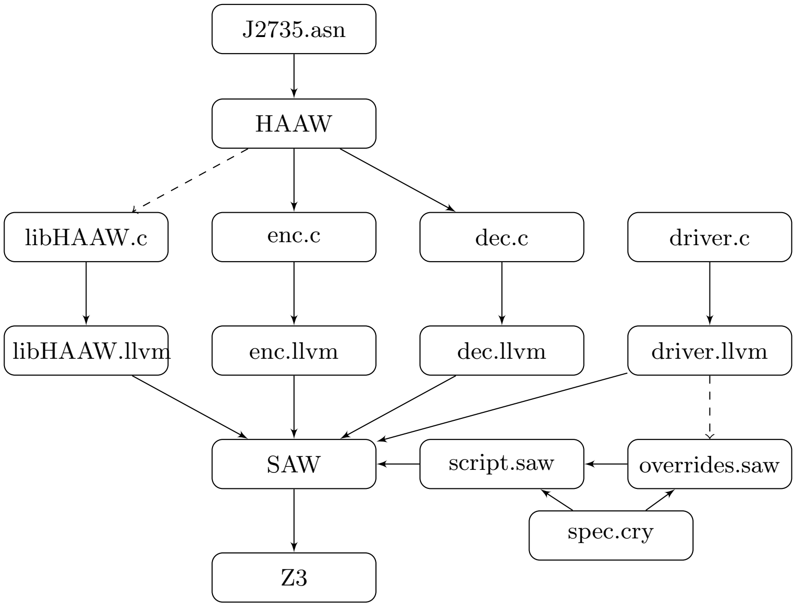

Fig. 2. Code generation and verification flow.

```
  dec(msg_out, str);
  return equal_msg(msg_in, msg_out);
}
```

The actual `round_trip` property is slightly longer as we need to deal with C level setup, allocation, etc. This is why we chose to implement this property in C (rather than in SAWScript).

Now all we need to do is verify, in SAWScript, that the C function `round_trip` returns 1 *for all inputs*. At this point, it would be nice to say the power of our automated tools was sufficient to prove `round_trip` without further programmer intervention. This, unsurprisingly, was not the case. Most of the applications of SAW have been to cryptographic algorithms where code typically has loops with statically known bounds. In our encoder/coder code we have a number of loops with unbounded iterations: given such code we need to provide some guidance to SAW.

In the following sections we present how we were able to use SAW, as well as our knowledge of our specific code, to change an intractable verification task into one that could be done (by automated tools) in less than 5 hours. An important note: the rest of this section describes SAW techniques that allow us to achieve tractability, they do not change the soundness of our results.

6.2 Compositional Verification with SAW Overrides

SAW supports *compositional verification.* A function (e.g., compiled from Java or C) could be specified in Cryptol and verified against its specification. That Cryptol specification can then be used in analyzing the remainder of the program, such that in a symbolic simulation, the function is replaced with its specification. We call this replacement an *override.* Overrides can be used recursively and can dramatically improve the scalability of a symbolic simulation. SAW's scripting language ensures by construction that an override has itself been verified.

Overrides are like lemmas, we prove them once, separately, and can re-use them (without re-proof). The lemma that an override provides is an equivalence between a C function and a declarative specification provided by the user (in Cryptol). The effort to write a specification and add an override is often required to manage intractability of the automated solvers used.

6.3 Overriding "copy_bits" in SAW

There are two critical `libHAAW` functions that we found to be intractable to verify using symbolic simulation naively. Here we describe generating overrides for one of them:

```
copy_bits
  ( unsigned char * dst
  , uint32_t *dst_i
  , unsigned char const * src
  , uint32_t *src_i
  , uint32_t const length)
{
  uint32_t src_i_bound = *src_i + length;
  while (*src_i < src_i_bound) {
    copy_overlapping_bits (dst, dst_i, src, src_i, src_i_bound);
  }
  return 0;
}
```

The above function copies `length` bits from the `src` array to the `dst` array, starting at the bit indexed by `src_i` in `src` and index `dst_i` in `dst`; `src_i` and `dst_i` are incremented by the number of bits copied; `copy_overlapping_bits` is a tedious but loop-free function with bit-level computations to convert to/from a bit-field and byte array. This library function is called by both the encoder and decoder.

One difficulty with symbolically executing `copy_bits` with SAW is that SAW unrolls loops. Without a priori knowledge of the size of `length` and `src_i`, there is no upper bound on the number of iterations of the loop. Indeed, memory safety is dependent on an invariant holding between the indices, the number of bits to copy, and the length of the destination array: the length of the destination array is not passed to the function, so there is no explicit check to ensure no write-beyond-array in the destination array.

Even if we could fix the buffer sizes and specify the relationship between the length and indexes so that the loop could be unrolled in theory, in practice, it would still be computationally infeasible for large buffers. In particular, we would have to consider every valid combination of the length and start indexes, which is cubic in the bit-length of the buffers.

To override copy_bits, we write a specification of copy_bits in Cryptol. The specification does not abstract the function, other than eliding the details of pointers, pointer arithmetic, and destructive updates in C. The specification is given below:

```
copy_bits : dst_n, src_n
            [dst_n][8] -> [32] -> [src_n][8] -> [32] -> [32]
            -> ([dst_n][8], [32], [32])
copy_bits dst0 dst_i0 src src_i0 length = (dst1, dst_i1, src_i1)
  where
  dst_bits0 = join dst0
  src_bits0 = join src

  dst1 = split (copy dst_bits0 0)
  copy dst_bits i =
    if i == length
    then dst_bits
    else copy dst_bits'' (i + 1)
    where
    dst_bits'' = update dst_bits (dst_i0 + i)
                   (src_bits0 @ (src_i0 + i))

  dst_i1 = dst_i0 + length
  src_i1 = src_i0 + length
```

We refer to the *Cryptol User Manual* for implementation details [10], but to provide an intuition, we describe the type signature (the first three lines above): the type is polymorphic, parameterized by dst_n and src_n. A type [32] is a bit-vector of length 32. A type [dst_n][8] is an array of length dst_n containing byte values. The function takes a destination array of bytes, a 32-bit destination index, a source array of bytes, a source index, an a length, and returns a triple containing a new destination array, and new destination and source indices, respectively. Because the specification is pure, the values that are destructively updated through pointers in the C implementation are part of the return value in the specification.

6.4 Multiple Overrides for "copy_bits" in SAW

Even after providing the above override for copy_bits, we are *still* beyond the limits of our underlying solvers to automatically prove the equivalence of copy_bits with its Cryptol specification.

However, we realize that for the SAE J2735 encode/decode, copy_bits is called with a relatively small number of specific concrete values for the sizes of the dst and src arrays, the indexes `dst_i` and `src_i`, and the length of bits to copy `length`. The only values that we need to leave symbolic are the bit values within the `dst` and `src` arrays. Therefore, rather than creating a single override for an arbitrary call to `copy_bits`, we generate separate overrides for each unique set of "specializable" arguments, i.e., `dst_i`, `src_i`, and `length`.

Thus we note another feature of SAW: SAW allows us to specify a set of concrete function arguments for an override; for each of these, SAW will specialize the override. (I.e., it will prove each specialization of the override.) In our case this turns one intractable override into 56 tractable ones. The 56 specializations (which corresponds to the number of SEQUENCE fields in the BSM specification) were not determined by trial and error but by running instrumented code.

It is important to note that the consequence of a missing overrride specialization cannot change the soundness of SAW's result: Overrides in SAW cannot change the proof results, they only change the efficiency of proof finding. If we had a missing override specialization for `copy_bits` we would only be back where we started: a property that takes "forever" to verify.

This approach works well for the simple BSM Part I. However, once we begin to verify encoders/decoders for more complex ASN.1 specifications (e.g., containing `CHOICE` and `OPTIONAL` constructs), this method will need to be generalized.

6.5 Results

A SAW script (`script.saw`) ties everything together and drives the symbolic execution in SAW and lifts LLVM variables and functions into a dependent logic to state pre- and post-conditions and provide Cryptol specifications as needed. Finally, SAW then generates a SMT problem; Z3 [5] is the default solver we use.

Just under 3100 lines of C code were verified, not counting blank or comment lines. The verification required writing just under 100 lines of Cryptol specification. There are 1200 lines of SAW script auto-generated by the test harness in generating the override specifications. Another 400 lines of SAW script is hand-written for the remaining overrides and to drive the overall verification.

Executed on a modern laptop with an Intel Core i7-6700HQ 2.6 GHz processor and 32G of memory, the verification takes 20 minutes to prove the round-trip property and 275 minutes to prove the rejection property. The round-trip property is less expensive to verify because symbolic simulation is sensitive to branching, and for the round-trip property, we assert the data is valid to start, which in turn ensures that all of the decodings succeed. In rejection, on the other hand, we have a branch at each primitive decode, and we need to consider both possibilities (success and failure).

7 Discussion

7.1 LLVM and Definedness

Note that our verification has been with respect to the LLVM semantics not the C source of our code. SAW does not model C semantics, but inputs LLVM as the program's semantics (we use CLANG to generate LLVM from the C). By verifying LLVM, SAW is made simpler (it need only model LLVM semantics rather than C) and we can do inter-program verification more easily. The process of proving that a program satisfies a given specification within SAW guarantees definedness of the program (and therefore memory safety) as a side effect. That is, the translation from LLVM into SAWCore provides a well-defined semantics for the program, and this process can only succeed if the program is well-defined. In some cases, this well-definedness is assumed during translation and then proved in the course of the specification verification. For instance, when analyzing a memory load, SAW generates a semantic model of what the program does if the load was within the bounds of the object it refers to, and generates a side condition that the load was indeed in bounds.

Verifying LLVM rather than the source program is a double-edged sword. On the one hand, the compiler front-end that generates LLVM is removed from the trusted computing base. On the other hand, the verification may not be sound with respect to the program's source semantics. In particular, C's undefined behaviors are a superset of LLVM's undefined behaviors; a compiler can soundly remove undefined behaviors but not introduce them. For example, a flaw in the GCC compiler allowed the potential for an integer overflow when multiplying the size of a storage element by the number of elements. The result could be insufficient memory being allocated, leading to a subsequent buffer overflow. CLANG, however, introduces an implicit trap on overflow [12].

Moreover, the LLVM language reference does not rigorously specify well-definedness, and it is possible that our formalization of LLVM diverges from a particular compiler's [13].

7.2 Other Assumptions

We made some memory safety assumptions about how the encode/decode routines are invoked. First, we assume that the input and output buffers provided to the encoder and decoder, respectively, do not alias. We also assume that each buffer is 37 bytes long (sufficient to hold a BSM with Part I only). A meta argument shows that buffers of *at least* 37 bytes are safe: we verify that for all 37-byte buffers, we never read or write past their ends. So, if the buffers were longer, we would never read the bytes above the 37th element.

For more complex data schemas (and when we extend to BSM Part II) whose messages require a varying octet size, we would need to ensure the buffers are sufficiently large for all message sizes.

7.3 Proof Robustness

By "proof robustness" we mean how much effort is required to verify another protocol or changes to the protocol. We hypothesize that for other protocols that use UPER and a similar set of ASN.1 constructs, the verification effort would be small. Most of our manual effort focused on the `libHAAW` libraries, which is independent of the particular ASN.1 protocol verified. That said, very large protocol specifications may require additional proof effort to make them compositional.

In future work, we plan to remove the need to generate overrides as a separate step (as described in Section 6.2) by modifying HAAW to generate overrides as it generates the C code.

8 Conclusion

Hopefully we have motivated the security threat to V2V and the need for eliminating vulnerabilities in ASN.1 code. We have presented a successful application of automated formal methods to real C code for a real-world application domain.

There are some lessons to be learned from this work:

(1) Fully automated proofs of correctness properties are possible, but not trivial. The encoding of properties into C and SAWScript and getting the proofs to go through took one engineer approximately 3 months, this engineer had some experience with SAW; we were also able to get support and bug-fixes from the SAW developers. (It also helped that the code was bug-free so no "verification" time was spent on finding counter-examples and fixing code.)

(2) The straightforward structure of the C used in the encode/decode routines made them more amenable to automated analysis (see Section 6). It certainly helped that the code verified was compiler-generated and was *by design* intended to be, to some degree, manifestly correct. The lesson is not "*choose* low-hanging fruit" but "*look*, low-hanging fruit in safety critical code" or possibly even "*create* low-hanging fruit!" (by using simpler C).

(3) For non-trivial software, the likelihood of having a correct specification at hand, or having the resources to create it, is quite slim! For instance, to fully specify correct UPER encoding/decoding for arbitrary ASN.1 specifications would be a Herculean task. But in our case, we formulated two simple properties—Round-Trip and Rejection—and by proving them we have also shown memory safety and some strong (not complete, see Section 5) guarantees of functional correctness. This technique could be applied to any encode/decode pair.

There are many ways we hope to extend this work:

(1) We plan to extend our verification to the full BSM. This now gets us to more challenging ASN.1 constructs (e.g., `CHOICE`) that involve a more complicated control-flow in the encoders/decoders. We do not expect a proof to be found automatically, but our plan is to generate lemmas with the generated C code that will allow proofs to go through automatically.

(2) Once we can automatically verify the full BSM, we expect to be able to perform a similar fully-automatic verification on many ASN.1 specifications (most do not use the full power of ASN.1). We would like to explore what properties of a given ASN.1 specification might guarantee the ability to perform such a fully-automatic verification.

(3) By necessity, parts of our SAWScript and the verification properties have a dependence on the particular API of the HAAW compiler (how abstract values are encoded, details of the encoding/decoding functions, memory-management

design choices, etc.). Currently the authors are working on generalizing this so that one can abstract over ASN.1-tool-specific API issues. The goal is to be able to extend our results to other encode/decode pairs (generated by hand or by other ASN.1 compilers.).

(4) Note that the self-consistency property is generic (and has no reference to ASN.1). As a result, we believe our work can be extended to encode/decode pairs on non-ASN.1 data.

Acknowledgments. This work was performed under subcontract to Battelle Memorial Institute for the National Highway Safety Transportation Administration (NHTSA). We thank Arthur Carter at NHTSA and Thomas Bergman of Battelle for their discussions and guidance. Our findings and opinions do not necessarily represent those of Battelle or the United States Government.

References

1. Barlas, K., Koletsos, G., Stefaneas, P.S., Ouranos, I.: Towards a correct translation from ASN.1 into CafeOBJ. IJRIS **2**(3/4), 300–309 (2010)
2. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.5. Tech. rep., Department of Computer Science, The University of Iowa (2015), available at w.SMT-LIB.org
3. Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Proceedings of the 22nd International Conference on Computer Aided Verification (CAV). pp. 24–40. Springer-Verlag (2010)
4. Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., Kohno, T.: Comprehensive experimental analyses of automotive attack surfaces. In: USENIX Security (2011)
5. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 337–340. Springer-Verlag (2008)
6. Dockins, R., Foltzer, A., Hendrix, J., Huffman, B., McNamee, D., Tomb, A.: Constructing semantic models of programs with the software analysis workbench. In: Verified Software. Theories, Tools, and Experiments - 8th International Conference, VSTTE, Revised Selected Papers. pp. 56–72 (2016)
7. DSRC Technical Committee: Dedicated Short Range Communications (DSRC) message set dictionary (j2735_20103). Tech. rep., SAE International (20016)
8. Dubuisson, O.: ASN.1 Communication between heterogeneous Systems. Elsevier-Morgan Kaufmann (2000)
9. Fouquart (P.), D.O., (F.), D.: Une analyse syntaxique d'ASN.1:1994. Tech. Rep. Internal Report RP/LAA/EIA/83, France Télécom R&D (March 1996)
10. Galois, Inc.: Cryptol: the language of cryptography. Galois, Inc. (2016), available at http://www.cryptol.net/files/ProgrammingCryptol.pdf
11. Goodin, D.: Software flaw puts mobile phones and networks at risk of complete takeover. Ars Technica (2016)
12. Lattner, C.: What every C programmer should know about undefined behavior #3/3. Online blog (May 2011), http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_21.html

13. Lee, J., andYoungju Song, Y.K., Hur, C.K., Das, S., Majnemer, D., Regehr, J., Lopes, N.P.: Taming undefined behavior in LLVM. In: Proceedings of 38th Conference on Programming Language Design and Implementation (PLDI) (2017)
14. MITRE: Common vulnerabilities and exposures for ASN.1. Website (February 2017), available at https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=ASN.1
15. U.S. Dept. of Transportation: Fact sheet: Improving safety and mobility through vehicle-to-vehicle communications technology. Online (2016), available at https://icsw.nhtsa.gov/safercar/v2v/pdf/V2V_NPRM_Fact_Sheet_121316_v1.pdf