KTH ROYAL INSTITUTE
OF TECHNOLOGY

DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERNIG

SECOND CYCLE, 30 CREDITS

Implicit Message Integrity Provision

In Heterogeneous Vehicular Systems

PAUL MOLLOY

Stockholm, Sweden 2023

Implicit Message Integrity Provision

in Heterogeneous Vehicular Systems

PAUL Molloy

Master's Programme, ICT Innovation, 120 credits
Date: April 4, 2023

Supervisors: Dr. Mohammad Khodaei, Dr. Per Hallgren
Examiner: Professor Panagiotis Papadimitratos
School of Electrical Engineering and Computer Science
Host company: Einride AB
Swedish title: Implicit Integrity In Heterogeneous Vehicular Systems
Swedish subtitle: Implicit Integritet i Heterogena Fordonsmiljöer Systems

Abstract

Vehicles on the road today are complex multi-node computer networks. Security has always been a critical issue in the automotive computing industry. It is becoming even more crucial with the advent of autonomous vehicles and driver assistant technology. There is potential for attackers to control vehicles maliciously. Traditionally Original Equipment Manufacturers have relied on physical security and a firewall to secure vehicles but with network connected and autonomous capable vehicles this is not enough. The concept of defence in depth is required. This means not trusting that internal traffic inside the firewall is benign. Each node in the vehicles network should be able to verify the authenticity and validity of communications it receives from other nodes. Implementation of the crypto-graphic systems for doing this is error prone. Therefore a key issue in the thesis is reducing the attack surface by developing these checks in the autonomous vehicle stack in a scalable way so the programmer does not have to be aware of this security layer on a day-to-day basis nor re-implement it for each node in these heterogeneous systems.
Although message integrity and authenticity verification have been studied and implemented in many fields, the area of heterogeneous autonomous systems present unique research challenges. There are tight performance constraints due to the real time requirements for vehicle control systems and data publishing rates. It is an open question if this approach can achieve performance within the bounds required for a reliable autonomous vehicle. Additionally the security benefit of scalably automatically generating the message integrity verification code across all of the nodes in a heterogeneous system would help the field quantify the defect reduction and security benefit of this kind of code generation on complex software systems.

Keywords

Privacy, Code Generation, Vehicle-to-infrastructure, Vehicular ad hoc Networks, Standardization, Remote Procedure Calls, Safety

Sammanfattning

Dagens fordon på vägarna är komplexa datanät med flera noder. Säkerheten har alltid varit en viktig fråga inom bilindustrin. Det blir ännu viktigare i och med tillkomsten av autonoma fordon och förarassistentteknik. Det finns en potential för angripare att styra fordon på ett illvilligt sätt. Traditionellt har tillverkare av originalutrustning förlitat sig på fysisk säkerhet och en brandvägg för att säkra fordonen, men med nätverksanslutna och autonoma fordon räcker detta inte längre. Begreppet försvar på djupet är nödvändigt. Detta innebär att man inte kan lita på att den interna trafiken innanför brandväggen är godartad. Varje nod i fordonets nätverk bör kunna kontrollera äktheten och giltigheten hos den kommunikation som den tar emot från andra noder. Genomförandet av kryptografiska system för att göra detta är felkänsligt. En viktig fråga i avhandlingen är därför att minska angreppsytan genom att utveckla dessa kontroller i det autonoma fordonet på ett skalbart sätt så att programmeraren inte behöver vara medveten om detta säkerhetslager dagligen eller implementera det på nytt för varje nod i dessa heterogena system. Även om meddelandeintegritet och äkthetskontroll har studerats och genomförts inom många områden, innebär området heterogena autonoma system unika forskningsutmaningar. Det finns snäva prestandabegränsningar på grund av realtidskraven för fordonskontrollsystem och datapubliceringshastigheter. Det är en öppen fråga om detta tillvägagångssätt kan uppnå prestanda inom de gränser som krävs för ett tillförlitligt autonomt fordon. Dessutom skulle säkerhetsfördelarna med en skalbar automatisk generering av koden för verifiering av meddelandets integritet över alla noder i ett heterogent system hjälpa fältet att kvantifiera felminskningen och säkerhetsfördelarna med denna typ av kodgenerering i komplexa programvarusystem.

Nyckelord

Integritet, Kodgenerering, Fordon-till-infrastruktur, Ad hoc-nät för Fordon, Standardisering, Samtal om fjärrprocedur, Säkerhet

Acknowledgments

I would like to thank Per Hallgren for proposing the thesis Project and Alexadre Thenorio for mentoring me through the collaboration with Einride. I am extremely grateful for the guidance, advice as well as patience shown to me by Mohammad Khodaei my supervisor as well as Panagiotis Papdimitratos my examiner. I would also thank them for helping to push to publish a paper together on the work done as part of this Masters thesis project.

Stockholm, April 2023
Paul Molloy

Contents

List of Figures

List of Tables

Listings

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

ASN.1 Abstract Syntax Notation One

AST Abstract Syntax Tree

BSM Basic Safety Message

BSSID Basic Service Set Identifier

CA Certification Authority

CAM Cooperative Awareness Message

CAN Controller Area Network

CI/CD Continuous-Integration/Continuous-Deployment

CPU Central Processing Unit

C2C-CC CAR 2 CAR Communication Consortium

DENM Decentralized Environmental Notification Message

DSA Digital Signature Algorithm

DUKPT Derived Unique Key Per Transaction

E2E End-to-End

ECC Elliptic Curve Cryptography

ECDSA Elliptic Curve Digital Signature Algorithm

ECU Electronic Control Unit

EMV Europay Mastercard Visa

ETSI European Telecommunications Standards Institute

FPGA Field Programmable Gate Array

GPU Graphics Processing Unit

gRPC gRPC Remote Procedure Call

HMAC Hash-based Message Authentication Code

IDE Integrated Developer Environment

IEEE Institute of Electrical and Electronics Engineers

IP Internet Protocol

JSON JavaScript Object Notation

LCM Lightweight Communications and Marshalling

LTC Long Term Certificate

LTCA Long Term Certification Authority

MAC Message Authentication Code

MSB Most Significant Bit

ms millisecond

NSRC Dedicated Short-range Communications

NIST National Institute of Standards and Technology

OBU On-Board Unit

OEM Original Equipment Manufacturer

OSI Open Systems Interconnection

PCA Pseudonym Certificate Authority

PER Packed Encoding Rules

PII Personally Identifiable Information

PKCS Public-Key Cryptography Standards

PKC Public Key Cryptography

PKI Public-Key Infrastructure

PSS Probabilistic Signature Scheme

RCA Root CA

RPC Remote Procedure Call

RSA Rivest–Shamir–Adleman

RSU Roadside Unit

SHA Secure Hash Algorithms

SLC Short-Lived Certificate

SOAP Simple Object Access Protocol

SSH Secure Shell

SeVeCom Secure Vehicle Communication

SoC System-On-A-Chip

TBCG Template-Based Code Generation

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

UML Unified Modeling Language

URI Uniform Resource Identifier

URL Uniform Resource Locator

V2C Vehicle-to-Cloud

V2I Vehicle-to-Infrastructure

V2N Vehicle-to-Network

V2V Vehicle-to-Vehicle

V2X Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I)

VANET Vehicular Ad-hoc Network

VANET Vehicular Ad-hoc Network

VC Vehicular Communication

VM Virtual Machine

VPKI Vehicular Public-Key Infrastructure

W3C World Wide Web Consortium

WAVE Wireless Access in Vehicular Environments

WS-Security Web Service Security

WSMP WAVE Short Message Protocol

XML Extensible Markup Language

YAML YAML Ain't Markup Language

Chapter 1

Introduction

This thesis project focuses on the challenge of improving the security of heterogeneous vehicular systems by creating a way of automatically generating security code to enable signing and verifying of communications for the component software sub-systems.

Please note that this thesis is based on the same work as the accompanying paper [2]. As such, some of the context, background and discussion overlaps with this paper.

1.1 Background

Traditionally vehicles had relatively simple internal networks with components communicating with the Electronic Control Unit (ECU) over a Controller Area Network (CAN) bus which is a simple wired multi-cast network protocol. Such vehicles were made with the assumption that physical access would be needed to access the network. In other words relying primarily on physical security rather than cryptographic security. The real-time nature of these communications also meant that cryptographic security checks were less feasible.

Modern and next generation vehicles are moving to be more highly networked both internally and externally. Vehicles are increasingly networked internally using more complex protocols such as Internet Protocol (IP) using UDP and TCP to connect machine learning and connected sensors need for new use cases. Externally vehicles are moving to become more networked externally through trends such as V2V and V2I (V2X) connecting vehicles with each other, infrastructure and cloud services [5].

To help illustrate this, Fig. 1.1 shows a use-case for remote driving, in

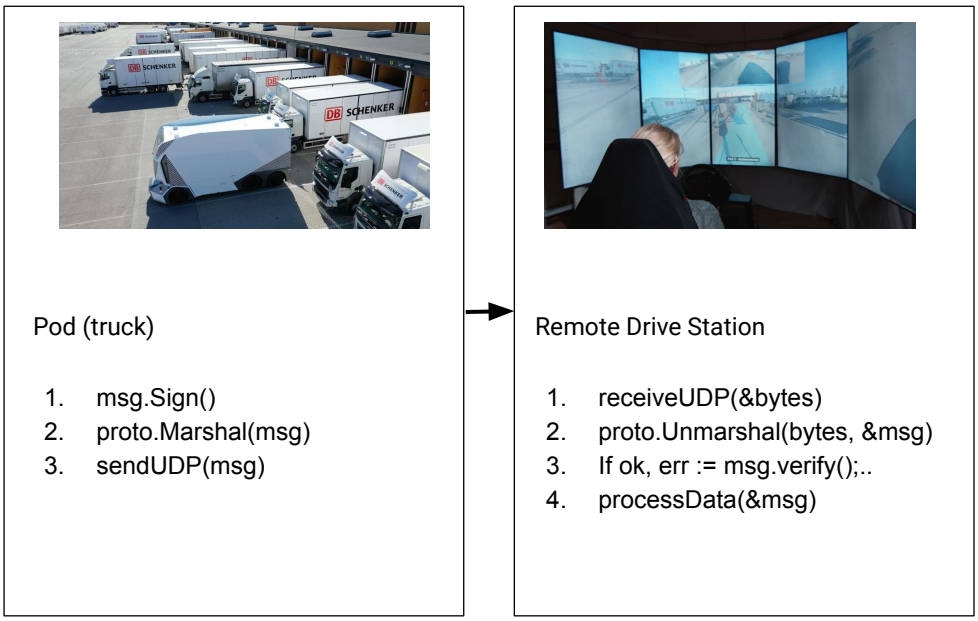

Figure 1.1: A Use-case Scenario: Data-stream to Remote Driver Station Over UDP (taken from [1]).

which vehicles interact with a remote driving station every 10 ms. The interfaces for signing and validating are also shown [2]. 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 may have numerous distinct message types, all of which must be signed to maintain security guarantees, and thus requiring custom code for digital signing and verification [2].

There are new risks such as eavesdropping on private information and man-in-the middle attacks [6] which be come more practical over wireless networks. For this reason such heterogeneous networked vehicles will need to provide security guarantees using cryptographic algorithms to help protect against these risks. At the same time, Original Equipment Manufacturers (OEMs) are mandated to comply with the Institute of Electrical and Electronics Engineers (IEEE) 1609.2 standards [7] and implement the vehicular communication standard as well as provide security protection guarantees for integrity, non-repudiation and authentication [6].

As the complexity of multi-node systems increases, the risk of introducing new vulnerabilities trying to implement security mechanisms also grows. To

mitigate against such a risk, a security in depth approach is needed [8]: all in-vehicle communications require security guarantees for interactions among different components and sub-systems instead of just of edge nodes [2]. This is further complicated by the adoption of a micro-service architecture-orientated design in many Vehicular Communications (VCs), with a multitude of computing sub-systems and sensors using different programming languages [2] and frameworks.

In order to provide these required security guarantees, it is needed to add security code to each software subsystem of the vehicle and all of the other networked systems it communicates with. This requires writing code for each component, which must be carefully written and audited by security engineers due to its critical nature. This process can be time-consuming and costly for vehicular systems developers and their employers, especially when each component requires nearly identical code with only minor modification based on the structure of the messages being communicated between nodes.

Most V2X developers are not information security professionals and are likely to be the source of security bugs if working on security critical code; The code to locate and handle the signature in the structured data, particularly enveloped signatures nested in the structured data sent. It is possible an inexperienced engineer may slip up when writing the verification code and forget to clear the signature field before the message is compared to the expected signature. In general this security code is bug prone and may take inordinate developer time. It is also difficult to always remember to verify signatures and message-integrity codes before use. It has been found in one study [9] that for every thousand lines of code an additional 7.4 defects are added to a software system. 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.

Clearly adding protection mechanisms correctly and consistently in a heterogeneous system is an extremely difficult and error-prone task when done manually. As a solution an automated system for generating this code for signing, verifying and accessing the keys automatically would alleviate much of this chore work, critical security bugs likely to spawn from it as well as crucially cost for the companies developing these systems.

To ensure that a system correctly enforces security policies, this thesis proposes an architecture where the necessary functionality is not individually re-implemented and manually for each VC node, but rather consists of reusable components that can be audited separately [2]. This requires a security layer

that provides *code generation* of such functionality as needed that does not need to be touched by programmers in their day to day work, but is *implicit* for every message sent [2].

The World Wide Web Consortium (W3C) Extensible Markup Language (XML) signatures standard [10] is widely used: 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 [7]: Abstract Syntax Notation One (ASN.1) with a compact and performant binary encoding format [11, 12, 13]. 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 [2].

To simplify this process, code generation can be used. This is discussed in Section 2.10 of the background. Code generation refers to the process of creating new code automatically using a program based on some input such as the structure of a message. This technique is used successfully across the software industry and may be fruitfully applied here also.

1.2 Problem

The research questions are as follows:

- **Research Question One:** Can message integrity verification be added to the Protocol Buffer tool-chain for the requirements of complex heterogeneous vehicular systems?
- **Research Question Two:** Can such a system have an acceptable performance impact in terms of end-to-end performance time and effect on message size?

  As such, the framework should aim to be able to complete end-to-end signing serializing, de-serializing and verifying within *1 millisecond (ms)*. In terms of message size the framework should aim to add only constant extra space to serialized messages over a message signed using manually created code.
- **Research Question Three:** Qualitatively can such a system be easily extendable to add new features and algorithms as need to support the creation of practical VC systems?
- **Research Question Four:** Can this framework save development time for the creation of VC systems?

1.3 Original Problem and Definition

The original thesis project proposal is written below. The final problem statement has been iterated upon based on this:

To reduce the likelihood of an exploit, this thesis aims to minimize the attack surface of a complex micros-service architecture where each unit secures its own perimeter. To this end, message integrity must be assured.

It would be an incredibly complex task to ensure that these required protection mechanisms are implemented in both a correct and consistent manner. Additionally this process is likely to be highly error prone when carried out manually [2]. To ensure that such large VC systems correctly enforce a security policy, we must therefore use an architecture where the implementation is not designed again for each new node but it is a reusable component that can be audited in isolation [2]. This calls for a security layer that is transparent to a programmer in their normal development work, but is implicitly used for every message that is sent over the wire [2] between nodes.

This thesis is about adding digital signatures to all messages in a system where all peers use Protobuf for data serialization. The Protobuf (or gRPC Remote Procedure Call (gRPC)) tool-chain is a good place to instrument peers to add signatures without putting this effort on the daily work of the programmer.

Originally proposed deliverables for the thesis were described as follows in the project proposal: Add support through the Protocol Buffer tool-chain to verify the integrity of Protocol Buffer payloads. Decide on which scheme to use (e.g. as in Transport Layer Security (TLS), or just Hash-based Message Authentication Code (HMAC)) to assure integrity. Report on the efficiency impact of the additional cryptographic verification.

A extra milestone was described as a stretch goal if time allowed:

The weakest link in many authentication schemes is the cryptographic key used for signing. Therefore, it is common to rotate keys - as seen in TLS, Secure Shell (SSH), and all long-lived cryptographic protocols. For credit card payments (Europay Mastercard Visa (EMV)) a protocol called Derived Unique Key Per Transaction (DUKPT) is used.

Enable ephemeral (preferably per-message as in Double-Ratchet (used by WhatsApp, Facebook Messenger and many others), but can also be per-session as in TLS) keys to be exchanged, signed using a pre-shared key

1.3.1 Scientific and Engineering Issues

There are several scientific and engineering issues that need to be overcome to solve this research problem. Designing a generalized system that will work for an arbitrary structured message is a challenging task. The framework needs to flexible enough to be broadly useful for many tasks. That is it should not be useful for some toy applications but instead a large subset of the heterogeneous system communications that are in need of security guarantees.

It will also require analysis and thought to decide on cryptographic schemes to use (e.g. as in TLS, or just HMAC) to assure integrity and any other security guarantees identified. Additionally the underlying the environment to develop the tool will need to be identified out of several possible options not limited to LCM, gRPC and Protocol Buffers.

The framework (and systems developed using it) will need to have good performance characteristics. This means that the framework should run quickly and using minimal resources during the development process. On top of this the security code generated by the framework should also be efficient in terms of Central Processing Unit (CPU) and memory as the code will in many cases run on hardware with limited resources.

The framework will also need to be engineered with use-ability in mind so that the average developer can use it easily and intuitively while developing software for the V2X use case. This should mean that the interface is simple and exposes the right information so that it is difficult for the user to inadvertently make security bugs regarding such security code.

1.4 Purpose

The purpose of the thesis is to investigate and develop a prototype framework for automatically generating security code needed for communications between and within network vehicular systems.

This benefits vehicle OEM as it will reduce development time of the software components of heterogeneous networked vehicular systems. This will lead to these systems being able to be brought to market much more rapidly. Through the same vein it also benefits from reduced cost due to the developer time saved. It also reduces the risk of security bugs and vulnerabilities in such systems which protects long term reputation for the OEM. Einride the supporting company benefit for these reasons particularly the development time saved. Additionally this framework proposed for this degree project would greatly reduce the amount of time spent on security

reviews of security code as the security code would just need to be written once in the framework and audited there.

There should also be benefits to the academic field of computer security in networked vehicular environments more broadly. A framework for automatic generation of security code also practically speeds up development of simulated vehicular networks and V2X scenarios as the security code does not need to be re-developed for each heterogeneous node in the network. This will allow larger scale simulations to be tested with less work by researchers. The framework can also be built on top of by other researchers so that different security protocols can be added. Support for other languages can be added as the framework is open source. The openness and extendable nature of the framework is designed with the aim to make the thesis project as widely useful to other researchers and industry as possible.

One of the primary benefits of this thesis is on Privacy and Security of Users of Networked vehicles. This thesis aims to better protect users from being endangered due to malicious actors exploiting vulnerabilities in vehicles that they use. The framework aimed to be constructed can also help to improve privacy chiefly if it is used to write better systems to authenticate uses.

This project benefits society as it makes the realization of Networked Vehicles more easily achievable. This will positive externalities on both sustainability and social issues.

Enabling Networked vehicle use cases (such as safety and traffic messages, between vehicles, platooning remote driving and autonomous driving will provide increased efficiency in driving and reduced congestion. This should have a positive effect on the environment [14]. Similarly these use cases such as safety messages and remote driving enabled by this additional networking will make roads safer. They will do this by providing more information to improve the decision making of both local human drivers, remote drivers and autonomous systems.

1.5 Research Methodology

The engineering design process was used in this thesis project. This entailed defining the problem, then conducting background research to understand the problem space. Finally quantitative measurements were made in evaluation tests to evaluate if the problem has been adequately solved by the engineered solution.

This was an iterative process of researching and clarifying problem in more detail. Particularly a closer link was made to real VC standards used in

industry as more research was done into the literature surrounding vehicular communication both for the thesis and preparing for publishing the paper. Additionally the thesis problem focused more overtime on the core research questions over time and away from key management and ephemeral keys sections as that was considered to large of a scope. It also became clear that the thesis should not only take a quantitative but also a qualitative angle to the evaluation to show that the framework could be practically be used by a developer.

1.6 Delimitations

The chief aim of this thesis is to explore the possibility of creating a set of tooling to automatically generate security code to enable security guarantees such as authentication automatically in heterogeneous vehicular networked systems. The second aim is to create a working prototype of such a system and evaluate it for both performance and functionality.

Such tooling would really show its usefulness though time saved and bugs prevented in a large scale vehicular software development project. It is not aimed to evaluate the performance of this proposed system in such an environment due to time and resource constraints.

The system aimed to be developed for this thesis would be most effective if it worked across all languages. Initially for the scope of this thesis it will be just aimed to be functional in generating this security code for a single programming language. It is proposed that once such a system has been proposed and proven to work in one language it should follow directly that it can be created for the other leading programming languages needed in networked vehicular systems and autonomous vehicular system environments.

Such a system of automatic Remote Procedure Call (RPC) security code may have additional usefulness outside of the field of vehicular systems and more generally in software engineering due to the scope of the thesis only this field was focused on for the thesis.

1.7 Structure of the Thesis

Chapter 1 - Introduction is the current chapter and outlines the purpose, definition and scope of the thesis project. **Chapter 2 - Background** presents relevant background information on Vehicular Networks and V2X, critical security concepts as well as meta-programming concepts such as static code

analysis and code synthesis. **Chapter 3 - Method** outlines the research methodology chosen for this degree project. It also covers to evaluation method used and describes the steps taken to ensure the accuracy and validity of both the results obtained and the conclusions taken from those results. **Chapter 4 - Design & Implementation** details the design of the solution decided on to solve the problem outlined in the introduction. It describes the software components created to solve the problem and the internal algorithms used by them. The user interfaces of the components are described and the user work flow of a software developer using the framework is outlined. **Chapter 5 - Results & Analysis** showcases the results obtained through measuring the performance characteristics of the framework that was designed and implemented. Additionally the results of manual testing of functionality and use-ability of the framework is also presented. **Chapter 6 - Discussion** In this chapter implications, significance and relevance of the results obtained are discussed. **Chapter 7 - Original Work** provides information as to the peer reviewed research published as part of the work towards this masters thesis. **Chapter 8 - Conclusion & Further Work** The final chapter provides a summary of the research problem and the designed solution as well as the significance of the results. Finally it highlights limitations of the thesis project and suggestions of potentially fruitful avenues for future work.

Chapter 2

Background

This chapter provides basic background information about Vehicular Ad-hoc Networks (VANETs) and V2X, as well critical network and security concepts. Finally related work related meta-programming concepts such as static code analysis and code synthesis are introduced.

2.1 Security Requirement Definitions

2.1.1 Integrity

In the field of information security, Integrity of communication refers to the certainty that the content of a message has not been modified by interference or a third party between the time when the sender sent the message and when it is read by the recipient. This is typically achieved by using a secret key shared by the sender and intended recipient which is used to hash the content of the message to create a digest of the message. This digest is easy to recalculate on the receiver side using the received message and the secret key. Message Integrity algorithms are typically designed so that it is a very hard problem to craft a different message for a secret key which would have the same digest [15]. It is important to note that in the case where the secret key is symmetric then message integrity is only guaranteed against parties who don't have the secret key. Parties with the secret key can all create, modify and then sign or resign messages and their integrity will appear valid to other holders of the secret key.

In VC systems [6] Message integrity is need for example so vehicles can trust safety messages or remote driving commands are coming from the agent in the network that you believe it has and importantly that it has not been

modified along the way.

2.1.2 Authentication

Authentication refers to the way that the identity of a party in communications can be determined [6, 16]. Specifically in this context we are not concerned with human user auth-entication but networked computer systems authenticating each other. So password based authentication will not be considered. The main focus is cryptographic authentication in this thesis as network based authentication has the weakness of sometimes being vulnerable to network address imp-ersonation and is therefore not the gold standard [16].

Authentication can be provided through both symmetric and asymmetric cryptography. With HMAC two actors can authenticate each other if they have a shared key as they know if they didn't sign a message then it must be signed by the other party. Asymmetric cryptography provides a more robust authentication capability as if an actor signs a message with their private key any other actor who has the public key can authenticate that the message was signed by the signer if they have the signer's public key.

In VC systems [6] Authentication is needed to make sure that for example a vehicle or piece of infrastructure in question is who it claims to be and that the data sent from it e.g. collision avoidance information can be trusted due to their identity.

2.1.3 Non-repudiation

Non-repudiation [16] means that a party which has signed a message is not able to deny that they are the party who signed the message. This is due to the cryptographic guarantees of the algorithm chosen. This cannot be done with secret key symmetric cryptography as both parties share the same secret key and a party can always claim that they did not sign a message but another party that with the same key did. With asymmetric cryptography, as only one actor has the private key no-one else can sign messages with it and pose as the identity represented by it (unless the key falls into the hands of a bad actor). This means that the holder of the key cannot deny having signed a message. In VC systems [6] non-repudiation is needed for example to prove that a VANET node e.g. a vehicle was the one that was sending spam Basic Safety Message (BSM) for example forward collision warnings to make vehicles stop and create traffic problems. There should be proof that the driver was the one who created the spam messages and not that they are

being framed by another actor for example a Certification Authoritys (CAs) or V2I node. Non-repudiation allows this guarantee.

2.2 Symmetric Cryptographic Primitives

2.2.1 Hash-based Message Authentication Code

HMAC [17] is a standard cryptographic message authentication protocol which utilizes symmetric cryptography. It is used as it addresses a weakness in more basic Message Authentication Code (MAC) algorithms which are vulnerable to length extension attacks [16].

A naive MAC works by concatenating the secret key on the end of a message and getting the hash of this using a common hash algorithm such as Secure Hash Algorithms (SHA)-1. The problem with this is that an attacker who knows the message and the MAC can simply add padding to the message and then append extra data to the message at the end and this will also match the same MAC for both the original and longer message [16]

HMAC can be used with any underlying cryptographic hash function. HMAC works by applying the hash function twice. For a given message (M) and secret key (K), message authentication code will be generated. If K is shorter than the underlying cryptographic algorithm it is padded with zeros, if it is too long it is hashed with the algorithm to make it the right length. Two keys are generated from K, the inner key k_{inner} and k_{outer}. This is done by xor-ing them with repeated *0x36* bytes and *0x5c* bytes respectively. The message is concatenated to k_{inner} and then hashed. The result of this is then concatenated to k_{outer} and then this is hashed again. The result is the message authentication code. The extra steps of hashing with a second key protects against the length extension attack described above.

2.3 Asymmetric Cryptographic Primitives

2.3.1 Rivest–Shamir–Adleman

Rivest–Shamir–Adleman (RSA) [18] is a form of asymmetric cryptographic where key pairs of public and private keys are generated between parties based on initial starting values. In terms of signatures Alice can sign a payload using her private key and then it can be verified using Alice's public key which can be distributed over clear text. RSA can also be used to send keys between

parties Bob could encrypt a new session key with Alice's public key and Alice would be the only party able to de-crypt this message using her private key. This session key could then be used to sign many messages in the future. The problem is that this session key or the RSA private key could be compromised at some point in the future and fraudulent payloads could be signed from then on.

2.3.2 Digital Signature Algorithm

Digital Signature Algorithm (DSA) is an asymmetric cryptographic algorithm created by the National Security Agency and standardized by National Institute of Standards and Technology (NIST)[16]. It is based on ElGamal, another asymmetric algorithm. Both ElGamal and DSA rely on the discrete logarithm problem. DSA requires random number for each new signature and so a weak-point of the algorithm is the randomness of the random number generator used [19].

2.3.3 Elliptic Curve Digital Signature Algorithm

Elliptic Curve Digital Signature Algorithm (ECDSA) is another asymmetric cryptographic algorithm which relies on the discrete logarithm problem the key difference from DSA which it is based on, is that it uses elliptic curves. ECDSA is thus an Elliptic Curve Cryptography (ECC) algorithm. The most efficient solutions to this problem are exponential [19]. This means that generally ECDSA requires shorter key sizes for the same amount of security in comparison to RSA which uses the integer factorisation problem. Like DSA, ECDSA relies on a random number generator to sign a message. The quality of this random number generator is thus also a weak-point of this algorithm. ECDSA was chosen as the main algorithm use in the IEEE 1609.2 WG standard. See later in the background at Section. 2.8.2 regarding VANET.

2.3.4 Ephemeral Keys

Preventing key reuse entirely by generating new keypairs for each message/short session reduces the size of the security risk. If a key is swapped several times a minute then there is a very narrow window where a bad actor could make use of a compromised key. This is difficult to orchestrate clearly as there is an orchestration cost of making sure every party has the correct keys at the correct time. If the key used for signatures was updated every time ephemerally it would mean that compromised keys would be useless in the future.

Often Schemes for providing ephemeral keys in cryptography are used for encrypted messaging where forward secrecy is important. The keys generated are discarded by all parties once new ones are created to prevent previously sent messages being decrypted again. For signature verification it may be useful for auditing to be able to verify old messages at a later point. In this case although new keys are use each time, it may be desirable to store old signing keys. Developed by signal used in Signal, Whatsapp, Facebook Messenger, Skype, Google Allo [20]. Derives new keys for each message being sent, using ratchet such that there is forward secrecy and ephemeral keys [21].

2.4 Network Protocols & Encoding Formats

There are several transport layer protocols used in VC systems, in Einride Protocol Buffers are used over UDP and TCP and LCM is used over the CAN bus. In the Wireless Access in Vehicular Environments (WAVE) standard set TCP UDP and used alongside WAVE Short Message Protocol (WSMP). The fact that so many different transport protocols are used in VC communication hints at the heterogeneity of such systems.

Table 2.1 summarizes these characteristics and contrasts the various transport protocols used in VC systems studied as part of this thesis. The characteristics compared are whether they provide a reliable channel and how suitable they each are for safety critical real time systems. In addition an example use-case where the protocol would be suitable is given.

Table 2.1: Transport Layer Protocol Characteristics

Protocol	**Reliable Channel**	**Real-time**	**Use-case**
TCP	Yes	Low	Non-safety traffic info
UDP	No	Medium	Camera Feed Streaming
WSMP	Yes	High	Collision Avoidance Message
LCM w/ CAN	Yes	High	Torque sensor data

2.4.1 TCP

This is a reliable channel protocol used broadly in computer networking. The TCP datagram is larger than UDP as can be seen in Figure 2.1 [22]. This protocol allows for re-sending messages if they do not reach their

destination but at the cost of higher latency and more acknowledgement messages verifying that messages have been received. There is also a larger handshake for creating a persistent connection. TCP is used in VC systems as part of the WAVE [23] standard set as well as in non-standard use-ages. It can be used for a broad range of non-safety critical examples such as requesting predicted weather data to provide to sensors and the driver or updating map data.

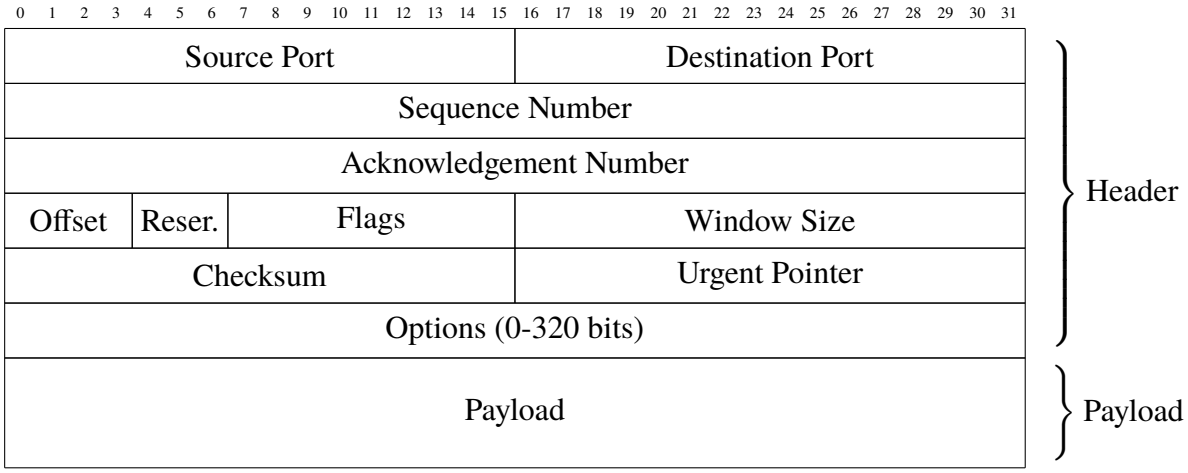

Figure 2.1: TCP Datagram Format

2.4.2 UDP

This is an unreliable channel protocol used broadly in computer networking. This protocol safes space by having a smaller packet size as seen in Figure [24]. UDP is designed to be fast and and connection-less. The header of the packets are smaller than in UDP 2.2. It does not allow for any control flow or re-transmission if messages do not arrive in time. Crucially UDP allows multi-cast functionality. Which is where one sender can send the same data to a subset of IP addresses at once. In VC systems this protocol is often used for streams of data where the latency to receive the latest data is the most important. An example of this would be streaming remote driving commands to a vehicle.

2.4.3 WSMP

WSMP is a protocol that carries out both the transport and network parts of the Open Systems Interconnection (OSI) model stack, this protocol's packet layout

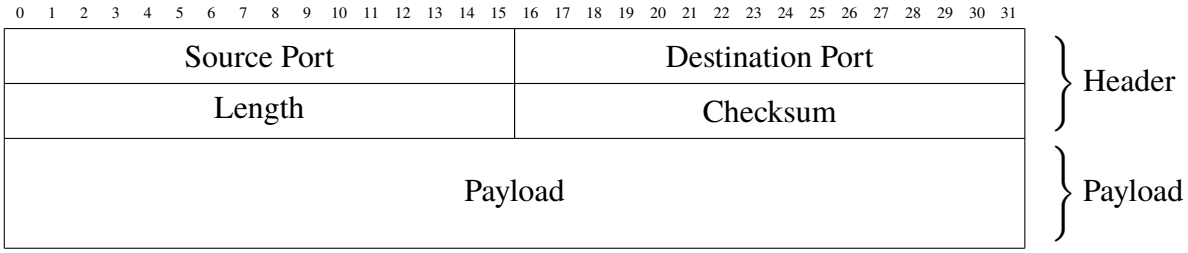

Figure 2.2: UDP Datagram Format

is shown in Figure 2.3. This protocol covers roles comparable to both internet protocol and TCP together. It was needed as a protocol was needed that would be able to prioritize smaller high priority and safety critical messages that are time-sensitive [23].

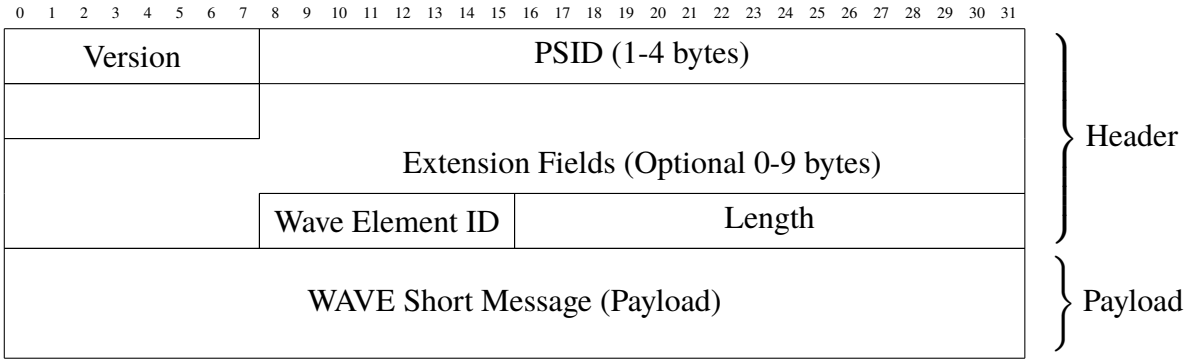

Figure 2.3: WSMP Datagram Format

2.4.4 Lightweight Communication and Marshalling

LCM is a common data communication format and library intended for inter-process communication low-latency use-cases [25]. The protocol uses a publisher and subscriber model and allows for automatic generation of marshalling and un-marshalling code for many languages. The protocol uses UDP multi-cast and allows easy logging and debugging of messages.
There are two data-gram formats a simple version if the data is under 64kB which is show in Figure 2.4 and a fragment data-gram format as in Figure 2.5 with extra metadata. If the payload is too long it is broken up into numbered fragments.

* The channel name is included for the first fragment of the sequence only.

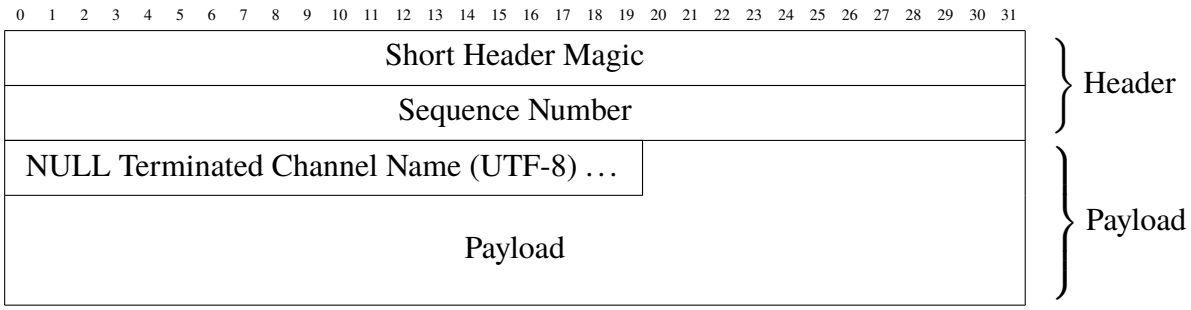

Figure 2.4: LCM Small Message Format (64 kB maximum)

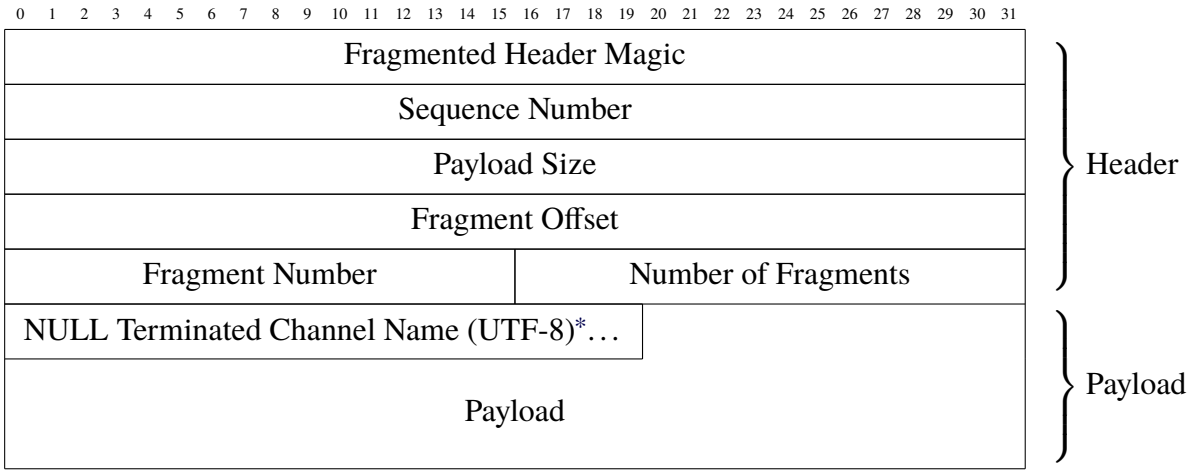

Figure 2.5: LCM Fragmented Message Format

2.5 Networked Vehicular Systems

V2X is the overarching term for vehicular commutation between the many potential parties that such a system can communicate with enabling vehicular networking use-cases. Figure 2.6 illustrates several of the categories within V2X. V2V refers to the communication between networked vehicles which can enable use-cases such as platooning, signalling, co-operative merging and co-operative overtaking [26]. V2I refers to communication with infrastructure in the road such as traffic lights, roadside information systems and parking. Vehicle-to-Network (V2N) is simply the direct connection of the vehicle to the internet chiefly through 5G. Vehicle-to-Cloud (V2C) means communication with cloud services and providers over the internet, this enables use-cases such as remote driving, information retrieval such as weather forecasts and offloading non-time sensitive compute tasks.

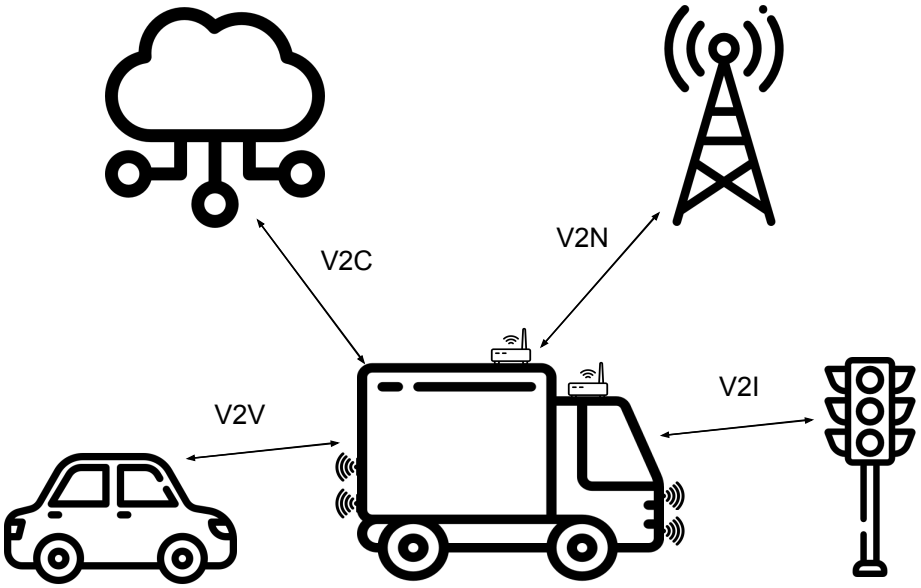

Figure 2.6: V2X Network Example

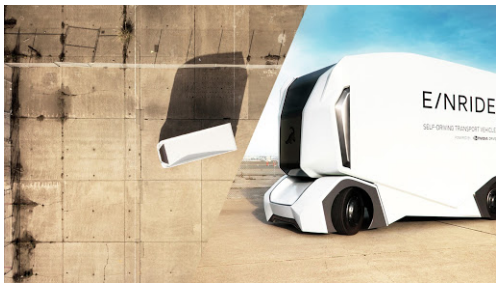

Figure 2.7: Einride's Autonomous Truck (Pod) [1]

In VC systems, vehicles regularly transmit Cooperative Awareness Messages (CAMs) and Decentralized Environmental Notification Messages (DENMs) at a frequent tempo. These messages are used to facilitate transportation safety and efficiency [2]. CAMs messages are used for sharing information about nearby vehicle positions. DENMs meanwhile is the standard system for safety messages about hazards on the road for road users as well as traffic conditions.

The susceptibility of VC systems to information security attacks has been shown and is well-understood [27]. A lot is at stake as such vulnerabilities have the potential to compromise both the privacy and even physical safety of VC users. To address these potential risks, security and privacy solutions

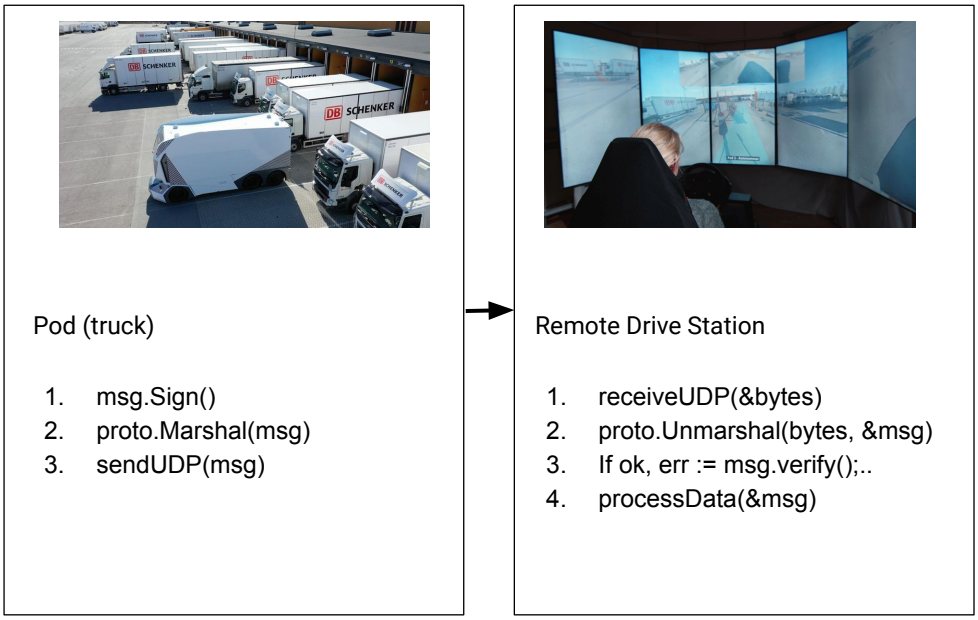

Figure 2.8: A Use-case Scenario: Datastream to Remote Driver Station Over UDP (Taken from [1, 2]).

have been proposed by standardisation bodies such as IEEE 1609.2 WG [7] and ETSI [28]), harmonisation efforts (C2C-CC [29]), and projects (SeVeCom [30, 31, 32], PRESERVE [33], and CAMP [34, 35]). Following this, there has been a general consensus to use Public Key Cryptography (PKC) to protect V2X communication through using a set of short-term anonymized certificates, known as *pseudonyms* which are issued to registered vehicles by a Vehicular Public-Key Infrastructure (VPKI) [36, 34, 37]. Users privacy is thus protected as vehicles regularly switch between unlinkable pseudonyms to maintain unlinkability [2].

The number of CAM and DENM data structure types, as specified in IEEE 1609.2 [7] and European Telecommunications Standards Institute (ETSI) [38], increases greatly as V2X systems are scaled and deployed [2]. These messages are nested data structures, needing custom code to traverse, sign and verify them. The WAVE *HeaderInfo* [7] data structure illustrates the format of basic safety messages with multiple nested sub-data structures. The ISO standard 15628 [39] specifies 20 different types ($dsrcApplicationEntityId$) for vehicular networking applications, such as parking, electronic fee collection and emergency warning [2]. Fig. 2.8 shows a use-case for remote driving

with vehicles interacting with the remote driving station every 10 ms [2]. A common process for signing (sign, serialize, transmit) and validating (receive, de-serialise, verify and handle, process) is shown [2]. All messages need to be signed before transmission to the back-end 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 [2].

2.6 Heterogeneous Real Time Systems

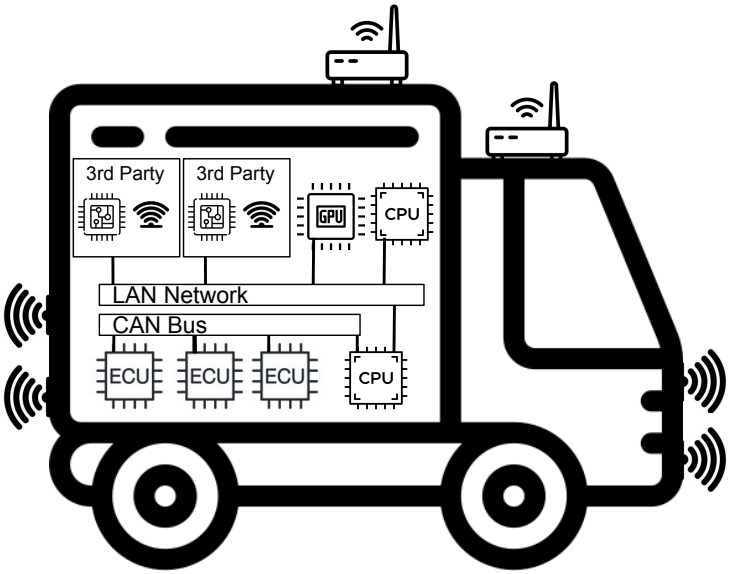

Figure 2.9: Heterogeneous Vehicular System Architecture [3]

A typical architecture for a networked vehicular system with capabilities such as remote drive, autonomous driving as well as remote drive and other V2X use-cases is shown in Figure 2.9. Such a system will commonly have a multitude of different compute units of different architectures across the gamut from low level ECUs, specialized Field Programmable Gate Arrays (FPGAs), Graphics Processing Units (GPUs) as well as other Application-Specific Integrated Circuits (ASICs) and even integrated sensor-System-On-A-Chip (SoC) packages [40].

ECUs are low level compute modules often with safety critical real time requirements. They generally control low level components in a vehicular system and use the CAN bus to communicate with each other. GPUs, FPGAs and ASICs compute units are often used for highly specialized compute task common in autonomous vehicular systems such as machine learning and linear algebra heavy workloads as well as sensor processing.

2.7 Micro-service Architecture in Vehicular Systems

A micro-services architecture is a strategy for designing software systems where the overall system is composed of many sub-systems which each are responsible for one thing and can run independently using different technologies such as languages, platforms and hardware. These micro-services communicate with each other using RPC which are requests from external micro-services asking a micro-service to execute a procedure and return a response with data. These RPC are sent between micro-services over the network and thus a micro-service system can flexibly operate across processes whether they are on the same machine in a local network or across the internet.

An example micro-architecture of an Autonomous vehicle system capable of lane following is shown in Figure 2.10. In this diagram each box is a micro-service which does one sub task which is part of the overall task of driving a vehicle. Each arrow represents an RPC sending response data in the direction of the arrow. These micro-services can then be composed together to do more complicated tasks. For example micro-services providing date from the Camera's, LiDAR and Motor Encoders are shown in the the architecture diagram Figure 2.10. These sub-components run independently and just run the task of providing the data needed when requested. The odometry unit, which is another micro-service requests this data from these sensor micro-services and when requested by other micro-services returns the latest 2D Pose of the vehicle.

This architecture generally has benefits for system development as the components can be tested and developed independently making development easier. In heterogeneous environments such as modern autonomous and networked vehicular systems, micro-service architectures have several benefits. As the compute environment is heterogeneous the many different compute nodes on the vehicle need to communicate with each other for example the

data from different ECUs need to be communicated to higher level systems such as odometry, planning, prediction which may run on different compute platforms. In addition they may use different languages and technologies. As their interaction is over a RPC these micro-services can communicate between different components within the car but also just as easily between micro-services across V2X a network. An example of this would be communication between a remote driving station for controlling a vehicle remotely and a networked vehicular system.

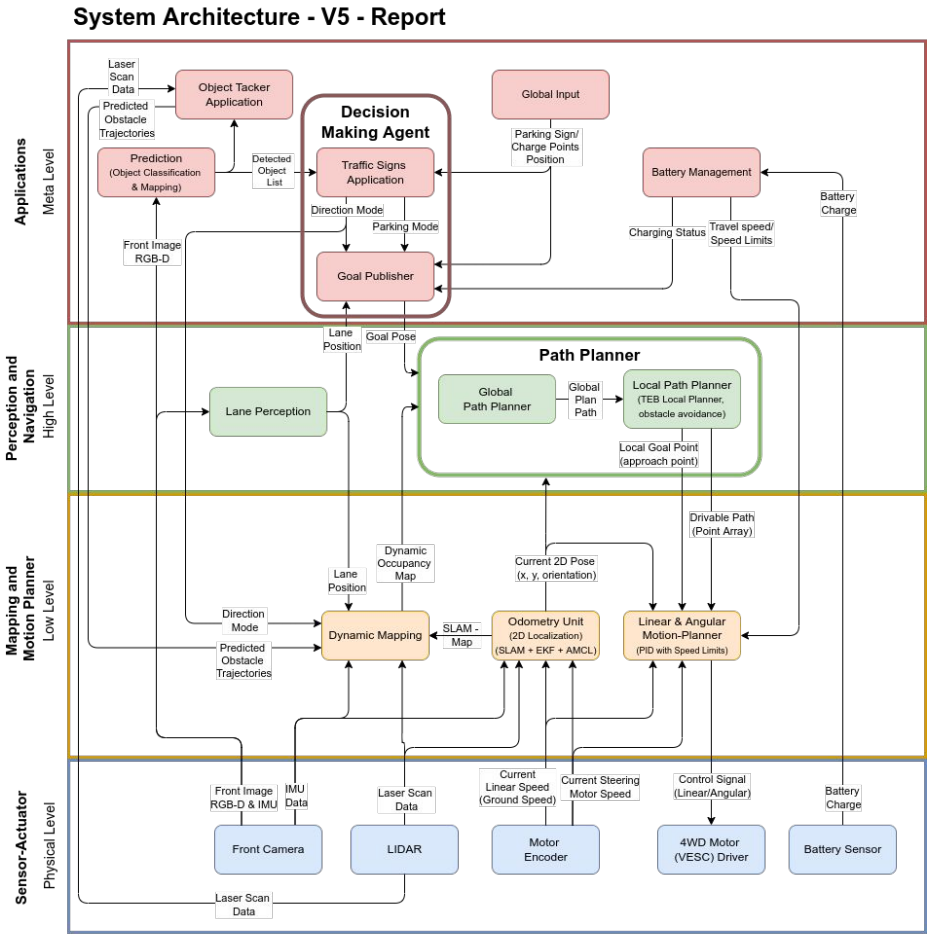

Figure 2.10: Illustrative Autonomous Vehicle Micro-Service Architecture [3]

2.7.1 Intra-vehicle Communication Authentication

There has been some work in the area of adding authentication to the CAN bus network such as CANAuth [41, 42, 43]. This is a difficult area in particular as messages on the CAN bus often have real time requirements which are difficult to balance with security requirements.

2.8 Security in Vehicular Communication Systems

2.8.1 Security and Privacy Risks

The main **privacy risk** from VC systems is data being misused. Often the most valuable information is location data. Other potential data could include.

- GPS Data (this could provide a users routine, business, home workplace locations etc).
- Driving behavior e.g. acceleration, harshness of breaking.
- Data from sensors e.g. proximity, LiDar, Cameras.
- Car model information.

Such data is valuable to advertisers as it would help them segment users to advertise to them more correctly. At the same time users have not given consent to such use-cases and so should be protected from their data being used except for VANET use-cases either mandated in the future or that they choose to use. Many of these pieces of data would also be highly valuable to an insurer for example as **data misuse** includes the case of an honest-but-curious [44] actor in a VANET for example an CA. This type of actor might try to use its position as a central piece of infrastructure to connect various pseudonyms a user is using. This could be done with timing attacks. An example of a timing attack would be a hostile actor with access to PCA data and correlating location data to de-anonymise a user and track them. The VANET standards have been designed to prevent this being possible.

There are also many **security risks** regarding VC systems. A user could have private Personally Identifiable Information (PII) data stolen from them e.g. Location data. This could be done by a compromised actor similarly to an honest-but-curious especially if they are able expand their access to compromise multiple actors e.g. both a Long Term Certification Authority (LTCA) and Pseudonym Certificate Authority (PCA).

A nefarious VC user getting multiple certificates and spamming messages is another risk. This is especially apparent for example in cases where information is crowd-sourced. A bad actor could report a traffic accident or ice on the road [6] at certain location with an aim to create traffic chaos.

This flooding of messages could also have an effect on the network. With enough messages a denial of dervice effect for legitimate messages on the network [45] could be created which is difficult to counter.

2.8.2 Vehicular Communication Security Standards

The technical standards such as IEEE 1609.2 WG and ETSI and CAR 2 CAR Communication Consortium (C2C-CC)

To solve these risks listed above the Secure Vehicle Communication (SeVeCom) [30] project funded by the European Commission, as an approach to mitigate these security and privacy problems. The project focused on identifying these risks, specifying a security architecture suitable for the threat-surface of VC systems as well as define security primitives needed for the environment.

2.8.3 Wireless Access in Vehicular Environments

The WAVE standard is an overarching standard for VC including security of such VANETs. The standard proposes the data link layer, network and physical standard for vehicular networking. The identifier for the network standard is IEEE 802.11p this is an amendment to the *Wi-Fi* standard *802.11p*. [7, 46].

Practically VC systems can be moving at high speeds i.e. driving and so the time when a vehicle is within range of a piece of VANET infra structure could be very short; in the order of seconds. Thus the main requirement for this amendment is the requirement is increased reliability and lower latency. One of the the main ways to achieve this is lower set up time. For this reason 802.11p removes the need to go through some of the traditional 802.11 initial authentication and association procedures. For example every Basic Service Set Identifier (BSSID) uses the same Media Access Control address. This also allows for smaller packet sizes.

This standard also used the *5.9 Ghz* frequency reserved specifically for VCs. The standard specifies that WSMP must be used for safety messages but allows for protocols like TCP and UDP for non-safety applications, with dedicated channels in the spectrum for safety, control and non-safety signalling.

The WAVE standards settled on a common set of several message types for use-cases of V2X systems,

- Basic Safety Message
- Intersection Collision Avoidance
- Map data
- Traveller Information
- Emergency Vehicle Alert Message

These are originally specified in the Dedicated Short-range Communications (NSRC) SAE J2335 standard [47, 48].

2.8.4 Vehicular Public-Key Infrastructure

In terms of key management the idea was that there would be a hierarchy of CAs that would provide sign public private key pairs for their children with the route-nodes being VC systems on and around the roads such as vehicles and Roadside Units (RSUs) and sensor systems on the road [4]. The CA tree would function as follows as specified in IEEE 1609.2 WG (illustrated in Figure 2.11. There would be a Root CA (RCA) that would be at the top of a given tree this would often represent a nation-state or Government. This RCA would then sign keys for OEMs and large organizations. These organisations would serve as LTCA on the next level down the hierarchy.

Each vehicle would be given a Long Term Certificate (LTC) by an LTCA that would be tied to the vehicle for its life in most circumstances. As a vehicle operates it would routinely obtain pseudonyms that it would use to sign messages. It would do this by public-private key pairs locally and sending the public keys to the PCA over a secure channel. The PCA would create pseudonyms by signing the vehicles public keys and returning a pseudonym for each public key, made up of a tuple of:

1. The identifier of the PCA
2. The lifetime of the pseudonym Short-Lived Certificate (SLC) that it would use to sign messages
3. The public key
4. The signature of the PCA

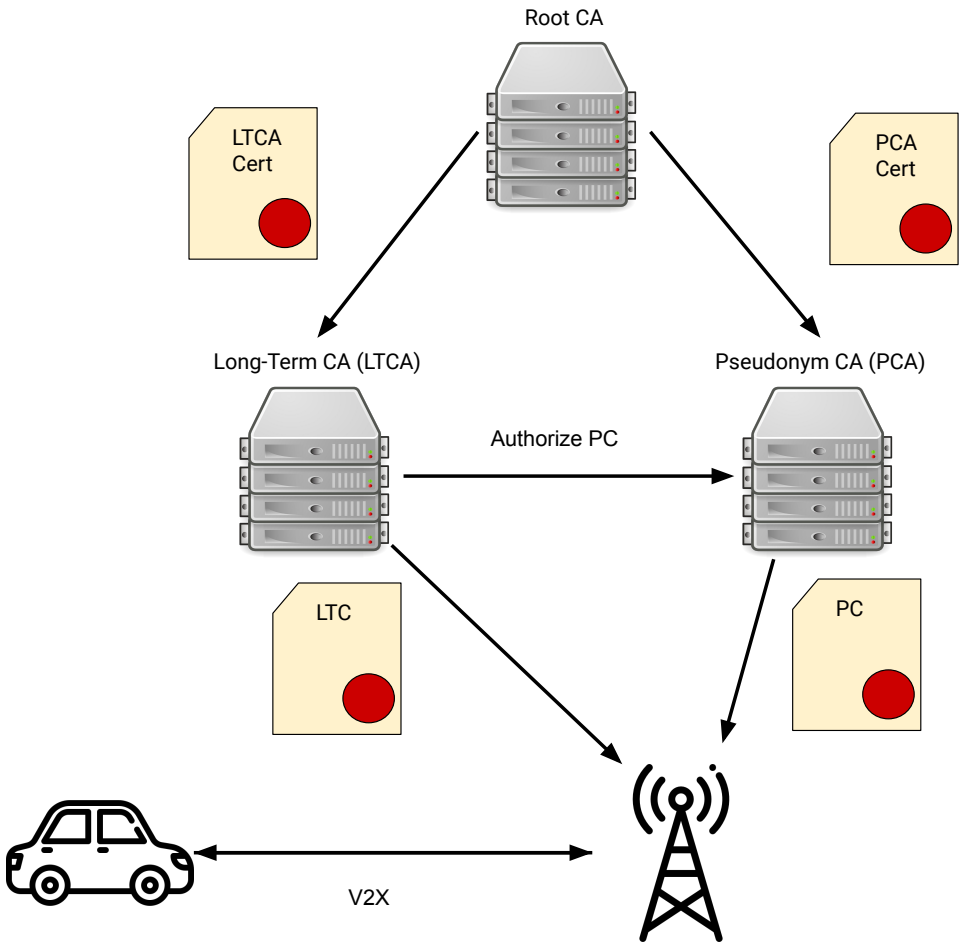

Figure 2.11: 1609.2 EU VKPI High Level Overview [4]

If only LTC were used then a CA could use this information to track vehicles as they move around. There is also the concept of a Linkage-Authority that traces misbehaving devices, revoking their certificates to then allow them to be tracked.

2.8.5 Institute of Electrical and Electronics Engineers 1609.2 WG

This is the standard for Secure Vehicle based communication, this standard builds on top of the lower layer standard of 802.11p [34].

Due to the design decision to not handle authentication at the data-link-

layer (802.11p) the authentication, integrity and non-repudiation security guarantees must be ensured at the application layer.

Standard for VC messages that can be used generally and flexible for many V2X use-cases.

The standard IEEE 1609.2 deals with the security related aspects VC communication. There are other related standards for different aspects of VC systems, for example 1609.2.3 covers Networking Services.

The standard 1609 defines several Message Types for security messages for example:

- Ieee1609Dot2Data-Signed
- HeaderInfo
- P2pcdLearningRequest
- BasePublicEncryptionKey
- PKRecipientInfo

These standards standardizes structures for the signatures, payloads and other metadata [7].

2.9 Signatures in Data interchange formats

2.9.1 Message Integrity Verification for XML

Several methods for data integrity verification of XML data have been proposed and deployed using signatures. The W3C XML Signature Standard [10] specifies the XML syntax for signatures, how they should be processed and how they should be sent with XML data.

Listing 2.1: XML Standard Signature

```
<?xml version="1.0" encoding="UTF-8"?>
<Signature xmlns="http://w3.org/2000/09/xmldsig#"
    Id="MyFirstSignature">
    <SignedInfo>
        <CanonicalizationMethod Algorithm="http://w3.
            org/TR/2001/REC-xml-c14n-20010315" />
        <SignatureMethod Algorithm="http://w3.org
            /2000/09/xmldsig#dsa-sha1" />
```

```
        <Reference URI="http://w3.org/TR/2000/REC-
            xhtml1-20000126/" >
            <Transforms>
                <Transform Algorithm="http://w3.org/TR
                    /2001/REC-xml-c14n-20010315" />
            </Transforms>
            <DigestMethod
            Algorithm="http://w3.org/2000/09/xmldsig#
                sha1" />
            <DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=<
                /DigestValue>
        </Reference>
    </SignedInfo>
    <SignatureValue>MC0CFFrVLtRlk=...</
        SignatureValue>
    <KeyInfo>
        <KeyValue>
            <DSAKeyValue>
                <P>...</P>
                <Q>...</Q>
                <G>...</G>
                <Y>...</Y>
            </DSAKeyValue>
        </KeyValue>
    </KeyInfo>
</Signature>
```

The Signature element contains signatures as well as metadata for XML data the data object which is signed is referenced by a reference Uniform Resource Identifier (URI) Listing 2.1. This Signature can be either embedded in the data object (enveloped signature) as in Figure 2.12 or the data object itself can be embedded in the Signature (enveloping signature) as in Figure 2.13.

One popular XML based protocol for exchanging structured data between web servers is Simple Object Access Protocol (SOAP) [49].Web Service Security (WS-Security) is an additional component on top of SOAP. SOAP is used as an RPC protocol to request processes to run in other web-servers and receive responses. To provide security guarantees such as integrity, authentication and non-repudiation signatures need to be integrated into this protocol. For this reason an additional component of the specification called WS-Security [50] was defined as part of the extended protocol. WS-Security

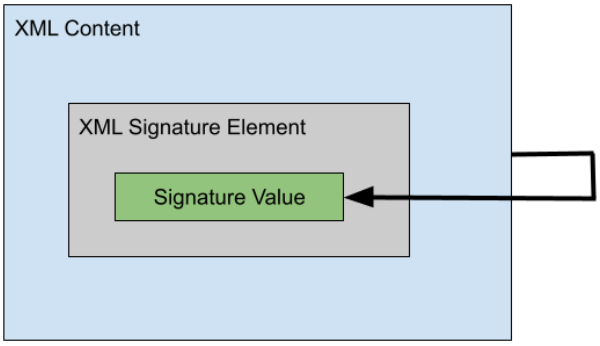

Figure 2.12: Enveloped XML Signature

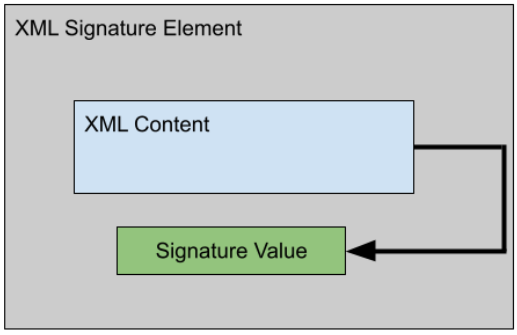

Figure 2.13: Enveloping XML Signature

makes use of the standard XML Signature elements. Specifically it uses the enveloped version of the element. WS-Security stores the Signature elements in a top level element in the SOAP request <wsse:Security> element alongside additional metadata such as expiry and creation timestamps. WS-Security also includes the Public Key of the signer of the the SOAP message. This can be useful for ensuring the correct public key is used for verifying the messages in the verification logic. This additional information comes at the cost of increasing the overall message size significantly. This extension allows SOAP

services to sign requests and responses. It also allows the receiving SOAP services to verify the signature of the requests to ensure security guarantees needed for the use-case. For example to the verify the identity of the sender (authentication) using the signature, to verify that the message content has not been tampered with (integrity) or that the sender cannot deny that they signed the request (non-repudiation).

2.9.2 Implicit Signatures

Work has been done into systems for automatically updating signing keys in a heterogeneous networked vehicle environment [51]. Rotating keys regularly is a crucial step in reducing security risk by minimising the window of attack if a key is compromised without its users being aware [52]. Automatically handling the rotation in distributed heterogeneous environment is a challenging task.

2.9.3 Protocol Buffers

Protocol Buffers * are a cross language data format which allows serialization and de-serialization on either end of the wire. Protocol buffers compared to other data interchange formats such as XML. Protocol buffers can be marshalled into a binary format which is smaller than XML or JavaScript Object Notation (JSON) minimizing bandwidth usage. Protocol buffers also allow the automatic generation of code for the parsing objects and methods for use in different languages such as GO, Java and C++. This saves developer time and speeds up development across multiple languages [53]. The compiler for translating Protocol Buffer message into a different language e.g. C++ is *proto-c*. *proto-c* creates a structure or object native to the language being compiled containing all of the fields in a message defined by a Protocol Buffer message. It assigns appropriate types for the fields in the message. It also generates methods to serve as setters, getters and clearing methods for the fields. Additionally it also generates internal code necessary for dealing with the marshalling i.e. serialisation of the message as well as un-marshalling i.e. de-serialisation of the code to and from this high level object type and the binary compressed format which the message is sent over the network with.

Protocol Buffer also allows customization to add additional functionality on top of the basic functionality. Plugins can be created that add additional functionality to the primary compiler for Protocol Buffer to a target language.

* https://developers.google.com/protocol-buffers

Plugins have been created to add additional methods to types generated from Protocol Buffers. They have also been used to add validating code that adds additional constraints to a type of message. As well as custom compilers there is also the concept of Options * which are optional pieces of information that do not change the meaning of a piece of Protocol Buffer configuration in themselves. They simply annotate parts of Protocol Buffer messages. They can be placed at the message level or added to specific fields inside of message definitions. These options can then be read by programs parsing the message types. Often they are used by Protocol Buffer plugins to identify messages that should be treated in a certain way by the plugin.

2.9.4 Protobuf Encoding

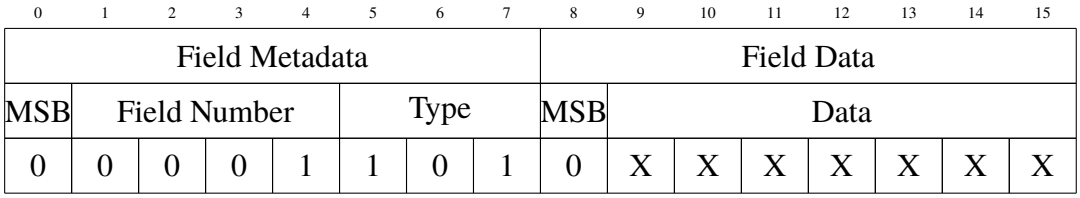

Figure 2.14: Protobuf Field Encoding

Protocol Buffer messages are serialized into an encoding where data is stored in groups of seven bits where the eighth bit Most Significant Bit (MSB) in every byte indicates whether the section of data continues to the next byte. If the bit is 1 or stops if it is 0. This allows for variable length integers to be more compact [54]. The fields in a message are not serialized in a set order. Each field in the message will start with a Field Metadata section which will indicate the field type and field number. Proceeding that there will be the data of the field. So for example in the case of Figure 2.14 the most significant bit is not set indicating that this is the end of the metadata after one byte. The field bits indicate that it is field 1. The Type bits indicate that the field's type is 32-bit in this case a float. The MSB is not set so the var int finishes at this point. The next field if there is one would begin right after this.

2.9.5 gRPC

gRPC is a cross language framework for defining RPC Application Programming Interfaces (APIs) using Protocol Buffers. gRPC itself is an instance of a

* https://developers.google.com/protocol-buffers/docs/proto#options

Listing 2.2: Example gRPC Service Protocol Buffer Definition

```
// The square number service definition.
service SquareNumberService {
  // Sends the number requested Squared.
  rpc AddOne (SquareNumberRequest) returns (
     SquareNumberReply) {}
}

// The request message containing the number to be
   squared.
message SquareNumberRequest {
  int64 number = 1;
}

// The response message containing the squared
   number.
message SquareNumberReply {
  int64 response_squared_number = 1;
}
```

Protocol Buffer plugin. By using the special keyword service around a message as well as messages with RPC written before then an entire RPC services can be spun up with essentially no other code. The Protocol Buffer messages define the interface of messages that define the requests and also the message formats that describe the response. An example gRPC annotated Protocol Buffer message is show in Listing 2.2 would represent a service which takes a number as a request and returns a squared version of that number in a response message. The gRPC framework generates all of the code needed to create a working RPC server for this service. The developer simply implements the generated functions for converting a request message into a response message. Servers and clients can be created easily and quickly for many languages as this framework has been built to work for nearly any language. The same Protocol Buffer message definitions can be use used for a server or client in any of these languages.

2.10 Code Generation

Code generation is an essential enabling technology for this degree project. The core concept of code generation is that of writing a program which itself can write other source code for new programs. This is a very powerful idea as

intuitively we can imagine that the productivity of a developer can be greatly amplified by this. If a developer can write X lines of code per day but a code generation program can be reused 100 times then a developer writing a code generation program is 100 times more productive than a develop who attempts to write the repetitive code manually each time.

A more subtle benefit of code generation is the reduced amount of bugs in a software system. The code generation program needs to be reviewed thoroughly and tested well to ensure that it works correctly for the various edge cases of code that will be generated. Once this is done though the generated code can be trusted with a higher certainty to have less bugs and also less variation in bugs compared to a human writing the repetitive code by hand. Each time a human would write such code there is additional chance of bugs being introduced [9]. With generated code if there is a bug found in the implementation the code generation template can be fixed and then the affected code can be simply regenerated. In comparison if the repetitive code was manually fixed in each instance where there could be a bug in the implementation.

There are several sub-genres of Code Generation, often referred in literature as code synthesis. Template-Based Code Generation (TBCG) is one of the most common approaches [55] in which a template source code file is marked up with parameterised sections which are customised based on input to the source code generating program. The input to the source code generating program is often written in a structured configuration language such as JSON, XML, YAML Ain't Markup Language (YAML) or Protocol Buffer.

2.11 Static Code Analysis

Static Code Analysis is a term that describes the process of gaining insights about the source code of a program without compiling or running the code [56]. Static analysis of code generally refers specifically to programs which do this analysis on source code automatically. Most common use-cases of this outside of academic research of programming languages are those of static analysis code checkers. These include programs finding bugs in code, finding anti-patterns in code, programs that help to aid with understanding of code and tools which provide statistical analysis on code bases such as usage of library and code patterns. The use-case of detecting bugs in code is the one of primary interest to this thesis. Practical example of this are the static analysis checkers which run inside ID's and give warnings about missing semi-colons and unused variables by adding marks or warning outputs inside the Integrated

Developer Environment (IDE).

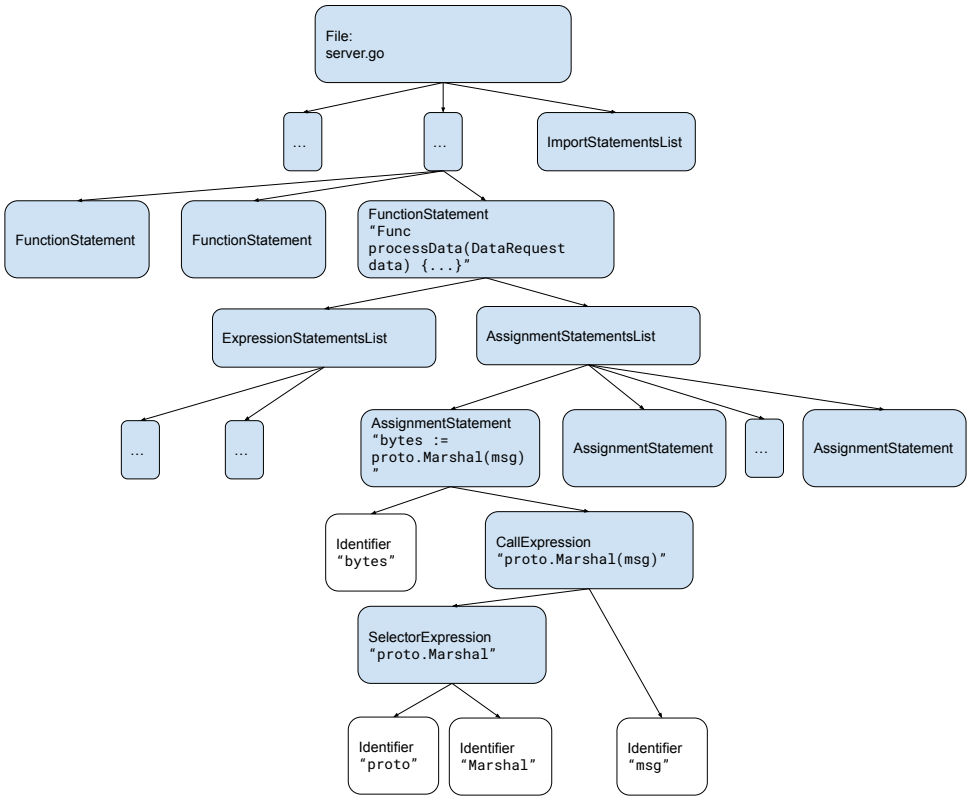

Figure 2.15: Abstract Syntax Tree Example Showing an Assignment Using a Method Call Inside a Go Program File

There are several steps to creating a code checker. The first of which is the routine to actually read the source code files that need to be checked. After this the source code text needs to be converted into a model of the code stored in a data-structure that is more straight forward to reason about. For this the obvious choice is often a tree structure specifically an Abstract Syntax Tree (AST) [57]. An AST is a tree like structure where the route node is the highest level concept in a program the file all of its child nodes are the next largest entities the package declaration, list of import statements and list of functions. Each of these sub-concepts in the tree are gradually smaller. Down lower in the tree there may be a Call Expression which has only two leaf elements and Identifier which represents the method name and another identifier which represents its one method parameter. This can be seen in Figure 2.15 towards the bottom of the tree. ASTs can be traversed easily using graph algorithms

and since the nodes in the tree represent semantically meaningful concepts in the programming language it is easy to reason with the data-structure to make inferences about the code. The third step in creating a code checker is to ascertain some facts about the program. This usually means traversing the tree one or more time to find information required information. For example a code checker could be created to find unused variables. This could be done by traversing the tree depth first and taking note of variable names as they are assigned in the tree. Then if the variable is later found in an expression statement in the tree remove the variable from the map. If a variable is not referenced again until the depth first traversal ends up going back up out of the level of the tree where the variable then we know it is an unused variable. This is a very simplified example but it is illustrative.

The language Go in particular has a strong AST library as well as a powerful introspective analysis library for parsing code inside the standard library. Similar libraries can be found for most popular programming languages.

2.12 Summary

To close of the background section the following is a short summary of the main ideas that are aimed to be to built upon during this thesis. In terms of security concepts it is clear that Integrity, Non-repudiation and Authentication are strong requirements in terms of security guarantees for V2X systems and these requirements should be possible to implement in the framework which is the goal of this thesis. The exploration of the topic of Vehicular networks and heterogeneous vehicular systems as well of the exciting use-cases of these brought into clear focus the benefit to society of this work by both helping to bring about these use-cases but also the need to protect future vehicles users from the increased risk to their privacy and security if concerns are not addressed. In terms of protocols, Protocol Buffers particularly stood out due to their good tooling for static code analysis and code generation as well as Einride's use of Protocol Buffers which makes it more practical for them if that is used. Meta-programming concepts such as code generation and static code analysis are crucial enabling technologies for this thesis and exploring them in the background helped clarify the concepts before they are practically put into practice in the design and implementation.

Chapter 3

Method

As stated in the introduction, engineering design process was used in this thesis project. This entailed defining the problem, then conducting background research to understand the problem space. Finally quantitative measurements were made in evaluation tests to evaluate if the problem has been adequately solved by the engineered solution.

3.1 Research Process

Firstly a literature review will identify the most promising message integrity options. The assumed focus will be on developing the functionality into the Protobuf compiler and adding key fields. Although other options such as working on the gRPC level of the stack and using the LCM protocol will be explored for feasibility. [25].

Then message verification will be designed and implemented into the tool-chain using options identified in the literature review.

During the design process diagrams partially based on Unified Modeling Language (UML) diagrams will be used to help visualize and understand the use case and user flow of the envisioned developer who will use the framework.

Profiling of performance of the generated Protobuf code with message integrity verification added. The planned profiling would include the latency, bandwidth, possibly CPU and memory as well as other useful metrics to Einride.

The message integrity verification component can be tested on testing messages as well potentially real world message flows replayed from components of autonomous truck components message data. The measurements from all different implementations will be compared to the baseline system without any

message integrity verification to understand the effect of this added overhead. Full end to end testing of the whole heterogeneous system can identify any macro effects due to the added overhead such as cascading delays that do not appear in simple two node evaluation. All of these measurements will be compared to each other and to acceptable metrics for Einride's performance requirements.
The aim of this work is to investigate the effect of implementing message authentication centrally in RPC serialization/serialization code to reducing the redundant re-implementation of code higher on the stack and security issues avoided due to this. This improvement can be shown by the number of lines of code saved by centralizing this message authentication.

3.2 Research Paradigm

The research paradigm chosen for this thesis is Positivist, this means that the core aim is the gain factual knowledge about the performance of such a framework as proposed. This will be done primarily through a quantitative approach; that is conducting small experiments measuring the performance of the system once it is built. There are some qualitative aspects of the research paradigm chosen as some elements of evaluation such as does the system function are more qualitative. Another qualitative aspect is assessing how useable the framework is for a developer although this is not the main aim through developing this some critical assessment of the tool can be made. Overall the main paradigm is positivist taking advantage of quantitative methods to get data to assess the performance.

3.3 Data Collection

Data will be collected primarily once the system has been developed. Timers will be used to measure the running time of different parts of the system both running time of code generated by the framework as well as time taken to generate code. Additionally if the static analysis Linter is completed then its running time will also be evaluate to see that their performance is adequate enough to be used practically. As the programs are not necessarily highly computationally dependent there should not be a significant environmental impact from the energy used to evaluate the system. Personal data is not needed in order to evaluate the system as the system can be run repeatedly

and does not need multiple people to run it individually.

3.4 Experimental Design / Planned Measurements

The experiments planned are as follows.

1. Measure the running time for signing and verifying a small message with only a signature and one other field end-to-end using HMAC and ECDSA.

2. After this the same will be repeated but with varying byte size of the message payload this will be varied to see the effect on the overall message size.

3. Measure the running time to evaluate the time taken to generate the signing and verification code created by SecProtobuf for a simple message with only a signature and one other field.

4. Measure the run-time of the static analysis Linter when evaluating a program with several integrity based security bugs.

5. Evaluated the number of bugs out of the total expected to be caught by the Linter that are caught.

3.4.1 Test Environment

All of the performance evaluation tests of SecProtobuf (the system developed) as well as programs developed using it were conducted on the same platform. The specifications for the system were as follows: It was running Ubuntu 20.04. It had 16 GB of RAM memory. The processor was an Intel® Core© i7-7700HQ CPU @2.80GHz [2]. The evaluation of the system as well as small scale tests of programs created can be reproduced easily The vehicle software stack was running on a set of containers using Docker v20.10.7 [58] running the Ubuntu Virtual Machine (VM). Further details are proprietary to the company and hence is out of scope of this section.

3.4.2 Hardware/Software to be Used

The original plan was to eventually evaluate the performance and practically it of the framework developed directly on the Einride's electric trucks compute

hardware and hopefully test drive the vehicle while running code using the framework.

In terms of software used, the main programming language to be used will be Golang. The main libraries which are planned to be used are primarily within the standard library. Note that the URIs listed here are Go package names not web Uniform Resource Locators (URLs). For static analysis components of the system the Analysis package (golang.org/x/tools/go/analysis) is aimed to be used additionally the AST Package (go/ast) will be used. Additionally outside of the core language libraries the Go Protocol Buffer library (google.golang.org/protobuf) will be used. Additionally the *protoc-go* compiler will be used and modified by adding a custom plugin to it. Additionally a proprietary vehicle simulation software will be used to verify that the system can be used end-to-end as part of a real vehicular software system. Further details about this simulation system cannot be given for company confidentiality reasons.

3.5 Assessing Reliability and Validity of the Data Collected

Steps were taken to ensure the reliability and validity of the date collected as part of the evaluation of the system.

3.5.1 Validity of Method and Data

]The validity of the results will be verified by several steps. While the evaluations are ongoing to results of the cryptographic checks will be verified to ensure they are occurring and not giving misleadingly low figures. The tests will be run first with print statement logging enabled to ensure the cryptographic steps are occurring and not being avoided or saved due to some bug. The tests will each be run several times and the average of all of the runs will be taken to try and avoid the risk of an outlier result. Keys will randomly picked to avoid some chance of an outlier key which could make the results predictable or invalid in some way.

3.6 Planned Data Analysis

3.6.1 Data Analysis Technique

As the data will not be very large in size much of it can simply be displayed in tables. For the effect of payload size on message size. The data will be graphed and visualized.

3.6.2 Software Tools

For the data-analysis python was used, the libraries NumPy and MatPlotLib were used also. In addition analysis tools such as Unix time command and Protoscope which are discussed later were used for performance analysis.

3.7 Evaluation Framework

The results will be evaluated against reasonable benchmark values. Einride provided a goal of 1 ms total for end-to-end signing, serialising, deserialising and verifying and this will be used as a bench marks for the different configurations of SecProtobuf.

3.8 System Documentation

It is crucial that the framework resultant from this thesis project is well documented so that it can both be used easily by developers as a tool, but also such that it can be further built upon, improved and extended by others. As the framework is designed to be open sourced using the MIT open source licence [59] it can be used, extended and commercialised by anyone once the licence and copyright notices are retained in the framework. For this reason it is also crucial to make sure the implementation is also well documented. The source code is required to be thoroughly commented so that future contributors can easily add to the framework. A Markdown 'README' will also be written to describe the use-age of the framework. Additionally it is hoped that this thesis along side the peer reviewed accompanying paper will serve as additional documentation.

Chapter 4

Design & Implementation

In this chapter the design decisions between possible solutions will be described. Then the design of the chosen solution is explained.

Through a combination of the knowledge gained through the background and related work research as well as evaluation of the type of solution which would be valuable to Einride while also an interesting engineering challenge three potential design paths were shortlisted as possible plans. These options roughly aligned to the level of the communication stack where the signing and verifying would be done.

The first option was building the framework at the data presentation layer. That is the using the Protocol Buffer library as a basis for the framework. Strengths of this option were versatility as Einride the company supporting the thesis utilize Protocol Buffers for most aspects of communication through out the systems of their company, low-level vehicle communications, higher-level communications as well as V2X communications. Additionally Protocol Buffers strong support for plugins as well as code generations support, particularly using the language Go were strong pluses.

The second option was building the framework at the Session Layer of the network specifically gRPC. This is a protocol for remote procedure calling which uses Protocol Buffers as discussed earlier. Einride uses this protocol for much of their higher level intra-vehicular communication as well as some V2X communication. This therefore would not be as universally practical as a solution for Einride as it would only solve the problem of Implicit Message Integrity for their communication using gRPC. From tooling point-of-view as gRPC uses Protocol Buffers, implementation should be equally possible as with lower level simple Protocol Buffers. Additionally it may be the more natural choice to integrity checks, signature checks at the data link layer

traditionally.

Finally basing the framework around the LCM was strongly considered. This would be another solution built between essentially at the Session layer as Einride uses protocol buffers as the data presentation layer inside LCM frames. This would also suffer from being less broadly useful as it would only work for messages using LCM. Additionally this protocol is primarily used for low level near-real time communications on the CAN bus, this was likely the area of operations where the additional delay of signing and verifying messages would be most felt as there are much tighter deadlines.

Weighing up these options it was decided to go with the first approach and design SecProtobuf the proposed framework using Protocol Buffers.

A framework was created to enable the automatic generation of the code needed to sign an verify any message written using the format Protocol Buffer. Additionally the framework has a static code analysis tool or *Linter* to help prevent the incorrect use-age of SecProtobuf.

4.1 Protocol Buffer Transpiler Plugin

A custom protocol buffer transpiler plugin was developed an published called *protoc-gen-messageintegrity* [2]. This plugin takes standard protocol buffer message definition files and generates additional Go code in a file `<proto_file_name>.messageintegrity.go`. This plugin is run alongside the standard *protoc-gen-go* plugin.

The plugin checks the protocol buffer message definitions for any messages that have the custom message integrity protocol buffer signature set to *OPTIONAL* or *REQUIRED* and if so add the following additional receiver methods for the *protoc-gen-go* generated Go types for these messages. The transpiler generates the needed security code for any of these protocol buffers types that will be needed for the code being compiled and that have the requisite custom SecProtobuf option set. Specifically the plugin [2] adds Sign and Verify methods on to the standard generated Go type corresponding to a protocol buffer messages. 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 4.1, as part of the normal build process [2].

Listing 4.1: Compiling ProtoBuf Message with SecProtobuf [2].

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

Listing 4.2: Calling Static Analysis *Linter* on a Go Source File [2].

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

4.1.1 Linter

To ensure that SecProtobuf [2] was used correctly and in the right places in systems a custom code analysis tool (*linter*) for the Go programming language was created [2]. The goal of the *Linter* is to give a warning if a developer tries to marshal a protocol buffer message which has a Message Integrity Signature option and they have not signed the message. Similarly if a protocol buffer message is unmarshalled and that protocol buffer message is not verified immediately in the code.

The *Linter* traverses the AST of source-code files to make its analysis to check that SecProtobuf is used correctly where required [2]. The Golang library for creating code analysis tools *"tools/analysis"* will be used to create the lint warnings. The *Linter* can be simply run through the command line interface (as seen in Listing 4.2) or integrated into common IDEs or continuous integration environments. The SecProtobuf Linter only applies to code containing data types generated from SecProtobuf Protocol Buffer messages custom option set to $REQUIRED$. For types where signing and verification is required, the *Linter* ensures that this is done for any instance of those types that appears in the code under analysis. If any instance is found where data is serialized or de-serialized without signing or verifying at the correct point at the correct point, then the tool will throw a *linter* error. For example an error is thrown if there is not a call to $Verify()$ immediately after de-serialisation of a message in the code. This provides a safeguard against entire class of errors caused by signing or verify messages being forgotten by engineers where it needs to be done [2].

The *Linter* is started as part of the pre-build process or automatically by an IDE integration so that it is run every time a file is saved. The *Linter* tool

can also be integrated to run as part of Continuous-Integration/Continuous-Deployment (CI/CD) process. The in depth process that *Linter* follows to identify potential problems with unsigned or unverified message payloads:

1. Find files containing Protocol Buffer based types by Going through the imports of each file and then see if any of them are Protocol Buffer imports by checking recursively if they import the internal proto implementation packages.

2. Go through each Protocol Buffer based structure found from the previous and see if any of them have the message integrity signature option set for a field.

   (a) Check the extensions on the fields and see if any of them are message_integrity_signature

   (b) Check that the option field in message_integrity_signature is set to $REQUIRED$

3. Make a short list of these structure types matching the previous criteria. From this criteria that these types have a field in them that is intended to store a SecProtobuf signature and they also have the piece of metadata that shows that the signature is required to be used.

4. Go through the file looking for proto.Marshal() calls on any of these types.

5. If found look that proto.Sign() is called immediately before it. If not add this finding to the list of errors to be reported by the linter.

6. Go through the file looking for proto.Unmarshal() calls on any of these types.

7. If found look that proto.Verify() is called immediately after it. If not add this finding to the list of errors to be reported by the linter.

After the *Linter* finishes it will print the various findings about signatures that may not be properly signed and/or verified across the code-base. The list of warnings are shown in Figure. 4.1. Each warning links to the potentially offending line in the code i.e. a point where a message is deserialised but does not have its signature checked afterwards for example. This enables the developer to catch these problems easily at development time instead of becoming critical security bugs.

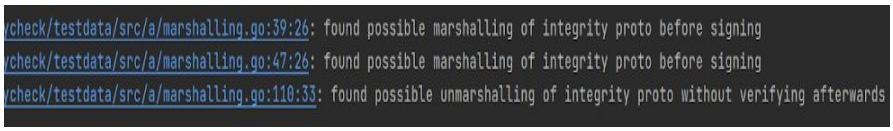

Figure 4.1: Integrity Lint Findings Returned for an Example Program with Un-signed and Un-verified SecProtobuf Enabled Messages.

4.2 User Flow of SecProtobuf

The following is a description of an envisioned usage flow of the SecProtobuf framework in the development of a new subsystem for reporting the steering angle of a vehicle.

1. Developer creates protocol buffer message representing a steering angle as well as a request for the current steering angle.
2. Developer compiles the protocol buffer message to Golang as they intend to develop the component in this language.
3. Developer develops the component of the system that needs to communicate using the generated message type in the language (Golang).
4. As the developer writes the code the *Linter* runs to identify locations where security signing and other security features are not used correctly for the messages being communicated.
5. The developer iterates based on the error and warning messages to fix the security bugs in the system.
6. The component is compiled into a binary which can be run and that is less vulnerable with fewer security bugs.

4.3 Implementation

4.3.1 Protocol Buffer Interface and Code Generation

The custom Protocol Buffer Message option message_integrity_signature was defined in Listing 4.3. Meanwhile Listing 4.5 shows an example of a Protocol Buffer message definition with the Message Integrity Option enabled and the signature field set to $REQUIRED$.

Listing 4.3: The message_integrity_signature Custom Option Definition

```
import "google/protobuf/descriptor.proto";

extend google.protobuf.FieldOptions {
    // Message Integrity Signature options.
    Signature signature = 1090;
}

// An indicator of if a field is a signature
// field and if it is required or not.
enum SignatureBehaviour {
    SIGNATURE_BEHAVIOUR_UNSPECIFIED = 0;
    // Don't use, unless to disable the signature
    // from being used.
    SIGNATURE_BEHAVIOUR_OPTIONAL = 1;
    // Allow if field does not have a signature.
    SIGNATURE_BEHAVIOUR_REQUIRED = 2;
    // Fail if field does not have a signature.
}

// Signature option for message integrity code
// generation.
message Signature {
    // Indicates if MessageIntegrity code
    // generation is enabled for the message
    // and if it is required or optional.
    SignatureBehaviour behaviour = 2;
}
```

Listing 4.4: Example Protocol Buffer Message with no Message Integrity Signature

```
import "integrity/signature.proto"
message Steering Command {
    bytes steering_angle = 0
}
```

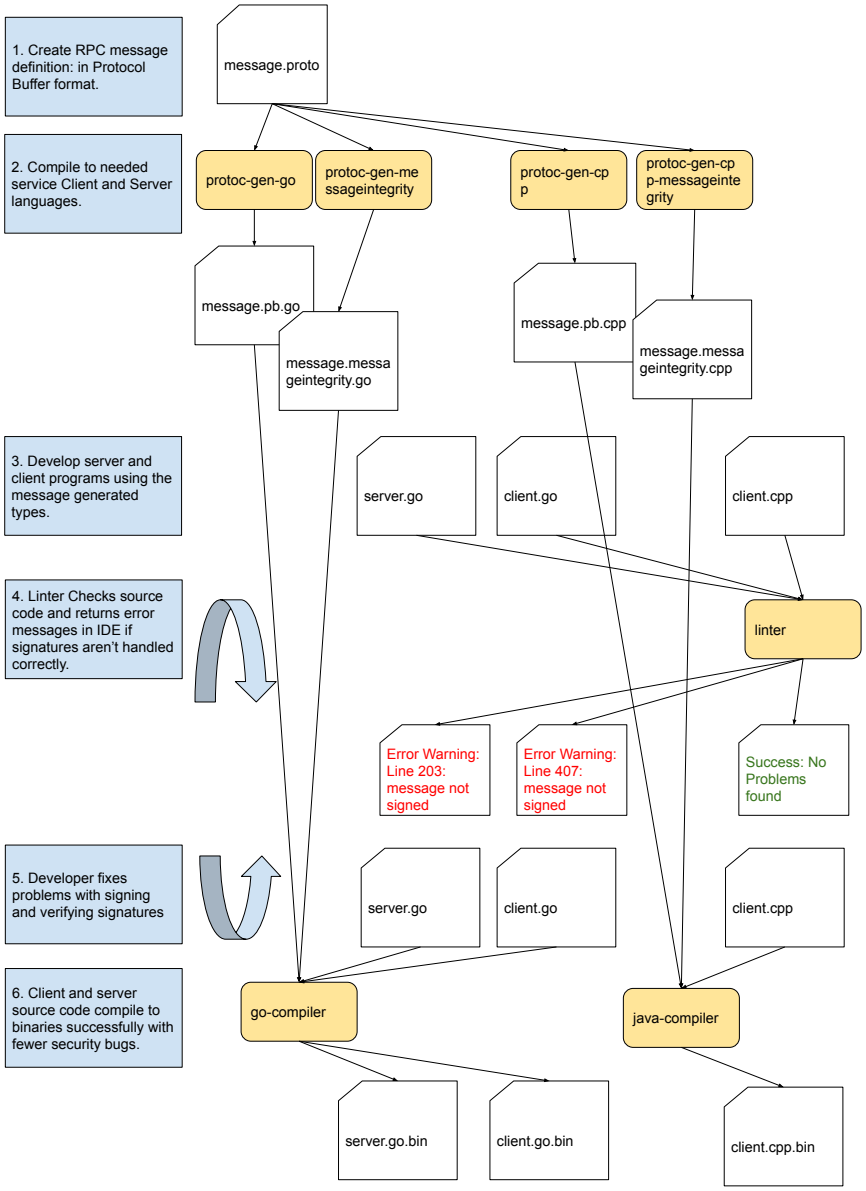

Figure 4.2: SecProtobuf User Development Process

When this protocol buffer is compiled using the message integrity plugin it will generate a file steering_command.message_integrity.go. An illustration of the content of one of these files is shown in Listing 4.6. This file is generated in addition to the standard steering_command.pb.go file which is generated for any Protocol Buffer file which is compiled with the protoc-go file without any

Listing 4.5: ProtoBuf Message with Signature Option Enabled [2].

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

Listing 4.6: SecProtobuf Plugin Generated Code for Protobuf Message (as Specified in Listing 4.5) [2].

```
func (x *SteeringCommandVerification) Sign()
   error {
        keyID := os.Getenv(
           ImplicitMessageIntegrityKeyID)
        return verificationRsaOption.
           SignPKCS1v15(x,
        verificationRsaOption.KeyID(keyID))
}

func (x *SteeringCommandVerification) Verify()
    (bool, error) {
        keyID := os.Getenv(
           ImplicitMessageIntegrityKeyID)
        return verificationRsaOption.
           ValidatePKCS1v15(
        x, verificationRsaOption.KeyID(keyID))
}
```

plugins. The message integrity generated file adds the extra functionality for message integrity signatures and verification to the generated types defined in the Protocol Buffer file.

There are two verification protocols implemented for the messageintegrity plugin. Initially a HMAC SHA-256 implementation was created. As this uses symmetric keys this offered message integrity guaranties to the messages generated using the plugin. This was implemented in the verification and verificationoption packages.

After this was completed an RSA based implementation was created in the verificationrsaoption package. The specific RSA implementation chosen was

was Public-Key Cryptography Standards (PKCS) #1 v1.5 due to its tenure of use compared to Probabilistic Signature Scheme (PSS).

Finally an ECDSA implementation was added the the framework so that SecProtobuf would be capable of using the same cryptographic algorithms as used in the WAVE standard.

The plugin was structured to take the protocol to be used as a parameter. This allowed the different protocols to be switched between easily.

Chapter 5

Results and Analysis

In this chapter, we present the results and discuss them.

For the ECDSA implementation the curve NIST P-256 as it is the main curve chosen for the 1609.02 Standard in the *ssh-keygen* tool this is listed in the tool as *prime256v1*.

5.1 Major Results

Some statistics of the delay measurements are shown in Table 5.1. The Baseline End-to-End (E2E) result is the amount of time that unmarshalling and unmarshalling on of the test messages takes. The non-baseline E2E include the time taken to sign a message, marshal it to bytes, then immediately unmarshal it and then verify the signature using the custom generated code for the message type. The Sign and Verify benchmarks just include the time for signing and marshalling, and for unmarshalling and verifying respectively.

Table 5.1: Performance of message-integrity in Terms of Processing Time (Base-line: Just Marshalling + Unmarshalling)

Benchmark	**Baseline (ms)**	**HMAC SHA256 (ms)**	**ECDSA (ms)**
E2E	0.000413	0.004754	2.266337
Sign	0	0.002165	2.104739
Verify	0	0.001976	0.125568

Given these benchmarks we can see that in these simple benchmarks the HMAC implementation of the message integrity protocol buffer plugin meets the performance benchmark of being sub 1ms. The ECDSA implementation

unsurprisingly does not meet it due to the extra complexity of the algorithm. From this it seems that the plugin can be useful for providing message integrity guarantees to any V2X communication for Einride's use-cases. The ECDSA implementation may be useful for less time critical use cases inside the vehicle as well as intra-vehicle communication. The two protocol implementations are also useful for broader use as they allow message integrity and optionally authentication and non-repudiation guarantees added to any protocol buffer messages.

5.2 Payload Size Increase

Evaluation was also carried out into the increase caused by the addition of the signature and metadata to the protocol buffer message types.

The results of this comparison are shown in Figure 5.1.

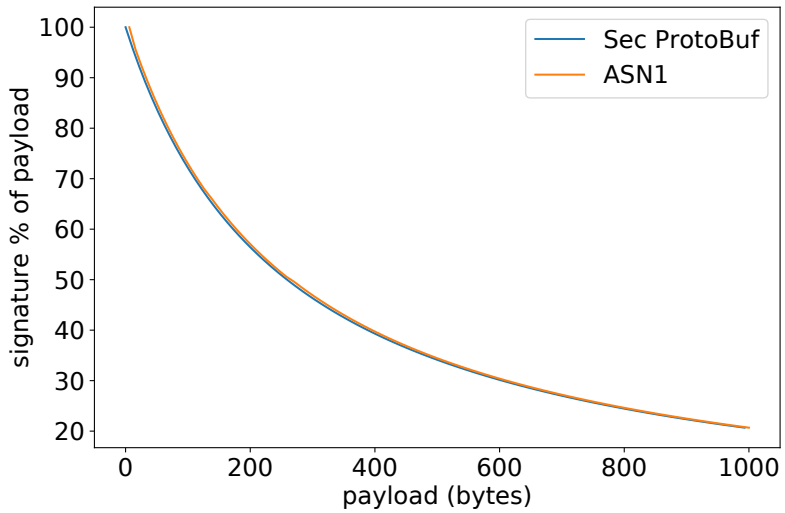

Figure 5.1: The Proportion of the Payload Taken up by the Signatures as the Payload is Increased [2].

As expected the Signature is a static size of 20 bytes and so for small messages the signature is a significant overhead on communications but for larger messages it is less significant.

Listing 5.1: Commands Used to Generate ECDSA NIST P-256 Key Pairs

```
#!/bin/bash
# Create private key.
$ openssl ecparam -genkey \
                  -name prime256v1 \
                  -noout -out private.pem
# Create public key.
$ openssl ec -in private.pem \
             -pubout -out public.pem
```

Listing 5.2: Generating Protoscope Representation of Serialized Signed Steering Command

```
#!/bin/bash
$ protoscope --explicit-length-prefixes \
             --explicit-wire-types \
             steering_ecdsa_id_4_asn1.protobin
```

Listing 5.3: ProtoScope Analysis of Signed Steering Message

```
1:I32 5.0i32   # 0x40a00000i32
2:LEN 71
  `304502207fc09ebb77af2cbf501e9dbe9f01fdc5
   e5703a6ec2c920272edccd49db220b9202210084
   ced54c2d82d9a8d1236879ad6f116d29ba94d492
   e1dda336c0e6b7f017be56`
```

5.3 Analysis of Wire Representation

The steering message from earlier was analysed to inspect how the signature is represented. This was done through a Protocol buffer utility called Protoscope [54] as well as raw analysis of the bytes using the documentation. For more information see the Protocol Buffers section in Chapter 2 Background.

The layout of the binary (shown in Figure 5.3) is simply the 32 bit float storing the value 5.0 for the steering angle. With Metadata showing the type and field number. This is followed by a ASN.1 encoding of the signature which turns out to be a byte array of length 71. The length of the byte array is encoded directly after the metadata for the byte data as seen in Figure 5.2

Figure 5.2: Analysis of Raw Protocol Buffer Bytes from Signed Steering Message

Listing 5.4: Profiling of integrity-lint Performance

```
#!/bin/bash
$ time ./bin/integritylint \
integritycheck/testdata/src/a/marshalling.go
```

5.4 Time Performance of integrity-lint

Basic profiling of the Linter was done by running the linter on the example ‘marshalling.go‘ file using the command shown in Listing 5.4. The average of 5 runs was 6.083 seconds.

5.5 Qualitative Comparison

A simple qualitative analysis was done as to the functionality of the framework vs the prior state. Previously every new message type required several hundred lines of new critical security code to be written. Now it has been written no new security code needs to be written for handling Protocol Buffer messages with SecProtobuf Signatures.

This is a huge improvement as compared to previous conditions.

5.6 Reliability Analysis

The code generation plugin, the generated code as well as the Linter were run many times during the evaluation. During all of this the results stayed consistent. This is not particularly surprising as the code should be deterministic given that the key and random noise source is the same for evaluation purposes. The system was also designed without parallelism and so there is no risk of race conditions so there was a low expectation of reliability issues being a concern.

5.7 Validity Analysis

Much thought was put into ensuring the validity of the implementation and verification of the code that was created by the system as mentioned in Chapter 3.

Chapter 6

Discussion

6.1 Research Question One

Can message integrity verification be added to the Protocol Buffer tool-chain for the requirements of complex heterogeneous vehicular systems?

In response to this research question the SecProtobuf framework was successfully created and validated. From the results it is clear that the framework can generate the code needed to sign and verify any messages with a SecProtobuf field option set.

6.2 Research Question Two

Can such a system have an acceptable performance impact in terms of end-to-end performance time and effect on message size?

On the second question regarding performance the results are mixed.

The performance of the ECDSA implementation for the end-to-end signing and verification was slower than the goal of 1 ms at 2.266 ms. As this is the algorithms used in the standard IEEE 1609.2 [4] this did not fully meet the bar of success for this research question. The path to reaching this goal for ECDSA e.g. using cryptographic enclave hardware to the cryptographic operations is discussed in the further work section.

The goal of Einride for the thesis would be satisfied with verified signatures simply using symmetric cryptography i.e. HMAC. The results for this were within the 1ms goal for the end-to-end score for this reason on performance is seen as being partially met.

In terms of effect on message size. It was found that the use of SecProtobuf did not add any additional overhead to the message payloads over a Protocol

buffer with a manually added signature. This is because the information about which field in a message is a SecProtobuf signature field is stored in the message definition Protocol Buffer file and not the serialized payload. Only metadata such as basic field type and field length are stored in addition to the signature. This means that the most significant effect on payload size is the key size.

This aspect of the performance evaluation is seen as a success and validates that the performance in terms of space is within bounds that mean that the framework can be used practically for time constrained use-cases.

6.3 Research Question Three

Qualitatively can such a system be use-able, easily extendable and save development time for the creation VC systems?

The framework was found to be easily usable to create new Protocol Buffer messages to send between a client and a server. The included *Linter* made it impossible to not call *Sign()* or *Verify()* on a message created using SecProtobuf as this is caught as a compile time error.

Three versions of the package were created supporting HMAC, RSA and ECDSA from this it was easy to modify the framework to add new cryptographic algorithms and features. As the code was automatically generated for verifying the messages by the framework. It did not need to be written manually and so would save development time.

Overall qualitatively this research goal was met although interviews and studies with potential users would have further backed up the answer to this question.

Chapter 7

Summary of Original Work

In this chapter, the summary of the papers in the context of this thesis, along with the contribution of the author, are given.

Paper A: SecProtobuf: Implicit Message Integrity Provision in Heterogeneous Vehicular Systems

Paul S Molloy, Mohammad Khodaei, Per Hallgren, Alexandre Thenorio, Panagiotis Papdimitratos
In IEEE Vehicular Networking Conference, Ulm, Germany, November 2021 [2]

Abstract: Novel vehicular applications, such as remote driving, platooning, and autonomous driving systems are increasing the complexity of networked vehicular systems. These 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 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 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*).

Contribution: The author of this thesis with the guidance of the co-authors created and evaluated the SecProtobuf framework. The conference paper was written by all authors.

Chapter 8

Conclusions and Future work

In the following sections the conclusions taken from this masters thesis will be described this will include the successful elements as well as limitations. The insights gained will be listed as well as suggestions for others aiming to further explore implicit message integrity through code generation.

8.1 Conclusions

In this thesis a Protocol Buffer plugin was designed and implemented which has the capability to generate the code needed to sign and verify messages at the point of serialisation and deserialisation [2]. Additionally to ensure that messages with SecProtobuf metadata are always signed and verified correctly a code *linter* was created [2]. This framework is not solely relevant to VC systems, but to any large scale heterogeneous software systems where there are multitudes of nodes running different software stacks and communicating using common messages. This framework has shown to be functional, practically useful and performant enough to be potentially used in industry in the future. In contrast to signature generation and validation in XML and ASN.1, SecProtobuf can generate the custom code for each data type automatically at compile time. This allows it to be statically analysed to help guarantee that the signatures are always used correctly and that the security and privacy requirements are upheld [2].

The framework and it's associated code is open-sourced under the MIT licence[59], available on Github*. A detailed 'README' has been provided† which serves as basic user documentation on how to use the framework.

* https://github.com/einride/protoc-gen-messageintegrity† https://github.com/einride/protoc-gen-messageintegrity#readme

8.2 Limitations

If both time allowed and Covid conditions were suitable for travelling to Gothenberg during my thesis in Stockholm it would have been beneficial to visit the Einride main offices and possibly evaluate the software on some of Einride's compute hardware and possibly one of the vehicles. Unfortunately this was not the case and so all of the evaluation was done on a personal computer. Having access to the specialised hardware may have been powerful enough to meet the 1 ms performance criteria proposed for the end-to-end serialisation-deserialisation. Otherwise there were not any other limitations.

8.3 Future Work

Due to the breadth of the problem, only some of the initial goals have been met. In these section we will focus on some of the remaining issues that should be addressed in future work including across evaluation, additional features and Public-Key Infrastructure (PKI).

The *Linter* was designed to take several passes over the source code being analysed in order to make observations on to correct usage of signatures. These passes were done to make the code easier to reason with, debug and test while developing, the *Linter* component. The performance in the evaluation of its performance evaluating these test file was six seconds which is large enough be disruptive to the developer flow when added to the build time of large software projects. The efficiency of the *Linter* could now be further optimised by restructuring it make fewer passes over code being analysed before it reports its findings [2]. Now that the viability of the SecProtbuf framework has been validated using Protocol Buffers one can prepare plugins that target different encoding formats, e.g., Packed Encoding Rules (PER) format message encoded in ASN.1. This would enable the framework to be more broadly useful to the wider VC community as it would allow for the automation of WAVE message signatures and message integrity codes.

If there were more time it would be interesting an valuable to test the framework against additional adversarial payloads of various kinds as measure the performance on more a more specialize compute platform such as an On-Board Unit (OBU), e.g., NexCom OBU [33].

The plugin can be expanded to include additional functionality, it would make the tool more practical, for example, if it integrated with a PKI system such as those seen in prior work [37, 60] as this tool does not manage the life-

cycle of the keys used in any way. Developing better interfacing between PKI and this plugin is left to further work. DSA and other algorithms could still be added. The performance cost of this extra metadata would also need to be evaluated.

The prototype does not meet one of the performance requirements for authentication as set out for the thesis as the ECDSA solution did not meet the 1 ms requirement for end-to-end signing serialisation, de-serialisation and verification. Possibly improvements to the code templates created for the plugin could be made to make the system slightly more efficient and pass this metric set for the thesis.

8.4 Reflections

After completing this thesis and taking some time to reflect on the work done there are some thoughts on the impact and implications the thesis could have on society in general. There are are it could have both economically, socially, environmentally and ethically. Primarily these impacts are positive. The economic efficiency of security code generation frameworks have potential to greatly speed up the large scale commercial advent of innovative technologies such as remote driving, modern electric vehicles and automatic road safety communication messages. In terms of environmental impact Einride the supporting company for this thesis is an Electric Truck company aiming to greatly reduce the environmental impact of truck haulage. This thesis project will hopefully help bring these trucks to large scale adoption soon reducing the carbon em-missions on our streets. In therms of the ethical impacts of this thesis improving the security and privacy of users of modern vehicles is a clearly ethically beneficial thing. Reducing the number of security bugs in networked vehicular systems significantly could have a clear impact on the safety and security of us all.

Overall this thesis project has been both a challenging and rewarding experience I have learned a lot about security, meta-programming and also about the academic research process which I feel has been greatly beneficial for my both my educational and professional career.

References

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

[2] P. Molloy, M. Khodaei, P. Hallgren, A. Thenorio, and P. Papadimitratos, "SecProtobuf: Implicit Message Integrity Provision in Heterogeneous Vehicular Systems," in *2021 IEEE Vehicular Networking Conference (VNC)*. IEEE, Nov. 2021. doi: 10.1109/VNC52810.2021.9644658 pp. 190–193.

[3] G. Parthasarathy, N. Dutta, D. Zebrowski, P. Molloy, R. Sahore, Z. Liu, and A. Dagar, "Applications of Robotics and Autonomous Systems Report 2: An Algorithm for Autonomous Vehicle Parking," Jul. 2020.

[4] W. Whyte, "IEEE 1609.2 and Connected Vehicle Security: Standards Making in a Pocket Universe," in *Secur. Standardization Res. Workshop*, Dec. 2016.

[5] P. Papadimitratos, A. de La Fortelle, K. Evenssen, R. Brignolo, and S. Cosenza, "Vehicular Communication Systems: Enabling Technologies, Applications, and Future Outlook on Intelligent Transportation," *IEEE Communications Magazine*, vol. 47, no. 11, pp. 84–95, Nov. 2009. doi: 10.1109/MCOM.2009.5307471

[6] P. Papadimitratos, V. Gligor, and J.-P. Hubaux, "Securing Vehicular Communications-Assumptions, Requirements, and Principles," in *ESCAR*, Berlin, Germany, Nov. 2006, pp. 5–14. [Online]. Available: http://infoscience.epfl.ch/record/94375

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

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

[9] S. McConnell, “Gauging Software Readiness with Defect Tracking,” *IEEE Software*, vol. 14, no. 3, p. 136, Jun. 1997. doi: 10.1109/52.589257

[10] D. Eastlake, J. Reagle, D. Solo, F. Hirsch, and T. Roessler, “XML-signature Syntax and Processing,” *W3C recommendation*, vol. 12, Feb. 2002. doi: 10.17487/RFC3075

[11] ISO, “ASN.1 Encoding Rules: Specification of Packed Encoding Rules (PER),” International Organization for Standardization, Geneva, CH, Standard, Feb. 2021.

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

[13] 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. doi: 10.4271/2015-01-0288

[14] A. Bibeka, P. Songchitruksa, and Y. Zhang, “Assessing Environmental Impacts of Ad-hoc Truck Platooning on Multilane Freeways,” *Journal of Intelligent Transportation Systems*, vol. 25, no. 3, pp. 281–292, 2021. doi: 10.1080/15472450.2019.1608441. [Online]. Available: https://doi.org/10.1080/15472450.2019.1608441

[15] S. G. Stubblebine and V. D. Gligor, “On Message Integrity in Cryptographic Protocols,” in *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE Computer Society, May 1992. doi: 10.1109/RISP.1992.213268 pp. 85–104.

[16] R. Perlman, C. Kaufman, and M. Speciner, *Network security: Private Communication in a Public World*, ser. Radia Perlman Series in Computer Networking and Security. Pearson Education, 2002. ISBN 9780132797160. [Online]. Available: https://books.google.de/books?id=wxMqaz4JMb0C

[17] H. Krawczyk, M. Bellare, and R. Canetti, “RFC2104: HMAC: Keyed-hashing for Message Authentication,” Feb. 1997.

[18] R. L. Rivest, A. Shamir, and L. Adleman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems,” *Commun. ACM*,

vol. 21, no. 2, p. 120–126, Feb. 1978. doi: 10.1145/359340.359342. [Online]. Available: https://doi.org/10.1145/359340.359342

[19] D. Johnson, A. Menezes, and S. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA)," *International journal of information security*, vol. 1, no. 1, pp. 36–63, Aug. 2001. doi: https://doi.org/10.1007/s102070100002

[20] J. Alwen, S. Coretti, and Y. Dodis, "The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer. Springer International Publishing, Apr 2019. doi: 10.1007/978-3-030-17653-2_5 pp. 129–158, https://eprint.iacr.org/2018/1037. [Online]. Available: https://eprint.iacr.org/2018/1037

[21] T. Perrin and M. Marlinspike, "The double ratchet algorithm," *GitHub wiki*, 2016.

[22] B. A. Forouzan, *TCP/IP Protocol Suite*. McGraw-Hill Higher Education, Jul. 2002.

[23] R. A. Uzcátegui, A. J. De Sucre, and G. Acosta-Marum, "Wave: A tutorial," *IEEE Communications magazine*, vol. 47, no. 5, pp. 126–133, May. 2009. doi: 10.1109/MCOM.2009.4939288

[24] "User Datagram Protocol," RFC 768, Aug. 1980. [Online]. Available: https://www.rfc-editor.org/info/rfc768

[25] A. S. Huang, E. Olson, and D. C. Moore, "LCM: Lightweight Communications and Marshalling," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2010. doi: 10.1109/IROS.2010.5649358 pp. 4057–4062.

[26] I. Llatser, T. Michalke, M. Dolgov, F. Wildschütte, and H. Fuchs, "Cooperative Automated Driving Use Cases for 5G V2X Communication," in *2019 IEEE 2nd 5G World Forum (5GWF)*, Sept. 2019. doi: 10.1109/5GWF.2019.8911628 pp. 120–125.

[27] S. Jafarnejad, L. Codeca, W. Bronzi, R. Frank, and T. Engel, "A Car Hacking Experiment: When Connectivity Meets Vulnerability," in *2015 IEEE Globecom Workshops (GC Wkshps)*, San Diego, CA, USA, Dec. 2015. doi: 10.1109/GLOCOMW.2015.7413993 pp. 1–6.

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

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

[30] 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. doi: 10.1109/MCOM.2008.4689252

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

[32] 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. doi: 10.1109/ITST.2007.4295890 pp. 1–6.

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

[34] W. Whyte, A. Weimerskirch, V. Kumar, and T. Hehn, "A Security Credential Management System for V2V Communications," in *2013 IEEE Vehicular Networking Conference*, Dec. 2013. doi: 10.1109/VNC.2013.6737583 pp. 1–8.

[35] "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.

[36] 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. doi: 10.1109/MVT.2015.2479367

[37] 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. doi: 10.1109/TITS.2017.2722688

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

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

[40] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, “The Architectural Implications of Autonomous Driving: Constraints and Acceleration,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’18. New York, NY, USA: Association for Computing Machinery, Mar. 2018. doi: 10.1145/3173162.3173191. ISBN 9781450349116 p. 751–766. [Online]. Available: https://doi.org/10.1145/3173162.3173191

[41] A. Van Herrewege, D. Singelee, and I. Verbauwhede, “CANAuth-a Simple, Backward Compatible Broadcast Authentication Protocol for CAN Bus,” in *ECRYPT Workshop on Lightweight Cryptography*, vol. 2011, 01 2011, p. 20.

[42] Black, J. and Halevi, S. and Krawczyk, H. and Krovetz, T. and Rogaway, P., “UMAC: Fast and Secure Message Authentication,” in *Advances in Cryptology — CRYPTO 99*, M. Wiener, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, Dec 1999. doi: 10.1007/3-540-48405-1_14 pp. 216–233.

[43] N. Ristanovic, P. Papadimitratos, G. Theodorakopoulos, and J.-P. Hubaux, “FAMIC-Fast Authentication and Message Integrity Check,” *Security and Cooperation in Wireless networks, Lausanne*, 2007.

[44] A. Paverd, A. Martin, and I. Brown, “Modelling and Automatically Analysing Privacy Properties for Honest-but-curious Adversaries,” *Tech. Rep*, 2014.

[45] H. Jin and P. Papadimitratos, “DoS-resilient Cooperative Beacon Verification for Vehicular Communication Systems,” *Elsevier Ad Hoc Networks*, vol. 90, p. 101775, July 2019. doi: https://doi.org/10.1016/j.adhoc.2018.10.003

[46] F. Arena, G. Pau, and A. Severino, “A Review on IEEE 802.11p for Intelligent Transportation Systems,” *Journal of Sensor and Actuator*

Networks, vol. 9, no. 2, Apr. 2020. doi: 10.3390/jsan9020022. [Online]. Available: https://www.mdpi.com/2224-2708/9/2/22

[47] J. B. Kenney, "Dedicated Short-range Communications (DSRC) Standards in the United States," *Proceedings of the IEEE*, vol. 99, no. 7, pp. 1162–1182, Jun. 2011. doi: 10.1109/JPROC.2011.2132790

[48] S. International, "Dedicated Short Range Communications (DSRC) Message Set Dictionary™." [Online]. Available: https://doi.org/10.4271/j2735_201603

[49] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, "Simple Object Access Protocol (SOAP) 1.1," May. 2000.

[50] B. Atkinson, G. Della-Libera, S. Hada, M. Hondo, P. Hallam-Baker, J. Klein, B. LaMacchia, P. Leach, J. Manferdelli, H. Maruyama *et al.*, "Web services security (WS-Security)," *Specification, Microsoft Corporation*, Apr. 2002.

[51] D. Püllen, N. A. Anagnostopoulos, T. Arul, and S. Katzenbeisser, "Using Implicit Certification to Efficiently Establish Authenticated Group Keys for In-vehicle Networks," in *2019 IEEE Vehicular Networking Conference (VNC)*, Dec. 2019. doi: 10.1109/VNC48660.2019.9062785 pp. 1–8.

[52] A. Everspaugh, K. Paterson, T. Ristenpart, and S. Scott, "Key Rotation for Authenticated Encryption," in *Annual International Cryptology Conference*, Springer. Springer International Publishing, Aug. 2017. doi: 10.1007/978-3-319-63697-9_4. ISBN 978-3-319-63697-9 pp. 98–129.

[53] G. Kaur and M. M. Fuad, "An Evaluation of Protocol Buffer," in *Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon)*, 2010. doi: 10.1109/SECON.2010.5453828 pp. 459–462.

[54] "Encoding Protocol Buffers Google Developers," https://developers.google.com/protocol-buffers/docs/encoding.

[55] E. Syriani, L. Luhunu, and H. Sahraoui, "Systematic Mapping Study of Template-based Code Generation," *Computer Languages, Systems & Structures*, vol. 52, pp. 43–62, Jun. 2018. doi: 10.1016/j.cl.2017.11.003.

[Online]. Available: https://www.sciencedirect.com/science/article/pii/S1477842417301239

[56] P. Louridas, "Static Code Analysis," *IEEE Software*, vol. 23, no. 4, pp. 58–61, Jul. 2006. doi: 10.1109/MS.2006.114

[57] A. I. Sotirov, "Automatic Vulnerability Detection Using Static Source Code Analysis," Ph.D. dissertation, Citeseer, 2005.

[58] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux journal*, vol. 2014, no. 239, p. 2, May 2014.

[59] H. E. Pearson, "Open Source Licences: Open Source—the Death of Proprietary Systems?" *Computer Law & Security Review*, vol. 16, no. 3, pp. 151–156, 2000. doi: 10.1016/S0267-3649(00)88906-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0267364900889062

[60] M. Khodaei, H. Noroozi, and P. Papadimitratos, "Scaling Pseudonymous Authentication for Large Mobile Systems," in *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '19. New York, NY, USA: Association for Computing Machinery, 2019. doi: 10.1145/3317549.3323410. ISBN 9781450367264 p. 174–184. [Online]. Available: https://doi.org/10.1145/3317549.3323410

TRITA-EECS-EX-2023:117