U.S. Department
of Transportation
**National Highway
Traffic Safety
Administration**

NHTSA

DOT HS 812 484

March 2018

Development of a Verified Message Parser For V2V Communications

Disclaimer

This publication is distributed by the U.S. Department of Transportation, National Highway Traffic Safety Administration, in the interest of information exchange. The opinions, findings, and conclusions expressed in this publication are those of the authors and not necessarily those of the Department of Transportation or the National Highway Traffic Safety Administration. The United States Government assumes no liability for its contents or use thereof. If trade or manufacturers' names or products are mentioned, it is because they are considered essential to the object of the publication and should not be construed as an endorsement. The United States Government does not endorse products or manufacturers.

Suggested APA Format Citation:

Batelle Memorial Institute. (2018, March). *Development of a verified message parser for V2V communications* (Report No. DOT HS 812 484). Washington, DC: National Highway Traffic Safety Administration.

Technical Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
DOT HS 812 484		

4. Title and Subtitle	5. Report Date
Development of a Verified Message Parser for V2V Communications	March 2018
	6. Performing Organization Code

7. Authors	8. Performing Organization Report No.
Mark Tullsen, Thomas Bergman, Eric Woldridge, Arthur Carter	

9. Performing Organization Name and Address		10. Work Unit No. (TRAIS)
Battelle Memorial Institute 505 King Ave. Columbus, Ohio 43201	Galois 421 SW 6th Ave. Suite 300 Portland, OR 97204	11. Contract or Grant No. DTNH22-14-D-00327L/0002

12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
National Highway Traffic Safety Administration Office of Program Development and Delivery 1200 New Jersey Avenue SE Washington, DC 20590	Final Report
	14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

The National Highway Traffic Safety Administration vehicle-to-vehicle (V2V) communications proposal promises to increase safety and coordination among vehicles on highways within the United States. However, as evidenced by recent cyber attack media events, today's automobiles are vulnerable to cyber attacks. V2V capabilities will only increase the attack surfaces available for these types of intrusions. Worse, V2V raises the possibility of leveraging a successful attack on one vehicle to disrupt others, much like an internet worm.

This project focused on building high-assurance, mathematically verified parsers and serializers (i.e., decoders and encoders) for the SAE J2735 Basic Safety Message that are low-cost, reproducible, and reconfigurable (i.e., easily generating new proofs for additional functionality). The project leveraged tools that Galois previously built for the U.S. Government: (1) the High-Assurance ASN.1 Workbench and (2) the Software Analysis Workbench.

17. Key Words	18. Distribution Statement
DSRC, basic safety message (BSM), V2V, parser	Document is available to the public from the National Technical Information Service www.ntis.gov

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified		25	

Form DOT F 1700.7 (8-72) **Reproduction of completed page authorized**

Acronyms

DIMACS	Center for Discrete Mathematics and Theoretical Computer Science ("Center for" is not part of the acronym). It is a partnership of more than 350 members headquartered at Rutgers University.
AES	Advanced Encryption Standard, a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology in 2001
AIGER	a format, library and set of utilities for And-Inverter Graphs (AIGs).
ASN.1	an interface description language for defining data structures that can be serialized and deserialized in a standard, cross-platform way
BSM	basic safety message
C	a high-level and general-purpose programming language ideal for developing firmware or portable applications.
DER	Distinguished Encoding Rules, a subset of Basic Encoding Rules (BER), and give exactly one way to represent any ASN.1 value as an octet string.
DoS	denial of service
DSRC	Dedicated Short Range Communications
ECSD	elliptic curve digital signature algorithm
GCC	GNU complier collection, a compiler system supporting programming languages Such as C, C++, Objective-C, Objective-C++, Fortran, Java, Ada, and Go among others.
HAAW	High-Assurance ASN.1 Workbench, an ASN.1 compiler, interpreter, test case generator, and support for formal verification of the correctness and safety of the generated code
HASNC	High-Assurance ASN.1 Complier
IEEE 1609	IEEE DSRC Family of Standards
INFOSEC	information security
LLVM	lower level virtual machine
QuickCheck	a combinator library originally written in Haskell, designed to assist in software testing by generating test cases for test suites.
RV-Match	Runtime Verification match is a program analysis environment.
SAW	Software Analysis Workbench
SAE J2735 BSM	SAE DSRC Basic Message Set Standard
SCADA	Supervisory Control and Data Acquisition, a control system architecture that uses computers, networked data communications, and graphical user interfaces for supervisory management to interface to the process plant or machinery.
SHA-2	Secure Hash Algorithm 2
SQL	Structured Query Language
SMT	Satisfiability Modulo Theories
SMTLIB	Satisfiability Modulo Theories Library
UPER	Unaligned Packed Encoding Rules
V2V	vehicle to vehicle
x.509	a standard defining format of public key certificates.

Table of Contents

Executive Summary

Vehicle-to-vehicle communications technology offers great opportunities to enhance safety and coordination among vehicles on U.S. highways. However, as evidenced by demonstrated cases of cyber vulnerabilities,[1] today's vehicles that rely on software to a much greater extent for functionality and increasingly offer connectivity are also vulnerable to cyber attacks. The V2V communications path will add to the attack surfaces available for potential intrusions. V2V also raises a question around the possibility of a potential successful attack on one vehicle that could potentially disrupt others, much like an Internet worm.

Most cyber vulnerabilities occur at the interfaces. Attackers cannot penetrate a vehicle through secure interfaces. But discovery of only one vulnerability could enable penetrating a system. Traditional testing-based validation is insufficient to protect modern systems, because no matter how many tests are executed, there is no *guarantee* that the system is free of vulnerabilities.

This project focused on building a high-assurance, mathematically verified parser and serializer (i.e., decoder and encoder) for the SAE J2735 Basic Safety Message that is low-cost, reproducible, and reconfigurable (i.e., easily generating new proofs for additional functionality). The project leveraged tools that the contractor previously built for the U.S. Government: (1) the High-Assurance ASN.1 Workbench and (2) the Software Analysis Workbench.

This study generated an encoder and decoder for SAE J2735 BSM Part I and formally verified their correctness. Both are available to the public for use.

[1] 60 Minutes. (2015, February 6). "Car Hacked on 60 Minutes." (Web page). New York: CBS News. Available at www.cbsnews.com/news/car-hacked-on-60-minutes/

1 Summary

1.1 Purpose

The purpose of the project was to develop a formally verified, secure reference parser for the V2V Basic Safety Message. The parser must operate over 5.9 GHz Dedicated Short Range Communications, such that when the reference parser is implemented, the DSRC communication interface would be considered robust with mathematical proof.

Specific objectives include a parser that:

1. Precisely specifies all legal inputs using a grammar, whereby inputs only represent data, not computation, and all data types are unambiguous (i.e., not machine-dependent);
2. Specifies maximum sizes for messages and message content to help reduce denial-of-service (DoS) and overflow attacks;
3. Checks every input to confirm that it conforms to the input specification;
4. Provides for a mechanism that can link interface messages to mission–critical functionality; and
5. Rejects non-required messages.

1.2 Approach

This study leveraged two tools previously built for the U.S. Government: (1) the High-Assurance ASN.1 Workbench and (2) the Software Analysis Workbench. HAAW is a tool for analyzing, testing, and synthesizing parsers and serializers for ASN.1 specification. SAW is a mature verification tool in use by the U.S. Government to verify cryptographic programs. Because SAW is an automated tool, verification is mostly automatic, and the expertise required by users is greatly reduced.

1.3 Overview of This Report

- Section 2 provides the background and discusses the problem space.
- Section 3 describes the approach to developing a verified parser.
- Section 4 discusses the success and challenges in the execution of this project.
- Section 5 is the conclusion.

2 Background and Problem Statement

2.1 Introduction

Vehicle-to-vehicle communications will enhance the safety of vehicles implementing connectivity by periodically passing status information to surrounding vehicles. The basic safety message, defined in the DSRC Message Set Dictionary and documented in the SAE J2735 standard, provides the definition and structure of the current suite of DSRC-based messages. The BSM includes two parts, with Part 1 containing core data elements such as vehicle size, position, speed, heading, acceleration, and brake status. BSM Part 1 messages are transmitted approximately 10 times per second. BSM Part 2 contains a variable set of data elements (such as antilock brake and stability control status, which vary by make and model) and is sent less frequently.[2]

The BSM is intended for the low latency, localized broadcast required by V2V safety applications. However, it can also be used to provide vehicle data in support of mobility applications. Where roadside units are deployed, the BSM can support such mobility applications as cooperative adaptive cruise control, speed harmonization, queue warning, transit signal priority, and incident scene work alerts. When Part 2 data elements are added, weather data and vehicle data can also be exchanged.[3] The BSM is specific to vehicle usage as it contains mandatory parameters that can only be provided or make sense for a motorized-vehicle.

2.2 Problem Statement

Devices and systems are increasingly being connected for improved functionality, for greater ease of use, and for monitoring and maintenance. But the same connectedness that benefits legitimate users also allows others to connect, providing access they would not have otherwise had, sometimes from great distances. News stories are emerging about the vulnerabilities that have been exposed by connecting legacy SCADA systems to the Internet,[4] systems such as dams or temperature controllers in schools.

The world is moving to a place where vehicles are increasingly communicating with other networks or to one another. The risks and challenges multiply and the attack surfaces expand in this scenario. Many vehicle systems have already been demonstrated to include security vulnerabilities.[5] Broad-scale connectivity could expose these insecurities dramatically.

Fortunately, there are emerging approaches to design-in protections which could greatly limit impacts from adversarial attacks. This project contributes to security-by-design by addressing a vehicle's DSRC communication interface that will eliminate a whole class of security vulnerabilities – security risks at the DSRC communications interface by securing the message parsers and serializers.

2.3 Vulnerabilities at the Interfaces

Protecting interfaces have a profound effect on security because most insecurities in computing systems are manifested as (1) the ability to carry out unexpected computation on a target platform by (2) carefully crafting inputs that violate the input assumptions of the design. Inputs are provided to the system only at

[2] Office of the Assistant Secretary for Research and Technology. (n.d.). About ITS Standards. (Web page and portal). Washington, DC: Author. Available at www.standards.its.dot.gov/LearnAboutStandards/NationalITSArchitecture)
[3] Ibid.
[4] Shodan. In Wikipedia. Retrierval date unknown, from http://en.wikipedia.org/wiki/Shodan_(website)
[5] Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., … & Savage, S.*Experimental security analysis of a modern automobile*, 31st IEEE Symposium on Security and Privacy, SP 2010 - Berkeley/Oakland, CA, May 16-18, 2010..

its interfaces. Some types of input assumptions are visible in the source code; SQL injection attacks are an example of this. Careless web developers would ask the user for input such as a name or address, and - making the assumption that the user is only providing a name or address - would paste the input directly into an SQL code template, which they proceed to execute. So long as the user supplies just a name or address, everything may be fine. But if a user supplies a piece of active SQL code instead of a name, the website still pastes it into the code template, and now starts executing the supplied code, however damaging it may be. SQL injection attacks are prevented by carefully specifying the set of legal inputs, and then checking that the inputs strictly conform to the legal set. Only then is the data used in a query.

In contrast, some assumptions are not visible in the source code but arise from details of how the implementation language executes its programs. Buffer overflow is a classic example of this input assumption: the implementation is expecting an input of a certain size, but when the user provides a larger input, the system naively places the extra data right on top of where it was storing information of what code to execute next! Consequently, carefully crafted overflowing input can allow an attacker to run whatever code they want, right at the heart of the system.

Sometimes the assumptions are in library code. For example, the specific character sequences that are normally used to indicate string endings can cause different systems to behave differently. Such "null terminators" in the middle of a name in an X.509 security certificate can cause some systems to see different names than others.[6] If the input certificate had the name "bank.com00.badguy.com" some systems would see this only as "badguy.com" but Internet Explorer and Firefox will see just "bank.com". If one system incorrectly validates the certificate and then passes it to another who trusts the validation, then a security vulnerability is created.

Regardless of the type of assumption violation that results in a vulnerability, the input is provided to the system at its communication interface. Creating a secure parser as a part of the system, which carefully checks all inputs to verify they fall within the expected range of input expectations is one way to protect the communications interfaces and help to secure the system.

[6] Kaminsky, D., Sassaman, L., & Patterson, M. (2009, August). *PKI Layer Cake: New Collision Attacks against the 4 Global X.509 Infrastructure*. In R. Sion, (ed.), Financial Cryptography and Data Security. FC 2010. Lecture Notes in Computer Science, vol 6052. Berlin: Springer. doi.org/10.1007/978-3-642-14577-3_22

3 Technical Approach: Provably Secure Parsers

This project focused on creating a *verified* parser and serializer for the basic safety message of SAE J2735.[7] SAE J2735 (defined using ASN.1) only defines the messages between vehicles at the application layer. A given implementation will also have code that handles communication at the transport layer and below; i.e., implementing the IEEE 1609 family of standards.

In the following, the study addresses how to secure the most vulnerable code, the BSM parser (as defined in SAE J2735), i.e., at the application layer. Although there could be flaws in the implementations of lower layers, it is unlikely that such flaws would result in an exploitable vulnerability, given a secure ASN.1 parser. Creation of a verified parser and serializer requires functional correctness and execution safety.

3.1 Verification Properties

Verification requires proving *functional correctness*. The two properties a verified parser and serializer require to assure functional correctness are *conformance* and *rejection*.

- *Conformance*: parse(serialize(*m*)) = *m*, for all legal messages of a given type.
- *Rejection*: parse(*i*) = REJECT, for all non-valid inputs *i*.

The conformance property guarantees that any legal message (according to SAE J2735) can be serialized, that its serialization can be parsed, and finally, that the resulting message is equal to the original message. The rejection property guarantees that non-valid inputs will be recognized as such. The non-valid inputs, in this case, are byte streams (*bs*) such that there exists no valid message *m* such that

```
serialize(m) = bs
```

"REJECT" is left abstract here and results from a rejected message. Rejection may cause anything from logging a failure to evasive action; how to respond to a rejection is beyond the scope of the parser. Indeed, parser failure may also result from non-malicious sources, such as physical interference.

Conformance ensures that the parser and serializer work as expected; a brick satisfies the rejection property, but not the conformance property.

The conformance property can be formalized as follows, stating that any legal message that is serialized then parsed returns the original message:

```
∀m, bs. Legal(m) ∧ bs = serialize(m) ⇒ parse(bs) = m
```

[7] The terms "parser" and "decoder" are used as well as the terms "serializer" and "encoder" interchangeably in this report.

If this property holds, then it can be reasoned any legal message can be both serialized and parsed correctly. For the rejection property, the formalization is that for any byte stream that is parsed, the resulting value is either a rejection message or the message conforms (but not both). What cannot happen is that a parsed byte stream returns a value *m* that is not a legal member of the type being considered:

```
∀bs. parse(bs) = REJECT ⊻ Legal(parse(bs))
```

Functional correctness is only half of the story; if the program that implements the parser can be corrupted, then its behavior can be circumvented. For example, as mentioned in the introduction, buffer overflow attacks are a way for an attacker to change the control flow of a program so that the original program's properties no longer matter. The other required property is *execution safety*. Execution safety ensures that no unintended violation of the program's semantics is possible. Execution safety is a property of the methods used to implement the parser and serializer. Execution safety includes *memory safety*: that a program does not allow the reading or writing of unintended memory. Memory safety includes a variety of specific vulnerabilities such as buffer overflows, uninitialized variables, stack overflows, and null pointer access. An implementation must be free from memory vulnerabilities in order to guarantee execution safety. Other kinds of undefined behavior are undefined arithmetic operations such as division by zero, signed integer overflow, and oversized bit-shifts.

In the general case, execution safety is undecidable. For example, in a language like C that allows pointer arithmetic, one can construct arbitrarily complex mathematical expressions, the correctness of which affect memory safety.

3.2 Verification Approaches: Overview

Evidence is desired that the functional correctness and execution safety properties described above hold. Evidence can come from testing, code review, and certification. One of the strongest forms of evidence is *mathematical proof*. The collection of approaches to proving properties of programs is called *formal verification*.

Despite the mathematical assurances provided by formal verification, it is typically not used because of its expense and inability to scale. For example, a national lab in Australia undertook one of the largest verification projects to date to verify the correctness of a microkernel, providing the core services of an operating system. The microkernel is about 10,000 lines of code. Still, the effort was estimated to have taken 20 engineer years.[8]

However, not all verification approaches are created equal. At the one extreme is the microkernel verification approach. In that effort, manual theorem proving was used. Popular manual theorem provers include Isabelle, Coq, and ACL2. While arguably providing the greatest assurance, these approaches also are the most labor intensive, requiring manually formalizing the entire programming language semantics and manually providing proof rules.

[8] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., … & Winwood, S. (2009). Sel4: formal verification of an OS kernel. In J. N. Matthews & T. E. Anderson (eds.), SOSP '09 Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 207-220, Big Sky, MT, October 1 -14, 2009. doi:10.1145/1629575.1629596

Manual theorem proving has its place - it is particularly suited for deep mathematical theorems - but is costly and fragile. The fragility comes from manually having to model and reprove properties whenever the underlying software system changes.

At the other end of the spectrum are automated bug hunting tools, generally referred to as *static analysis*. Popular tools include Code Sonar[9] and Coverity Scan.[10] These tools suffer from both false positives and false negatives, meaning that the tools may not report bugs in the software (false negative) and report potential errors when there are none (false positive). While completely automated and useful, static analysis tools cannot be considered a proof. Furthermore, static analysis does not have the power to prove functional correctness properties.

The formal verification approach followed in this study falls between these two extremes: mathematical proof of security will be provided while being highly automated.

3.3 Verification Approach

The verification approach followed here uses a "correct by construction" approach along with automated tools to provide strong mathematical evidence of correctness (at a fraction of the cost of manual theorem proving). The approach leverages two state of the art verification and validation tools, previously developed by Galois:

- The High-Assurance ASN.1 Workbench: a tool for parsing ASN.1, automatically generating parsers and serializers for a given specification, and automating test-case generation, debugging, and inspection.
- The Software Analysis Workbench: a comprehensive symbolic simulator for C, Java, and other languages.

Each tool and how those tools are leveraged to generate code and proofs of functional and execution correctness are described next.

3.4 The High-Assurance ASN.1 Workbench

HAAW is a suite of high assurance tools 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 with varying levels of maturity. The most mature tools in *HAAW* are the Interpreter and Validator. Given an ASN.1 specification, the Validator will generate test suites to test the acceptance behavior of ASN.1 encode and decode routines (hand-written or compiled). These test suites consist of randomly generated ASN.1 messages, each guaranteed to conform to the given specification. The tests help developers to automatically test code that is challenging to test manually.

[9] GrammaTech, Inc. (2017). GrammaTech CodeSonar. Delivering resiliency for today's IoT devices. (Web page). Retrieved from www.grammatech.com/codesonar

[10] Synopsys, Inc. (n.d.). Coverity Scan Static Analysis. (Web page). Retrieved from https://scan.coverity.com/

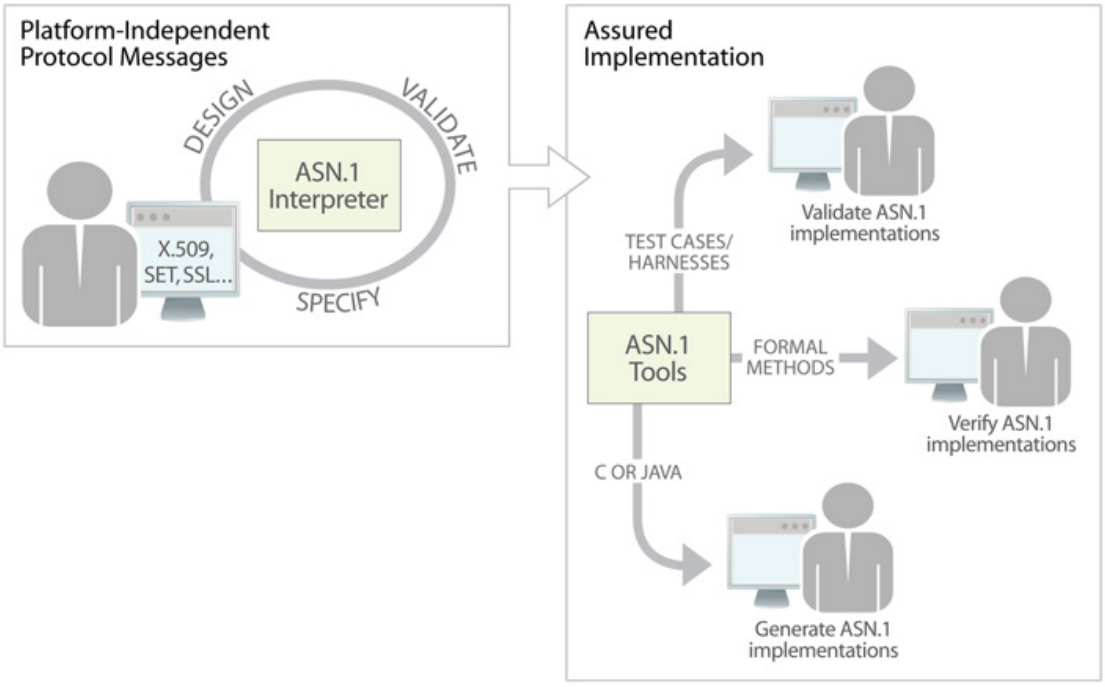

Figure 1: HAAW Workflow

HAAW was developed for the InfoSec branches of the U.S. Government.

Figure 1 shows the HAAW workflow. A user begins by designing new protocols or analyzing already specified protocols in ASN.1. The Workbench provides the user with the following tools:

Interpreter. An interactive, developer-friendly ASN.1 Interpreter that gives useful error messages and allows protocol designers, infrastructure designers, and developers to check their ASN.1 specification for conformance to the ASN.1 standard. The encode and decode facilities provided by the Interpreter may be used as a reference.

The Interpreter can also be used to generate test vectors (currently only supported on a subset of ASN.1 and DER encoding). The test-generation capability is similar to QuickCheck. As with QuickCheck, it is based on Haskell's type class overloading. This allows test-generators to be written for each ASN.1 primitive type and "higher order" test-generators for each ASN.1 composite type (e.g., SEQUENCE), as a result, specific test-generators are able to generated for any ASN.1-defined type.

Compiler. Using the same parsing and type-checking components as the Interpreter, the Compiler (HASNC, High Assurance ASN.1 Compiler) produces high assurance parser and serializer routines targeting C. HASNC was developed to demonstrate the feasibility of a high assurance ASN.1 compiler, using a combination of careful, semi-formal design processes (motivated by manifest correctness) and innovative testing, based upon randomly-generated test data (tuned to cover the input space). Sophisticated automatic test generators were utilized to generate (1) valid ASN.1 programs, (2) invalid ASN.1 programs, and (3) valid ASN.1 values and byte streams (given a specification and a type from it). The test generators currently only support a subset of ASN.1 and DER encoding.

Validator. Given an ASN.1 specification, the Validator generates a test harness, which tests both the expected acceptance and rejection behavior of parser and serializer routines against the Interpreter's reference routines. To maximize flexibility, the routines to be tested may come from the Compiler, from some other compiler, or may even be hand-written. Currently they are only supported on a subset of ASN.1 and DER encoding.

3.5 The Software Analysis Workbench

The Software Analysis Workbench is a set of tools developed previously by the contractor for generating formal models from existing systems code using symbolic simulation. SAW is capable of extracting models from Java code via a symbolic simulator for Java Virtual Machine bytecode, and from C code via a symbolic simulator for Lower Level Virtual Machine bytecode. The primary goal of SAW is to enable developers and security analysts to reason about and prove properties of programs. SAW can show that a program satisfies a functional specification and avoids undefined behavior such as accessing invalid memory locations or dividing by zero. SAW uses automated verification tools including a built-in term rewriter and external SMT (Satisfiability Modulo Theories) and model-checking engines for performing the verification. Users may script multiple verification tactics together to solve a single problem.

The SAW workflow is shown in Figure 2. A user begins with a program, written in C, Java, Cryptol,[11] or a combination of these languages. A program is automatically synthesized into an intermediate representation called *SAWCore*. SAW operates on SAWCore, so that one proof engine can be used on multiple programming languages. While SAW is primarily a fully automated verification tool, SAW also exposes a scripting language that allows a user to provide guidance in proof search. After calling out to various proof engines, SAW returns either a proof or counterexample (mapped back into the source language) for the user.

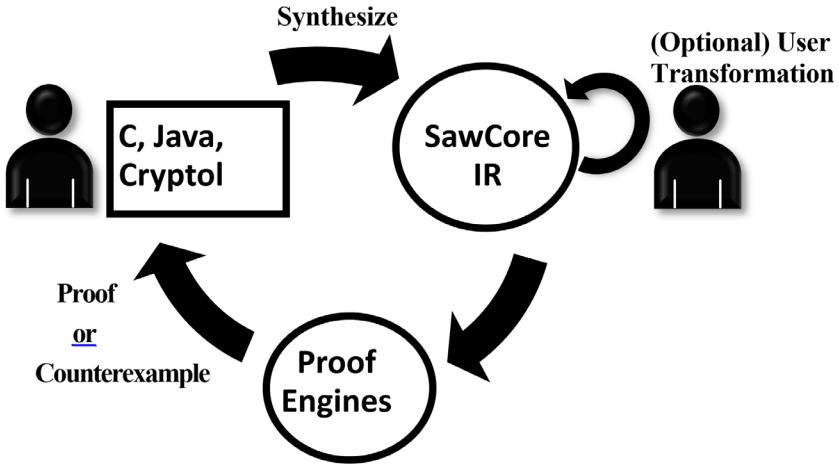

Figure 2 SAW Workflow

Internally, SAW uses symbolic simulation to build a representation of a program's operations on all execution paths. Although the number of execution paths in a program may be infinite, it is small for some classes of critical programs. Such classes include, for example, cryptographic applications. These programs typically contain loops that execute a fixed number of times and avoid data dependent branches that would expose the cryptographic primitive to advanced key recovery attacks, such as timing-based attacks. As the symbolic simulator evaluates the program, it maps program expressions into SAWCore, a dependently typed formal representation shared by all the tools. SAWCore expressions can be simplified using a built-in rewriter, and then translated into verification formats supported by third party tools such as SMTLIB, AIGER, and (Q)DIMACS.

SAW had been used to prove the functional correctness of different AES implementations from libgcrypt, OpenSSL, and Bouncy Castle with minimal help from the user. Verification of secure hash functions and public key cryptography requires that the problem be decomposed into multiple verification steps, and compositional techniques to be used to obtain a verification of the whole system. This capability has been used for verifying libgcrypt's implementation of SHA-2, and a Java implementation of ECDSA signature verification. For compositional verification, the basic approach is to verify equivalence between low-level operations first, and then use the verification results from low-level operations to simplify the verification

[11] Cryptol is a special-purpose language developed by Galois, Inc., for specifying cryptographic protocols. See www.cryptol.net/

of higher-level operations that use the low-level operations. This process may be repeated for algorithms that involve multiple levels.

SAW is open-source and available online.[12]

3.6 Summary of Approach

To develop a formally verified, secure reference parser for the basic safety message, the project was structured with the following key tasks:

1. *Analyze SAE J2735 specification in HAAW.*

   Import the SAE J2735 specification into the HAAW interpreter, checking for inconsistencies, errors, or duplications. If HAAW cannot parse the full specification, either (1) update HAAW to handle the required extensions or (2) surgically rewrite the SAE J2735 specification to preserve semantics but to simplify the syntax.

2. *Generate C Encoder/Decoder Routines for the BSM of SAE J2735.*

   Use the HASNC compiler to generate C parsers and serializers for the Basic Safety Message of SAE J2735. This task includes developing extensions to HASNC's encoder/decoder generation capabilities and writing new primitive encoder/decoder routines.

3. *Verify Execution Safety of the Generated C Code.*

   Use SAW to verify safety properties of the generated code. This task includes adding verification conditions to the HASNC generated parser/serializer as needed.

4. *Verify Functional Correctness of the Generated C Code.*

   Use SAW to verify the functional correctness of the generated code.

[12] See http://saw.galois.com/

4 Project Execution

4.1 Summary

The following is a high-level summary of the results and notable considerations. Section 4.2 details specific results by task.

Regarding code generation with HASNC:

- A revised "gap analysis" was performed for new features needed to support the MAR2016 version of the SAE J2735 specification.
- HASNC was updated with the new UPER backend, while new C code was written for the UPER run-time libraries.
- HASNC was updated to support the ASN.1 feature gaps. Support was added for AUTOMATIC TAGS, ENUMERATED types, extension markers in multiple contexts, and size constrained types, etc.

Regarding verification:

- Scalability was investigated for verification of both the acceptance and rejection properties for BSM Part I, making improvements to SAW, particularly to handle aspects of integer-based reasoning in loops.
- Formal specifications were developed for intermediate functions. These intermediate specifications improved scalability by allowing compositional verification to be performed.

Summary of results:

- HASNC was used to generate C encoders/decoders for BSM Part I (of the MAR2016 J2735).
- Using SAW, two functional correctness properties were verified on the BSM Part I encoder/decoder functions.
- Sufficient features were added to HASNC (along with some specification refactoring) to support BSM Part II and III.

4.2 Tasks

4.2.1 Task 1. Analyze SAE J2735 specification in HAAW.

4.2.1.1 Original Task Description

The first task was to take the SAE J2735 ASN.1 specification - which includes the Basic Safety Message, parts I & II - and import it into the HAAW workbench. Currently, HAAW does not fully support three features required to accomplish this task: (1) full multi-module support, (2) the "…" extension marker, and (3) the AUTOMATIC TAGS clause in a module definition. For each of these features, the feasibility of extending HAAW to support it sufficiently was assessed. In each case, HAAW was either extended or the SAE J2735 specification (preserving the semantics) simplified so as to proceed. The expectation is that HAAW will be extended to handle the SAE J2735 specification unmodified.

4.2.1.2 Commentary

Adding the above features was accomplished essentially per plan. The specification was refactored into a single ASN.1 module rather than changing HASNC. The primary challenge was adding other features to support UPER: supporting various type constraints that affect the UPER encoding (in DER constraints can be dropped before the backend because they do not affect encoding).

4.2.2 Task 2. Generate C Encoder/Decoder Routines for SAE J2735.

4.2.2.1 Original Task Description

The compiler, HASNC, currently generates C for a large subset of ASN.1. HASNC also provides C libraries that perform the encoding/decoding for ASN.1 primitives.

The C code generated by HASNC is simple by design. It performs no memory allocation (memory for parsed messages is assumed to be pre-allocated by the caller), and it does not require libc functionality. Loops are bounded by a constant upper bound. It was hypothesized that these characteristics of the generated C code dramatically simplify the task of verifying its execution safety and function correctness. There is also an amount of library code (e.g., encoding/decoding for ASN.1 primitives) that must be linked with the generated code. Some of this library code may not have the same simplicity as the generated code. The library code that is required by the BSM encoder/decoder routines will be part of the code base for verification in the following steps.

HASNC was thought to likely require modifications to generate code for the entire SAE J2735 specification. This could involve (1) handling more ASN.1 features in the "front-end", (2) handling more ASN.1 features in the "backend" (code generator), or (3) extending the HAAW C libraries.

4.2.2.2 Commentary

As planned, additional features to support HAAW and run-time libraries were added. The biggest impact was the unplanned work to implement UPER encoding. To support UPER it was effectively turned from a single-back-end compiler into a two back-end compiler. Although some code could be re-used, the run-time C libraries for the UPER generated code needed to be written.

4.2.3 Task 3. Verify Execution Safety of Generated C Code.

4.2.3.1 Original Task Description

A key aspect of the code generation was to make it easy to verify functional correctness and to ensure the generated C code is (mostly) correct by construction. The correct by construction approach provides evidence that the generated software is correct with respect to certain properties due to how it is generated.

For example, a "use after free" memory-safety bug is impossible if a program never allocates (and then frees) memory. If the C code generator does not have the capability to generate C code that does allocation, then the property holds. Because of the restrictions of the parser/serializer generator in HAAW, most execution safety properties hold by construction.

There are a few properties, however, that needed to be proven explicitly. In particular, it was hypothesized that the main properties in the generated parsers relate to showing the absence of integer overflow. SAW is capable of verifying arithmetic properties and was used to complete the execution safety proofs.

Based on the needs of the following tasks, extending the HAAW code generator was explored so that verification conditions are also emitted. For example, for an expression

```
int32_t x = a + b;
```

add the following verification condition:

```
   assert(add_ovf_int32(a, b));
   int32_t = a + b;
where int32_overflow() is defined as
add_ovf_int32(int32_t x, int32_t y) {
 return (x >= 0 && y >= 0 && INT32_MAX - x >= y)
  || (x <= 0 && y <= 0 && INT32_MIN - x <= y)
  || (signum(x) != signum(y));
}
```

The assertion becomes a verification condition to ensure no overflow occurs during addition.

4.2.3.2 Commentary

SAW is a symbolic simulator that analyzes programs compiled to the *Lower Level Virtual Machine* intermediate language. LLVM is a simpler language to analyze, and it is a popular intermediate representation for many languages so that verification at the LLVM level allows inter-language reasoning. However, LLVM semantics differ in some important ways from C semantics. For example, integers in LLVM are assumed to be twos-complement integers of a fixed bit width. Thus, C signed and unsigned integers are not distinguished, and in particular, it is assumed that arithmetic operations are on twos-complement integers that overflow. So signed integer arithmetic that overflows and is undefined in C is defined in LLVM.

Any LLVM-based C compiler (e.g., CLANG) respects LLVM semantics, so a SAW verification is sound if the program is compiled with such a compiler. However, if the program is compiled with a non LLVM-based compiler (e.g., GCC), undefined behavior could be present in a verified program. However, there are some mitigations. First, with respect to memory safety, the fragment of C generated by HAAW is memory safe if the corresponding LLVM is memory safe. Principally, this includes array access. Thus, the SAW verification guarantees memory safety.

With respect to potential vulnerabilities due to other undefined, unspecified, or implementation-defined behavior, C semantic evaluators were experimented with. For example, experimentation with *RV-Match.*[13] took place. RV-Match interprets a C program on concrete data, testing for the presence of undefined or implementation-defined behavior. However, it is does not prove the absence of undefined or implementation-defined behavior, since it only works on concrete tests.

Other mitigations are discussed in Section 5.

[13] See https://runtimeverification.com/match/

4.2.4 Task 4. Verify Functional Correctness of Generated C Code.

4.2.4.1 Original Task Description

The final step of the verification plan involves using SAW to verify functional correctness. First, the pre and post conditions as discussed above were formalized. One potential difficulty noted is that the precondition for the rejection property includes a program fragment. However, SAW performs symbolic execution, transforming programs into propositions. Therefore, there is no distinction between a program itself and pre and post-conditions, as in traditional Hoare logic.

The initial approach to verifying the properties is to use the full automation of SAW in which a symbolic program is extracted from the C code, and a "push button" proof attempted. There are, however, two challenges to highlight:

First, program verification is inherently undecidable. Static analysis engines are usually incomplete and unsound to handle arbitrary programs in which there is no proof of correctness. It was deemed that a better approach would be to take advantage of the fact that C code generated by HAAW is extremely restricted, making automated verification feasible. The restrictions that make execution safety hold are the ones that generally make functional correctness proofs feasible.

Second, even if program verification is made decidable, it may not be tractable using automated techniques. This is because automated techniques, like symbolic simulation, generally scale exponentially with respect to the number of state variables required to model the program. State variables generally scale linearly with respect to the number of control-flow decisions made, such as in loops and branches.

SAW employs state-of-the-art optimizations for symbolic analysis, and it is conceivable that an end-to-end verification is possible for functional correctness. If not, SAW has a scripting language that allows human guidance. It was hypothesized that the regular structure of the generated parsers and serializers will make compositional proofs straightforward.

For example, consider the VehicleSafetyExtension from Part II of the basic safety message:

```
  VehicleSafetyExtension ::= SEQUENCE {
 events  EventFlags OPTIONAL,
 pathHistory PathHistory OPTIONAL,
 pathPrediction PathPrediction OPTIONAL,
 theRTCM RTCMPackage OPTIONAL,
 ...
}
```

A parser for the VehicleSafetyExtension type is built from component parsers for the EventFlags type, pathHistory type, etc., and likewise for serializers. C functions are denoted for parsing by "parse_TYPE", so for example, the parser for VehicleSafetyExtension messages is a C function "parse_VehicleSafetyExtension()", and the parser for EventFlags is "parse_EventFlags()". Likewise, for serializers, so that the serializer for EventFlags is serialize_EventFlags().

To generate a symbolic program for parse_VehicleSafetyExtension, the simple approach is to inline the calls to parse_EventFlags() and the other parsers for the fields. But inlining may not scale well.

An alternative method is to take an algebraic composition approach, in which component parsers are verified independently, and those proofs are used compositionally.

For example, if parse_EventFlags() is a parser function and serialize_EventFlags() is the serialization function, respectively, then first use SAW to prove that

```
parse_EventFlags(serialize_EventFlags(m)) = m
```

for any valid message m belonging to the EventFlags type. Refer to this proof as "conformance(EventFlags)". Then in proving the conformance property for the VehicleSafetyExtension type, "conformance(EventFlags)" was used as an axiom. Thus, the proof becomes compositional, built up from the proofs of correctness for parsers and serializers for the component types.

The strategy requires pairing the parsers and serializers for component types. These pairings can be constructed automatically by HAAW, since it generates parsers and serializers simultaneously. It is envisioned that HAAW can automatically generate a script used by SAW, which provides the correspondence between parsers and serializers that guide SAW to compositionally complete an automated proof.

4.2.4.2 Commentary

During the initial experiments, it turned out that the code base and complexity of the encoders and decoders were too great to symbolically simulate to verify correctness. Consequently, the proof had to be modularized.

To do so, the general process is to modularize by function. For example, if function `foo()` calls function `bar()`, then first specify what correctness means for `bar()` alone, and verify that `bar()` satisfies that specification. Then when verifying `foo()`, instead of symbolically simulating `bar()`, the body of `bar()` is *overriden* with the specification for `bar()`.

For example, suppose that `bar()` is a complex function, but all that needs to be known is to verify `foo()` is that a positive integer is returned no matter what the input to the function. Then when verifying `foo()`, it not required to simulate `bar()`; just assume that whatever it does, the specification holds. In this way, the complexity of the verification can be significantly controlled.

Multiple overrides have to be developed for particular functions. Some functions in the encoders and decoders are called with a relatively small set of fixed parameters, along with a symbolic value. For example, in integer range decoding, there are constant lower and upper bounds respectively from the ASN.1 specification. The integer decode function is called with a small number of these constant parameters, along with the actual value to decode. Thus, an override is generated for each set of unique lower and upper bounds. Indeed, these override specifications can be generated programmatically, and are done so using a small script.

To summarize, the general outline of the verification approach is as follows:

1. Formalize the conformance and rejection properties as C functions that are "drivers" for the HAAW encoders and decoders, respectively.
2. Concretely execute the encoders and decoders and collect the parameter's key intermediate functions. Using these parameters, programmatically generate specifications/overrides for these intermediate functions.
3. Develop a top-level SAW script (and associated libraries) to execute SAW to generate the proof.

Below, the C functions that formalize the encoding conformance and rejection properties are shown. First, the property that decode inverts encoding:

```
/* Encode 'x_in', then decode that encoding and check that it equals
* 'x_in'. */
int decode_inverts_encode(BSMcoreData *x_in) {
 AsnOctet os[BUF_SIZE];
 // Reading an uninitialized buffer is undefined in C, and SAW fails
 // accordingly with "invalid load address" if we don't initialize
 // here.
 for (int i = 0; i < BUF_SIZE; ++i) {
 os[i] = 0;
 }
 StreamPtr sp = {os, 0};
 AsnStatus st;

 if ( (st = encV_BSMcoreData(x_in, &sp)) != ASN_OK ) {
 printf("decode_inverts_encode: enc failed!\n");
 return 0;
 }

 StreamPtr buf;
 buf.pOctet = os;
 buf.offset = 0;
 BSMcoreData *x_out;

 if ( (st = decV_BSMcoreData(&x_out, &buf)) != ASN_OK ) {
 printf("decode_inverts_encode: dec failed!\n");
 return 0;
 }

 return BSMcoreData_Equal(x_in, x_out) ? 1 : 0;
}
```

The C function `decode_inverts_encode()` takes a pointer to an arbitrary BSM Part I message. After initializing memory for placing a decoded message, the encoder is run. If an error is encountered during encoding, the property fails. Otherwise, the decoder is run the on the encoded result, again failing if there is any error during decode. Finally, the call to `BSMcoreData_Equal()` checks for equality between the resulting value and the input.

This function, when executed *symbolically* by SAW for an arbitrary input, proves the encoding property. Next the rejection property considered, which is slightly more subtle:

```
/* Decode 'os_in', and if that succeeds, then reencode the decoded
 result and check that we get back the bits we started with. */
int decode_rejects_non_encode(StreamPtr buf_in) {
 BSMcoreData *x_out;
 AsnStatus st;

 if ( (st = decV_BSMcoreData(&x_out, &buf_in)) != ASN_OK ) {
 // It's OK for decoding to fail, since not all buffers need be
 // valid encodings.
 return 1;
 }

 AsnOctet os[BUF_SIZE];
 for (int i = 0; i < BUF_SIZE; ++i) {
 os[i] = 0;
 }
 StreamPtr sp = {os, 0};

 if ( (st = encV_BSMcoreData(x_out, &sp)) != ASN_OK ) {
 printf("decode_rejects_non_encode: enc_int failed!\n");
 return 0;
 }

 AsnLen length = 0;
 ASN_CHECK_RETURN(addLenV_BSMcoreData(x_out, &length), 0);

#ifdef GEN_OVERRIDES
 printf("enc length: %i\n", length);
#endif

 if (big_endian_compare_bits(buf_in.pOctet, 0, sp.pOctet, 0, length) != 0) {
 printf("decode_rejects_non_encode: big_endian_compare_bits failed!\n");
 return 0;
 }
 return 1;
}
```

For the rejection property, the C function takes a pointer to an arbitrary bit stream and then calls the decoder on the bit stream. It is acceptable if the decoding fails since some bit streams may be invalid. (Indeed, the null decoder that *never* decodes any bit stream satisfies this property. However, it is guaranteed that the decoder is not null, since it must also satisfy the acceptance property, mentioned previously.) If decoding succeeds, then encoding of the resultant BSM message is checked; it must have a valid encoding, and furthermore, the resulting bit stream must equal the original bit stream.

With the infrastructure described, the conformance property takes approximately 20 minutes and the rejection property approximately four hours on a modern laptop. The difference is due to the additional nondeterminism when considering arbitrary bit streams in the rejection property.

5 Conclusions

Assessments based on the findings in this project are summarized below.

5.1 Assessments

ASN.1 is complex, and encoders/decoders can be subtle, supporting the need to build formally verified ASN.1 encoders/decoders. Interfaces are where a system meets outside threats and are critical to get correct. Fortunately, they are small enough, and their properties are simple enough to feasibly verify.

While the verification is specific to the SAE J2735 ASN.1 specification and the code generated by HAAW, the core of the approach can be extendable to other ASN.1 compilers for SAE J2735 as well as other ASN.1 specifications. This is due to two reasons. First, the most difficult aspect of the verification revolved around verifying the C library used in UPER encoding. This portion of the verification is relevant to *any* verification for HAAW-generated encoders/decoders. Second, any correct C implementation for an ASN.1 implementation for a particular encoding rule will have a very similar implementation. Because the SAW proofs are mostly automatic, the framework could be reusable for other implementations.

Moreover, while the work is an important part of guaranteeing the correctness of vehicle-to-vehicle communications, there are other important aspects that have not been addressed. This includes IEEE 1609 and IEEE 802.11p covering the other aspects of the system. The security and correctness of these other systems should also be addressed.

5.2 Deliverables

As part of this report, several software products and datasets are being made available to the public:

- The supported SAE J2735 specification
- The C Encoder/Decoder routines supporting the SAE J2735 BSM Part I, II and III
- The API specification detailing the interface to the C Encoder/Decoder routines
- The Verification results
    - C Encoder/Decoder routine supporting the BSM Part I (verified against the two properties)
- The Galois open source Software Analysis Workbench (SAW[14]) and the verification reproduction, including all artifacts required to reproduce verifications including:
    - A readme with step by step instructions to run the verification (which is minimal as this is predominantly automated)
    - The encoder/decoder routines supporting the BSM Part 1 that have verified the two properties

[14] See saw.galois.com

DOT HS 812 484
March 2018

U.S. Department
of Transportation

**National Highway
Traffic Safety
Administration**

13264-030718-v4a