SecProtobuf: Implicit Message Integrity Provision in Heterogeneous Vehicular Systems

Paul Molloy
Networked Systems Security Group
KTH Royal Institute of Technology
Stockholm, Sweden
molloy@kth.se

Mohammad Khodaei
Networked Systems Security Group
KTH Royal Institute of Technology
Stockholm, Sweden
khodaei@kth.se

Per Hallgren
Einride AB
Gothenburg, Sweden
per.hallgren@einride.tech

Alexandre Thenorio
Einride AB
Gothenburg, Sweden
alexandre.thenorio@einride.tech

Panos Papadimitratos
Networked Systems Security Group
KTH Royal Institute of Technology
Stockholm, Sweden
papadim@kth.se

***Abstract*—Novel vehicular applications, such as remote driving, platooning, and autonomous driving systems are increasing the complexity of networked vehicular systems. These Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I) (V2X) use-cases require strong security (and privacy) guarantees, *authentication*, *integrity*, and *non-repudiation*. Standardization bodies and harmonization efforts provide complex data structures for basic safety messages, mandated to be digitally signed and validated. Due to the complex data structures, the multiplicity of use-cases, the rapid deployment, as well as the need for interoperability among Original Equipment Manufacturers (OEMs), developing the code needed to provide security becomes a more challenging, error prone, and time consuming task; even more so as the scale of Vehicular Communication (VC) systems grow. In order to tackle this challenge, we propose *SecProtobuf*, a novel security framework to automate the signature generation and validation procedures for any VC safety and non-safety data structures. Our framework facilitates the serialisation and deserialisation processes for arbitrarily complex data types, thus, mitigating potential security defect risks and catalyzing the deployment. In order to ensure the correct usage of the framework by developers, SecProtobuf is provided with a static code analysis (*linter*).**

***Index Terms*—Automatic Integrity Checks; Code-generation.**

I. Introduction

In Vehicular Communication (VC) systems, vehicles beacon Cooperative Awareness Messages (CAMs) and Decentralized Environmental Notification Messages (DENMs) periodically, at high rates, to facilitate transportation safety and efficiency. It has been well-understood that VC systems are vulnerable to attacks and that the privacy of their users is at stake [1], [2]. As a result, security and privacy solutions have been developed by standardisation bodies (IEEE 1609.2 WG [3] and ETSI [4]), harmonisation efforts (C2C-CC [5]), and projects (SeVeCom [6]–[8], PRESERVE [9], and CAMP [10], [11]). A consensus towards using Public Key Cryptography (PKC) to protect Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I) (V2X) communication has been reached: a set of short-lived anonymised certificates, termed *pseudonyms*, are issued by a Vehicular Public-Key Infrastructure (VPKI) [12], e.g., [10], [13], for registered vehicles. Vehicles switch from one pseudonym to a non-previously used one towards message unlinkability, as pseudonyms are per se inherently unlinkable.

The number of CAM and DENM data structure types, as specified in IEEE 1609.2 [3] and ETSI [14], increases greatly as V2X systems are scaled and deployed. These messages are nested data structures, needing custom code to traverse, sign and verify them. The Wireless Access in Vehicular Environments (WAVE) *HeaderInfo* [3] data structure illustrates the format of basic safety messages with multiple nested sub-data structures. The ISO standard 15628 [15] specifies 20 different types ($dsrcApplicationEntityId$) for vehicular networking applications, such as parking, electronic fee collection and emergency warning. Fig. 1 shows a use-case for remote driving with vehicles interacting with the remote driving station every 10 ms. A common process for signing (sign, serialize, transmit) and validating (receive, deserialise, verify and handle, process) are shown. All messages need to be signed before transmission to the backend infrastructure; at the same time, all messages need to be validated on the remote driving station when received. Each new application would have many different message types, all of which would need to be signed, thus requiring custom code to digitally sign and verify.

Widely used standards include the World Wide Web Consortium (W3C) Extensible Markup Language (XML) signatures standard [16], it allows for enveloped signature elements, stored inside the data structure of the message. There exist other standards for message integrity and authentication in vehicular contexts such as IEEE 1609.02 [3]: Abstract Syntax Notation One (ASN.1) with a compact and performant binary encoding format [17]–[19]. However, these standards do not provide support for automatic generation of the code for signing and verifying. Rather, a developer needs to program it manually.

At he same time, Original Equipment Manufacturers (OEMs) need to provide guarantees of integrity, non-

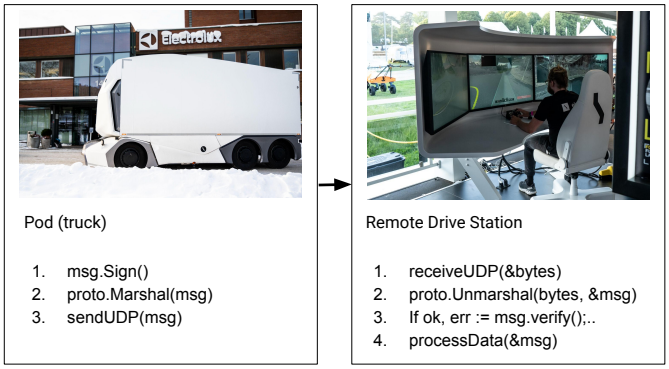

Fig. 1: A Use-case Scenario: Datastream to Remote Driver Station Over User Datagram Protocol (UDP) [taken from [22]].

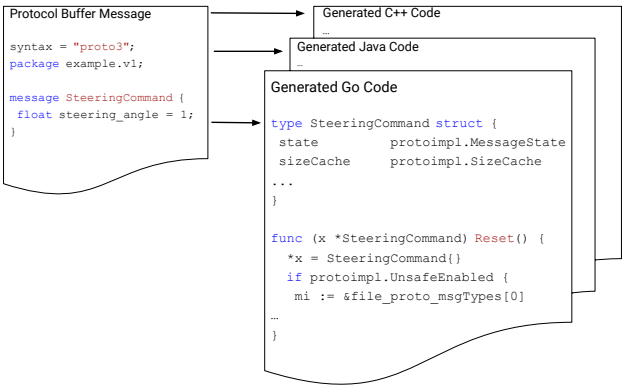

Fig. 2: Code Compilation of Steering Command Protocol Buffer Message into Multiple Languages.

repudiation and authentication [20], and implement vehicular standards such as IEEE 1609.2 [3]. As the complexity of multi-node systems increases, the risk of introducing new vulnerabilities increases while trying to implement security mechanisms. A security in depth approach is needed [21] to mitigate against such a risk: all in-vehicle communications require security guarantees for interactions among different components and sub-systems instead of just of edge nodes. Making this worse, many networked vehicle systems are adopting micro-service architecture orientated designs, with many computing sub-systems and sensors using different programming languages.

Adding protection mechanisms correctly and consistently in a heterogeneous system is an immensely complex task, and error prone when conducted manually. To assert that a system correctly applies security policies, we propose an architecture where necessary functionality is not re-implemented manually and uniquely for every entity; instead, it consists of reusable components that can be audited in isolation. This calls for a *code-generated* security layer that a programmer does not touch on a day-to-day basis, but is *implicit* for every message sent.

Most V2X developers are not security engineers and may 'mis-implement' security critical components; for example, the code to locate the signature in the structured data, particularly so with enveloped signatures nested in the structured data sent. The message must have the nested signature field copied and then cleared before it is compared to the signature (on the receiver side). Such code could be bug prone and consume developer time. It is also difficult to always remember to verify signatures and message-integrity codes before use. It has been found [23] that for every thousand lines of code an additional 7.4 defects are added to a software system. A system to generate the code for signing, verifying and accessing the keys automatically would alleviate much of this chore work and the critical security bugs likely to spawn from it. This extra work to program custom signature code for every custom data structure message in every language will have a heavy cost in terms of both developer time and money for large networked vehicular software systems.

Contributions: We propose a plugin for the language-neutral Protocol Buffer [24] serialisation framework, to automatically generate the code for adding and checking the signatures of arbitrary nested V2X data structures. Protocol Buffer was chosen due to the strong tooling for code generation and multi-language support, as well as its use in the automotive and V2X software [22], [25]. A code *linter* [26] was also written to ensure that for any data type, generated using this plugin, the data structure is signed and verified at the points of serialisation and deserialisation. This framework can co-exist with, and lead to improved tooling for standards such as IEEE 16.09 [3]. The code is open-sourced under the MIT licence, available on Github[1].

II. Implementation

Protocol Buffers [24] are a serialised structured data format developed by Google. As shown in Fig. 2, structured message types are defined by a '.proto' file, which can be compiled into native code with relevant objects for one of many languages, e.g., C++, Go, Java, Rust, JS. Custom plugins can be added to the Protocol Buffer compiler to add extra functionality. This was chosen chiefly because it is the main format used in several autonomous systems [22], [25]. Einride is applying a microservice architecture both for the remote operator station and the autonomous vehicle, with a large number of services communicating using Protocol Buffers over gRPC Remote Procedure Call (gRPC), Lightweight Communications and Marshalling (LCM) and UDP. Einride's technology is primarily written in Go with some components written in Rust and C. Some of the advantages of Protocol Buffers are the language agnostic nature, automatic code generation support and performance [27]. There is particularly strong support for code generation and *protoc-gen-go*, which translates '.proto' files into Go code. This code generation facilitates communication among entities regardless of programming language, by generating corresponding Application Programming Interfaces (APIs). Code generation, in this use-case, could automatically write the code needed to handle the signatures and message integrity codes for any message with any nested data structure.

Custom Protocol Buffer Compiler Plugin: A custom Protocol Buffer option was created, as shown in Listing 1,

[1]https://github.com/einride/protoc-gen-messageintegrity

Listing 1: ProtoBuf Message with Signature Option Enabled.

```
message SteeringCommand {
 float steering_angle = 1;
 bytes signature = 2 [(integrity.v1.signature)={
    behaviour: SIGNATURE_BEHAVIOUR_REQUIRED}];
}
```

Listing 2: SecProtobuf Plugin Generated Code for Protobuf Message (as Specified in Listing 1).

```
func (x *SteeringCommand) Sign() error {
 keyID := os.Getenv(MessageIntegrityKeyID)
 return verification.Sign(x, keyID)
}

func (x *SteeringCommand) Verify()(bool,error){
 keyID := os.Getenv(MessageIntegrityKeyID)
 return verification.Validate(x, keyID)
}
```

Listing 3: Compiling ProtoBuf Message with SecProtobuf.

```
#!/bin/bash
$ protoc --proto_path=src --go_out=gen --
    messageintegrity_out=gen --go_opt=paths=
    source_relative src/
    steering_command_example.proto
```

Listing 4: Calling Static Analysis *Linter* on a Go Source File.

```
#!/bin/bash
$ integritylint example.go
```

TABLE I: The Processing Overhead for Serialisation and Deserialisation.

Run	(De)serialisation	HMAC	ECDSA
Sign	0.000254 ms	0.002165 ms	2.104739 ms
Verify	0.000206 ms	0.001976 ms	0.125568 ms

to mark a field that stores a signature. This allows metadata to specify if the signature is required to be signed and verified. Additionally, a custom Protocol Buffer compiler was created to generate additional Go code for the generated Go message types as shown in Listing 2. This generated code adds functionality to sign the Protocol Buffer messages and verify the signatures using keys stored in a location specified by plugin configuration. The generated code uses the standard Go $crypto$ library. Based on the implementation, a developer simply needs to call *msg.Sign()* and *msg.Verify()* instead of manually writing the code to sign and verify an enveloped signature each time.

Within the plugin, several message integrity verification packages were created, supporting Hash-based Message Authentication Code (HMAC)-SHA256, Rivest–Shamir–Adleman (RSA) (2048 key size) and Elliptic Curve Digital Signature Algorithm (ECDSA) (256 key size). The system was designed to be extensible; other protocols can be added to the plugin in a similar manner.

Custom Protocol Buffer Compiler Plugin: A custom plugin for the Protocol-Buffer-to-Go compiler ($protoc\text{-}go$) was created. This plugin adds Sign and Verify methods to any generated Go type corresponding to a protocol buffer message that has the custom option enabled in it. This has the effect of automatically adding message integrity-checking code to all Go source language types that require message integrity checking. The compiler plugin is executed as part of the *protoc-gen-go* compiler command, as in Listing 3, as part of the normal build process.

Linter: A custom code analysis tool (*linter*) for the Go programming language was created to ensure correct usage of SecProtobuf. It traverses the abstract syntax tree of source code files to make its analysis about the correct usage of SecProtobuf. The *linter* can be integrated into an Integrated Development Environment (IDE) or executed from the command line interface (as seen in Listing 4). The tool first identifies any data types generated based on Protocol Buffer messages with the custom SecProtobuf option set to $REQUIRED$. The *linter* ensures that any instance of those types is always signed and verified where required. The tool will throw a *linter* error if an instance of one of these types is serialised/deserialised before it is signed/verified; e.g., an error is thrown if $Verify()$ is not called right after deserialisation. This takes away an entire class of errors due to developers forgetting to sign or verify messages where required.

III. Evaluation

All of the benchmarks were conducted on a Intel(R) Core(TM) i7-7700HQ CPU @2.80GHz with 16 GB of RAM with the components running on Ubuntu 20.04. The vehicle software stack was running on a set of containers using Docker v20.10.7 [28]. Benchmarks to measure the delay caused by using generated signing and verifying on protocol buffer messages are shown in Table I.

The latency due to the (de)serialisation process and HMAC-SHA256 is ten times faster than the same process for the ECDSA module, as expected, and would work well under tight performance requirements. The different algorithms evaluated here are not being directly compared, rather their performance is shown to illustrate the functionality and viability of our plugin use.

Of the protocols provided by the plugin, it was found that the HMAC component met performance requirements for the intra-vehicle system it was designed for (sub 1 ms latency to serialise + deserialise + generate + verify). Meanwhile the components that provide non-repudiation and authentication, RSA and ECDSA respectively, are better suited to use cases with less stringent performance requirements.

We found that there was limited effect on the payload size of the signatures added by the protoc-message-integrity plugin. The increase in payload size is proportional to the

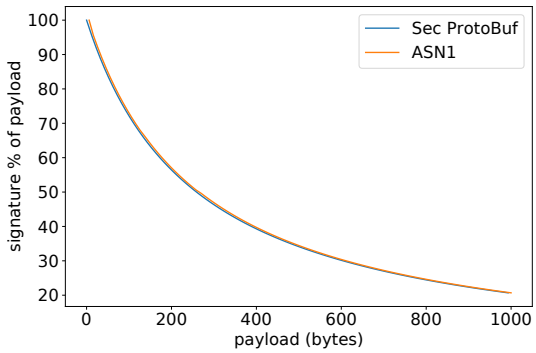

Fig. 3: The proportion of the payload taken up by the signatures as the payload is increased.

signature length encoded in Packed Encoding Rules (PER) ASN.1 (259 bytes on average) for ECDSA public/private key pairs (256 key sizes as per the standard [3]). Fig. 3 shows, as expected, that the overhead is proportionally higher for small messages. Note that the only difference between the two is that ASN.1 encoding takes a constant 6 bytes extra than ProtoBuf. For example, a 32-bit steering angle message (See Fig. 1) is proportionally overshadowed by the signature that is added. For more complex messages with large byte arrays, the signature overhead is less significant.

IV. Conclusions & Further Work

An implementation of a Protocol Buffer plugin to generate code for signing and verifying messages at the point of serialisation and deserialisation was created. In addition, a code *linter* was created to ensure messages with the correct metadata are always signed and verified. This framework is more broadly relevant to heterogeneous software systems beyond VC systems, where the numbers of messages and nodes are at scale. Unlike signature generation and validation in XML and ASN.1, SecProtobuf automatically generates custom code for each data type and can be statically analysed to ensure signatures are always used correctly.

The plugin can be expanded to include additional functionality, e.g., integration with a Public-Key Infrastructure (PKI) system [13], [29]. The efficiency of the *linter* could be optimised to make fewer passes over code being analysed before it reports its findings. Furthermore, one can prepare similar plugins for other encoding formats, e.g., ASN.1 to automate WAVE message signatures and message integrity codes. Evaluation against additional adversarial payloads as well as performance using an On-Board Unit (OBU) with asymmetric hardware support, e.g., NexCom OBU [9] are ongoing and future work.

References

[1] S. Jafarnejad, L. Codeca, W. Bronzi, R. Frank, and T. Engel, "A car hacking experiment: When connectivity meets vulnerability," in *IEEE globecom workshops (GC Wkshps)*, San Diego, CA, USA, Dec. 2015.

[2] A. Ghosal and M. Conti, "Security issues and challenges in V2X: A Survey," *Computer Networks*, vol. 169, p. 107093, Jan. 2020.

[3] "IEEE Standard for Wireless Access in Vehicular Environments (WAVE)–Certificate Management Interfaces for End Entities," *IEEE Std 1609.2.1-2020*, pp. 1–287, Dec. 2020.

[4] ETSI, "Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Definitions," ETSI Tech. TR-102-638, Jun. 2009.

[5] PKI-Memo, "C2C-CC," http://www.car-2-car.org/, Feb. 2011.

[6] P. Papadimitratos, L. Buttyan, T. Holczer, E. Schoch, J. Freudiger, M. Raya, Z. Ma, F. Kargl, A. Kung, and J.-P. Hubaux, "Secure Vehicular Communication Systems: Design and Architecture," *IEEE Communications Magazine*, vol. 46, no. 11, pp. 100–109, Nov. 2008.

[7] A. Kung., "Security Architecture and Mechanisms for V2V/V2I, SeVeCom," https://sevecom.eu/Deliverables/Sevecom_Deliverable_D2.1_v3.0.pdf, Feb. 2008.

[8] P. Papadimitratos, L. Buttyan, J.-P. Hubaux, F. Kargl, A. Kung, and M. Raya, "Architecture for Secure and Private Vehicular Communications," in *IEEE ITST*, Sophia Antipolis, Jun. 2007.

[9] PRESERVE-Project, www.preserve-project.eu/, Jun. 2015.

[10] W. Whyte, A. Weimerskirch, V. Kumar, and T. Hehn, "A Security Credential Management System for V2V Communications," in *IEEE VNC*, Boston, MA, Dec. 2013.

[11] "Vehicle Safety Communications Security Studies: Technical Design of the Security Credential Management System," https://www.regulations.gov/document?D=NHTSA-2015-0060-0004, July 2016.

[12] M. Khodaei and P. Papadimitratos, "The Key to Intelligent Transportation: Identity and Credential Management in Vehicular Communication Systems," *IEEE Vehicular Technology Magazine*, vol. 10, no. 4, pp. 63–69, Dec. 2015.

[13] M. Khodaei, H. Jin, and P. Papadimitratos, "SECMACE: Scalable and Robust Identity and Credential Management Infrastructure in Vehicular Communication Systems," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 5, pp. 1430–1444, May 2018.

[14] "Intelligent Transport Systems (ITS); Security; Security Services and Architecture," *ETSI Standard TS 102 731*, Sept. 2010.

[15] "Intelligent Transport Systems — Dedicated Short Range Communication (DSRC) — DSRC Application Layer," International Organization for Standardization, Standard, Nov. 2020.

[16] D. Eastlake, J. Reagle, D. Solo, F. Hirsch, and T. Roessler, "XML-signature syntax and processing," *W3C recommendation*, vol. 12, Feb. 2002.

[17] "ASN.1 encoding rules: Specification of Packed Encoding Rules (PER)," International Organization for Standardization, Geneva, CH, Standard, Feb. 2021.

[18] "Abstract Syntax Notation One (ASN.1): Specification of basic notation," International Organization for Standardization, Geneva, CH, Standard, Feb. 2021.

[19] V. Kumar and W. Whyte, "Performance Analysis of Existing 1609.2 Encodings v ASN.1," *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, vol. 8, no. 2015-01-0288, pp. 356–363, Apr. 2015.

[20] P. Papadimitratos, V. Gligor, and J.-P. Hubaux, "Securing Vehicular Communications-Assumptions, Requirements, and Principles," in *ESCAR*, Berlin, Germany, Nov. 2006.

[21] R. Ward and B. Beyer, "Beyondcorp: A New Approach to Enterprise Security," *login*, vol. 29, pp. 5–11, Dec. 2014.

[22] Einride Tech, https://www.einride.tech/, Jul. 2021.

[23] S. McConnell, "Gauging Software Readiness with Defect Tracking," *IEEE Software*, vol. 14, no. 3, p. 136, Jun. 1997.

[24] K. Varda, "Google Protocol Buffers: Google's data interchange format," http://code.google.com/apis/protocolbuffers/.

[25] "We're building the World's Most Experienced Driver", https://waymo.com/, Jul. 2021.

[26] A. Gosain and G. Sharma, "Static Analysis: A Survey of Techniques and Tools," in *Intelligent Computing and Applications*, Feb. 2015.

[27] G. Kaur and M. M. Fuad, "An Evaluation of Protocol Buffer," in *Proceedings of the IEEE SoutheastCon*, Concord, NC, USA, Mar. 2010.

[28] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, May 2014.

[29] M. Khodaei, H. Noroozi, and P. Papadimitratos, "Scaling Pseudonymous Authentication for Large Mobile Systems," in *ACM WiSec*, Miami, FL, USA, May 2019.