CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Designing Secure Applications for the Internet of Vehicles

Exploring how existing languages and techniques impact future security & safety

Master's thesis in Computer science and engineering

Karl David Hedgren

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Master's thesis 2024

Designing Secure Applications for the Internet of Vehicles

Exploring how existing languages and techniques impact future security & safety

Karl David Hedgren

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Designing Secure Applications for the Internet of Vehicles
Exploring how existing languages and techniques
impact future security & safety
Karl David Hedgren

Supervisor: Magnus Almgren, Computer Science & Engineering
Advisor: Emmanuel Mellblom, Carmenta Automotive AB
Examiner: Magnus Almgren, Computer Science & Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Designing Secure Applications for the Internet of Vehicles
Exploring how existing languages and techniques
impact future security & safety
Karl David Hedgren
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Connected vehicles are becoming increasingly important in light of autonomous driving, new features, and improved road safety. The recent phenomenon of vehicles communicating with the cloud, other vehicles, and user devices is often referred to as the Internet of Vehicles and promises more features while also promising increased road safety, leveraging the increased available computational power and cloud data. As vehicles begin communicating with each other and rely on cloud connectivity for features such as media systems, hazard warnings, and route planning, security concerns are raised as these features also increase the attack surface and make vehicles more susceptible to hacking, possibly leading to property damage or loss of life.

This thesis aims to investigate how the use of modern languages and development techniques may prevent malicious actors from exploiting vehicles, causing harm to individuals and society. More specifically we propose a framework for evaluating the advantages and drawbacks of new developments technologies. We then follow this framework by implementing a promising vehicle-to-vehicle network protocol across several ecosystems, showing how the industry might adopt more secure tools and languages in future automotive development.

We find that these paradigms can provide benefits to automotive development in the form of improved security. In particular, we find that the memory paradigm of Rust provides ample protection against memory-based attacks while also providing an ecosystem that is actively working towards better security in the supply chain. Meanwhile, languages that require runtimes or work at a high abstraction level such as MicroPython are found to be unsuitable due to lack of support and performance costs.

Keywords: Internet of Vehicles, Networking, Security, V2V, Language Security.

Acknowledgements

First I would like to thank my supervisor & examiner Magnus Almgren for his insightful wisdom on both the process and the content of my thesis. Following this, I would like to thank my supervisor at Carmenta Automotive AB, Emanuel Mellblom, for providing insights into the automotive industry, its challenges, and potential solutions.

I would also like to thank William Albertsson who was my thesis partner for the first month of this thesis. Although we only worked together for a short while, his ideas were used as inspiration in the final thesis. Most of his reflections, with modification, can be found in section 3.1.

Karl David Hedgren, Gothenburg, 2024-02-20

Contents

List of Figures

List of Tables

1

Introduction

Today, vehicles are becoming more akin to computers on wheels. The systems onboard are increasing in computational power and becoming more connected in order to support new features such as cloud-generated road hazard warnings, modern media systems, and cameras to detect what happens in the vehicle's surroundings. Autonomous driving is also becoming a hot topic for most vehicle manufacturers, as it promises to increase fuel efficiency and road safety.

However, as features expand, so does the attack surface. Vehicles that were previously air-gapped are now communicating over many popular wireless protocols such as Bluetooth and WiFi [1] across already used frequencies such as 2.4 GHz but also new bands such as 5.9 GHz. The risk to personal safety is also increasing as autonomous vehicles are progressively implemented. An attacker may not only gain insight into your GPS coordinates or current speed but may in the near future also gain control of the control system of the vehicles [2].

One can also see that with the rise of autonomous vehicles and their risks, the industry also requires better toolboxes to empower its developers. While the European Union and other regulatory bodies are preparing abstract methodical frameworks [3], we also require knowledge of the concrete tools that developers have available.

1.1 Motivation

Due to the safety and integrity hazards involved, it is important that the algorithms running on our vehicles remain as secure as modern techniques allow them to be. Code is always vulnerable to developer errors but modern techniques can be used to ensure that many common mistakes are avoided.

We also have to consider performance implications as these algorithms will be running in constrained environments. It is therefore imperative to begin designing and evaluating algorithms and their implementations to make them as secure as possible.

1.1.1 Performance

Although recent developments have provided the industry with more powerful microcontrollers, requiring less energy per calculation, the strict requirements on the industry and novel product developments still limit developers. Vehicles have limited

bandwidth [4], computational capacity, memory, and energy compared to consumer computers or server infrastructure. The systems are also real-time and time-critical as they are required to react within milliseconds. This poses an issue as the security systems running on the vehicle must be efficient to not disrupt normal operations or drain the energy reserve. It also makes certain types of workloads impossible as there simply is not enough memory to perform the operation.

1.1.2 Security principles

It is important to always consider the security implications when designing digital systems. All components, from hardware to software, may pose a risk to the stability, security, and integrity of the system. This is especially true for automotive systems, where a lack of security may result in a lack of safety. In the event of failure, the systems must either fail gracefully or otherwise provide backup systems or procedures to guarantee the safety of passengers and pedestrians.

Another important factor is the development principles themselves. Most embedded development is done with a mix of C and C++. In the last decade, memory-safe languages have become increasingly popular with Rust being ranked as the most popular programming language according to StackOverflow [5]. These languages promise strong guarantees that the compiled code executes as expected, in comparison to C where memory management is manual and error-prone [6]. Memory mismanagement is also a common source of security flaws, as can be seen in common attacks such as buffer overflows or segmentation faults [7]. Even if a single programmer may be proficient and able to avoid mistakes, it becomes impossible to have total control when orchestrating a code base of hundreds, if not thousands of contributors over millions of lines of code. This necessitates the introduction of tools to coordinate development efforts, to avoid bugs and exploits.

1.2 Aim

This thesis aims to provide knowledge on the principles of development for secure development and networking in Vehicle-to-Vehicle (V2V) environments. First, we will look at the current industry standard and evaluate why the landscape is shaped the way it is. Following this, we will research how new technologies may help ensure program correctness and memory safety in ways that are currently unavailable as these techniques help developers to produce secure programs by providing safety nets even before testing and deployment. We also research how these choices affect program performance in several dimensions, such as power usage, processing speed, and memory usage.

1.3 Limitations

Each of the main goals of this thesis has a set scope, as written below.

1.3.1 Development Principles

One of the primary goals of this thesis is to understand the advantages of different development principles and tools. We will be looking at different languages, taking into account the core features they present, in the context of providing executable binaries without undefined or unintended behavior. Libraries and techniques used in these languages that further improve the security and integrity of the code, e.g. providing safe parallelism primitives, may also be of interest as some languages rely heavily on libraries to provide these features.

1.3.2 Testing

The algorithm will be tested on consumer hardware with similar specifications as vehicular Electric Control Units (ECU) to provide faithful tests in the absence of real ECUs. It will additionally be tested on simulated hardware, as this environment better reflects where most developers will prototype their algorithms.

1.3.3 Thesis outline

First, we introduce the motivation behind the thesis and our goals in chapter 1. Chapter 2 introduces the literature upon which this thesis is built upon. This is followed by chapter 3, where we provide a general description of the technologies explored and utilized within the scope of this thesis. We then describe our method in chapter 4, providing concrete aspects to analyze. Later, in chapter 5 we share the details of the work as outlined in chapter 4. Subsequently, chapter 6 presents the concrete findings derived from our research, while in chapter 7, we delve into further thoughts and discussions related to the results. Finally we present our final thoughts and conclusion in chapter 8

2

Related Work

This chapter presents an overview of the literature related to this thesis. Beginning with those who provide context to the motivation of the thesis, and ending with those who focus on utilizing programming languages to bring greater overall security.

2.1 Automotive Networks

Ray et al. present a general overview of current security challenges within the automotive domain [8]. These include both recent or future attack vectors but also development aspects such as the long-expected lifetime of vehicles. We used this paper to outline potential problems for us to solve, which ultimately led us to consider how other languages could be leveraged to provide more robust software while maintaining real-time requirements.

In *Automotive requirements for future mobile networks* [9] Alieiev et al. provide context on the future expectations of automotive networks by presenting several use cases for systems in the field of assisted driving. They advocate for more advanced cellular networks before the industry can adopt data-heavy vehicular applications. However, the study emphasizes centralized networks and does not cover mesh networks. Inam et al. and Al-Saadeh et al. both investigate how the next generation of cellular connectivity (5G) can enable automotive network applications using physical experiments in their papers *Feasibility assessment to realize vehicle teleoperation using cellular networks* [10] and *End-to-End Latency and Reliability Performance of 5G in London* [11] respectively. They find that 5G enables very low latencies, although they can not be considered real-time in many scenarios, and that the requirements are very demanding. Following this, we decided that mesh networks or other alternatives to centralized communications should be evaluated given that centralized communication technologies are already actively researched and commercialized but may not yet be ready for this scenario.

However, there is also work conducted to investigate non-centralized approaches to automotive networking. Dey et al. look at heterogenous networks and their reliability. They ultimately find that the switch between local and non-local networks does not allow for safe operations due to the latency of such handoffs. Cesana et al. find more promising results in their paper *C-VeT the UCLA campus vehicular testbed: Integration of VANET and Mesh networks* [12]. Showing that indeed local mesh networks can sustain many potential future applications without relying on a

central uplink. In this thesis, we further investigate the potential of these networks using consumer hardware.

2.2 Formal Verification

Formal verification is a tool to mathematically prove the correctness of a program. Lauser et al. used Tamarin to formally verify AUTOSAR's Secure Onboard Communication [13]. The Tamarin prover was used to verify that state transitions within the applications were valid. As the complexity of protocols grows, the risk of human error increases. Bojjagani et al. further explored how formal verification can be used to secure key management and authorization [14].

We used these papers as inspiration when selecting the focus of this thesis. A way of formally ensuring program correctness can be found within the languages and development process itself by assuring that the tools and languages enforce conditions upon the program during compilation and development.

2.3 Language Security

Pinho et al. in their paper *Towards Rust for Critical Systems* [15] show how Rust may enhance the security of critical systems by highlighting existing issues in languages such as C or ADA, many of which are solved by design in Rust. Further, Balasubramanian et al. in *System Programming in Rust: Beyond Safety* [16] show how new languages, in this case, Rust, may enhance the security of critical systems. Additionally, they show how Rust could help in the certification process, providing economic incentives for manufacturers to adopt new technologies. They show how Rust can be used to provide zero-cost abstractions using its' linear type system which is both performant and secure. We further develop this idea by testing an implementation for the automotive domain and evaluating the ecosystem.

Finally, Sareen and Blackburn investigate the benefits and costs of automatic memory management techniques, such as garbage-collection, in the context of microarchitecture in their paper *Better Understanding the Costs and Benefits of Automatic Memory Management* [17]. We further compare how automatic memory management stands in comparison to other novel methods, namely the borrow system of Rust.

2.4 Efforts within the Internet of Things

The automotive domain and the Internet of Things are moving closer together and we will most likely see much development between these two fields. One effort towards safer embedded development is introduced by Valliapan et al. in the paper *Towards secure IoT programming in Haskell* [18]. They present an embedded domain-specific language in Haskell which assists developers in writing programs that compile to safe

C-code. We build upon this by investigating how a type system, or lack thereof, can impact the security of an application in non-functional languages.

3

Background

In this chapter, we go through the technologies that are used and referenced within this thesis. In particular, we go through some of the history of vehicular communication, the technologies that enable vehicle-to-vehicle communication, and some platforms and languages that may prove useful to the automotive industry in the future.

3.1 Vehicular Communication

Throughout the history of automated transport several different network types have been developed. The most relevant for this thesis are presented below.

3.1.1 Mobile Ad-Hoc Networks

Mobile Ad-Hoc Networks (MANETs) have been around for several decades and were initially developed for military applications to enable communication between military personnel in remote areas without the need for any existing communication infrastructure. MANETs operate by allowing wireless nodes to establish a network without any preexisting infrastructure, making them highly adaptable and versatile in various situations [19].

With the advancement of technology and communication protocols, MANETs have been transformed into Vehicle Ad-Hoc Networks (VANETs) and the Internet of Vehicles (IoV) to enable communication between vehicles and the surrounding infrastructure. The main difference between MANETs and VANETs is that VANETs are designed specifically for vehicular communication, while MANETs are more general-purpose networks.

3.1.2 Evolution towards Vehicle Ad-Hoc Networks

Vehicle Ad-Hoc Networks (VANETs) and MANETs are wireless communication networks that are designed to operate without any preexisting infrastructure. However, there are some key differences between these two types of networks.

One of the main differences between VANETs and MANETs is their mobility patterns [20]. In MANETs, nodes are typically mobile and move in a more random and unpredictable way compared to VANETs. In contrast, vehicles in VANETs move in a

more organized and predictable manner, following established roads and traffic patterns. This difference in mobility patterns can impact the design and functionality of these networks, with routing protocols for VANETs being able to take advantage of the more constrained movement patterns.

Another key difference between VANETs and MANETs is the types of applications and services that are supported. Although both types of networks can support a wide range of applications, VANETs are specifically designed to support vehicular applications, such as collision avoidance, traffic management, and infotainment systems. These applications require high-bandwidth real-time communication, which can be challenging to achieve in a dynamic, high-speed vehicular environment. In contrast, MANETs are more general-purpose networks that can support a wide range of applications, including military and emergency response applications, but they may not be optimized for the specific requirements of vehicular applications [1].

3.2 The Internet of Vehicles

The Internet of Vehicles (IoV) refers to the interconnected network of vehicles, roads, and other infrastructure that enables the exchange of information between different vehicles and transportation systems. VANETS are subsequently a part of this concept. The IoV has the potential to change the transportation industry, by providing data that can lead to increased safety, reduced traffic congestion, and improved fuel efficiency. However, the IoV also presents a range of challenges and security risks that must be addressed to realize its full potential.

One of the key benefits of the IoV is improved safety on the roads. By allowing vehicles to communicate with each other and with the infrastructure around them, the IoV can help reduce the risk of accidents and improve overall road safety. Additionally, the IoV can help reduce traffic congestion by providing real-time information about traffic conditions, allowing drivers to make more informed decisions about their routes and travel times.

3.2.1 Protocols for the Internet of Vehicles

The technology is still in its infancy and the industry is still deciding on how to best use the available data to enrich the user experience [21]. As such the latency and throughput requirements are still undecided. Using VANETs for the majority of V2V communication may be necessary to ensure that vehicles respond in time to signals sent by other vehicles in close proximity. This has several drawbacks as it requires much more complex communication protocols. On the other hand, the industry speculates that 5G technology could allow for centralized communication, leveraging contemporary technology to provide secure communication and still delivering within the latency requirements [22].

It is not yet clear exactly what data will be sent through the future IoV. Some propose simple warnings that may only require kilobytes of bandwidth, while others suggest that it may be desirable to transmit the whole spectrum of sensor data

including camera feeds. Therefore, it is important that the protocols designed today provide a high bandwidth to not restrict future use cases.

The network must also ensure secure communication, as the data sent on these networks will be valuable or sensitive in several aspects. An attacker with access to this data may pose a safety risk, slow down normal road operations, or contain data that is private to the vehicle manufacturer. Furthermore, an attacker capable of manipulating the normal communication flow either by denying communication or injecting malicious packets may use this to gain access to vehicular computer systems or connected clouds [23]. This is especially concerning, as vehicles are, by nature, dangerous when operating outside of their specified parameters. A novice attacker may, by accident, misinform autonomous vehicle systems in a way that leads to personal injury or death.

To secure communications, we require authentication systems and integrity alongside confidentiality for both the data and network routes [24].

3.2.2 Communication Protocols

As part of our thesis, we searched for a peer-to-peer algorithm that could be used for communication between vehicles and infrastructure. This would serve as a baseline implementation allowing us to more precisely measure the effectiveness of our tools and techniques. Although we are not concerned with the specific terminology used to categorize the algorithm, they all stem from the idea of MANETs and VANETs, which can be read more about in section 3.1.1 and section 3.1.2.

OLSR (Optimized Link State Routing) is a MANET protocol capable of building mesh networks by proactively exchanging topology information between nodes designated as routing nodes [25]. The algorithm is most efficient in densely populated networks by assigning router roles to nodes with many links. It does not make assumptions about the link layer and provides bi-directional (symmetric) communication as messages travel the same route both ways. As VANETs may not always be dense, they offer fewer optimizations for those cases.

SAODV builds upon an older unsecured MANET protocol called AODV by introducing routing security, explicitly protecting from black hole attacks [26]. The protocol assures that a node dropping data packets, but otherwise participates willingly, will be unable to prevent communication for those nodes routing through it. The algorithm does not assure confidentiality nor data integrity, only route integrity.

IPSec IPSec is a widespread communication protocol based on TCP/IP [27]. It protects nodes from source spoofing, eavesdropping, and data modification. In addition, it is cipher agnostic and integrates with existing IP infrastructure. The reliance on IP makes it suitable for centralized communication but may be problematic for MANET deployment in the event that the industry decides on using protocols other than TCP/IP which seems likely considering the current suggested requirements of V2V communication. It is nevertheless an interesting protocol due to its widespread use and scrutiny.

SUPERMAN is a communication protocol designed for MANETs [24]. The protocol aims to resolve the issues present in previous protocols that are only intended to secure the data or the route. The protocol provides end-to-end encryption using Authenticated Encryption with Associated Data to also ensure the integrity of the payload. By leveraging a central certificate authority (provided by the vehicle manufacturer) and vehicle certificates, the protocol ensures that only authorized nodes are allowed to participate. The protocol also verifies the integrity of each message on each hop.

SUPERMAN is comprehensive in scope, mostly due to the attempt to solve many different issues in one single protocol, much like TLS. In addition to the algorithm itself, the system requires a relatively complex certificate management infrastructure. Certificates will need a central authority, in the same way as TLS, and certificates will have to be stored safely in each vehicle.

The algorithm is in its principles similar to IPSec, but is more efficient in MANETs, requiring less data to be transmitted when attempting to secure a complete network graph. It also requires less overhead compared to IPsec during communication.

There are many different takes on secure algorithms for MANETs. In the end, we will need a full protocol suite as we have found for the web with HTTP over TLS via TCP/IP. In the meantime, we can analyze and evaluate the proposed components. OLSR and SAODV have promising aspects as network layers; although both lack components. OLSR provides little to no routing security but would be efficient in metropolitan road contexts. SAODV provides better resistance to attacks but there are still holes to fill. In particular, the protocol does not authenticate nodes, nor does it protect against man-in-the-middle attacks.

IPSec is a battle-tested protocol and provides good security given secure ciphers. It is, however, a session protocol and does not consider routing, aside from encapsulating TCP/IP. SUPERMAN instead draws inspiration from TLS, another battle-tested protocol. Provided that the protocol can provide the same guarantees once scrutinized by the cryptographic community it looks promising as a session layer protocol. Just like IPSec, it only partly involves itself with routing, but in contrast to the other network layer protocols, it provides better route and message integrity along with confidentiality. However, an important question is still left unanswered. Considering that the protocol relies on a certificate hierarchy, the industry must decide on how to secure and deploy certificates. If a vehicle's certificate is lost, there must also be some sort of revocation and re-issuing of certificates.

3.2.3 Energy & Performance Requirements

Energy requirements are a critical consideration when designing vehicular communication systems, as vehicles rely on limited energy sources, such as batteries or fuel, to power their operations. Balancing energy savings with the need for reliable communication, sufficient bandwidth, and maximum mileage is a crucial challenge in the design of these systems.

In vehicular communication systems, reducing energy consumption is crucial to ex-

tending the lifetime of the vehicle's energy source and ensuring efficient operation. However, there is often a trade-off between energy savings and the need for sufficient bandwidth and reliable communication. High-bandwidth communication requires more energy than low-bandwidth communication and more reliable communication often requires redundant transmissions, which can consume additional energy.

However, reducing energy consumption too much can lead to compromised communication quality and reduced mileage. For example, if a vehicle's communication system has to spend too much energy on transmitting or receiving data, it could drain the vehicle's battery faster, reducing the distance it can travel before requiring recharging or refueling.

Thus, finding the right balance between energy savings, bandwidth, and mileage is crucial in designing vehicular communication systems that can efficiently and reliably operate in the dynamic and challenging vehicular environment.

3.2.4 Autonomous Vehicles

As autonomous vehicles become more prevalent, concerns about their security have grown. Risks associated with autonomous vehicle security include potential cyber-attacks and unauthorized access to vehicle systems. With the increasing use of connected technology in autonomous vehicles, it becomes essential to ensure that they are designed and developed with robust security measures that can protect them against potential threats. Attacks can result in loss of control of the vehicle or even remote control. This would enable a new kind of threat as connected emergency vehicles could become controlled or monitored by crime syndicates or malicious actors from other nations.

The emergence of autonomous vehicles has brought to light a variety of legal and regulatory issues that require careful consideration by lawmakers [3]. One key area of concern is the role of legislation in regulating autonomous vehicles. While these vehicles have the potential to revolutionize transportation, they also pose unique safety and liability challenges that must be addressed through effective legislative frameworks [28]. To ensure that policymakers feel safe in this legislation, the industry must assure lawmakers that the systems are developed methodically to produce safe hardware and software. These are in development by companies and government organisations [29].

3.3 Known Security Concerns

Here we outline some of the security issues that this thesis references. We chose these based on the new risks posed to vehicles as they become more connected. We further elaborate on their nature and reason for inclusion within each section.

3.3.1 Good Practices for Security in Smart Cars

In 2019 the European Agency for Cybersecurity (ENISA) released its report on what it considers good practice when developing for the future automotive industry [30]. This report provides a good baseline for our investigations as it highlights risks associated with increased connectivity and vehicular autonomy that the industry has previously ignored, mostly due to vehicles historically being air-gapped systems. According to the report, the general public is positive about the development of autonomous vehicles but may not know the risks involved with this technology.

The report also highlights many different types of assets with which future vehicles may interact. This includes both physical and abstract assets. Infrastructure, such as traffic lights, may connect to self-driving vehicles using wireless technology to relay their status, allowing vehicles to make more informed decisions, than solely relying on computer vision. For example, if the camera is somehow obstructed it may not be able to ascertain the status of a streetlight. Local broadcasts of the current signal status could then be used instead. The techniques are not mutually exclusive either, they could also be combined to provide further grounds for making a decision. Two other interesting entities are the home and phone of the owner. They suggest that this can become a new attack vector as these systems are comparatively less scrutinized, in terms of security. Even benign functions such as fuel status may become attack vectors if the system is poorly designed.

Another important topic is threat taxonomy. The report catalogs many types of attacks and rates them in terms of impact, ease of detection, affected assets, recovery effort, and where OEMs may find gaps in knowledge. It also provides several countermeasures to these attacks, which automotive manufacturers are expected to adhere to. One such attack is the deployment of malicious firmware on OEM update servers. An attack where the deployment server is tricked into thinking an illegitimate binary is valid, transmitting the update to vehicles. As the update comes from a valid deployment server, the vehicle proceeds to update its firmware. At this point, the firmware may entrench itself by persisting in backup memory or installing itself onto other ECUs.

Few domains are as security critical as the automotive domain as visualized by this report. In the example attack, the ramifications could be severe as it would possibly allow for full access to the automotive systems. This results in both loss of integrity and safety. The report considers only *full* and *high* automation of vehicles, meaning that current vehicles where the driver is in final control are not included. The many different assets introduced also support the argument that the cloud may not be able to support the sheer volumes of data produced in real-time. Systems will nevertheless have to be split into critical and auxiliary sections.

3.3.2 Memory Safety

Computer memory has always come with many types of security risks [31]. Some of these are caused by malicious actors, utilizing underlying systems outside the control of the developers. Exploits such as Rowhammer target the physical nature

of DRAM to change memory that is supposed to be protected, allowing the attacker to attempt to escalate their privileges [32].

Others are self-inflicted, instead relying on poorly written code within the application itself. Such is the case with errors such as use after free, and buffer overflows. These are often caused by human error and are not inherent algorithmic design flaws, making them difficult to tackle.

Programming languages have been developed with different memory paradigms. Some require manual effort, such as C and Assembly. In these languages it is up to the programmer to imperatively define when to allocate and free heap memory. Attempting to use a resource before or after this interval will result in undefined behavior occurring since memory is neither clear nor protected from access outside of the scope of a variable.

Others use garbage collection, making memory reclamation automatic. Most often the method used to determine whether a variable is fit to be reclaimed comes from a tracing process where any variable no longer referenced by variables, on the stack or those marked as global variables, is deemed fit for reclamation. Another method is reference counting, where each variable has an associated counter, which is incremented for every reference created, and respectively decremented each time a reference goes out of scope. This method is relatively simple to implement naively but has several pitfalls which may cause memory leaks or poor performance. A reference-counting garbage collector may have issues with cyclic references, causing them to persist after they are no longer referenced by the rest of the program. Freeing up variables with many references may also create a recursive cascade of frees, negatively impacting performance.

New methods are also being developed. One promising method is the memory borrowing used in Rust, allowing for runtime-free reclamation of memory by marking and tracing the *lifetime* of a variable during compilation [33]. A variable is *owned* by the scope it is created in and this ownership can be transferred to other scopes either by passing the variable to a new scope via a function call or simply using it within a deeper scope, or returning the variable to a higher scope by use of a *return* statement. Passing a variable to another scope makes the variable unusable during the remaining portion of the scope as the variable may have been destroyed within an inner scope.

Further, a variable may be *borrowed* by other scopes. Borrowing creates a mutable or immutable reference. To protect against data races, a variable can have either any number of immutable references borrowed at any time or one single mutable reference. This constraint makes programs more difficult to reason about but provides stronger memory guarantees. Although the system itself does not incur any performance costs compared to manually managed memory, some optimizations are impossible within the constraints of the system. At such times the programmer may choose to either manually manage the memory or forgo the optimization in lieu of stronger guarantees.

3.3.3 Network Security

Automotive applications are by necessity locally networked, mostly using CAN [34]. This network is often considered a trusted network, however, in dialogue with the industry [22], current trends show that vehicles tend to include more third-party modules on the network than they did in the past. This trend further highlights the importance of security even on local networks within vehicles as compromised modules would otherwise be able to potentially gain further access.

External communication is under further scrutiny and vehicles are leveraging this in several ways. Infotainment systems using the same building blocks as modern TVs come to mind. As communication is performed wirelessly, data must be encrypted on the vehicle to guarantee privacy. Some vehicles are already connected to manufacturer clouds where data such as position, system status, and more is uploaded to the cloud regularly. As vehicles open up to more external services, organized crime may target these networks to obtain information, lock vehicles for ransom, or otherwise control vehicles for malicious purposes [35].

As these networks encompass a wide ecosystem of devices and networks with differing requirements, these systems must be architected with a holistic approach. Peer-to-peer protocols require different security mechanisms when compared to central communication algorithms using 5G or similar technologies.

Likewise, manufacturers have to consider that their vehicles or cloud systems will require service for many years to come. Selecting future-proof ciphers will thus be important even in the light of OTA (over-the-air) updates becoming more common, seeing as hardware support for certain cryptographic operations may be necessary.

3.3.4 Supply Chain Attacks

Supply chain attacks are when systems used to deliver services during development become compromised [36], resulting in either intrusion on development machines or poisoning of production code for the attacker's benefit. Supply chain attacks are closely related to network security in general but have unique aspects important to the adoption of new software within the automotive industry.

Libraries, development tools, and deployment platforms are the primary attack vectors. These vectors may be malicious by the intent of their authors, or due to third parties exploiting other vulnerabilities to somehow infect the library or platform. Libraries may be published to harvest user credentials, for example, a malicious entity may create a simple library for writing logs in the browser. Initial versions may be free from malware in the hope of gaining more users. When a critical mass of users is achieved, the library author devises a way to transmit valuable information such as input fields using HTTP requests. There are many ways to elaborate on such an example but it demonstrates the core principle.

It could also be possible that the platform hosting the libraries gets exploited so that an attacker may replace a build artifact with an infected version. This case is tricky to avoid but can at least be partially mitigated using hashing or code signing

to verify blobs or source code.

3.4 Hardware Platforms

In this section, we introduce some hardware platforms and emulators that show promise for future development within the automotive domain. We describe the boards' general capabilities and architecture as well as some current use cases.

3.4.1 Adafruit ESP32 Feather V2

The Adafruit ESP32 feather is a compact microcontroller board that is designed for use in a wide range of projects, especially those requiring wireless connectivity [37]. It features a dual-core Tensilica LX6 microprocessor, which is capable of running at up to 240 MHz. This provides plenty of processing power for most applications and allows the board to run complex programs with ease. The board also features 4 MB of flash memory and 520 KB of SRAM, which provide plenty of space to store code and data.

One of the key features of the Adafruit ESP32 feather is its built-in WiFi and Bluetooth connectivity. The board features a dual-band 2.4 GHz and 5 GHz WiFi module, which supports the 802.11b/g/n and 802.11ac standards. It also supports Bluetooth 4.2 and Bluetooth Low Energy (BLE), which allows it to communicate with a wide range of devices, including smartphones, tablets, and other microcontrollers. This makes it ideal for use in Internet of Things (IoT) projects, where it can be used to connect to sensors, displays, and other devices. This is also beneficial for automotive applications as they share similar connectivity requirements.

In addition to its connectivity features, the Adafruit ESP32 feather also includes a range of other useful features. These include a USB-C port for power and programming, a JST battery connector for portable projects, and a range of GPIO pins for connecting sensors, displays, and other peripherals. Overall, the Adafruit ESP32 feather is a versatile and powerful microcontroller board that is well suited to a wide range of projects, from simple sensor monitoring applications to complex IoT systems.

3.4.2 Raspberry Pi Pico W

The Raspberry Pi Pico W is a wireless variant of the Raspberry Pi Pico microcontroller board [38]. The Pico W is designed to be low-power and easy to use, making it ideal for IoT projects, wireless communication, and sensor applications.

The Pico W has a number of specifications which makes it similar to the capabilities of the ECUs used in vehicles. Among these are a dual-core ARM Cortex-M0+ processor running at up to 133 MHz, 264 KB of on-chip RAM, and support for a wide range of peripherals, including SPI, I2C, UART, and ADC. In addition, it features built-in support for Wi-Fi and Bluetooth Low Energy (BLE), allowing for easy wireless communication with other devices.

One of the most significant advantages of the Raspberry Pi Pico W is the wealth of developer support available for it. The Pico W is supported by a number of programming languages, including C, C++, Python, and MicroPython, making it easy for developers to quickly get up and running with the board. In addition, the Raspberry Pi Pico W has a large and active community of developers, providing access to a wide range of tutorials, examples, and support resources.

It also features a user-friendly bootloader, exposing itself as a flash drive where even a developer without terminal experience can simply drag and drop a program onto the filesystem to flash the board. Other boards often rely on specialized software or direct serial access to flash the programmable memory.

3.4.3 QEMU

QEMU (Quick Emulator) is an open-source software tool that enables the simulation of different architectures and platforms [39]. Platforms include, among others, the ESP32 and ARM microprocessors, which are commonly used in the development of electronic control units (ECUs) for automotive applications.

QEMU provides a range of features that make it useful for embedded development, including support for a wide range of hardware architectures and operating systems and the ability to simulate peripheral devices such as sensors and communication interfaces. Developers can use QEMU to test and debug their software in a simulated environment, which can help identify and fix issues early in the development process. Additionally, QEMU can be used to test the performance of the software in different scenarios, which can help optimize the software for specific use cases. This approach can save time and resources and can ultimately lead to more reliable and robust ECU systems.

3.5 Languages

In this section we provide background and context on promising programming languages that may have a future within automotive development.

3.5.1 C

C is often considered the de facto standard for embedded programming. This is in part due to its great execution performance, as it provides few abstractions allowing developers to program closer to hardware. Likewise, C and its companion language C++ are considered the standard for the vehicular industry [40] [41].

C makes few assumptions about the code and there are no canonical compiler or ecosystem provided. The International Organization for Standardization (ISO) publishes the standard but unlike most other programming languages, there is no governing body that publishes a reference compiler or promotes an ecosystem with tooling and systems. The ecosystem has instead grown organically by contributions from different industries and parties. The ISO C-standard does not define the expected

behavior of all syntactically valid expressions, such as reading uninitialized memory, causing unexpected behavior when compiling with different compilers, which furthermore may itself differ between executions.

As a consequence, a C-program may not be compatible with all compilers as memory representation and optimizations can be different depending on the compiler used. This is unfortunate due to many programmers relying on C to either perform or not perform a specific optimization to squeeze out every last bit of performance. Thus, an optimization on one compiler may result in a slowdown on another. Several compilers, including GCC, Clang, and MVSC, are in use today, and no compiler has gained enough use to be called the standard compiler, with each compiler having its own intended use cases and focus.

Keeping this in mind, portable code must be carefully written in order to support different hardware and compilers. Relying on features available in later versions of the ISO standard increases the risk of further incompatibilities. The wide ecosystem also leads to difficulties when attempting to improve the language, as each party maintaining the compilers must either agree on an implementation or create their own interpretation of how to solve the problem, further diverging from the common path. This process results in a very stable language, though one that is unable to continuously and efficiently integrate new syntactic and semantic constructs compared to languages with a more central guiding body.

No package manager is provided, and the most common workflow is to compile libraries beforehand and later link to them during the compilation of the target codebase. This promotes the sharing of libraries between applications and was beneficial in reducing storage usage back when applications were a considerable part of the total storage size. This practice has since lost some of its value as a result of lowered costs of external storage and bandwidth. It is also not an important factor when developing monolithic applications for microcontrollers as only one codebase will be running on the microcontroller at a given time. The lack of a package manager does however increase the burden of introducing new libraries into the codebase, especially considering the previous information on C-compatibility.

3.5.2 Rust

Rust is a relatively new language that promises modern language features such as runtime-free automatic memory management, powerful meta-programming tools, and performance close to a C implementation [42]. The Rust language supports the most common processor architectures with first-class support for x86 and ARM. It also supports most other production-ready architectures. Rust also supports compilation targets without a C standard library. We will now go further into some specific areas where Rust shows promise.

3.5.2.1 Memory Management

There are several popular options for memory management, with two common methods being manual and garbage-collected systems. When using a garbage collector,

```
{
    let s = String::from("hello"); // s is valid from
                                   // this point forward

    print(&s)                 // Function taking a read-only
                              // reference to s (borrowing)

}                             // scope ends and s is no
                              // longer valid
```

Listing 1: An example of how borrowing works in Rust.

each reference is associated with a runtime counter which keeps track of how many copies of a reference the program is currently using. As this is a run-time feature, it naturally follows that we cannot statically guarantee anything about the state of the counter at a certain point in our program during compilation.

This is in contrast to the system used by the Rust, called *Ownership* [43]. In Rust, each scope may take ownership of a variable and create references when calling other functions or creating new scopes. At the end of each scope, the reference is dropped. An example can be seen in listing 1. Since the calling function still owns the variable the compiler can assume that the variable is still necessary and keep it around. If, however, an owned variable reaches the end of the scope to which it belongs, the compiler assumes that the variable should be dropped. The references function like an R/W lock, in that a function may create as many read-only references as possible, but only one single read-write reference.

As mentioned above, this system creates restrictions when developing in Rust, which may impact performance. It is possible to create scopes that ignore the borrow checker, called *unsafe* blocks. This is normally frowned upon as memory management becomes manual but it may be necessary if performance is absolutely critical or when interacting with C libraries.

3.5.2.2 Meta-programming

Rust also offers a powerful declarative and hygienic meta-programming system. A hygienic macro-system means that the abstract syntax tree (AST) produced by the macro is verified to be valid; this is in contrast to C-like macro-systems where a macro does not have to be syntactically valid. Macros can be used to offset some of the performance loss, due to borrow-checker requirements. Macros are also very useful when defining hardware abstraction layers, as they allow code to be written generically, for example, by having a macro that changes endianness depending on the platform compiled for. An example can be seen in listing 2.

```rust
#[macro_export]
macro_rules! vec {
    // The macro takes an arbitrary number of expressions
    // as input
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x); // $x can be any
                                   // valid Rust expression
            )*
            temp_vec
        }
    };
}
```

Listing 2: An example of a macro that creates a vector. In the standard library, this macro also pre-allocates the right amount of memory instead of progressively growing the vector.

3.5.2.3 Developer Experience

An expressive type system capable of representing different data and procedures may for some feel restraining but stops certain kinds of erroneous code from executing in a production environment, ultimately saving time and increasing security [44].

Rust's type system is sophisticated and capable of representing both sum and union types in addition to a wide support for generics and experimental support for dependent types. This, combined with the extensive macro support enables developers to rapidly write complex, yet type-safe applications. However, this also results in a steeper learning curve.

Though some consider Rust to be difficult to learn, the Rust compiler is known for providing detailed error messages that help developers quickly identify and fix issues in their code. The error messages often include suggestions on how to resolve the issue, making it easier for developers to understand and learn from their mistakes.

3.5.3 MicroPython

MicroPython is a specialized version of the Python programming language designed explicitly for embedded systems [45]. It offers a lightweight implementation of Python 3, enabling it to run efficiently on microcontrollers with limited resources, making it a popular choice for a wide range of embedded development projects.

One of the key strengths of MicroPython is its simplicity and ease of use, making it accessible to developers who do not possess experience with embedded systems development. Furthermore, MicroPython provides a familiar programming environment for developers already well-versed in the Python language, facilitating smooth

transitions between desktop and embedded development projects.

Portability is another notable strength of MicroPython. As it is written in Python, the language can run on a diverse array of hardware platforms, including ARM, AVR, and ESP32. This versatility makes it an attractive option for developers who work with various hardware configurations.

The compact size of MicroPython further adds to its strengths, as it can be effectively utilized on devices with extremely limited resources. The language is specifically designed to cater to microcontrollers with as little as 256 KB of flash memory and 16 KB of RAM, significantly less than what a full version of Python would require.

However, MicroPython does have certain weaknesses that need to be considered. One such drawback is the limited number of available libraries. Due to its lightweight implementation, MicroPython lacks access to the full range of Python libraries. While some libraries are designed specifically for MicroPython use, developers may need to create their own libraries or find workarounds for certain tasks.

Another weakness of MicroPython lies in its performance. As an interpreted language utilizing garbage-collection, it falls short in comparison to manually managed and compiled languages such as C or C++. This could pose challenges in applications where speed and performance are critical factors.

3.5.4 WebAssembly

WebAssembly is a novel platform that was primarily intended to enhance or replace the current ECMAScript (commonly referred to as JavaScript) browser runtimes with a general-purpose bytecode target [46]. This new runtime can be enhanced without being tied to ECMAScript development or foundations. While the initial target was browsers, new projects are now attempting to extend WebAssembly's capabilities to other domains, such as embedded and systems programming.

The inception of this project can be traced back to previous efforts in producing alternative web runtimes. One of the earliest examples was ActiveX [47], a Microsoft product that allowed signed binaries to run in a browser context. Similarly, Flash [48] and Java Applets enabled users to program web pages using languages other than ECMAScript. Although these runtimes found use during their release they were eventually replaced by new and standardized DOM APIs due to safety and performance concerns.

While the web protocols themselves are hardware-agnostic, historically, browsers were predominantly limited to the x86 architecture, with only minimal implementations running on other platforms. However, the web has undergone significant transformations since then, and modern browsers are now required to provide support for diverse architectures across various device types, including desktops, media systems, and smartphones. To ensure efficient execution on these diverse platforms, WebAssembly has been designed to accommodate heterogeneous hardware. This capability to support varied computer architectures makes WebAssembly intriguing as another attempt to establish a lingua franca within the computer science domain,

akin to languages like Java or .NET.

3.5.5 Zig

Zig is a minimalistic language that aims to replace C [49] while preserving a stable C-compatible binary interface and incorporating additional functionalities. Zig does not guarantee total memory safety and requires manual memory management. It does however address common flaws with the C-model of memory management, primarily by reducing language constructs that may create undefined behavior. Additionally, Zig promises superior performance out-of-the-box by automatically leveraging advanced processor features, including SIMD, as well as applying code optimization steps that are often performed manually in C-like languages. Furthermore, Zig is statically compiled, aiming to avoid any particular standard library whenever possible for enhanced compatibility with embedded systems.

To mitigate occurrences of undefined behavior, Zig embraces various modern language features. One notable feature is the adoption of monadic result and error types with built-in language support, effectively replacing null values. This approach translates null-pointer exceptions into compile-time errors, contributing to improved code robustness. Moreover, Zig prioritizes simplicity and comprehensibility by avoiding hidden control flows. Consequently, accessors, operator overloading, and exceptions are considered anti-features and deliberately omitted. While Zig foregoes complex meta-programming due to similar reasons, there is provide support for compile-time function evaluation.

3.6 Other Development Technologies

In addition to the previously explained technologies, this thesis also relies on other practical applications not directly related to one specific implementation environment.

Docker is a powerful tool for creating containerized applications that can run anywhere, providing a lightweight and secure environment for running applications [50]. With Docker, developers can specify a root filesystem, usually derived from the Ubuntu or Alpine Linux distributions, allowing them to execute programs and even develop within the virtual environment. This approach is similar to virtual machines, but it relies on kernel features to provide isolation between the host and client, resulting in a more lightweight execution context.

Using Docker images has become increasingly popular for developing and deploying applications in various environments. Images can be exported and shared among other developers, making it easier to set up new team members and reducing the setup time for complex projects. This is especially useful for large codebases requiring complex dependencies, where setup and configuration can often be time-consuming and error-prone. With Docker, developers can easily create and share images containing all the necessary dependencies and configuration settings, making it easy for other team members to get started quickly. It also makes it easier

to manage and test multiple applications on a single host as each application is sandboxed.

4

Method

In this chapter, we present the overall methodology of the thesis and how it relates to the purpose of the thesis. We also describe some of the preliminary decisions required before beginning the actual implementation.

4.1 Overview

If the industry is to adopt new technologies, they must be assured that the proposed technologies can meet their demands, while providing meaningful benefits. The first step is to identify technologies that hold the potential to enhance security within the automotive domain. These technologies must then be tested in settings close to real-world applications. One such critical component is the inclusion of modern programming languages and their ecosystems.

New languages may offer security techniques unavailable to current development practices. As many vehicular applications are currently written in C, they face issues with manual memory management, difficult integration of libraries, relatively low type safety, and difficult compilation steps. However, a significant benefit is that the code is about as performant as modern architectures can make it.

Thus, the languages we test should offer something novel to the industry in terms of security. These may come in the form of language features such as information control or memory management. They may also originate from the ecosystem. Build systems and package managers have great influence over the authenticity of imported code, while static analysis tools provide insight before the code is tested.

Before beginning we must decide on the variables that will help us compare each language. The first variable is of course the languages that we wish to investigate. They must provide interesting benefits that would enhance the security of automotive platforms. Further, we must ensure that the languages work on multiple platforms common to the automotive industry. Languages that do not work on the platforms used within the industry have a higher resistance towards adoption as much energy must be invested for the languages to run, never mind the libraries on which the application may depend. Finally, we must decide on one or several algorithms to implement. They must be relevant to our domain and contain security challenges important to current and future vehicular applications.

4.2 The Framework

To assess the different programming languages when developing secure network applications for vehicular systems, we propose a framework mirroring the development cycle of applications. The framework is visualized in fig. 4.1 and further explained below.

The evaluation process for each language, platform, and algorithm tuple is divided into three steps. The first is to research the tools and ecosystems available for the language and hardware in question. This is followed by an examination of the capabilities of the language itself by implementing the algorithms selected. This will require self-reporting and input from a reference group which we describe in the following section. Finally, we test the implementation to show how the different parameters affect performance and resources.

To test portability between architectures and to give more faithful test data, the algorithm will be tested on both virtualized hardware running on consumer hardware and commercial microcontroller boards. This is important since developers will not be expected to develop on actual hardware during the entire development process.

Qualitative assessments will be graded on a three-point scale, where we assess difficulty and/or accessibility in relation to a reference individual. The reference individuals will be interviewed on their experiences and will be asked questions about a few code snippets in the different languages assessed. We will also present our findings through our discussion, highlighting issues and solutions present within each implementation and the evaluation as a whole.

4.3 Selection Phase

In the first phase, we select our variables which will be varied during the planning, implementation, and testing phase and finally evaluated in the final phase. These may be different depending on the domain but a general set can be found in fig. 4.1.

For example, programming languages give a good start because of their great variance of intended use cases. Some trade development speed for security or performance. Others provide strong runtime guarantees by providing a strong type system. If testing embedded systems it might be necessary to limit the languages based on available hardware but it could also be interesting to evaluate the languages themselves based on domain requirements rather than existing technical restrictions.

Libraries and tools are often specific to programming languages which makes it difficult to use them as base variables. Theoretically, we could select a static set of variables and instead change the variables that arise during the planning stage. This is however out of the scope of this thesis.

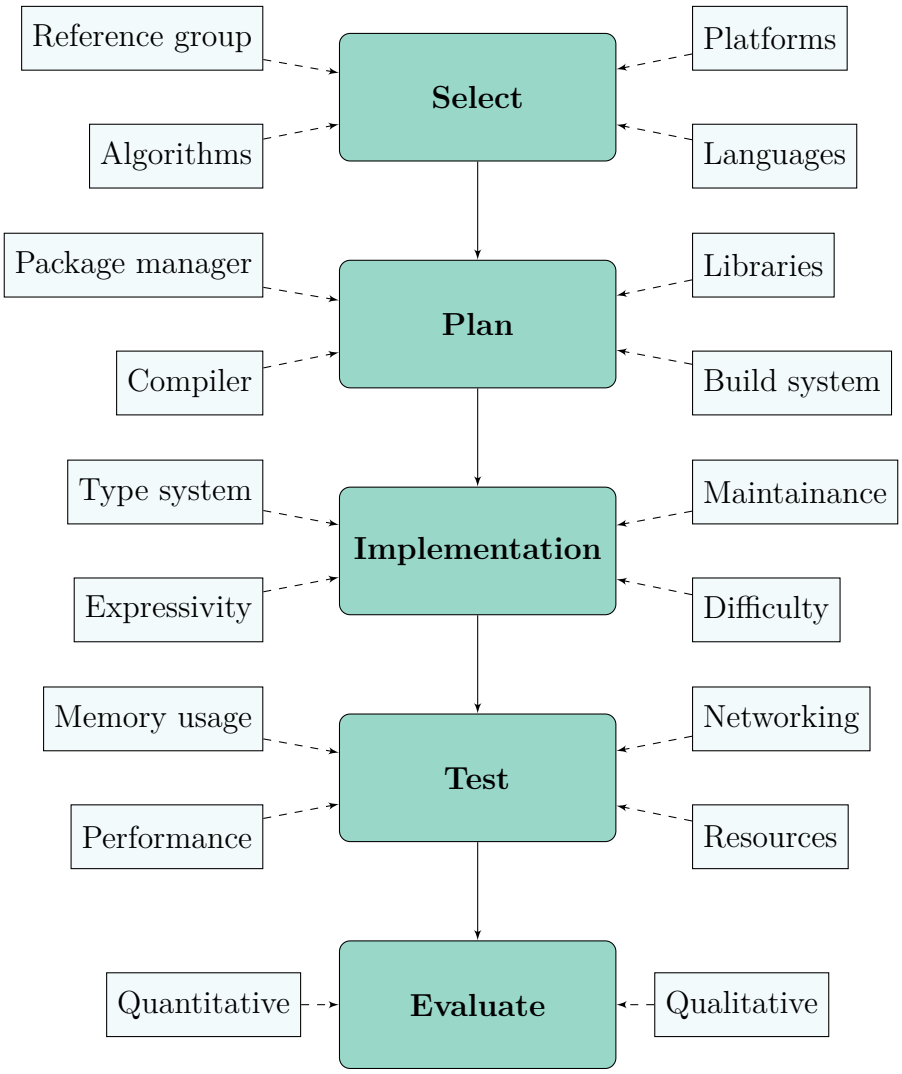


Figure 4.1: The evaluation process used during the thesis.

4.4 Planning Phase

When planning the implementation of an algorithm, several aspects must be taken into account. First, we have to select the toolchain to be used for our project. This includes the compiler, linter, package manager, and other development technologies to use. For example, we could decide to use Docker to set up a complex build chain for a specific development board.

We also have to select which parts of the algorithm can be levied by the available libraries. This will become a tradeoff between learning and integrating the library in contrast to writing the functionality oneself. For cryptography, this almost always equates to finding a vetted library, as it is extremely difficult to implement modern cryptography in a provably correct manner while also being performant. However, it may not be worth it to preemptively exhaust all available libraries for small tasks, even if they are more performant than a hand-made solution.

Finding a library is after all a qualitative process in and of itself. Efforts must be made to discover, compare, and integrate each library. If we can see that a library is unfit at the discovery stage we save much time as the steps become increasingly expensive to perform. Different languages will also have different integration costs. Some languages have ecosystems and type systems that will guarantee whether a library is compatible with the current project configuration; others do not, and it is up to the developer to ensure that the final binary is valid.

In addition, we must also evaluate any package manager used as they can introduce supply chain attacks. Many package managers allow arbitrary code to be executed during the build or fetch stage of a library and this risk must be evaluated in a high-risk setting such as automotive development.

4.5 Implementation Phase

The implementation phase is where the application code will be written and the process evaluated. During the implementation phase, we consider language aspects such as correctness and support for complex expressions. For example, we consider how the language protects the developer from performing illegal actions, such as writing to unallocated memory.

We will use our model developers when analyzing qualitative aspects of implementation, such as learning curves and familiarity with tooling. Another important aspect is how much knowledge there is available. Obscure tools and programming languages may incur significant overhead if they are used in niche environments where developers cannot rely on information available online or the experience of colleagues.

4.6 Testing Phase

During and after execution, we will measure several quantitative measurements. We will select these to verify if the languages, hardware, and algorithms chosen are suitable for the real-time conditions present in automotive environments. We do not anticipate either hardware or algorithm to be used in their presented forms but that they will be similar in key criteria that make the knowledge transferable when developing the algorithms that will be running in our future vehicles. Because of this, we won't be measuring specifics about the algorithm or hardware as they are subject to change.

4.6.1 Test 1A. Cryptography

Cryptographic operations are often expensive, especially so on embedded hardware in real-time situations. Because of this, the first test will be to measure the execution speed of key generation. Generating keys is a slow process even on full-powered machines so we expect this parameter to change drastically depending on the efficiency of processor instructions. The ability to generate keys in close to real-time

will determine the viability of connecting to hundreds of nodes per minute, a potential requirement for mesh networks. If keys are unable to be generated there is a significant risk that developers will opt to use less secure mechanisms.

The test will be executed 100 times and will show the minimum, average, and maximum time to completion for each language, hardware, and algorithm tuple.

4.6.2 Test 1B. Cryptography

The second test is to measure the encryption speed of one payload during key exchange. This provides similar data to the first test but is a separate issue. Even if key generation is not real-time, keys could be provided through other means than direct generation within the algorithm itself. However, if encryption speed is not sufficient then even this will not be enough.

The test will be executed 1000 times and will show the minimum, average, and maximum time to completion for each language, hardware, and algorithm tuple.

We assume that the algorithms will contain some sort of symmetric encryption, and we will not be testing the encryption performance of any such algorithms as they can already be deemed fast enough even with low-performing hardware [51]. We will however note whether there is hardware support for encryption as that will be key when transmitting high-bandwidth media.

4.6.3 Test 2. Network

Any networked algorithm will require some sort of setup before communication can be considered private. This test will measure when a channel can be regarded as such. If the handshake is too slow then it will be unreasonable to expect moving vehicles to establish a connection before moving too far away.

The test will be executed 100 times and will show the average time to completion for each language, hardware, and algorithm tuple.

We will also measure Round-trip-time (RTT) to understand how many packets we can expect the hardware to handle per unit of time. We will perform this test by measuring the average RTT for a conversation of 1000 messages.

The networked tests will not be able to capture the jitter expected during normal operations, and will likewise have some jitter of its own due to surrounding electromagnetic interference. Removing this interference is beyond the scope of this thesis but we will still consider it when evaluating our results.

4.6.4 Test 3. Application Lifetime Assessment

This test will test the whole algorithm by running it through a simulation of expected parameters. In essence, this is a combination of the setup of both the algorithm and the network channel and also includes several messages being transmitted and reacted to.

The purpose of this test is to determine if there is any poorly distributed load balance during execution and whether the memory requirements of the algorithms are reasonable for commercial hardware.

4.7 Deciding the Variables

Before beginning the implementation of the framework we must decide on which languages, platforms, and algorithms to use. We begin by explaining the requirements in further detail, followed by what choices we made based on these requirements.

4.7.1 Programming Language Requirements

Programming languages form the basis of which programs can be written and how the programmer will be protected from making mistakes, either from a security or performance perspective. It is therefore imperative to evaluate different languages for their strength and weaknesses, in order to assess which tools to choose when designing the next generation of applications. Using older more proven languages may provide a stronger certainty over that of a modern language which may have fewer complex applications from which to draw experience. On the other hand, some features may prove vital to protect developers from making fatal mistakes present in the older generation of languages.

Languages will be primarily chosen for their suitability for development on automotive platforms. This gives us two hard requirements. First, they have to compile into machine code for the specific instruction sets or provide lean interpreters. Since interpreters are rare in embedded environments, this will most likely not significantly affect the search. The program may also not rely on any major C standard library as these are rarely available when targeting bare metal and reduce the portability.

We also want our language to be different in some aspects. High-level languages allow developers to be more efficient when writing code and with microcontrollers gaining more horsepower they may be worth exploring even though they might have worse performance. Some examples of higher-level features that are not common in C but often praised in other languages include: garbage-collection and variable immutability. A richer type system will also allow programmers to set better rules for their programs. Today, languages such as Haskell, Rust, and TypeScript allow programmers to define sum and product types, define constrained generic types, and give advanced type inference hints allowing programmers to be terser in their programming.

The reference will be implemented in C, as that is the de facto standard for automobile development as outlined in the background. This also allows us to evaluate the process of porting C code to other languages. The implementation process will be documented, and the final executable tested using various metrics.

These features come at a cost of learning, performance, or maintenance which we would like to explore during our evaluation through our reference group and implementation experience.

Rank	Language	Loved	Dreaded
1	Rust	86.73	13.27
2	Elixir	75.46	24.54
3	Clojure	75.23	24.77
4	Typescript	73.46	26.54
5	Julia	72.51	27.49
6	Python	67.34	32.66
7	Delphi	65.51	34.49
8	Go	64.58	35.42
9	SQL	64.25	35.75
10	C#	63.39	36.61
...	...	...	...
25	C++	48.39	51.61
...	...	...	...
33	C	39.68	60.32

Figure 4.2: Top 10 most desired programming languages from the StackOverflow Developer Survey 2023. With C and C++ added. Used with permission [53].

4.7.2 Programming Language Choices

When searching for target languages we wanted to find languages with unique features and ecosystems. To have something to compare to we first chose C as our reference due to its prevalence within the industry [52]. StackOverflow conducts yearly surveys in which developers vote on their preferred or disliked development technologies [53]. Among these are questions about programming languages and their desirability by professional developers. We assumed that languages with high interest promised new features that programmers wished to experience and then went more in-depth with each language, trying to find what makes them unique. The highest results of this survey can be found in fig. 4.2.

Among the top-rated languages, most are not suitable for embedded development. In the top ten, we only found a few languages that, according to the owners of the respective programming language projects, support bare-metal deployment for the most common architectures. The first was Rust, rated number 1 by a significant margin. Additionally, we also note that C++ and C score very low in this ranking which we will keep in mind when investigating the ergonomics of the language.

We also decided to include Python in the form of MicroPython as the language is very similar and provides a different take on embedded development. We do not expect this implementation to mirror the others in performance or platform support, instead, it will serve as the opposite side of the simplicity spectrum.

Finally, unrelated to this previous list we also wished to look into WebAssembly as a future intermediate for bare-metal deployment. WebAssembly is not strictly a language as much as an instruction set but we nonetheless thought it fitting to evaluate.

4.7.2.1 C

C was chosen to provide a reference implementation. C is the primary language used by many vehicular frameworks such as AUTOSAR [52]. Due to being so close to hardware and battle-tested, we can assume that such an implementation will be close to the performance maxima.

4.7.2.2 Rust

Rust gave a very positive first impression, promising performance rivaling C while also providing modern language features. We searched for real-world uses of the language and found that it is currently used in the Linux kernel for certain drivers [54]. Additionally, Amazon uses Rust as the main language in their Firecracker VM [55]. Firecracker is used in the AWS Lambda service, proving that it is possible to write reliable and performant applications in Rust [56].

Further, Ferrous Systems has developed several open-source projects for embedded systems in Rust [57]. In addition to these projects, they have recently released a Rust toolchain with ISO 26262 certification, a standard for developing safe automotive applications. The Rust project also references several companies that have had success developing embedded applications in Rust [42].

4.7.2.3 MicroPython

When researching MicroPython, we found that it offers the high-level syntax and feel of the Python language, while still optimizing for embedded development. It has first-class support for the ESP32 and Raspberry Pi Pico W along with several other architectures, allowing it to be used on a variety of microprocessors commonly found within the ECU ecosystem.

The Python syntax will allow programmers from outside the embedded development sphere to comfortably program in a familiar environment. If the manufacturer uses Python in other products, developers may also share libraries and components among its services. In the StackOverflow census, we can also see that professional developers have more experience in Python, in contrast to programming languages commonly referred to as systems programming languages such as C or Rust. This would imply that MicroPython might enable more developers to become productive within the embedded domain.

However, it is uncertain whether MicroPython will be able to deliver performance close to other common languages targeting embedded platforms. Features are rarely free of runtime costs.

4.7.2.4 WebAssembly

The primary focus of this thesis is the implementation of an embedded application within the automotive domain using the programming languages decided on earlier. However, we have also opted to explore the potential of WebAssembly as an alternative approach. WebAssembly serves as a unified binary compilation target for

multiple platforms, akin to the Java Virtual Machine or the LLVM intermediate representation format. This offers the advantage of accommodating various programming languages within a single project, allowing the same codebase to operate on diverse platforms, even if they do not inherently support it as a native binary target.

We can from our preliminary research already tell that the technology is not yet ready for production use. We will not be able to fully implement our system for our platforms. Nevertheless, WebAssembly shows promise in its capacity and thus we wish to further research the possibilities in the near future. To that end, we will rely on smaller samples and focus more on the theoretical aspects of WebAssembly. The main objective will similarly be to evaluate the security, performance, and development aspects of WebAssembly.

4.7.3 Platform Requirements

The automotive industry relies on a wide spectrum of system architectures. Common platforms include general architectures such as ARM, ESP, AVR, and MCUs targeting specifically the automotive industry such as Infineon, Power, and Renesas RH850 microcontrollers [58]. We expect that future vehicle-to-vehicle communication will be communicated through a dedicated gateway node, either retrofitted from existing ECUs with a similar purpose or a new ECU sharing many characteristics with those currently available and in use.

The assumptions about which platforms are used within the industry provide the basis for our choice of platforms. It was not possible to gain access to industrial ECUs for this thesis. Fortunately, they do not differ greatly in function compared to the boards available for hobbyists and IoT development, and can therefore be used as a substitute. Industrial projects attempting to implement similar evaluations could improve upon this work by using hardware expected to be used by their vehicles.

4.7.4 Platform Choices

Following our framework, the platforms had to be similar to those currently used in ECUs. The physical platforms were selected to provide as informative test data as possible, as well as convey some of the development experience of running code on hardware. Meanwhile, the simulated platform was chosen to allow rapid and simplified development. Flashing a program to hardware is not always convenient and execution on hardware increases the number of failure points, as they are generally not as easy to inspect. It also requires constant access to hardware, which is not ideal. On the other hand software emulation is not always representative of the speed or accuracy of the final result.

The first physical platform we chose was the Adafruit Feather. We selected it due to its ESP32 Xtensa LX6 processor and wireless availability. It features two cores and runs at 240MHz which is significant horsepower for an embedded board. Furthermore, it has a well-supported SDK [59] with integrated cryptographic libraries and hardware translation layers. The ESP32 also has cryptographic hardware for

	Feather	Pico
Architecture	ESP32 Xtensa LX6	ARM Cortex-M0+
Clock speed	2 cores @ 240 MHz	2 cores @ 133 MHz
Memory	520 KB	264kB
Flash	8 MB	2 MB
Hardware-accelerated cryptography	Yes	No

Table 4.1: Comparison of the two hardware platforms

certain operations, which allows additional benchmarks to be performed. One downside of this board is the lack of support for ad-hoc wireless communication. This necessitates additional wireless routing which will impact our tests.

The second platform we decided to use was the Raspberry Pi Pico W. This is a board featuring a dual ARM Cortex-M0+ processor. It is similar to the ESP32 but is clocked at half the frequency (133MHz) and also features a similar wireless connectivity chip without ad-hoc support. ARM is commonly used within the automotive industry and can even be seen in consumer hardware such as most smartphones [60] or the Mac M1 chip [61].

We choose to emulate both chips using QEMU. Primarily, we want to emulate the selected hardware platforms. This is done to ensure that the simulated development is as close to the hardware development process as possible to ensure that the code that runs in the simulator also works in hardware. QEMU is chosen due to being open-source, available on many host platforms, and providing full system emulation, instead of just emulating the execution of a single program.

We used Docker as a development environment to further simplify the simulation environment. This allowed us to set up a Linux environment on all hosts without modifying the host filesystem when trying new toolchains, regardless of the host OS used.

4.7.5 Algorithm Requirements

When selecting algorithms, we must find an algorithm that covers many aspects of vehicle-to-vehicle networking. Several topologies are suggested in the literature, ranging from ad-hoc mesh networks to centralized cloud communication. These are different paradigms with different requirements. If traffic is centralized we increase latency; on the other hand, we may use much of the existing infrastructure and knowledge as the scenario is similar to contemporary IT. In contrast, mesh networks are more akin to IoT applications and as they are decentralized, require more complex security protocols.

We also want our algorithms to require both symmetric and public-key cryptography in order to test the capabilities of our platforms. Because cryptographic operations

are computationally intensive this provides us with an opportunity to both test pure performance in addition and to investigate whether there is benefit in offloading certain computations to hardware.

Finally, we wish our algorithm to be implementable using existing cryptographic libraries, as this is not an exercise in the implementation of cryptographic primitives. The implementation of cryptography is error-prone and requires extensive scrutiny, analyzing the cryptographic primitives would most likely shadow most of the other aspects of the thesis.

4.7.6 Algorithm Choices

Our initial assumptions require us to first decide whether we are focusing on meshed networks or centralized networks. As we have previously discussed, there are already numerous studies on the usage of centralized communication algorithms and thus we see a greater research value in basing our implementation on a mesh algorithm. A mesh algorithm will also allow us to reason more about security implications and trade-offs. In addition, we will not be able to rely on available frameworks, allowing us to further explore each language and the available tools.

However, our investigation of centralized algorithms provided us with the insight that developers would have several advantages when developing within this paradigm. Developers would be able to use existing tools, such as the TLS ecosystem and the certificate authorities currently in use by the global TLS root system. They would also be able to analyze and filter data on powerful server nodes, although it is still debated how efficient this would be, as the number of messages that circulate would be enormous [62].

Finally, we decided to only use one algorithm for our implementation as we considered this variable to be less interesting, given that we could find an algorithm that was comprehensive enough to fit our scope. Another implementation would also result in a significant amount of work, for little benefit, which would have impacted the timetable of the thesis.

4.7.6.1 Algorithm Candidates

We found several algorithms such as SUPERMAN [24], OLSR [25], IPsec [27] and SAODV [26], further described in chapter 3. They each provide different guarantees, some focusing on end-to-end security, while others focus on securing the route itself.

In the end, we settled on the SUPERMAN algorithm due to the usage of certificates, public key cryptography, and symmetric encryption which would allow us to fully utilize and investigate cryptographic library availability.

Despite the strengths of this algorithm, there is one particular step that does not contribute positively to this thesis. The primary issue is the reliance on certificate authorities. Although this is a clever way of ensuring the authenticity of each node, it requires the setup of a central trusted certificate authority. Another requirement is a circulation mechanism for root certificates, as well as the granting of new certificates.

In production, we believe that vehicles will be assigned a personal and root certificate when deployed and that the company either invests in a personal certificate authority or leverages existing certificate chains.

For this reason, we had to implement an alternative strategy that does not compromise the development aspects of the algorithm. We chose to rely on statically assigned self-signed certificates to test the program flow. This does not significantly change the codebase but leaves out certain development operations that are critical in a production environment. Despite this, we are still able to reason about the organizational requirements in our discussion, and our alternative may nevertheless prove valuable in development scenarios.

4.7.7 Cryptographic Choices

The protocol does not dictate which specific algorithms to use for key-sharing or encryption so the choice of these falls on the developer. More specifically, we had to decide on a key exchange algorithm and a block-cipher algorithm for communication.

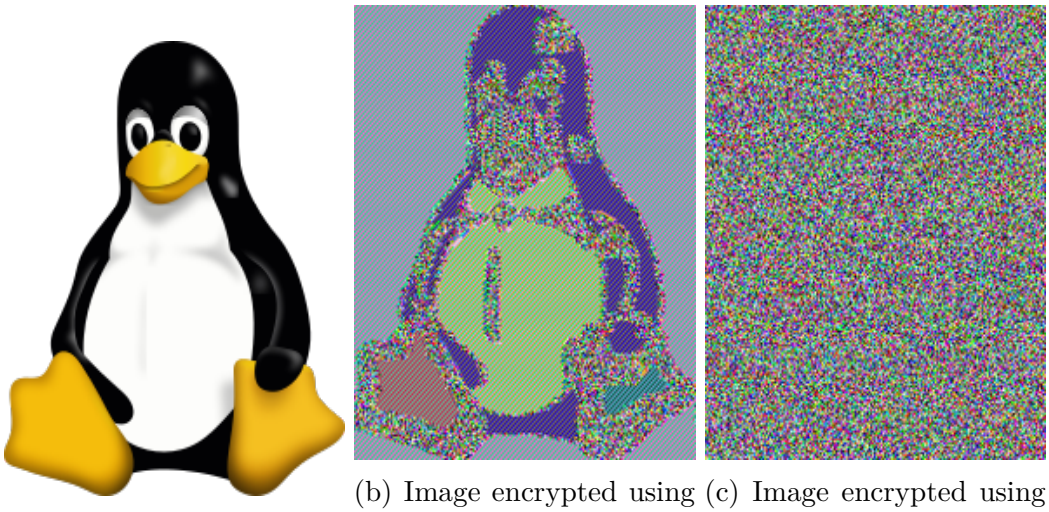
While abstract standards exist for the development of automotive platforms, no standards for specific cryptography algorithms exist [63]. However, we can draw inspiration from other fields where cybersecurity is critical. The American National Institute of Standards and Technology (NIST) provides several algorithms suitable for our protocol [64].

4.7.7.1 Key Exchange

NIST approves of the usage of either DSA, ECDSA, or RSA for key exchange. However, a draft for the upcoming recommendations has been published indicating that DSA is no longer considered suitable for contemporary applications [64]. ECDSA is relatively new compared to the other algorithms and many early libraries were shown to be insecure under certain circumstances [65] [66]. While considered difficult to implement, the benefits are that it requires smaller key sizes to provide equivalent security to RSA and that it is more performant than RSA. On the other hand, RSA has been around for a long time and many battle-tested implementations exist.

Additionally, concerns are being raised that common key exchange algorithms may be broken using quantum computers. While the time frame, or feasibility, of such attacks is currently unknown they could become a significant threat in the future. There are currently several algorithms attempting to solve the problem of quantum-secure key exchange. Many promising algorithms use a mathematical concept known as lattices [67]. NIST has recently selected several lattice-based algorithms for their *Post-Quantum Cryptography Standardization* project, with CRYSTALS-Kyber [68] being selected by the NIST as the recommended algorithm for general purpose encryption [69].

Post-quantum algorithms do however come with significant drawbacks [70]. The most significant are slower execution speeds, greater data overhead, and larger key sizes. Developers can expect at least three times longer execution speeds compared to ECDSA and between one and two magnitudes greater data overhead, this is



(a) Original image

(b) Image encrypted using ECB-mode

(c) Image encrypted using GCM mode

Figure 4.3: Using the incorrect cipher mode may successfully hide individual data points but reveal patterns in the data, images used with permission [71].

significant even for servers but even more so for embedded devices. The algorithms are also not yet formally verified to the same level as common algorithms such as RSA and ECDSA making their use riskier as the algorithm has a greater potential to be broken. Due to this, we decided against testing post-quantum cryptographic key exchange algorithms.

Choosing the certain over the uncertain, this thesis will use the RSA algorithm, but we would still recommend that the industry explore the usage of ECDSA as it remains a promising candidate, given a correct implementation. Similarly, NIST recommends a key size of 2048 bits or greater. As these keys will be both ephemeral we do not feel the need to use 4096-bit or larger keys particularly since they may result in performance degradation on the order of magnitudes. Moreover, they should be considered safe for at least another decade.

4.7.7.2 Symmetric encryption

For block-ciphers, NIST approves the usage of both Triple DES and AES. Although still approved, NIST has decided to deprecate the usage of Triple DES. The OpenSSL project considers Triple DES to be a weak cipher and no longer includes it by default within its cryptographic library [72]. Meanwhile, AES is still recommended and widely used. Many CPUs provide hardware acceleration for AES and those that do not are still capable of real-time encryption even on microcontrollers.

When using AES it is also important to select a suitable mode of operation. Modes offer different benefits, such as improved confidentiality or authenticity, by defining a process of how to apply the cipher to data. The modes may also help in hiding patterns within the data. A potent example of this is image encryption, where each pixel is encrypted as one block. The ECB mode would then leak the pattern of the pixels, producing an image in which the color is encrypted, but the general shape of

the image is not as in the example of fig. 4.3.

We chose the Galois/Counter Mode (GCM) since SUPERMAN requires a mode that provides *authenticated encryption with associated data* (AEAD). This is important because AEAD provides both privacy and tamper-protection, removing the need for an external signature algorithm.

4.7.7.3 Library requirements

After selecting the algorithms we can also decide on library requirements. For all categories we only consider support to be full if they do not require calling into other languages. Sub-modules compiled from other languages using a native interface are still valid for full support status. For cryptography, we consider support to be full if the libraries satisfy the following:

1. Supports 2048-bit RSA.
2. Supports 256-bit AES.
3. Supports AES-GCM mode (other AEAD modes were not considered).
4. Hardware support for AES instructions.

For serialization, we consider the following:

1. Does not require manual function definitions, i.e. serialization can be generalized over different types.
2. Must support common formats such as JSON and efficient binary encodings.

There is no universal standard for what encompasses the complete availability of utility libraries. We are only concerned with those required by this project, which are the following:

1. Hashmaps
2. TCP/IP abstraction
3. Buffers
4. Process management

4.7.8 Reference Group

The reference group is important to obtain varied input from individuals with different knowledge and experience. Since we are targeting the automotive domain, our members of the reference group must be selected so that they can answer questions relevant to the domain of the thesis. Thus, we decided on the following.

The first individual is a newly graduated computer science student with a master's degree. This individual was selected to represent someone who has no previous work experience within the domain but may be a candidate for new recruitment. This candidate will also not be colored by previous experience or industry dogma.

Hardware	**Languages**	**Algorithms**	**References**
ESP32	C	SUPERMAN	New CS graduate
ARM Cortex M0	Rust		Experienced programmer
QEMU emulation	MicroPython		Industry professional

Table 4.2: Final choice of variables.

The second individual is an experienced programmer who has none or only peripheral experience with the industry. This individual will have enough experience with large collaborative codebases to provide further insights but may not know how development within the automotive industry works.

The final individual is an industry professional. This person will provide valuable insight into the inner workings of why certain decisions were made historically and what current challenges may be.

4.8 Summary of decisions

In this section, we provide a summary of the various variables that have been introduced as seen in table 4.2. We also opted to use ECDSA and AES-GCM for encryption. The quantitative tests will focus on cryptographic and network performance while the qualitative will focus on language productivity and security aspects.

5

Implementation

In this chapter, we go in-depth about the planning and implementation process for each set of parameters. Results from tests and evaluations can be found in the following chapter.

5.1 C implementation

The C implementation was our first implementation and serves as a baseline when comparing our later results. We will be relying on the platform SDKs in both the Rust and MicroPython implementations so we will not repeat ourselves in the following sections.

5.1.1 Planning Phase

The planning stage was the most extensive of the implementations since this was to be the reference implementation and also served to introduce us to the specific hardware requirements.

5.1.1.1 Compiler Toolchain

As stated previously, there are several toolchains available for C development. In our early search, we could remove many of the candidates due to a lack of support for our target platforms. In particular, only GCC and Clang provided promising support for both the ARM and Xtensa instruction sets. The maintainers of the ESP32 SDK strongly recommended GCC as the Clang support is from a third party. Likewise, the Raspberry Pi Foundation recommends the ARM GCC Compiler for C projects targeting the Pico. Third-party support is not necessarily a strong weakness, but given that the GCC is officially endorsed on both platforms, it became the obvious option.

It should be noted that when targeting automotive platforms, the industry is required to meet several certification standards. Among these is using a compiler that meets several ISO standards such as ISO26262. These compilers are often based on or compatible with GCC and Clang. In the case of ARM, their certified compiler is built upon Clang but promises GCC compatibility and supports GNU Assembly syntax.

Both of these toolchains require considerable setup and are mostly targeted toward Linux environments. Fortunately, we can rely on Docker to create sandbox environments capable of running on all common operating systems.

5.1.1.2 Platform SDK

For the ESP32 we relied on the ESP Integrated Design Framework (ESP-IDF), a C API that provides a hardware abstraction layer for all of the different hardware components, such as wireless networking and task management. The SDK comes with a real-time OS, built on FreeRTOS, which allowed us to easily leverage the dual-core CPU out of the box. In addition, this SDK bundles the MbedTLS library, which provides access to cryptographic APIs, some of which are optimized to use hardware acceleration.

The ARM SDK was provided by the ARM Corporation and worked similarly to the ESP-IDF. Unlike the ESP-IDF no real-time OS was provided within the SDK and we had to rely on manual scheduling of the second core. Fortunately, the API was better documented, and libraries better integrated. The manual scheduling would also not be an issue since most of the work was single-threaded with the exception of the RSA key generation algorithm.

5.1.2 Libraries

In terms of libraries, we chose to be restrictive in their usage, as each library required manual intervention to ensure compatibility with the project. GCC provides little information as to whether linking is successful, which results in one significant problem, run-time errors. As long as the header files are available, GCC will happily compile our code, expecting the linker to later include the actual code. As much metadata has been stripped from the compiled library at this point, the linker cannot always guarantee that the linked binary is compatible with the final executable.

Another issue is that when programming in C, we are expected to change the source code itself to change the compilation variables, which may set the target platform, etc. Libraries may do this in many different ways, requiring the developer to explore the code or documentation in order to understand how to properly compile the library. This increases the overhead of evaluating and including new libraries.

As can be seen, significant energy has to be spent in order to debug library compatibility. In some cases, issues may not become obvious before the library has already been integrated into the project, perhaps requiring a restructuring of the codebase.

After evaluating cryptographic libraries, we decided that MbedTLS was our final choice. There are abundant implementations of AES and RSA available; some attempt to maximize performance, while others attempt to keep a low memory footprint. Others try to maintain wide compatibility by supporting both little-endian and big-endian targets by sacrificing performance. We tested both tinyAES [73] and LibTomCrypt [74] together with MbedTLS [75]. After integrating tinyAES we realized the value of having one large integrated cryptographic library as we would not have to verify each component. LibTomCrypt and MbedTLS both included all

necessary cryptographic components, however seeing that the SDK already bundled MbedTLS with optimizations, it became the clear candidate. LibTomCrypt provided a greater choice of algorithms to pick from, but also became more complex to build as features were selected and removed.

We decided against using a serialization library as there were no libraries that provided type safety without code generation. Protobuffers might have been an interesting path to investigate as they are supported by a wide range of languages and platforms. However code-generation tools often require more complex setups than integrated libraries and would have increased the scope unnecessarily. Because of this we manually implemented the serialization of this implementation.

5.1.3 Implementation Process

The implementation required much time to focus on reasoning about reasonable tests and due to the lack of many modern ergonomic language expressions in C it was difficult to write code without resorting to hidden control flows and global state. For our proof of concept implementation, we did not have to worry about this, but if this were a codebase containing hundreds of submodules contributed by hundreds of developers it would have been messy to document everything which risks unintended side-effects.

During development, we used simple integer counters as the timestamp for each packet. Although this seemed innocuous at first, we quickly ran into undefined behavior. Suddenly messages were rejected while looking valid. The source of this bug turned out to be an overflow within the counter code, where we had used signed integers of value -1 before the conversation with the connected node began. This resulted in timestamps becoming negative numbers hence resulting in ordering guarantees no longer holding.

Our solution made us change the encoding to two separate fields. One boolean confirming the status of the connection and one unsigned integer counter, carrying the timestamp. Here we also require a special case for when overflow is expected. We could either re-establish a connection, or we could allow rolling timestamps, essentially creating an unbounded channel.

5.2 Rust Implementation

The Rust implementation was the second implementation. We expected the Rust algorithm to be close in performance to its C counterpart. In contrast, we expected the development setup to be more complex as it is a less common language and platform combination.

5.2.1 Planning Phase

When planning our Rust implementation we did not have many choices to make as the ecosystem is mostly unified under one platform. Fortunately, the compiler, build

system, and package manager are all well-designed. The greatest variability came from the libraries which would later speed up development significantly compared to other implementations.

5.2.1.1 Compiler Toolchain

The Rust ecosystem provides a canonical compiler called *RustC*. This compiler is available for most available targets using different tiers of support. ARM is available with tier 2 support, which is "guaranteed to build" while the ESP32 is available with tier 3 support. Tier 3 builds are not officially provided, nor are they automatically tested. In practice, this means that users of tier 3 toolchains will have to test their programs more thoroughly as well as bring their own support for the compiler. For complex and niche use cases such as those in the automotive industry, this could result in long wait times for upstream fixes to be applied. Instead, the company might have to patch the code themselves possibly even providing the patch for the upstream project.

There are also other implementations of the Rust language standard. Unfortunately, their respective project owners do not consider their compilers ready for automotive use cases, nor even production-ready in general. Finally, no contending compiler provides feature parity with RustC.

The RustC toolchain can be downloaded using the official `rustup` tool that allows users to install all officially supported toolchain variants consistently. This is especially useful when the development environment requires cross-compilation for different targets. The unified backend of RustC, which leverages LLVM, allows for seamless cross-compilation when compiling on common operating systems for embedded targets that do not require native C dependencies.

RustC is known for having a complex compilation stage, which makes compilation slow compared to C or many other compiled languages. However, RustC also has several optimizations, of which the most important is incremental builds. By dividing each library into its own code unit and compiling in parallel, cold starts become much faster. Meanwhile, these compiled packages are cached so that future executions only have to compile the local code.

5.2.1.2 Build system

While possible to build Rust programs using only RustC, most complex applications will require a build system. There are examples of how to compile Rust projects using generic build systems such as Make, CMake, or Bazel. There is also the Rust-specific *Cargo* tool, which doubles as a package manager and a build system.

Cargo, like many other general build tools, provides build caching, and semantic support for common build actions such as selecting debug or release builds.

In addition to common features, Cargo also integrates testing and benchmarking of applications and libraries allowing for semantic embedding of tests within code. Benchmarks can be defined using a macro (see listing 3) giving access to several con-

venience functions which make tests easier to write. One especially useful function is the `black_box` function, which ensures that the function or expression is executed exactly as written, removing any non-obvious optimizations.

```
#[bench]
fn bench_add_two(b: &mut Bencher) {
    let a = 2;
    let b = 2;
    // Use `test::black_box` to prevent compiler optimizations
    test::black_box(|| a + b);
}
```

Listing 3: An example benchmark, using the black box function to prevent optimizations

The test framework also works well with both unit and integration test styles. It additionally allows for property-based testing and fuzzing, two tools that are becoming increasingly relevant in security and testing.

5.2.1.3 Libraries

Libraries for Rust are officially provided via the Crates repository, reachable using the officially endorsed Cargo package manager and build system. The repository is open-source and self-hostable, allowing organizations to self-host their libraries or simply use git repositories as library sources.

Libraries that provide different implementations depending on platform features or cipher suites may implement feature flags that can easily be toggled in the application manifest. For example, many libraries may be compiled without `std` or `malloc` to comply with certain embedded targets.

Our Rust implementation relied on several libraries. Perhaps the most important were `serde` and `postcard`, both providing serialization and deserialization functionality. `serde` allows the programmer to use macros to automatically serialize and deserialize data structures using the same API. `postcard` was the data format that we used to interchange messages, relying on `serde` to provide the underlying mechanisms.

We had hoped to rely on the same cryptographic library between our platform targets but the support for the ESP32 was not good enough to warrant an extended effort in this direction. No such issue was present for the ARM implementation, and we could rely on the well-vetted `ring` library to provide cryptographic primitives. As support for the ESP32 platform evolves we expect libraries to be further supported, allowing more unified code bases.

The `Mutex` type in the Rust Standard Library has several restrictions, namely that it is not available in static contexts due to relying on memory allocation and lock poisoning [76]. It is important to note that poisoning has use cases, where the program is expected to unwind the call stack to recover from errors. This advanced

use case is not important for this thesis but can serve as a further incentive to look into the ecosystem if you are developing applications that may never crash unintended. The `parking_lot` library provides a mutex without these issues and several other benefits, which caused us to switch implementation.

5.2.1.4 ESP32 SDK

Like in the C implementation, some functionality had to be derived from the ESP-IDF. The Rust `std` library provides abstractions on top of the most important SDK functions such as networking and concurrency, allowing us to reuse more code between architectures.

As previously explained the ESP32 could not compile most cryptographic libraries properly. It was however still possible to rely on the `MbedTLS` library bundled within the SDK. This provided us with an opportunity to further experiment with *unsafe* Rust as it was required to interface with the C library.

5.2.2 Implementation Process

The development time of the Rust implementation was surprisingly short after the initial setup. As Rust promises to do away with most undefined behavior, debugging memory issues became only a small part of the implementation process. Most of the heavy lifting could be done by our libraries, while we could focus on the algorithm at hand instead of its many subproblems.

5.2.2.1 To be Safe or Unsafe

As specified in section 3.5.2.1 Rust has two programming environments. The first is the safe environment where the compiler guarantees that program behavior is well-defined. The other is the unsafe environment where no such guarantees are made. Here, the developer may access memory without safeguards and in general disregard the borrow checker completely.

The safe environment does this through the borrow checker described earlier. Most programmers who are not targeting embedded platforms will spend the majority if not all of their time in a safe Rust environment as there are few use cases requiring unsafe environments. Even during our development, this was where we spent most of our time. In the next section, we will go into more detail on how the *borrow checker* helped us.

The unsafe environment is primarily intended for three use cases: Accessing C functions, manually optimizing code, or accessing memory constructs (such as sensors or system calls) manually. The goal is to implement such functions as small, contained, and well-tested pieces of code to produce results that can be handed to the safe environment where most of the logic will be located.

Our codebase used the ESP-IDF SDK wrapper provided by the `esp-idf-sys` family of libraries. This library helps integrate the SDK with the Rust `std` library. However, some functionality is only provided as raw C functions. This was the case

```
unsafe {
    // This struct has a fixed size
    let struct_size = size_of::<mbedtls_pk_context>();
    let layout =
        Layout::from_size_align(struct_size, 1)
        .unwrap();
    let public_key_context =
        System.alloc(layout)
        as *mut mbedtls_pk_context;
    mbedtls_pk_init(public_key_context);
    return public_key_context;
}
```

Listing 4: Allocating a struct manually in Rust. Note that we can unwrap the Result as we know that the parameters are valid before-hand.

for the MbedTLS library included within the SDK as well as some of the network functionality. As such we had to implement our own safe wrapper functions for these functions. The Rust compiler and included libraries help the developer in this process by providing zero-cost abstractions for memory allocation which leverages the type system, see listing 4. Further, we could create structs wrapping the pointer, and provide safe implementations of the operations pertaining to it within the struct. In the safe code, we could then reason about the struct as if it were any Rust struct from a library providing the functionality required.

5.2.2.2 The Borrow Checker

The borrow checker managed to capture many inconsistencies during development. In general, we experienced very few memory faults and only one outside of the unsafe environment. There were two areas in which the borrow checker was particularly helpful: race conditions and resource use and release.

Rust is usually very restrictive with static variables. They are however required in certain embedded contexts to represent singleton structures wrapping hardware components. Our implementation first wrapped the pointer within a Rust struct and provided a safe API on top of the C implementation. We leveraged the `lazy_static!` macro, allowing us to run arbitrary code, to initialize the struct, before beginning the actual program from `main`. This macro also allowed us to perform memory allocation during static variable initialization. Continuing, we could then wrap the resource within a `Mutex` and `Arc` (Atomically Reference Counted) to provide thread safety.

Another significant helper was the `Drop` trait. This trait is built into the compiler and can be used to instruct the compiler how to clean up resources after they are no longer needed. This is mostly only useful when creating safe abstractions from unsafe code as the automatic trait derived for each struct almost always works as intended in safe environments. As previously mentioned in section 3.5.2.1, the compiler tracks

```rust
impl Drop for RSAContext {
    fn drop(&mut self) {
        unsafe {
            // Free the C struct using the library function
            esp_idf_sys::mbedtls_pk_free(self.handle);
        }
    }
}
```

Listing 5: The Drop implementation of the MbedTLS RSA context wrapper. Any remaining memory resources allocated through Rust are still freed automatically.

references to variables within the program to know when to free memory, this means that we never have to manually call `free()`, or the Rust-equivalent `drop()`, unless we want more granularity (for example, releasing locks before the end of a scope).

In our implementation, we could use the `Drop` trait for the AES contexts as they were MBedTLS structs behind the scenes. Whenever the key had been used for the final time we would simply remove the entry from our security table and the `drop()` function would run automatically as soon as the scope ended. This allows for greater separation of concerns and makes memory leaks less common. In practice, the only way for memory leaks to occur is to abstain from, or incorrectly implement the `Drop` trait, which our language server may warn us of. Each individual `drop()` method is also simpler to debug, as it is universal within the application, instead of sprinkled around the codebase, as seen in our implementation of the RSA context wrapper in listing 5. As no manual steps are required, it is also not possible for the developer to simply *forget* to run the free function.

5.2.2.3 Types & Macros

During the implementation of the Rust version we once again had to design our packet timestamps. Seeing as we had already developed the C version, we would not be repeating the same mistake (section 5.1.3). However, we also speculated on how Rust could help us prevent this from occurring in the first place. Rust provides several variants of integer addition, those being: wrapping the number back to 0, saturating the integer at the limit (maximum for addition, minimum for subtraction), or we could use a checked arithmetic operation. This returns an `Option` [77] which forces us to handle edge cases. There is also a variant of wrapping which also returns a boolean, stating whether we have wrapped or not. However, using booleans does not force the developer, through the type system, to take action as the value can simply be ignored, perhaps with the intent to return to it later. As a final note, this does indeed incur a performance loss. However, it is negligible in the scope of the algorithm as defined by this thesis.

```
#[cfg(test)]
mod tests {
  quickcheck! {
      fn prop(start: u16) -> bool {
          let client = Client::new(start);
          let timestamp = client.send_message(TEST_MESSAGE);
          return timestamp > start;
      }
  }
}
```

Listing 6: A simplified example of using property-based testing to assure that timestamps are monotonous.

5.2.2.4 Property-based Testing

Property-based testing can also be used to prevent flawed algorithms by testing inputs that are likely to be edge cases. Property-based testing allows the developer to specify a function with arguments that will be varied by the Quickcheck algorithm. Rust has a library called `quickcheck` that provides macros, usable with the Cargo test framework, to allow for property-based testing. Any type that implements the `Testable` trait can be used as input to be used in the test. We generated starting values for timestamps and tested whether they were monotonous after using the client to send a dummy message.

5.3 MicroPython Implementation

MicroPython sticks out as the only language running in a Virtual Machine (VM). This results in interesting performance trade-offs in return for simplicity and portability.

5.3.1 Planning Phase

As MicroPython is a fairly new language, it currently lacks full support for the ESP32 platform. As such our implementation will focus on the Pico instead.

5.3.1.1 Compiler Toolchain

There is only one compiler available for MicroPython, which is hosted by the project owners. It supports most recent Python language features with a few exceptions such as asynchronous generators. None of which significantly impact the developer experience in this case.

5.3.1.2 Libraries

MicroPython has restricted support for Python libraries in general. Any library interacting with the system using system calls and relying on an underlying OS will not work. Examples are network or filesystem libraries. Each board supported by MicroPython has its own library where external functionality is implemented, although many implementations do implement these using an API reminiscent of the built-in libraries that Python developers may be used to.

Cryptography libraries are similarly hard to discover. AES support comes built-in with the MicroPython standard library. Support for asymmetric cryptography with RSA can be found in AdaFruits repositories for hobby programming. This implementation is written in pure Python and intended for hobbyist use which is less than ideal for usage in the automotive domain. To further add to the situation, there are no standalone libraries for X.509 management that compile on the Pico. Instead of extracting code from the available TLS modules, we opted to create a simplified certification verification process, merely simulating an actual exchange.

5.3.2 Implementation Process

The implementation in MicroPython was very simple in comparison to the C and Rust implementations. However, due to being an interpreted embedded language requiring specific hardware layouts, it is difficult to test without hardware. MicroPython also lacks any significant debugging utilities making hardware issues difficult to diagnose. Programs thus have to be tested using print statements or other custom methods.

The simplicity of the language can also be a strength. The lack of manual memory management mechanisms simplifies development as we do not have to manage arrays or allocate structures. Consequently, memory errors were practically non-existent. On the other hand, it was difficult to predict when garbage collection would kick in. The garbage collection algorithm also caused usage spikes beyond those normally seen in servers or desktops due to the restricted hardware and complex algorithms used to reclaim memory.

6

Results

In this chapter, we outline our results and benchmarks. We briefly explain which factors might influence the results and how they do so.

6.1 Overview

Using the metrics defined in section 4.6, our platforms and languages perform as expected. Cryptographic operations were expensive but mostly allowed real-time communication. The exception was key generation which was too slow to be of any practical use.

Among the programming languages C and Rust had similar performance profiles, with MicroPython falling behind significantly. MicroPython also lacked key components on the ESP32 platform which disallowed a full implementation and therefore testing.

All simulation results were executed on an Intel Core i7-1260 using external power. We did not experience any throttling of the CPU on the host but did note significant delays when interacting with simulated hardware.

Each of the tests targets different stages of the algorithm lifecycle, beginning with the cryptography elements and ending with the overall performance of the system over time.

6.2 Performance

As will be discussed below, the Feather is the most performant platform, mostly due to the high clock speed and instruction set. This was expected as the specifications in table 4.1 would point to this when combined with our metrics as defined in section 4.6. The Feather beats the Pico by a significant margin during cryptographic operations. The emulator is not a contender in this comparison but the metrics are useful to understand the increased cost of operations when developing on standard user platforms.

The dual-core architecture helps alleviate some of the congestion as one of the cores is available for urgent computation. Time-sensitive code can be identified and run in

	QEMU			Feather			Pico		
	min	max	**avg.**	min	max	**avg.**	min	max	**avg.**
C	8.83	139.04	**40.93**	6.34	144.45	**39.02**	27.55	341.58	**96.34**
Rust	6,87	177.50	**37.24**	5.94	156.32	**34.87**	29.01	289.63	**87.13**
MicroPython	N/A	N/A	**N/A**	N/A	N/A	**N/A**	39.55	321.62	**112.41**

Table 6.1: Time to generate a 2048-bit RSA key-pair (seconds), 100 iterations

	QEMU			Feather			Pico		
	min	max	**avg.**	min	max	**avg.**	min	max	**avg.**
C	2	19	**7**	2	14	**6**	8	44	**12**
Rust	6	35	**7**	4	22	**7**	9	46	**13**
MicroPython	N/A	N/A	**N/A**	N/A	N/A	**N/A**	12	53	**17**

Table 6.2: Encryption with a 2048-bit RSA key-pair (milliseconds), 1000 iterations

a separate task pinned to one core, which is reserved for these critical tasks, leaving the other core to do background work or long-running tasks.

6.3 Test 1. Cryptography

As we can see in table 6.1, RSA key generation is potentially very slow (note that the table metrics are in seconds not milliseconds!). The time it takes on average to generate a key pair for use in communication with a single vehicle may surpass the time during which two vehicles will be in range of each other.

The large interval between minimum and maximum values shows how unpredictable an execution run is. We expected C to be the fastest but it was unexpectedly faster to generate keys in Rust most of the time. This discrepancy is difficult to localize but we suspect that it is most likely affected by the entropy available in the RNG source. Further experiments must be done to determine how this influenced executions across platforms. We suspect that the Rust compiler performs some optimization not used or available in the C implementation. It is also possible that the different SDKs perform different optimizations. Rust generally optimizes for performance at the cost of code size which may also explain some of the differences as the different processors have different instruction cache sizes, perhaps resulting in greater cache misses.

Key generation is not interruptable, making the real-time OS lock on the executing core for the duration of the generation. As all platforms have more than one core, this could be offloaded to a dedicated core. However, it is still important to note

	QEMU	Feather	Pico
C	1802	1343	5541
Rust	3591	3122	8924
MicroPython	N/A	N/A	11878

Table 6.3: Average handshake completion time (milliseconds)

	QEMU	Feather	Pico
C	44.6	26.5	65.6
Rust	38.9	20.3	61.1
MicroPython	N/A	N/A	79.1

Table 6.4: Average round-trip-time for a conversation of 1000 messages (seconds)

that the hardware is incapable of generating keys in real-time.

Hardware makes a great difference and is obvious from both table 6.1 and table 6.2 where the Feather greatly outperforms both the emulator and Pico. On the other hand, the differences between languages are not very large when compared to the platforms they run on. Key generation relies on large segments of native code even for Micro-Python which may explain why the performance is not as terrible as one might expect when looking at later tests for this language choice. The Rust libraries rely on C-bindings to perform their cryptography, meaning that the performance difference is mostly due to overhead caused by interop routines and differing memory management.

6.4 Test 2. Network

The handshake is not very speedy, as seen in table 6.3, but could most likely be enhanced further by attempting to limit dynamic memory allocations. In general, the C implementation performs the fewest allocations due to the use of frequent memory casting and reassignment. This required great finesse as small deviations from memory assumptions frequently caused bugs. The Rust implementation could most likely be sped up using similar ideas but would instead lose some memory safety as it would require more unsafe code environments. Since we have almost no control of the memory in the MicroPython implementation we were forced to copy memory frequently when it could have been transmuted in place.

The round-trip-time test showed significant differences between our variables. The actual network latency was not measured within the emulator but the slow hardware interactions still increase the execution time significantly, most likely outweighing any would-be signal latency. For the hardware platforms, we must keep in mind that unfavorable network conditions may significantly change the outcome of the

test.

The test was not performed in a shielded area, therefore surrounding traffic in the same wavelength may have interfered with individual runs or tests. Similarly, we noticed performance degradation when having both cores accessing the Wi-Fi antenna. It is difficult to ascertain whether the performance is an algorithmic flaw in our interactions with hardware or a physical limitation of the platform. As we will later see, the CPU usage did not show increased load, yet we observed significant latency. Further profiling would be necessary to show the exact nature of this issue.

6.5 Test 3. Application Lifetime Assesment

When looking at resource usage we found that the algorithm has bottlenecks on all platforms and languages which must be mitigated before any production use would be deemed acceptable.

6.5.1 Load Average

We measured the load average by sampling the CPU usage using the provided platform SDKS. This worked well for understanding how much raw performance the application required but does not explain the overall latency.

We observed that symmetric encryption operations did not alter the RTT test significantly compared to sending raw packets. Furthermore, the algorithm itself exhibited minimal resource usage, and there were no significant processor spikes during interactions with hardware, such as radio communication. Nevertheless, the emulator did significantly increase execution time when interacting with simulated hardware. The only language that spontaneously showed load spikes was MicroPython due to the garbage collection kicking in, they were not significant enough to halt the program but most likely contributed to the overall slowness of the MicroPython implementation.

Regarding asymmetric operations, the results presented in table 6.2 show that they were mostly inconspicuous in terms of workload impact. However, there was one noteworthy outlier, the RSA key generation process. During key generation, one of the cores had to be locked for a significant amount of time, up to several minutes.

In summary, the load averages for the different platforms remained consistently low, except for instances involving RSA key generation. A solution has to be found for this problem in order for this or any other communication protocol using asymmetric cryptography can be utilized. Further, more tests are required to measure which particular operations cause latency and how it can be mitigated.

6.5.2 Memory

When analyzing the memory (shown in table 6.5) usage we found that the biggest contributor was the code size. Rust and C both appear to optimize code size significantly better on Arm platforms resulting in a slight reduction of total memory

	Feather	**Pico**
C	156 KB	151 KB
Rust	286 KB	208 Kb
MicroPython	N/A	240 KB

Table 6.5: Peak memory used during handshake (percent)

usage. Meanwhile, the different implementations have similar total memory footprints during execution but some had a higher total throughput during certain parts of the algorithm.

The C implementation required almost no additional overhead with the exception of the SDKs for the respective platforms.

The case is similar for Rust, albeit with a slightly higher usage due to the greater availability of libraries which sometimes provided more functionality than was used. Rust performs dead code elimination through LLVM to reduce binary size which removes any unused code paths in the majority of the libraries. Most of the memory usage was due to Rust optimizing performance over binary size. This could be further balanced should it be necessary. Finally, it should be noted that Rust on ARM is more mature, perhaps also contributing to the lower overall memory size.

In contrast, MicroPython requires a large runtime and does not perform dead code elimination. Measuring code size was also more difficult but it seemed to use almost all of the memory available on the Pico.

6.5.3 Performance Summary

When we compare the results of our programming languages we see that C and Rust compete on performance.

6.6 Implementation Comparison

The critical findings found during the process explained in chapter 5 will be summarized here. The qualitative assessments, categorized according to the level of expertise, are presented below in each table, accompanied by an explanation of the three-point scale. Throughout this section, we will utilize the abbreviations **NG**, **PP**, and **IP** to refer to the three respective individuals: new graduate, professional programmer, and industry professional.

6.6.1 Library Availability

When researching and comparing library availability there were many pitfalls which in the end resulted in some unclarities. Library support for wide topics such as serialization or cryptography varies on several grounds but mostly due to different

programming paradigms or due to the many different ways to reach the same goal. As an example, C requires code generation in order for serialization to be automatic and type-safe. Rust on the other hand has a much wider hygienic macro system, allowing programmers to embed the logic within the code itself instead of relying on external tools to generate code. One method is not inherently superior to the other but there are several metrics from which we could base our final result.

As can be seen in table 6.6, the availability is extremely varied and the results have to be carefully interpreted. MicroPython does indeed show only barebones support for most critical libraries, however, it would potentially be possible to port existing Python libraries with relatively low effort. Further inquiries have to be made into the viability of porting libraries as that was not in the scope of this thesis. Rust seemingly showed the widest support. This is in part due to the batteries-included approach of the standard library, together with there only being one de facto standard library which is centrally developed. C had partial support in all categories. This was accounted to very few libraries being cross-platform and a high on-boarding process which meant that it was sometimes not worth it to find libraries to perform simple tasks. This can be somewhat mitigated with experienced developers but is still a drawback.

6.6.2 Memory Management

Each implementation used very different methods of primary memory management. The MicroPython implementation required no interaction with memory mechanics, though to provide performant algorithms the programmer was required to understand the repercussions of garbage-collected memory. This manifests mainly in activity spikes during the garbage collection process itself, where the system freezes briefly before beginning to execute once more. Some optimizations were of course not available due to the nature of the management method. It is impossible to control the memory layout or control where a variable will be stored.

The C implementation required the most extensive test suite as few mistakes could be caught by the compiler. This was further exacerbated by the requirement to

	Cryptography	**Serialization**	**Utility**
MicroPython (Arm)	Partial	Partial	Partial
MicroPython (ESP32)	None	None	Partial
Rust (Arm)	Partial	Full	Full
Rust (ESP32)	Partial	Full	Full
C (Arm)	Partial	Partial	Partial
C (ESP32)	Partial	Partial	Partial

Table 6.6: Diversity and availability of libraries.

manually cast types as it requires the programmer to make soundness calculations.

The Rust implementation was unique in that it relied on both the unique borrow checker and manual memory management. However, the extensive type system and well-thought-out APIs make manual memory management more understandable and easier to reason about. The algorithm itself is written fully in safe Rust as only the C and hardware interfaces must be written in unsafe scopes. This combination allowed for a balance between safety and control over memory management. As can be seen in the tables table 6.3 and table 6.2, this sometimes comes at a cost. Given further time, critical paths could be analyzed and optimized further, possibly allowing the Rust code to be more efficient or as efficient as the naive C implementation. However, it is also likely that similar optimizations could be performed in the C implementation.

Further, we only had issues with stack overflows once during the Rust implementation. We had initially set the stack size to be only slightly larger than in our C implementation as we knew that Rust had a habit of storing variables on the stack more often than C. This was not enough as our Rust implementation used significantly fewer heap allocations than our C implementation, requiring us to double the stack size in the end. Rust thus required further working memory but over time the total memory requirements were similar to the C implementation.

In table 6.7, the data reveals that garbage collection stands out as the method with the least overhead during development. However, measuring the borrowing method's performance was challenging due to Rust's requirement of manual memory management in *unsafe* development, particularly when interacting with C libraries or hardware interfaces. An alternative is to bypass the borrow checker by cloning a value, creating an independent copy, and avoiding the complexities of memory management, albeit sacrificing some performance benefits obtained by adhering to borrowing rules. Regardless, all participants found mastering borrowing difficult.

Graduates and professional programmers agreed that memory management in C was challenging. Those with extensive experience in manual memory management languages still reported high productivity. Nonetheless, they acknowledged the difficulty of identifying errors in large codebases or code produced by other programmers, which remained an ongoing issue in their projects. On the other hand, industry specialists employed strict formatting rules and programming frameworks that helped mitigate errors, such as use after free or unintended side effects.

	NG	**PP**	**IP**
Manual	Significant	Significant	Relative
Borrowing	Significant	Relative	Relative
Garbage-collection	Relative	Low	Low

Table 6.7: Difficulty of using different memory management paradigms

	Package signing	ACE protection	Auditing
C (N/A)	None	Partial	None
Rust (Cargo)	3rd party	None	Full
MicroPython (pip)	Partial	None	Full

Table 6.8: Support for library security features within development environments

6.6.3 Security Features

When analyzing the security features of each language environment we must take into account that each ecosystem is different in its process. Seeing as there is no official package manager for C we instead compare the other package managers to the manual process of library management. Similarly, some build systems are integrated into their respective package managers, while others are not.

6.6.3.1 Package Managers

All tested package managers include features to automatically scan codebases for known vulnerabilities, usually by providing command-line auditing tools to check package versions against a database of versions with known errors. While there exist vulnerability scanners for C code they are often commercial, meaning that auditing the scanner itself is difficult and the product more expensive. It also requires one or more external tools depending on the coverage of the databases in each tool.

The package management systems endorsed by the entities holding stewardship over each language are diverse in their security features. We can see a trend in package managers towards supporting package signing but we are not yet there. This trend can be seen both in the ones used within this thesis and in the landscape at large.

6.6.3.2 Build Systems

Most build systems seem to allow arbitrary code execution when downloading and building packages. This is by design to allow more complex builds to occur. A balance must be struck here to ensure that security and usability are both met. In a scenario where code is signed and entrusted, this practice may not, in and of itself, pose immediate issues for most threat models. The assumption of code authenticity and integrity can mitigate potential security threats.

Since C notably does not provide a canonical build system it is difficult to assess the status of arbitrary code execution during build. Most C projects still utilize *some* build system, often make or a derivative of make, which are just as vulnerable by design as any other. Similarly, it might be interesting to look at code generation tools as they might also hold potential for security threats similar to those incurred by build systems.

7

Discussion

With our metrics and results presented, we move on to the discussion of our findings. We begin by discussing the developer experience and security implications through the lens of the different programming languages we tested. We then explain potential performance indicators to further justify the feasibility of changing the status quo. Finally, we present ethical considerations relevant to our thesis and suggestions for future work.

7.1 Developer Experience

We could see quite varying responses to the different programming paradigms. The reference group's answers matched rather well with the survey conducted by StackOverflow. C seems to be held in a negative light, in part due to the lack of ergonomics, the great work required to change the workflow, import new libraries, or review code reduces motivation and perhaps also innovation. The ranking of newer languages like Rust seems in part to be attributed to hype. The reference group was positive to the features present within Rust but none were certain of the long-term consequences of using the language in larger codebases. When it comes to MicroPython, the experienced developer and new graduate were positive about MicroPython and its development. The industry professional was however skeptical due to the lack of security mechanisms and control over resources, stating that it would require a tremendous effort to even begin porting parts of the codebase to such a language.

When asked about WebAssembly most had not heard of it and they all stood skeptical of the long-term viability of the project as a platform for embedded development, comparing it to Java which started as an alternative for embedded devices but has since lost favor with the community. Perhaps if WebAssembly gains more traction within the IoT where requirements are less stringent, it could become a possible tool for automotive development in the future.

7.2 Security Considerations

We had to take into account several potential security issues during the implementation. They are discussed below, where we present where the issues arose and potential issues.

7.2.1 Platform Security

When investigating the differences in security offerings between platforms none could fully attest to the security of a library, though efforts are being made towards that goal. The platforms that offered code signing can at least be considered authenticated and, or unmodified by attacks targeting the delivery mechanisms of the packages. Of course, in the end, trust still has to be manually assigned and external organizations without proper security procedures are still vulnerable to social hacking. Code attestation may never change that.

One strategy to minimize the effects of code execution during builds is to perform them in sandboxed environments. Normal compilation stages may need access to resources such as the filesystem but may not need internet access. Sandboxing has become common in mobile environments such as the iOS and Android operating systems but is yet to attain widespread use in desktop systems. Such restrictions could be designed to be a part of the build languages or as part of the OS, though it may be difficult to retrofit such a system onto existing operating systems. This would protect the development environment but could still leave malicious code within executables, only partially solving the problem.

Research into information control flow has delivered promising models that allow the programming language to sandbox variables and data, although they require a somewhat sophisticated type system to be useful. Of the languages used within this thesis, only Rust's type system can be considered advanced enough to provide such restrictions in a useful manner. Languages such as MicroPython, with an integrated runtime, could perhaps instead attempt to include this metadata dynamically and then allow the virtual machine to verify data flow during runtime.

The lack of a canonical vulnerability database for C libraries with developer integrations creates friction during the integration and upgrading of libraries. Third-party services are available but were not evaluated as part of this thesis. Aggregating and verifying the vulnerabilities may become too large of a task for smaller contractors that lack the funds available to larger manufacturers. Teams may be encouraged to write their own library replacements instead. This requires discipline and expertise that may not be available or otherwise result in code with other vulnerabilities. Furthermore, updating libraries may become a hassle and encourage development teams to stay on buggy versions, fixing the problems themselves, or postponing upgrades until resources are available. Finally, it is possible that organizations instead decide to trust upstream sources blindly, ignoring the initial vetting process and potentially including harmful code.

7.2.2 Network Security

While the different algorithms powering secure communications are agnostic to the implementation, different languages have different security assurances. C provided no guard rails, complicating the formal reasoning of the program. In a sense MicroPython provides just as few guard rails due to the lack of a type system, meaning that the logical reasoning is still fully the responsibility of the developer even though

```rust
struct SecureData {
    credentials: String,
}

trait Secret {}
impl Secret for SecureData {}

fn transmit<T>(value: T)
where
    T: !Secret,
{
    // Serialize and transmit data
}

fn main() {
    let secret_data = SecureData {
        credentials: "password".to_string(),
    };
    // Example usage with a type that implements 'Secret'
    transmit(secret_data); // This won't compile!

}
```

Listing 7: An example of a function that will fail to compile if secret data is used as input.

certain memory bugs are impossible due to the automatic memory management. In contrast, Rusts type system is very expressive allowing more complex safety nets to be implemented.

In this thesis, we did not implement any cryptographic algorithms ourselves but our experience in these languages did give us insight into what it means for critical systems to be implemented in languages without protection from logic errors. A type system is of course not a panacea that protects against **all** logic errors, but we still advocate that *more* protection is greater than none.

In Rust, one way to implement further protection from accidental data leaks occurring during transit would be to tag any critical data structures with a blank trait `Secret` and disallow any such structs from being passed to the function that transmits data over the network. An example can be seen in listing 7. In listing 8 we instead see the example of an inverse function that only accepts safe types.

7.3 Memory Safety

While difficult to measure objectively, we did collect the number of times we ran into memory issues during development. In C, the most troubling issues were memory

```rust
// A wrapper around a byte vector, tagging it as encrypted
struct Encrypted(Vec<u8>);

fn transmit(data: Encrypted) {
    // transmit data
}

fn encrypt(data: Vec<u8>) -> Encrypted {
    // encrypt data
}

fn main() {
    let unencrypted_data = vec![0, 1, 2, 3];
    let encrypted_data = encrypt(unencrypted_data);

    transmit(encrypted_data); // This is ok!
    transmit(unencrypted_data); // This won't compile!
}
```

Listing 8: An example of a function that will fail to compile if the input is not encrypted.

leaks, use of null pointers, use after memory was freed, and heap allocation errors due to incorrect manual memory bounds calculations.

For Rust, the issues were mostly stack overflows due to the greater use of stack variables and a low stack bound by default on all hardware platforms. Heap allocation errors were absent as variable sizes are in general automatically calculated. Although null pointers remain a potential concern, Rust's robust typing extends to null pointers, requiring greater and more deliberate intent in manual pointer casting within unsafe scopes. Safe code provides cleaner abstractions such as the Option monad to represent the lack of a value, forcing the user to specify control flow operations for all cases. Another benefit of the unsafe scope is that it is very clear which portions of the codebase may panic due to memory errors. Unsafe scopes also enhance the clarity of the code review process by confining the expanse of hazardous operations to a small fraction of the entire codebase.

MicroPython gave rise to almost no memory errors but did suffer in performance instead. The errors that did arise were found inside external C libraries. The platform is also less mature and not battle-tested at all. Before using MicroPython or any other similarly interpreted language, the underlying runtime must the thoroughly vetted to remove potential security issues. One can also wonder whether writing an underlying runtime in C and then running an interpreted payload delivers significant benefits when compared to simply using C for the project itself. Though it does arguably provide a smaller attack surface.

It would be interesting to conduct further studies on the benefits and drawbacks

of functional programming languages for vehicular systems development. While not truly comparable, Rust borrows many constructs from the world of functional programming which this thesis regards as positive successors to previous constructs. Examples of such constructs are the aforementioned Option monad and property-based testing.

7.3.1 Supply Chain Security

Although various programming languages offer distinct mechanisms to guard against logic errors, they do not inherently safeguard against malicious code. Within package managers and build systems, different mechanisms are implemented to ensure that flawed libraries do not harm developers or their end-users.

Of all languages tested only Pythons PyPi provided the ability to sign packages. PyPi signs each package in the central repository, assuring that the package has not been tampered with during delivery between the repository and the client. The upload between the author and repository is secured in both Cargo and PyPi through multi-factor authentication.

Another potential security improvement that Rust currently offers is reproducible builds which assures that the compiler builds produce deterministic data. This is already possible but requires elaborate setups and versioning of sub-components which is only viable within large organizations but not within the ecosystem as a whole. In particular, one has to freeze the linker and LLVM versions used in the backend, as well as make sure that any macros used do not depend on the build setup. In addition, some factors such as build path may interact with the reproducibility further complicating the effort.

As for vulnerabilities, Cargo provides commands to audit your dependencies to find known vulnerabilities through *RustSec*. This service is provided by the Rust Security Team and receives updates from first and third-party auditors. This information is also used by crates.io and the Cargo specification supports a feature known as *yanking* to remove and flag releases known to contain harmful or unintended breaking changes.

Furthermore, it is important to note that Cargo also has support for packages through alternative channels, including git, local files, or hosted instances of the crates registry. This flexibility allows for organizations to establish their repositories, wherein solely audited and approved packages are made accessible. By leveraging these channels, organizations can exercise greater control over the dependencies used in their projects.

7.4 Performance

Even though some of the tests did not show satisfactory performance we are overall satisfied as the relative performance between languages gives great insight into the future possibilities of automotive development. Seeing as the cryptographic opera-

tions were the most computationally expensive, and networking a significant latency bottleneck, we focus on performance from the perspective of these two parameters.

7.4.1 Cryptographic Performance

Our tests show that there are several issues present in the available cryptography libraries. One of the prominent hurdles is the time-consuming nature of generating RSA keys. The process of generating these cryptographic keys can be computationally intensive, leading to delays in establishing secure connections between parties. To address this issue, a possible approach is to implement RSA key generation during idle periods. By utilizing idle system resources, key generation can be performed in the background, allowing the resulting keys to be readily available when needed and thus speeding up the handshake process during actual communication sessions. For this purpose, thousands of ephemeral keys may be stored on flash memory for later use. Care has to be taken to make sure that these keys can not be leaked, so they should preferably be stored in an encrypted form using a Trusted Platform Module to store the encryption key.

When attempting to further understand the issue we found that it may not only be an issue of clock speed but also an issue of randomness generation. This could account for why the spread is so large between runs. As debugging hardware modules can be very time-consuming we did not have the time to further investigate if a better RNG module would result in better performance.

While key generation was indeed expensive, other cryptographic operations did not show any significant performance impact. RSA is only used during the handshake, so even though it is significantly slower than AES, it is used seldom enough not to warrant any further optimization.

Apart from speed-related challenges, the diversity of platforms and their native libraries is another crucial aspect to consider. While most platforms include native libraries for cryptographic operations, some platforms lack these essential components. In such cases, developers have to resort to using FFI to access libraries from other languages, introducing complexities and potential security risks. In the case of Rust and MicroPython this means breaking out of the memory-safe environment and reducing, but not removing, the benefit of working with such languages.

Additionally, the limited availability of alternative libraries across different platforms further limits adoption. The lack of alternatives reduces the flexibility and adaptability for developers to choose the most suitable and performant library for their specific use cases. This lack of diversity can lead to less resilient systems, as developers might be compelled to use libraries that may not fully align with their security or performance preferences. With few libraries available, few or none may have undergone serious security audits or have been battle-tested to the extent required by critical software. In the best of worlds, manufacturers would be pushed to fund audits for the benefit of the community, or at the very least their user's interests, although the administrative hassle may instead encourage developers to remain silent and hope that the library does not contain any serious flaws.

7.4.2 Networking Performance

The networking issues we found seem to be based on inefficient usage of the attached radio modules as well as the relatively low power of the devices. Since the results show no particular overhead for each respective language when performing radio operations we only note that platforms must be evaluated to fit the goal of the module and that the choice of programming languages do not inherently affect the performance of the network.

7.5 Ethical Considerations

The field of computer science is often regarded as ethically neutral. Yet, the state of cybersecurity, or its absence, can carry significant consequences for both individuals and society as a whole. Manufacturers and service providers are obliged to develop systems that do not cause harm to their users as dictated by regulations from the European Union and other institutions. Furthermore, the vehicular industry is one of the largest sources of environmental concerns. New technology may demand new products further exacerbating the global environmental concerns.

7.5.1 Privacy and Integrity

Recent legislation has focused on aspects of integrity and the right to privacy. Vehicles have the capability of generating vast amounts of data, which is easily connected to the owner, some of which may be considered especially harmful to privacy. Among these, location data may be the most intrusive as it can be used to identify vital information about an individual including but not restricted to where they work, live, shop, and whom they associate with. This is perhaps not a great concern to those living in democratic nations, especially as most are willing to give their locations to smartphone service providers for free. This is however a concern for more unstable countries where the government may wish to use this data to control its citizens.

Furthermore, cyber-physical systems such as vehicles carry even greater risks as they have the ability to interact with the physical world. Remote-controlled vehicles are of great convenience. An office worker may take their bike to work in the morning, but once the afternoon rain hits, they might call their car in order to ride it home. While these features are lucrative for service providers they also come with risks. A hacker may take control of vehicles in order to cause disorder in society or outright steal vehicles. As we have learned from other incidents, no system can ever be 100% secure from attacks, thus we must decide between the benefits and the risks.

This thesis aims to diminish risks associated with the development of future vehicular systems. By leveraging more secure paradigms, we can enjoy fewer potential vulnerabilities in our code and thus fewer potential attack vectors. We believe that even though some of the proposed paradigms offer slower development or require greater expertise, manufacturers should be held responsible for their choices when producing potentially dangerous products. This may require further legislation and adoption of standards beyond those already in effect, and perhaps even the dis-

solution and replacement of old standards that no longer produce sufficiently safe products.

7.5.2 Environmental Impact

Vehicles are significant contributors to climate change, both by relying on the oil industry and requiring large amounts of both inert and dangerous materials, mostly in the form of metals. Fortunately, as manufacturers opt for more self-driving capabilities in vehicles, they also see that electrical control systems provide benefits in terms of feedback and control. It is possible that this feature alone could be a major force in the adoption of electric vehicles (EVs). EVs do not necessarily rely on the aging oil industry but are no panacea either. The fuel for EVs is only as clean as their sources, meaning that they could still be powered by dirty coal plants, now with even less transmission efficiency.

Greater security requirements can often lead to increased overall robustness, prolonging the service life of equipment. Vehicles that are difficult to physically manipulate can be built to further protect from adverse conditions such as weather or information but may also reduce repairability if they require specialized tools to repair. Connected vehicles may however also include sensors that detect wear of components, potentially signaling of a dangerous condition well before any actual harm occurs. Previously you often had to wait for the vehicle to stop working before a flaw could be detected. With connected fleets and big data it may be possible to use statistical tools to infer the state of components in a way not possible by current sensors which only measures the current performance.

7.6 Summary of Findings

Our initial hypothesis was that the more dynamic language targets, those being interpreted or executed using software virtual machines would deliver practical improvements to onboarding and the developer experience. Unfortunately, we can now see that the landscape is not yet sufficiently evolved to support automotive applications for anything other than the most simple of use cases, and even then, the hardware support is minimal. The immature support also spills over into the ecosystem as only a fraction of the ecosystem is available for these embedded use cases. This means that the proposition of an improved ecosystem quickly becomes void. WebAssembly will most likely evolve on its own, without the influence of the industry, and may become more mature in a few years when it has gained a foothold within the web domain. If WebAssembly as a project were to fail or change direction, efforts towards its use within embedded systems programming may be in vain. *If the industry is interested in utilizing the potential of high-level languages and targets they must first assure comprehensive hardware support and invest in already available open ecosystems.*

Meanwhile, we expected compiled languages closer to C, being the industry standard, to offer similar performance and hardware benefits to C but perhaps with new and improved ecosystems. Our only other tested language was Rust and compared to C

we found significant security and ecosystem improvements. Although C remains the language with the widest hardware support, Rust is quickly gaining support from a variety of platforms. The great interop between C and Rust also allows progressive adoption of Rust within existing codebases, with previous work within the Linux kernel showing the viability of such integrations. *With the recent announcement of an ISO-certified Rust compiler it may be a good time to invest in Rust and its ecosystem due to the long lead times within the industry. Improved safety may be necessary to ensure that safety within connected vehicles remains relatively uncompromised when exposed to new threat models.*

The algorithmic backbones provided by traditional cipher suites such as public key exchange, certificates, and symmetric encryption for established channels are natural candidates for future peer-to-peer networks within the automotive industry. It is difficult to predict exact throughput requirements due to the lack of real-use case studies performed by vendors. This thesis does not significantly regard consistency requirements, which may be of interest for future investigations. *In cases where tasks can handle some delay in network communication without real-time constraints, opting for a centralized communication approach might make sense. This would allow the utilization of existing infrastructure, algorithms, and security guidelines.*

It would be unwise to expect a new toolchain to be ready as-is to be used within the automotive industry, one of the most strict domains of software development. Whichever path the industry wishes to take, in order to make its products more secure and by extension safer, it is the belief of the authors that the industry must invest in new tools to meet the ever-increasing risks associated with connected vehicles and IT infrastructure in general.

7.7 Future Work

Future research efforts could be directed towards programming languages with even more formal grammar. Languages such as Haskell could allow for a very formal definition of operations, future efforts could investigate whether they allow for performant programs or whether such formalisms could be written efficiently for the diverse hardware components of an automotive vehicle.

There may also be much to learn from the research into Internet of Things. It is a similar field, where performance must allow for real-time conditions but has a higher degree of innovation owing to laxer legal requirements.

Finally, this thesis only briefly touches on the subject of information control. This is in part due to few practical applications being available for study. Nevertheless, should these efforts manifest into libraries or language features where these features could be seamlessly integrated within programs, they may promote code sharing in domains, such as the automotive industry, which are otherwise locked down due to stringent security constraints.

8

Conclusion

This thesis aimed to investigate the potential security benefits of moving towards new programming languages and paradigms other than C/C++ which has been and currently is the standard for vehicular development. We found that in general moving towards new technologies comes with advantages provided that they are mature enough to support the different architectures required by the automotive industry.

Languages with fully automatic memory management did not live up to our expectations. This is unfortunate as they could make automotive development more accessible to more developers. It is possible that with further development, these languages could be harnessed for specific purposes within the automotive domain.

In contrast, we found that the memory paradigm of Rust can be leveraged to produce code with significantly fewer bugs stemming from unintended memory behavior. Further, the ecosystem surrounding Rust enables developers to be simultaneously productive and secure. While it would be an impossible task to rewrite all hundred millions of existing lines of code, the industry could incrementally adopt Rust, starting with critical connected systems that interface with the outside world. Finally, with an ISO-certified compiler recently made available, along with Rust's C-interoperability, Rust may become the future language of choice for automotive development.

Bibliography

[1] Felipe Cunha, Leandro Villas, Azzedine Boukerche, Guilherme Maia, Aline Viana, Raquel A. F. Mini, and Antonio A. F. Loureiro, "Data communication in VANETs: Protocols, applications and challenges," *Ad Hoc Networks*, vol. 44, pp. 90–103, Jul. 1, 2016, ISSN: 1570-8705. DOI: 10.1016/j.adhoc.2016.02.017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570870516300580 (visited on 04/10/2023).

[2] WIRED, director, *Hackers Remotely Kill a Jeep on a Highway | WIRED*, Jul. 21, 2015. [Online]. Available: https://www.youtube.com/watch?v=MK0SrxBC1xs (visited on 05/30/2023).

[3] Ismail Ertug. "Parliamentary question | EU type approval regulations for autonomous vehicles | E-001676/2022 | European Parliament," [Online]. Available: https://www.europarl.europa.eu/doceo/document/E-9-2022-001676_EN.html (visited on 04/10/2023).

[4] Yinjia Huo, Wei Tu, Zhengguo Sheng, and Victor C.M. Leung, "A survey of in-vehicle communications: Requirements, solutions and opportunities in IoT," in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, Dec. 2015, pp. 132–137. DOI: 10.1109/WF-IoT.2015.7389040.

[5] "Stack Overflow Developer Survey 2022," Stack Overflow, [Online]. Available: https://survey.stackoverflow.co/2022/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022 (visited on 04/10/2023).

[6] "Safe manual memory management | Proceedings of the 6th international symposium on Memory management," [Online]. Available: https://dl.acm.org/doi/abs/10.1145/1296907.1296911 (visited on 04/26/2023).

[7] *Next Steps - OWASP Top 10:2021*. [Online]. Available: https://owasp.org/Top10/A11_2021-Next_Steps/ (visited on 09/19/2023).

[8] Sandip Ray, Wen Chen, Jayanta Bhadra, and Mohammad Abdullah Al Faruque, "Extensibility in Automotive Security: Current Practice and Challenges: Invited," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17, New York, NY, USA: Association for Computing Machinery, Jun. 2017, pp. 1–6, ISBN: 978-1-4503-4927-7. DOI: 10.1145/3061639.3072952. [Online]. Available: https://dl.acm.org/doi/10.1145/3061639.3072952 (visited on 09/26/2023).

[9] Roman Alieiev, Andreas Kwoczek, and Thorsten Hehn, "Automotive requirements for future mobile networks," in *2015 IEEE MTT-S International Conference on Microwaves for Intelligent Mobility (ICMIM)*, Apr. 2015, pp. 1–4. DOI:

10.1109/ICMIM.2015.7117947. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7117947 (visited on 09/26/2023).

[10] Rafia Inam, Nicolas Schrammar, Keven Wang, Athanasios Karapantelakis, Leonid Mokrushin, Aneta Vulgarakis Feljan, and Elena Fersman, "Feasibility assessment to realise vehicle teleoperation using cellular networks," in *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, ISSN: 2153-0017, Nov. 2016, pp. 2254–2260. DOI: 10.1109/ITSC.2016.7795920. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7795920 (visited on 09/26/2023).

[11] Osama Al-Saadeh, Gustav Wikstrom, Joachim Sachs, Ilaria Thibault, and David Lister, "End-to-End Latency and Reliability Performance of 5G in London," in *2018 IEEE Global Communications Conference (GLOBECOM)*, ISSN: 2576-6813, Dec. 2018, pp. 1–6. DOI: 10.1109/GLOCOM.2018.8647379. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8647379 (visited on 09/26/2023).

[12] Matteo Cesana, Luigi Fratta, Mario Gerla, Eugenio Giordano, and Giovanni Pau, "C-VeT the UCLA campus vehicular testbed: Integration of VANET and Mesh networks," in *2010 European Wireless Conference (EW)*, Apr. 2010, pp. 689–695. DOI: 10.1109/EW.2010.5483535. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5483535 (visited on 09/26/2023).

[13] Timm Lauser, Daniel Zelle, and Christoph Krauß, "Security Analysis of Automotive Protocols," in *Proceedings of the 4th ACM Computer Science in Cars Symposium*, ser. CSCS '20, New York, NY, USA: Association for Computing Machinery, Dec. 2020, pp. 1–12, ISBN: 978-1-4503-7621-1. DOI: 10.1145/3385958.3430482. [Online]. Available: https://dl.acm.org/doi/10.1145/3385958.3430482 (visited on 09/26/2023).

[14] Sriramulu Bojjagani, Reddy, Y. C. A. Padmanabha, Anuradha, Thati, Rao, P. V. Venkateswara, B. Ramachandra Reddy, and Muhammad Khurram Khan, "Secure authentication and key management protocol for deployment of internet of vehicles (IoV) concerning intelligent transport systems," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 12, pp. 24 698–24 713, Dec. 2022, Conference Name: IEEE Transactions on Intelligent Transportation Systems, ISSN: 1558-0016. DOI: 10.1109/TITS.2022.3207593. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9905232 (visited on 01/17/2024).

[15] André Pinho, Luis Couto, and José Oliveira, "Towards Rust for Critical Systems," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct. 2019, pp. 19–24. DOI: 10.1109/ISSREW.2019.00036. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8990314 (visited on 09/26/2023).

[16] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk, "System Programming in Rust: Beyond Safety," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17, New York, NY, USA: Association for Computing Machinery, May 2017, pp. 156–161, ISBN: 978-1-4503-5068-6. DOI:

10.1145/3102980.3103006. [Online]. Available: https://dl.acm.org/doi/10.1145/3102980.3103006 (visited on 09/26/2023).

[17] Kunal Sareen and Stephen Michael Blackburn, "Better Understanding the Costs and Benefits of Automatic Memory Management," in *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*, ser. MPLR '22, New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 29–44, ISBN: 978-1-4503-9696-7. DOI: 10.1145/3546918.3546926. [Online]. Available: https://dl.acm.org/doi/10.1145/3546918.3546926 (visited on 09/26/2023).

[18] Nachiappan Valliappan, Robert Krook, Alejandro Russo, and Koen Claessen, "Towards secure IoT programming in haskell," in *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*, ser. Haskell 2020, New York, NY, USA: Association for Computing Machinery, Aug. 9, 2020, pp. 136–150, ISBN: 978-1-4503-8050-8. DOI: 10.1145/3406088.3409027. [Online]. Available: https://doi.org/10.1145/3406088.3409027 (visited on 01/17/2024).

[19] Y Laxmi Prasanna, "An Overview of MANET: History, Challenges and Applications,"

[20] Melaouene Noussaiba and Romadi Rahal, "State of the art: VANETs applications and their RFID-based systems," in *2017 4th International Conference on Control, Decision and Information Technologies (CoDIT)*, Apr. 2017, pp. 0516–0520. DOI: 10.1109/CoDIT.2017.8102645.

[21] Soumya Kanti Datta, Jerome Haerri, Christian Bonnet, and Rui Ferreira Da Costa, "Vehicles as Connected Resources: Opportunities and Challenges for the Future," *IEEE Vehicular Technology Magazine*, vol. 12, no. 2, pp. 26–35, Jun. 2017, ISSN: 1556-6080. DOI: 10.1109/MVT.2017.2670859.

[22] *Dialogue with carmenta automotive AB*, in collab. with Emmanuel Mellblom, Mar. 14, 2023.

[23] Fatih Sakiz and Sevil Sen, "A survey of attacks and detection mechanisms on intelligent transportation systems: VANETs and IoV," *Ad Hoc Networks*, vol. 61, pp. 33–50, Jun. 1, 2017, ISSN: 1570-8705. DOI: 10.1016/j.adhoc.2017.03.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570870517300562 (visited on 04/10/2023).

[24] Darren Hurley-Smith, Jodie Wetherall, and Andrew Adekunle, "SUPERMAN: Security Using Pre-Existing Routing for Mobile Ad hoc Networks," *IEEE Transactions on Mobile Computing*, vol. 16, no. 10, pp. 2927–2940, Oct. 2017, ISSN: 1558-0660. DOI: 10.1109/TMC.2017.2649527.

[25] "RFC3626: Optimized Link State Routing Protocol (OLSR)," Guide books, [Online]. Available: https://dl.acm.org/doi/abs/10.17487/RFC3626 (visited on 04/19/2023).

[26] Songbai Lu, Longxuan Li, Kwok-Yan Lam, and Lingyan Jia, "SAODV: A MANET Routing Protocol that can Withstand Black Hole Attack," in *2009 International Conference on Computational Intelligence and Security*, vol. 2, Dec. 2009, pp. 421–425. DOI: 10.1109/CIS.2009.244.

[27] A. Ghosh, R. Talpade, M. Elaoud, and M. Bereschinsky, "Securing ad-hoc networks using IPsec," in *MILCOM 2005 - 2005 IEEE Military Communica-*

tions Conference, Oct. 2005, 2948–2953 Vol. 5. DOI: 10.1109/MILCOM.2005.1606111.

[28] Viktória Ilková and Adrian Ilka, "Legal aspects of autonomous vehicles — An overview," in *2017 21st International Conference on Process Control (PC)*, Jun. 2017, pp. 428–433. DOI: 10.1109/PC.2017.7976252.

[29] "WP.29 Cybersecurity Vehicle Regulation Compliance | Ultimate Guides | BlackBerry QNX," [Online]. Available: https://blackberry.qnx.com/en/ultimate-guides/wp-29-vehicle-cybersecurity (visited on 04/10/2023).

[30] "ENISA good practices for security of Smart Cars," ENISA, [Online]. Available: https://www.enisa.europa.eu/publications/smart-cars (visited on 04/25/2023).

[31] Victor van der Veen, Nitish dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos, "Memory errors: The past, the present, and the future," in *Research in Attacks, Intrusions, and Defenses*, Davide Balzarotti, Salvatore J. Stolfo, and Marco Cova, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2012, pp. 86–106, ISBN: 978-3-642-33338-5. DOI: 10.1007/978-3-642-33338-5_5.

[32] Onur Mutlu, Ataberk Olgun, and A. Giray Yağlıkçı, "Fundamentally understanding and solving RowHammer," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, Jan. 16, 2023, pp. 461–468. DOI: 10.1145/3566097.3568350. arXiv: 2211.07613[cs]. [Online]. Available: http://arxiv.org/abs/2211.07613 (visited on 01/04/2024).

[33] David J. Pearce, "A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust," *ACM Transactions on Programming Languages and Systems*, vol. 43, no. 1, 3:1–3:73, Apr. 2021, ISSN: 0164-0925. DOI: 10.1145/3443420. [Online]. Available: https://dl.acm.org/doi/10.1145/3443420 (visited on 09/19/2023).

[34] Studnia, Ivan and Nicomette, Vincent and Alata, Eric, Yves Deswarte, Mohamed Kaâniche, and Youssef Laarouchi, "Survey on security threats and protection mechanisms in embedded automotive networks," in *2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W)*, ISSN: 2325-6664, Jun. 2013, pp. 1–12. DOI: 10.1109/DSNW.2013.6615528.

[35] Todd Moore, *Council Post: Uncharted Territory: Managing The New Security Risks Of Connected Cars*, en, Section: Innovation. [Online]. Available: https://www.forbes.com/sites/forbestechcouncil/2023/03/22/uncharted-territory-managing-the-new-security-risks-of-connected-cars/ (visited on 09/20/2023).

[36] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. "Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages." arXiv: 2002.01139 [cs]. (Dec. 2, 2020), [Online]. Available: http://arxiv.org/abs/2002.01139 (visited on 07/17/2023), preprint.

[37] "ESP32-PICO-MINI-02 en | Espressif Systems," [Online]. Available: https://www.espressif.com/en/module/esp32-pico-mini-02-en (visited on 04/25/2023).

[38] "Raspberry Pi Documentation - Raspberry Pi Pico and Pico W," [Online]. Available: https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html (visited on 04/25/2023).

[39] "QEMU - a generic and open source machine emulator and virtualizer," [Online]. Available: https://www.qemu.org/ (visited on 04/04/2023).

[40] D. D. Ward, "MISRA standards for automotive software," en, pp. 5–18, Jan. 2006, Publisher: IET Digital Library. DOI: 10.1049/ic:20060570. [Online]. Available: https://digital-library.theiet.org/content/conferences/10.1049/ic_20060570 (visited on 09/20/2023).

[41] Miroslaw Staron and Darko Durisic, "AUTOSAR Standard," en, in *Automotive Software Architectures: An Introduction*, Miroslaw Staron, Ed., Cham: Springer International Publishing, 2017, pp. 81–116, ISBN: 978-3-319-58610-6. DOI: 10.1007/978-3-319-58610-6_4. [Online]. Available: https://doi.org/10.1007/978-3-319-58610-6_4 (visited on 09/20/2023).

[42] "Rust Programming Language," [Online]. Available: https://www.rust-lang.org/ (visited on 03/17/2023).

[43] Nicholas D. Matsakis and Felix S. Klock, "The Rust language," *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, Oct. 18, 2014, ISSN: 1094-3641. DOI: 10.1145/2692956.2663188. [Online]. Available: https://dl.acm.org/doi/10.1145/2692956.2663188 (visited on 04/10/2023).

[44] Aaron Pang, Craig Anslow, and James Noble, "What Programming Languages Do Developers Use? A Theory of Static vs Dynamic Language Choice," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, ISSN: 1943-6106, Oct. 2018, pp. 239–247. DOI: 10.1109/VLHCC.2018.8506534.

[45] "MicroPython - Python for microcontrollers," [Online]. Available: http://micropython.org/ (visited on 03/17/2023).

[46] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017, New York, NY, USA: Association for Computing Machinery, Jun. 14, 2017, pp. 185–200, ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062363. [Online]. Available: https://dl.acm.org/doi/10.1145/3062341.3062363 (visited on 07/17/2023).

[47] David Hopwood, "A comparison between Java and ActiveX security," *Network Security*, vol. 1997, no. 12, pp. 15–20, Dec. 1997, ISSN: 1353-4858. DOI: 10.1016/S1353-4858(97)88552-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1353485897885522 (visited on 08/31/2023).

[48] *Adobe Flash runtimes | Adobe.com*, en-US. [Online]. Available: https://www.adobe.com/products/flashruntimes.html (visited on 08/31/2023).

[49] "Home Zig Programming Language," [Online]. Available: https://ziglang.org/ (visited on 03/17/2023).

[50] "Docker: Accelerated, Containerized Application Development." (May 10, 2022), [Online]. Available: https://www.docker.com/ (visited on 04/10/2023).

[51] *ESP32 Crypto Benchmark | SSL TLS SSH IPsec TCP*. [Online]. Available: https://www.oryx-embedded.com/benchmark/espressif/crypto-esp32.html (visited on 09/22/2023).

[52] Ricardo Eito-Brun, "A Proposal for the Tailoring of AUTOSAR Coding Guidelines C++ to ISO 26262-6:2018," in *Systems, Software and Services Process Improvement*, Murat Yilmaz, Paul Clarke, Richard Messnarz, and Michael Reiner, Eds., ser. Communications in Computer and Information Science, Cham: Springer International Publishing, 2021, pp. 505–517, ISBN: 978-3-030-85521-5. DOI: 10.1007/978-3-030-85521-5_33.

[53] Stack Overflow, *Stack Overflow Developer Survey 2022*, 2022. [Online]. Available: https://survey.stackoverflow.co/2022/#technology-admired-and-desired (visited on 09/25/2023).

[54] Shao-Fu Chen and Yu-Sung Wu, "Linux Kernel Module Development with Rust," in *2022 IEEE Conference on Dependable and Secure Computing (DSC)*, Jun. 2022, pp. 1–2. DOI: 10.1109/DSC54232.2022.9888822.

[55] Madhur Jain. "Study of Firecracker MicroVM." arXiv: 2005.12821 [cs]. (May 26, 2020), [Online]. Available: http://arxiv.org/abs/2005.12821 (visited on 04/10/2023), preprint.

[56] "Firecracker," [Online]. Available: https://firecracker-microvm.github.io/ (visited on 04/10/2023).

[57] "Open Source - Ferrous Systems," [Online]. Available: https://ferrous-systems.com/open-source/ (visited on 04/10/2023).

[58] "Automotive Microcontroller Market Size, Share, Growth, 2029," [Online]. Available: https://www.fortunebusinessinsights.com/automotive-microcontrollers-market-104084 (visited on 04/19/2023).

[59] "ESP-SDK | Espressif Systems," [Online]. Available: https://www.espressif.com/en/products/software/esp-sdk/overview (visited on 04/19/2023).

[60] Ginny, Chiranjeev Kumar, and Kshirasagar Naik, "Smartphone processor architecture, operations, and functions: Current state-of-the-art and future outlook: Energy performance trade-off," *The Journal of Supercomputing*, vol. 77, no. 2, pp. 1377–1454, Feb. 1, 2021, ISSN: 1573-0484. DOI: 10.1007/s11227-020-03312-z. [Online]. Available: https://doi.org/10.1007/s11227-020-03312-z (visited on 04/19/2023).

[61] Zixuan Zhang, "Analysis of the Advantages of the M1 CPU and Its Impact on the Future Development of Apple," in *2021 2nd International Conference on Big Data & Artificial Intelligence & Software Engineering (ICBASE)*, Sep. 2021, pp. 732–735. DOI: 10.1109/ICBASE53849.2021.00143.

[62] Thomas Rosenstatter, Tomas Olovsson, and Magnus Almgren, "V2C: A Trust-Based Vehicle to Cloud Anomaly Detection Framework for Automotive Systems," in *Proceedings of the 16th International Conference on Availability, Reliability and Security*, ser. ARES 21, New York, NY, USA: Association for Computing Machinery, Aug. 17, 2021, pp. 1–10, ISBN: 978-1-4503-9051-4. DOI: 10.1145/3465481.3465750. [Online]. Available: https://dl.acm.org/doi/10.1145/3465481.3465750 (visited on 04/10/2023).

[63] SAE Mobilius, *Cybersecurity Guidebook for Cyber-Physical Vehicle Systems*. [Online]. Available: https://saemobilus.sae.org/content/J3061_201601/.

[64] Information Technology Laboratory NIST | Computer Security Division. "Cryptographic Standards and Guidelines | CSRC," CSRC | NIST. (Dec. 29, 2016), [Online]. Available: https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines (visited on 04/19/2023).

[65] Billy Bob Brumley and Nicola Tuveri. "Remote Timing Attacks are Still Practical." (2011), [Online]. Available: https://eprint.iacr.org/2011/232 (visited on 04/19/2023), preprint.

[66] Richard Chirgwin. "Android bug batters Bitcoin wallets," [Online]. Available: https://www.theregister.com/2013/08/12/android_bug_batters_bitcoin_wallets/ (visited on 04/19/2023).

[67] Daniele Micciancio and Oded Regev, "Lattice-based Cryptography," in *Post-Quantum Cryptography*, Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, Eds., Berlin, Heidelberg: Springer, 2009, pp. 147–191, ISBN: 978-3-540-88702-7. DOI: 10.1007/978-3-540-88702-7_5. [Online]. Available: https://doi.org/10.1007/978-3-540-88702-7_5 (visited on 09/22/2023).

[68] Peter Schwabe, *Kyber*, Text. [Online]. Available: https://pq-crystals.org/kyber/index.shtml (visited on 09/22/2023).

[69] "NIST Announces First Four Quantum-Resistant Cryptographic Algorithms," en, *NIST*, Jul. 2022, Last Modified: 2022-07-07T12:51-04:00. [Online]. Available: https://www.nist.gov/news-events/news/2022/07/nist-announces-first-four-quantum-resistant-cryptographic-algorithms (visited on 09/22/2023).

[70] Christian Paquin, Douglas Stebila, and Goutam Tamvada, "Benchmarking Post-quantum Cryptography in TLS," in *Post-Quantum Cryptography*, Jintai Ding and Jean-Pierre Tillich, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2020, pp. 72–91, ISBN: 978-3-030-44223-1. DOI: 10.1007/978-3-030-44223-1_5.

[71] *Block cipher mode of operation*, Page Version ID: 1166972375, Jul. 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Block_cipher_mode_of_operation&oldid=1166972375 (visited on 09/22/2023).

[72] "The SWEET32 Issue, CVE-2016-2183 - OpenSSL Blog," [Online]. Available: https://www.openssl.org/blog/blog/2016/08/24/sweet32/ (visited on 04/19/2023).

[73] kokke, *Kokke/tiny-AES-c*, Apr. 19, 2023. [Online]. Available: https://github.com/kokke/tiny-AES-c (visited on 04/19/2023).

[74] *Libtomcrypt*, libtom, Apr. 19, 2023. [Online]. Available: https://github.com/libtom/libtomcrypt (visited on 04/19/2023).

[75] *README for Mbed TLS*, Mbed TLS, Apr. 19, 2023. [Online]. Available: https://github.com/Mbed-TLS/mbedtls (visited on 04/19/2023).

[76] "Poisoning - The Rustonomicon," [Online]. Available: https://doc.rust-lang.org/nomicon/poisoning.html (visited on 05/29/2023).

[77] "Std::option - Rust," [Online]. Available: https://doc.rust-lang.org/std/option/ (visited on 05/30/2023).