CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Performance Evaluation of a Hardware Security Module in Vehicles

Master's thesis in Computer Systems and Networks (MPCSN)

MICHEL FOLKEMARK
VIKTOR RYDBERG

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Performance Evaluation of a Hardware Security Module in Vehicles

MICHEL FOLKEMARK
VIKTOR RYDBERG

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Performance Evaluation of a Hardware Security Module in Vehicles
MICHEL FOLKEMARK
VIKTOR RYDBERG

Supervisor: Tomas Olovsson, Department of Computer Science and Engineering, Chalmers
Advisor: Per Hermansson, Volvo Group
Examiner: Magnus Almgren, Department of Computer Science and Engineering, Chalmers

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2021

Performance Evaluation of a Hardware Security Module in Vehicles
MICHEL FOLKEMARK, VIKTOR RYDBERG
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

With the rapidly increasing computerization of vehicles, cyber security has more and more become a very important aspect of modern automobiles. A vehicle consists of a large number of electronic control units (ECUs), all connected by a network. The ECUs and the communication between them need to be protected from illegal use by vehicle owners as well as cyber attacks from malicious actors. This protection is provided through the use of cryptographic techniques such as message encryption and authentication. The operations and calculations related to cryptography can be performed by the processor in the ECU itself, but that puts an additional strain on the limited computational capabilities of the ECU. A hardware security module (HSM) is a device that has hardware acceleration for cryptographic operations. Using an HSM alongside an ECU to perform cryptographic operations could thus offload the ECU, which means the computational power of the ECU can be used to perform its regular duties.

In this thesis, we have evaluated the use of HSMs in a vehicle environment with regards to performance. This included comparing the performance of an HSM versus a cryptographic solution implemented purely in software, as well as investigating security and performance trade-offs of different HSM configurations. It was found that using an HSM considerably improves performance of using cryptography, both in terms of increasing the speed of cryptographic operations as well as offloading the ECU CPU. Furthermore, it was also found that adding a message authetication code (MAC) to messages in the Controller Area Network (CAN) protocol results in a relatively large amount of overhead data, which consequently contributes significantly to the bus load. This makes it an infeasible method to use in many cases. However, according to our work using CAN-FD alleviates this problem considerably.

Keywords: Performance, cybersecurity, automotive, HSM, AES, CAN

Acknowledgements

The authors would like to express gratitude and thanks to Volvo Group Trucks for supporting this thesis by offering time and resources. We would especially like to thank our supervisor Per Hermansson for his continuous support, as well as Magnus Stålesjö for giving us the opportunity to conduct this thesis. At Chalmers we would like to thank our supervisor Tomas Olovsson for his valuable guidance throughout this thesis.

Michel Folkemark, Viktor Rydberg, Gothenburg, December 2021

Contents

List of Figures

List of Tables

Terminology

AES	Advanced Encryption Standard
AURIX	Automotive real-time integrated architecture
AUTOSAR	AUTomotive Open System ARchitecture
CAN	Controller Area Network
CAN FD	Controller Area Network Flexible Data-Rate
CBC	Cipher Block Chaining mode
CFB	Cipher Feedback mode
CMAC	Cipher Message Authentication Code
CTR	Counter mode
DES	Data Encryption Standard
ECB	Electronic Code Book mode
ECM	Engine Control Module
ECU	Electronic Control Unit
GCM	Galois Counter mode
HSM	Hardware Security Module
IoT	Internet of Things
IV	Initialization Vector
IVN	In-Vehicle Network
MAC	Message Authentication Code
MITM	Man-in-the-middle
RNG	Random Number Generator
RTOS	Real-Time Operating System
TRNG	True Random Number Generator
OFB	Output Feedback mode
OPC	Operation code
SHA	Secure Hash Algorithms
SHE	Secure Hardware Extension
SSL	Secure Sockets Layer
TLS	Transport Layer Security
V2X	Vehicle to everything

1

Introduction

Vehicles are today becoming more and more complex as there is a growth of on-board devices and advanced functions provided by the vehicle, such as connectivity-based services and autonomous driving. There is an increasing need for more advanced vehicle functionality in general, where many are related to connectivity. The modern vehicle provides much more than just a basic means of transportation. It is connected to the outside world in several ways; through a user's smartphone, to remote servers, other vehicles and infrastructure devices.

In the future, the rise of connectivity will increase even more with new technologies such as Vehicle-to-Everything (V2X), which for example includes Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I). But also other technologies such as cloud computing, software over-the-air, and autonomous driving will impact the connectivity aspect.

These general developments come with many implications. One significant implication is the increase of interfaces to the vehicle, which is particularly related to the connectivity aspect. It is important to recognize how connectivity poses a challenge to security, as more interfaces leads to more potential attack vectors.

Connected vehicles have various entry points (interfaces), which all are potential points of attack. Attack paths can range from protocol attacks over-the-air to cryptographic attacks on hardware level. It is important to note that all the possible theoretical security threats are relevant and need to be addressed. These threats can e.g., be categorized in different models such as confidentiality, integrity, availability (CIA). Attack targets can for example be private data, to compromise safety and access control. It is worth noting that damages can range from passenger safety to damages of vehicle components and proprietary theft. Note that several different types of attacks are applicable to this area. Security threats need to be minimized, but this is often challenge and sometimes might be neglected to some extent. Thus, security is relevant and will likely be even more son in the future.

The existing system architectures, technologies and research must keep up with these challenges. The higher demand on security in vehicles is a challenge since a vehicle is a very complex system and has limitations. For example, one important limitation is real-time demands on communication, which is crucial for safety reasons. Furthermore, there are some communication links which are relatively slow in

bandwidth and small in possible data payload size. Based on all this, many security solutions might be unpractical as they could potentially generate too much overhead (e.g., authentication through signatures).

The more critical the system and data, the more important it is with security measures, e.g., the use of dedicated hardware for security. One such hardware is a Hardware Security Module (HSM), which has optimized cryptographic hardware (crypto accelerators) and secured storage (cryptographic keys, counters, data).

A more general name for an HSM is Hardware Trust Anchor (HTA). Some of the common features of an HTA are to protect sensitive data (e.g., cryptographic material) in such a way that software can not manipulate it, as well as to provide cryptographic functions that offload the host controller.

As stated earlier, a vehicle is a quite complex and restricted system. The question is how well suited an HSM is for providing stronger and efficient security in a vehicle, such as protection for the electronic control units (ECUs) in the in-vehicle network. The ECUs are the backbone of the in-vehicle communication and function control.

Up to recently, the commercial solution is to use embedded software (e.g., a software module) for providing the security in the in-vehicle network. For example, the AUTomotive Open System Architecture (AUTOSAR) is one such embedded software which can enable such functionality [1] [2]. It provides functionalities such as cryptographic functionality, including handling the storage of cryptographic keys. However, this solution creates additional load on the host controller. Without an HSM, cryptographic operations need to be calculated with a software library and if the software library is synchronous, the ECU will be blocked for the duration of the cryptographic operation.

For the last ten years there has been extensive research on HSMs specifically for a vehicle environment. The EVITA project [3] has defined a categorization of HSM by profiles: HSM full, HSM medium, and HSM small. Particularly, the HSM full provides features such as hardware accelerators for asymmetric cryptography, complex block ciphers as well as providing the highest performance. The idea with an HSM is to directly increase security, but also creating the prerequisites for stronger security w.r.t. performance.

The EU-funded research project EVITA [3], in which several companies from the auto-industry such as BMW, Infineon, ESCRYPT, and Bosch were involved, is relevant to our work. The following quote (taken from the EVITA website [3]) describes the objective of the EVITA project:

“Secure and trustworthy intra-vehicular communication is the basis for trustworthy communication among cars or between cars and the infrastructure. Therefore, the objective of the EVITA project is to design, verify, and prototype an architecture for automotive on-board networks where security-relevant components are protected against tampering and sensitive data are protected against compromise when transferred inside a vehicle.”

EVITA has developed detailed guidelines for design, verification, and prototyping of various security architectures for ECUs in the automotive industry. One such guideline is that all critical ECUs should be equipped with a chip that contains a dedicated HSM as well as the CPU. The EVITA project ended in 2012, and it was out of the scope of that project to do further implementation and testing when it comes to more real-life deployment.

The development of HSMs designed for vehicles have gotten so far that they are commercially available. Furthermore, the automobile market is in a early phase where some early adopters have started to integrate HSMs as a security solution for their ECUs.

1.1 Purpose

The general goal of this thesis is to study and create the prerequisites for stronger security in the in-vehicle network, especially the Controller Area Network protocol (CAN) w.r.t. performance.

In more detail, the purpose of this thesis is to assess and evaluate the use of HSMs as a cyber security solution for a vehicle environment in terms of performance. This includes a comparison of using an HSM versus only using the ECU together with a software cryptographic library.

This thesis also assesses limits on security in CAN communication with regards to performance. This is done by comparing security versus performance trade-offs of using an HSM. Since the CAN protocol has low bandwidth and small maximum data payload size (8 bytes), there likely could be a restriction on security because of this.

When it comes to which data that should be secured, the long-term goal is to efficiently encrypt the CAN data payload for high priority security messages. However, in our work we only perform encryption on arbitrary data outside of the CAN context. Nevertheless, we do provide theoretical bus load calculations for CAN.

1.2 Research Questions

In this section, we specify the research questions for this thesis. The first two research questions are the main focus of this thesis, whereas the third is of more secondary focus.

1.2.1 Research Question 1: What is the cryptographic performance of an HSM and how does it compare to the performance of executing cryptographic operations in software?

We want to investigate possible performance gains when using an HSM in a vehicle. In order to assess this, it is preferable to have a base case to compare the HSM to. The most natural choice is to use a solution that performs cryptographic operations purely in software (by the ECU). Therefore, as the base case, we chose to use a cryptographic module implemented in software. More specifically, we use the cryptographic library wolfCrypt, which is part of the wolfSSL embedded TLS library [4]. The same platform – the AURIX microcontroller (refer to section 2.4.1) – is used both for HSM and software cryptography.

1.2.2 Research Question 2: What are the security and performance trade-offs of different HSM configurations?

We want to take different HSM configurations of varying security level and measure the performance of these configurations in relation to each other. This way, we can establish what limitations there are on security with regards to performance. As an example, a certain security level may not be achievable if the performance costs to sustain it are too high in relation to some set performance requirements; for instance if the security level infers a bus load that exceeds a maximum allowed bus load. Moreover, we also want to investigate whether there may exist a "sweet spot" where the performance and security trade-offs are optimal.

To be more specific, when we say different HSM configurations, we are referring to different combinations of parameters that make the security provided by the HSM more or less robust. This includes using different key lengths, different cipher modes, different algorithm combinations – e.g., using encryption alone versus using encryption and MAC together.

1.3 Delimitations

In order to accomplish the main objectives of the thesis within the given time frame, some delimitations have been made:

- **Testing is not performed on a real vehicle:** There is not any possibility of doing any work on a real vehicle. Testing is limited to a testbed.
- **Asymmetric encryption (incl. signatures) is not considered:** There are two versions of Infineon AURIX modules provided: AURIX TC299 [5] and AURIX TC3xx [5]. Although the TC3XX provides hardware acceleration for asymmetric encryption, there is no stable software stack available to be able to work efficiently with the TC3XX. Working with the TC3XX would thus require a considerably larger effort, which would take time away from the

main objectives of the thesis.

- **Securing external communication is not considered:** In the scope of this thesis, only security for internal communication was interesting to look at. Therefore, external communication, such as communication to a back-end server or V2V/V2E communication, is not considered.
- **The work is restricted to the CAN protocol:** There is some possibility of using Ethernet instead of CAN as the communication protocol for in-vehicle communication. But since CAN is much more established and also likely to remain the de facto standard for in-vehicle networks for the foreseeable future, we chose to only focus on CAN.

1.4 Thesis Contributions

This project has resulted in deeper knowledge when it comes to how effective the use of HSMs is as a cyber security solution in a real vehicle environment.

We compare the performance of carrying out cryptographic operations in hardware versus doing it purely in software, in order to establish the difference in performance between the two methods. This provides guidance for which option to choose, not only in a vehicle environment but also in general.

Furthermore, we investigate what effects the use of different cryptographic algorithms and cipher modes have on performance, especially when it comes to bus load and the CAN protocol. This can provide some insights as to what algorithms and cipher modes are the most suitable to use, not only in a vehicle environment but also in general.

Lastly, we also provide detailed methodology on how to test cryptographic performance in general.

1.5 Disposition of the Thesis

The thesis is organised as follows:

- **Chapter 1 - Introduction** Contains an introduction to the thesis, the purpose, the problem formulation, and the scope of the thesis.
- **Chapter 2 - Background** Provides background knowledge to the central topics of this thesis. This includes specifics of an HSM, the AURIX microcontroller, vehicle infrastructure and more.
- **Chapter 3 - Literature Overview** This section provides an overview of the literature related to the work done in this thesis. It also provides a description of the research methodology that was used in order to find relevant literature.
- **Chapter 4 - Method** Describes the methods for answering the posed research

questions, i.e., how to evaluate the performance of an HSM. It also includes a methodology for how to conduct testing for performance. Lastly, this chapter also contains information about how testing was actually performed and details regarding the used testbed.

- **Chapter 5 - Results** This chapter consists of the results yielded from the testing of performance, on the testbed.
- **Chapter 6 - Discussion** A discussion about the findings from the thesis work.
- **Chapter 7 - Future work** This chapter consists of information about what more work could be done on this topic in the future.
- **Chapter 8 - Conclusion** In this chapter the conclusions of the thesis are presented.
- **Appendix - A, B, C** In these chapters extra details are provided regarding A) Possible extension to our work B) Background knowledge C) Results. The appendix acts as an additional source for more information.

2

Background

This chapter introduces the most fundamental underlying concepts and technical background of this thesis. The chapter is organized as follows. Section 2.1 gives a brief introduction on what a hardware security module is. Section 2.2 explains the main details of the Advanced Encryption Standard; the encryption algorithm used in this thesis. Section 2.3 provides an introduction on the vehicle infrastructure and its key components, such as the Electronic Control Unit and the Controller Area Network. Section 2.4 describes the hardware used in this thesis, including which specific microcontroller and HSM that are used. Section 2.5 describes wolfSSL/-wolfCrypt; the software cryptographic library used in this thesis. Lastly, section 2.6 gives an introduction on the performance metrics that are considered in this thesis.

2.1 Hardware Security Module (HSM)

A Hardware Security Module (HSM) is a physical computing device which can provide functionality for cyber security purposes, including authentication, authorization, data confidentiality and data integrity. One of the main functions of an HSM is to provide secure management of cryptographic keys. This includes safe storage of keys, which is made possible by encapsulating the keys in an isolated environment which has mechanisms that prevent tampering.

Another vital function of an HSM is to provide hardware acceleration for various cryptographic operations in order to increase the speed by which they are performed. Typical cryptographic operations which could benefit from hardware acceleration include key generation, encryption, decryption, signature generation and hash functions.

An HSM can be used in a wide variety of environments which could benefit from enhanced cryptographic functionality. This includes environments such as vehicles, web servers, card payment systems, cloud infrastructure and more. An HSM could come as a stand-alone device or plug-in card, or it could be built into other devices, as in the case with some Infineon AURIX microcontrollers.

2.2 Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) is a standardized encryption algorithm established by NIST in 2001. It replaced the Data Encryption Standard (DES) as the de facto standard for encryption of electronic data. It is a symmetric-key algorithm, i.e., the same key is used for both encryption and decryption. The algorithm comes in 3 different versions: AES-128, AES-192 and AES-256; where the versions differ in the key length used (128, 192 and 256 bits, respectively). The HSM studied in this thesis uses AES-128 as its encryption algorithm.

The AES algorithm is a block cipher, meaning that it operates on data of fixed size, i.e., blocks, where each block contains 128 bits. In order for the algorithm to be applied to data of size larger than 128 bits, different so called modes of operation have to be used.

2.2.1 Modes of Operation

In this thesis, we consider four different modes of operation. The choice was firstly based on the prevalence of the modes, as we wanted to include commonly used modes. But most important was to include modes of different security levels, so that the choice of mode of operation could be used as a parameter in the security/performance trade-off analysis. In this thesis, we also refer to mode of operation as "cipher mode".

Electronic Codebook (ECB) Mode

The most basic mode of operation is the electronic codebook (ECB) mode, where each block of data is simply encrypted/decrypted separately. This method has a major drawback in that the same plaintext always gets encrypted to the same ciphertext, which results in data patterns not being hidden efficiently. Therefore, ECB is not recommended for use in cryptographic protocols.

In order to solve this problem, a so called initialization vector (IV) can be included in the encryption process. An IV is basically an arbitrary number of some size. In order for the usage of an IV to be secure, the number should be unpredictable. An IV is therefore usually a randomly generated number. Note that the term nonce can be used interchangeably with IV.

The technique of using a nonce/IV is utilized in several modes of operation, including cipher block chaining mode and counter mode. When these modes are used; even if identical blocks of plaintext are encrypted with the same key, they will result in different blocks of ciphertext.

Cipher Block Chaining (CBC) Mode

In cipher block chaining (CBC) mode, a plaintext block is XORed with the preceding ciphertext block before undergoing encryption. This way, a ciphertext block is not only dependent on the corresponding plaintext block, but also on all previously

encrypted blocks. An IV is used to XOR with the first block. If the IV is random and unique, this method ensures that identical blocks of plaintext will be encrypted into different blocks of ciphertext.

Counter (CTR) Mode

In counter (CTR) mode, a counter is used in the encryption process. The counter is a value that has to be unique for each plaintext block that is encrypted. In general, the counter is initialized to some value, then incremented by 1 for each subsequent plaintext block. The initial counter is equivalent to an IV.

In CTR mode, the plaintext is not encrypted directly. Instead, the counter is encrypted; the ciphertext is then produced by XORing the plaintext with the encrypted counter. For this reason, the counter has to be equal to the block size, i.e., 128 bits. Just like for CBC mode, this method ensures that identical blocks of plaintext will be encrypted into different blocks of ciphertext.

CTR can be shown to be at least as secure as CBC [6]. Therefore, in this thesis, we consider CTR as more secure than CBC.

Even though CBC and CTR are more secure alternatives to ECB, they have some drawbacks. One of them is that these modes do not protect the integrity of data. For example, using a MITM attack, an attacker can corrupt a piece of ciphertext without the recipient of the data being aware of the corruption. This problem is solved by Galois/Counter Mode, a mode of operation that builds on CTR.

Galois/Counter Mode (GCM)

Galois/Counter Mode (GCM) is a mode of operation that extends CTR by also including message authentication into the encryption process. This is done by adding a message authentication code (MAC) to the encrypted data. This way, if some ciphertext is corrupted in an MITM attack, this will be detected by the message recipient, as the now corrupted data will not match the MAC. GCM is therefore considered as the most secure mode of operation used in this thesis.

2.2.2 Unpredictability of Cryptographic Material

The security of AES does not only depend on aspects such as key length and choice of block cipher mode of operation, but also on good cryptographic material, i.e. unpredictable random numbers. This is where the TRNG comes into the picture. In this section we give some background as to why unpredictability is relevant in a security perspective, and to some extent also a performance perspective.

Randomness or unpredictability is required due to that it is vital when it comes to making it harder to break the keys. For example, if we have a key generation algorithm that has the unique ID number of the ECU (which could in reality be neither completely obfuscated or secured) as input, which then apply padding and increment it resulting in a unique 128-bit symmetric key that the ECU can use.

This algorithm together with the assumption of unique input would in principle guarantee uniqueness. However, it would be predictable. If one knows the unique ID number of a given ECU together with knowledge how the algorithm functions, then one could easily calculate the symmetric key. Thus, randomness is required in a security perspective since it makes it harder for an adversary. The goal is to make a brute-force search the only viable option for an adversary. This also applies to IVs due to the same reasoning.

Note that the TRNG will not guarantee unique cryptographic material, i.e. it is still possible that the TRNG will generate the same random number multiple times. The TRNG only provides true random numbers, i.e. unpredictable numbers.

Generating good cryptographic material costs performance and the frequency of the generation becomes a vital factor to consider. For example, if each communication session between two nodes should have a unique symmetric key. Then when a communication session is initiated, the node that initiated the communication needs to generate a unique and preferably an unpredictable key. The node could fetch an unpredictable number for the key from the TRNG, then proceed along with some key exchange protocol so that both nodes have the key. However, this only occurs when a new session of communication is initiated, or when session keys need to be updated. Therefore, it is likely that the key update cycle time is relevant to consider. If the performance cost of the TRNG or the overall performance would be deemed to high, an alternative would be to use the same symmetric key for all communication sessions or unique keys for some groups of nodes. In summary, the number of unique keys and the key update cycle time is important for the TRNG performance aspect.

2.3 Vehicle Infrastructure

The modern vehicle is a computer system with a high complexity level, and in general it comes with some constraints in terms of performance. The whole vehicle network is rather complex; there are many different communication protocols deployed (both on a physical and virtual level), several subnetworks (aka. levels), gateways, and many nodes in the network (e.g., different Electronic Control Units/Modules).

Thus, it is hard to fully comprehend in detail how a modern vehicle functions. This is also one reason why security tends to become complex. Furthermore, the clear common trend is that the number of functions/services are increasing along with electrical components and devices. The possible security consequences are important, especially for connected vehicles. A connected vehicle is usually connected to some external networks (i.e., remote severs), in order to provide remote services such as updates and diagnostics. In general this leads to higher potential security threats to the internal network.

2.3.1 Electronic Control Unit (ECU)

The most fundamental device in a vehicle is the Electronic Control Unit (ECU), also known as electronic control module. An ECU has the primary function of controlling one or more of the electrical systems in a vehicle. For example, the ECU responsible for the engine functionality is called Engine Control Unit/Module (ECM). The ECM controls a series of actuators on the engine, in order to perform tasks such as optimizing engine performance based on some variables. The ECM achieves this through reading values from a multitude of sensors. Then the ECM needs to perform some interpretation e.g., using lookup tables, and based on that adjust engine actuators.

To give another example, a vehicle has crash sensors located around itself, which informs the relevant ECU when a crash has occurred. Based on this input the ECU can measure the current speed of the vehicle and decide to launch the airbags or not. This is a safety critical function, and this whole process happens in a few milliseconds.

Continuing on another safety related example, a vehicle is accelerating and suddenly an ECU triggers the airbag. This situation could occur due to a malicious attack. For example, unencrypted data between the ECU and the sensors can be read and easily manipulated. In this case it is fully feasible for an attacker to create a spoofed message that can result in triggering the airbag.

ECUs can differ greatly as they control a wide range of systems/functions. In other words, the demands differ significantly and thus the ECUs differ greatly. Thus, it is natural that different ECUs require different levels of security measures. For example, authentication might not be necessary or worth to implement for every single ECU and every type of message, as it is likely the case that the costs in terms of performance would outweigh the gains in terms of security. Therefore, in the bigger picture one need to take into account both the perspectives of performance and security. Then based on the decided safety and security requirements choose an appropriate level of security in relation to performance costs. For example, it is most likely the case that not every message type in an in-vehicle network requires message authentication, both in a security and safety perspective.

2.3.2 In-Vehicle Network (IVN)

This section provides an overview of an in-vehicle network (IVN). Note that since most of the work done in this thesis is restricted to the delimited domain of an ECU, the purpose of this section is primarily to provide a context for this work.

A modern vehicle contains a so called in-vehicle network, which consists of different ECUs. As described earlier the ECUs are in fact responsible for most of the functionality in the vehicle. Currently a modern vehicle can have well over a hundred ECUs onboard. However, the normal case is usually a lower number. The functionality that an ECU provides can, as described earlier, range from everything from unlocking doors to more advanced features such as engine control, automatic

brake systems and Advanced Driver-Assistance Systems (ADAS). Thus, there are clearly different demands, in terms of performance, security and safety, on the ECU depending on the functionality it is responsible for providing.

Vehicle functions are divided into many different systems and sub-systems in the vehicle network. All these different systems are together responsible for providing the passenger the different functionalities in a vehicle, such as infotainment, comfort, safety and powertrain control.

To make it more complex, there can also exist local/private and global networks in a domain. This creates several layers of communication. In Figure 2.1 a more detailed illustration is shown, with emphasis on an arbitrary system/domain. Examples of such systems/domains are e.g., the group of ECUs responsible for the powertrain system. Other systems/domains are also applicable for this model. However, it is worth noting that they might utilize a different communication protocol. This abstracted system has CAN (FD) as its communication link/protocol in the arbitrary domain. Furthermore, the ECUs of this arbitrary domain are connected to both global CAN networks as well as local/private CAN networks. However, the specific details may differ greatly between implementations and also depend on the sub-network, such as powertrain, vehicle safety, comfort, infotainment and telematics.

Note that there are multiple other domains connected to the Vehicle Master Control Unit (VMCU, see Appendix B.1), alternatively a Central Gateway, either trough domain controllers (gateway units) or directly through the domain's global networks. For example, in this abstraction the arbitrary sub-network does not have a gateway. Instead it is as Figure 2.1 shows; the domain's global CAN networks are directly connected to the VMCU. The VMCU in principle acts as a CGW. But the VMCU can be assumed to have no security mechanisms, it only has a static routing table. Furthermore, it is the VMCU that has the On-Board Diagnostics (OBD) contact. For details on the functionality of the different nodes see Appendix B.1.

With the assumption of no security in the CGW and no domain controllers (gateway units) in this model, there are no layers of security. Thus, in this model security is primary based on the Telematic Gateway (TGW or TCU, see Appendix B.1) capabilities. However, how advanced the TGW is can greatly vary. For example, it might only have a basic firewall, no HSM, and no IDS/IPS.

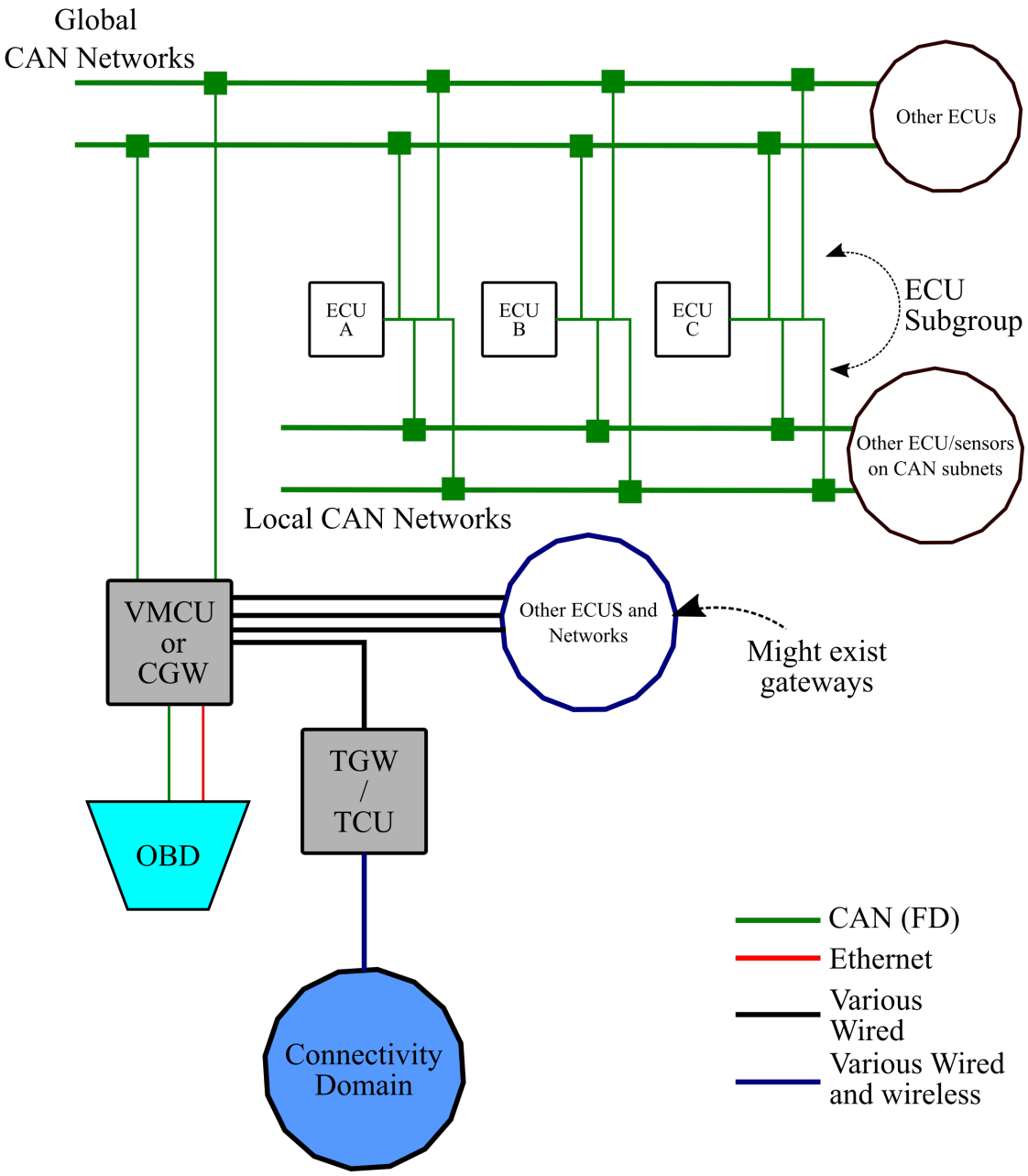

Figure 2.1: An illustration of a reality-based conceptual in-vehicle network. Emphasis is on an abstracted sub-network, for example powertrain. The Vehicle Master Control Unit (VMCU) in principle acts as a central gateway for the in-vehicle network.

2.3.3 Controller Area Network (CAN)

CAN is a serial communication technology and is a robust vehicle bus standard (i.e., nodes connected through a bus), that is especially known to be deployed for reliable data exchange between ECUs in vehicles (in-vehicle network) and satisfies the real-time requirements of target usage areas in vehicles. The communication is done through messages, where each message can contain 64 bits of data and an error correction code. The CAN bus is widely deployed in vehicles, e.g., to enable networking between the ECUs and elsewhere. As normally with bus protocols, only one node/device at a time can send data and everyone receives it. See Section B.2.1 for further details on CAN.

CAN Network

The structure of the CAN network is quite straightforward. The CAN network consists of a number of nodes which are all linked via a physical transmission medium, the CAN bus. The CAN network is usually based on a line topology with a linear bus, to which a a number of ECUs are connected through their respective CAN interface.

2.3.4 CAN FD

As the names implies; the main difference between CAN and CAN FD is the Flexible Data-rate (FD). CAN FD supports dual bit rates. First, it has the nominal data-rate (utilized during arbitration) limited up to 1 Mbit/s, as in the classical CAN protocol. And secondly it has the flexible data-rate, which depends on the actual network topology and its nodes. Nominally it can go up to 10 Mbit/s, but a more realistic data-rate is lower. Still, it is a significant increase in bandwidth compared to classical CAN. CAN FD also supports up to 64 bytes of data per data frame, compared to classical CAN which only supports 8 bytes of data.

In more detail, CAN FD retains the standard CAN bus bandwidth during arbitration, and can increase the bandwidth for the actual data phase. The CAN FD will return back to the nominal (arbitration) bandwidth, especially the normal bit time, at the CRC delimited, before the receiver nodes send their ACK bits.

In conclusion, the CAN FD protocol has an adjusted CAN data frame, which enables flexible larger data payloads, and flexible higher data-rates without requiring any changes of the physical layer of CAN. Thus, ECUs with CAN FD can dynamically switch to different data-rates and with different message sizes, both which can be significantly better than classical CAN. So with CAN FD it is possible to have a lower protocol overhead and higher efficiency in communication.

2.4 Hardware

In this section, background information is presented regarding hardware involved in the project. It mostly involves the embedded devices and hardware tools in the

testbed.

2.4.1 AURIX

In this thesis, a specific microcontroller of the brand AURIX is used. The AURIX series is Infineon's family of microcontrollers, designed for embedded, automotive and industrial applications. It builds on Infineon's TriCore CPU technology. In the context of automotives, an AURIX microcontroller is commonly used as the main computing unit in an ECU.

AURIX microcontrollers target a large variety of automotive applications, including control of combustion engines, transmission control units, braking systems and airbags.

AURIX microcontrollers are developed according to multiple industry standards, including ISO 26262 (automotive standard) and IEC 61508 (functional safety standard). This ensures the functional safety and security properties of the microcontroller. For this purpose, the microcontroller hosts a wide range of hardware peripherals and safety mechanisms which aid in aspects such as safe computing, safe intro chip communication, safe data and code storage and more. AURIX microcontrollers also contain support for AUTOSAR.

The first generation of AURIX microcontrollers is called TC2xx. The different models of the TC2xx series mainly differ in the number of cores they have, clock frequency, memory size and peripheral set (including whether they have an HSM or not). In this thesis the TC299TF is used, which is the most advanced model in the TC2xx series, featuring three 32-bit super-scalar TriCore V1.6.1 CPUs running up to 300 MHz, 8MB of flash memory as well as an HSM. It has support for a wide variety of communication protocols, including Ethernet, CAN, LIN and CAN FD.

2.4.2 AURIX HSM - TC2xx

The production of the first generation HSMs of Infineon Technologies started in 2015. The HSM inside AURIX provides a robust response to many cyber security requirements. For example, an important aspect is that it is highly flexible and programmable. To start off, the AURIX HSM functions as an anchor of trust through a separated logical protection domain. It supports access control, data confidentiality, integrity and authenticity needs (e.g. secured boot). All the following information is complied only from public documentation [7] [5] [8] [9] [10]. Note that all the deeper information and technical details are confidential, due to security and proprietary reasons.

The integrated HSM in the 32-bit TriCore AURIX TC2xx is a medium EVITA compliant HSM. A dedicated core for security functionality, which is fully detached, provides a significant advantage, since it allows the creation of a trusted execution environment with a separated domain. The HSM AURIX TC2xx series is a medium, whereas the HSM AURIX TC3xx is a full.

The HSM is designed to perform cryptographic operations and contains functions such as a True Random Number Generator (TRNG) and an AES (128-bit) crypto accelerator for encryption and decryption. The HSM has a 32-bit ARM Cortex-M3 processor with 100 MHz CPU clock frequency. Furthermore, it supports AES CMAC with a minimum rate of 25 MBytes/s. Thus, the HSM offloads the ECU's resources through its optimized hardware. For example, AES is computed through embedded hardware accelerators, which are optimized for AES.

The hypothesis is that an HSM with its dedicated and optimized hardware has better performance compared to a cryptographic software solution. In other words, that cryptographic algorithms implemented in hardware should have lower latency of cryptographic operations and higher throughput compared to a software solution. These are vital performance aspects which need to be taken into consideration when securing the CAN protocol. As one cannot disregard the risk of e.g., excessive delays in the communication, or overloading the main CPU (ECU).

Other major features are to provide a secured storage primarily for cryptographic keys. This includes a Memory Protection Unit (MPU) and the secure key storage is in a separate HSM P/DFlash portion.

Other security aspects that are worth mentioning are that there is support for secured boot and also protection on the AURIX debug interface (debugger protection).

Other more specific security requirements (for TC2xx) such as HASH or asymmetric cryptography, can still be implemented in software (SHE and SW). Thus, a combination of hardware and software is a functional solution.

Different security demands implies the need for different specialized dedicated hardware. Thus, it is important to understand what the security requirements for a specific ECU is, in order to decide on what dedicated hardware is needed.

The main general goal of this project is to study and test how effective a solution HSMs are for securing on-board communication (CAN). The security of the CAN protcol can be improved using the AURIX HSM. The important security aspects confidentiality, integrity, and availability (CIA) are not built into the protocol. Availability is the only aspect that is somewhat built in through the arbitration process of the CAN protocol. However, it is not secure and can be broken by an attacker. For example, a connected pirate device can relatively easily shut down the CAN bus by continuously sending out high priority messages.

The use of an AURIX HSM can help in solving the overall security situation, but it does not offer a complete solution. For example, one can use the HSM to encrypt CAN messages and use CMAC to provide some authentication.

AES 128-bit Cryptographic Accelerator

AES is implemented and optimized in hardware for fast encryption and decryption via 128-bit key length. The cryptographic accelerator operates on single 128-bit data blocks (plaintext or ciphertext), or on a multitude of 128-bit data blocks. Thus, the

AES algorithm performs cryptographic operations in blocks of 128 bits of data.

There are several supported block cipher modes of operations in AURIX HSM:

- Electronic Code Book (ECB)
- Cipher Block Chaining (CBC)
- 32 bit Counter (CTR)
- OFB (Output Feedback)
- CFB (Cipher Feedback)
- Galois Counter Mode (GCM)
- XTX (TCB - Tweaked Code Book based in XEX with CTS - Ciphertext Stealing)

AES CMAC

Cipher-based Message Authentication Code (CMAC) is a block cipher-based message authentication code algorithm. It is based on symmetrical encryption, like the CBC-MAC algorithm. CMAC can be used to provide authentication. For example, the secure boot of the HSM uses CMAC for tamper detection and prevention.

True Random Number Generator (TRNG)

TRNG is implemented and optimized in hardware. As the name implies it generates random numbers. These random numbers can be used to generate cryptographic keys and be used in different protocols (such as challenges and padding bytes) that supports several different protocols. This TRNG is fully compliant with the AIS 20/31 standard.

Secured key storage in separate HSM P/Dflash portion

The HSM offers secure storage for keys, data, and counters in a separate HSM-DFLASH (DF1) portion of 64 KBytes. It is worth noting that alternative secure key storage is feasible in dedicated HSM-PFLASH sections.

A dedicated HSM Data Flash (DFlash) allows executing the TriCore (host/ECU) application to fetch, read code/data from Program Flash (PFlash) while updating secured non-volatile information. Furthermore, the segregation for the sensitive information stored in the HSM-DFlash can be enforced using exclusive access features, which allows read/write access to the core only.

HSM Integration

The HSM is connected to the host device (TriCore/ECU) via the System Peripheral Bus (SPB). There the HSM acts as a system on-chip and is a bus master. Further-

more, there exists a firewall that helps in protecting the HSM's internal resources from unwanted access. Lastly, the program and data flash (P/DFlash) of the HSM is shared with the host. But they can be protected through an exclusive access from the TriCore and other masters accesses in the system.

2nd Generation HSM - AURIX TC3xx

The second generation of the HSM is TC3xx, of which the production was started in 2019. It is a full HSM by the EVITA compliance. This means that it offers both asymmetric and symmertric cryptographic accelerators. Thus, it is suitable for both securing internal (on-board) and external communication. The central improvements compared to the first generation is asymmetric cryptographic accelerators (PKC ECC 256-bit), HASH SHA-2, and a secure watchdog.

2.4.3 Debugger - Lauterbach TRACE32

The debugger consists of Lauterbach Power Debug module (generic, a universal base module) and a debug cable for the specific case (in this case AURIX TriCore). The debugger acts as a communication link between a PC (e.g., USB 3 or Ethernet interface is possible with an adapter). [11]

Then there is software for the debugger which is Lauterbach TRACE32. It supports debugging for up to three TriCore cores and all its auxiliary controllers, and has both a GUI menu and a command line. TRACE32 has several vital features such as High-Level-Language (HLL) Debugging, which includes debugging aspects such as reading/writing to registers, memory, variables (incl. structs, linked lists) and also provides the ability to use breakpoints and advanced stepping. Secondly, it provides advanced breakpoints such as in software, on-chip, program and conditional breakpoints. Note that there exist special debuggers with extended functionality.

Furthermore, the debugger has C/C++ debugging, AUTOSAR-OS aware debugging, FLASH programming, access to all peripheral devices, and more. But most importantly for our work is that it supports debugging of all auxiliary controllers which includes HSM (Cortex-M debugger). Thus, the debugger enables direct access and provides many debugging features.

2.5 wolfSSL/wolfCrypt

wolfSSL is a lightweight and portable SSL/TLS library written in the C programming language. It is primarily targeted at IoT, embedded, and RTOS environments because of its size, speed, and feature set, but it also works well in desktop, enterprise, and cloud environments.

wolfCrypt is a cryptographic library which serves as the cryptographic engine behind wolfSSL. It is a subcomponent of the wolfSSL library. Multiple FIPS Certificates have been issued for wolfCrypt, and it is used in millions of applications and devices worldwide [4].

wolfCrypt features many of the common cryptographic algorithms today. This includes hash functions such as SHA up to 512 bits, symmetric encryption algorithms such as AES (including many cipher modes) as well as public key algorithms such as RSA and various ECDH (Elliptic Curve Diffie-Hellman) variants.

In section 4.3.4 we elaborate on the features of wolfCrypt that motivated us to choose it over other software cryptographic libraries that are available.

2.6 Performance and Metrics

The performance of an HSM can be measured in a variety of ways. In Chapter 3, a collection of papers are presented which propose various metrics to use when evaluating the performance of HSMs and other related systems/modules. This section provides an introduction to some of these metrics from a general viewpoint. In Chapter 4, we explain more in depth how these metrics apply to the context of this thesis.

2.6.1 CPU Load

CPU load refers the amount of computational work that the CPU performs or has to perform, and is often calculated as an average over a period of time. CPU load can be defined in two ways. One way is by how much time the CPU is used/is active, and is measured as the percentage of time that the CPU is doing work. Another way is by the number of processes using or waiting to use the CPU. For example, if one process is using the CPU, the CPU load is 1 (or 100%). Whereas if one process is using the CPU and a second process is waiting to use the CPU, the CPU load is 2 (or 200%), and so on. For the context of this thesis, the first definition is the most relevant, as we are not so concerned about the number of processes that are currently active and wanting to use the CPU, but instead about how much the CPU is utilized on average over a certain period of time.

A major objective of an HSM is to reduce the computational load on a system by offloading cryptographic operations performed by the system onto the HSM. Therefore, the CPU load of the offloaded system is a relevant metric to use when evaluating the performance of an HSM.

2.6.2 Bus Load

In the context of computer architecture, a bus is a communication network through which data can be transferred between multiple components. The bus load is then the amount of data being transferred on the bus. Bus load can be measured as a percentage of the maximum bus load, i.e, if half of the bus capacity is being used, the bus load is 50%. In the context of an in-vehicle network and the CAN bus specifically, the bus load is the amount of data currently being transferred on the CAN bus.

The use of cryptographic mechanisms to secure messages is likely to result in an

increase in message size. This could be due to, e.g, added padding from message encryption as well as data overhead generated from using techniques such as MAC, hashing and signatures [12]. In the context of a CAN bus, this increase in message size will contribute to a higher bus load. Since a CAN bus has a limited capacity, measuring differences in bus load when using different cryptographic techniques and algorithms is thus a relevant metric to use in this thesis.

2.6.3 Memory Utilization

Memory utilization, or memory usage, simply refers to the amount of memory that is currently being used. This could be memory usage for a process, a program or the system as a whole, depending on the context.

When a system performs cryptographic operations, some amount of memory needs to be allocated for this purpose. Thus, when cryptographic operations are offloaded to an HSM, the system's memory usage is likely to be affected. Although the memory usage is likely to go down as a result of this, this may be counteracted by memory that needs to be allocated in order for the system to be able to communicate with the HSM. Moreover, different cryptographic algorithms require different amounts of memory, as shown in [13], which needs to be accounted for as well. Therefore, memory utilization is a relevant metric to use in this thesis.

2.6.4 Cryptographic Latency and Throughput

Cryptographic latency refers to the time it takes to perform cryptographic operations. More specifically, it refers to the time between when a cryptographic operation is started until it is finished. As an example, encryption latency is the time it takes to convert a piece of plaintext to ciphertext.

Cryptographic throughput is the rate at which cryptographic operations can be performed. Whereas cryptographic latency refers to the time it takes to perform a cryptographic operation on a single piece of data, cryptographic throughput is more concerned with a continuous flow of data, and the rate at which cryptographic operations can be applied to this data stream.

One of the main incentives of using a hardware security module is to speed up the execution of cryptographic operations. Therefore, cryptographic latency and throughput are key metrics for evaluating the performance of an HSM.

2.6.5 Communication Latency

In the context of a communications network, communication latency refers to the the time it takes for a message to travel from one point to another in the network. This time can be measured either as end-to-end – the time it takes for the message to travel from source to destination – or round trip time (RTT) – the time it takes for the message to travel from source to destination and then back to the source again.

Communication latency generally only accounts for message travel time, and does not include any processing of messages that may be performed at the source and/or the destination. Nonetheless, in the context of this thesis, it is more relevant to include cryptographic latency in the calculation of communication latency. This way, we can assess the overall effects that the application of cryptography has on communication delay. Therefore, communication latency is a relevant aspect to consider in this thesis.

Moreover, as stated earlier, applying cryptographic operations to messages may result in an increase in message size. If the data overhead from cryptographic operations is large enough, this may even result in a message having to be split up into multiple messages. This is therefore another effect of cryptography which could contribute to increasing communication latency.

2.6.6 Key/IV Generation Time (TRNG Latency)

Key/IV generation time simply refers to how fast a cryptographic key or IV can be generated. In general, generating a cryptographic key or IV involves the use of a random number generator (RNG) – such as a TRNG – to generate a random number, which then acts as the key/IV. With that in consideration, testing the key/IV generation time of a cryptographic module which uses a TRNG to generate random numbers (e.g., an HSM), is practically equivalent to testing the performance of the TRNG of the module. Therefore, in the context of this thesis, this metric could be further generalised to TRNG latency.

Secure cryptographic keys/IVs are essential when using cryptography. It does not matter too much if the cryptographic algorithm is secure, if in fact the key/IV used in conjunction with the algorithm is easily broken. If many keys/IVs need to be generated though, this process could have a toll on the performance of a system. Therefore, key/IV generation time (i.e., TRNG latency) is a relevant aspect to consider when evaluating the performance of an HSM.

2.6.7 Key Update Time

Key update refers to the process of updating a cryptographic session key for communication between two nodes. This could refer to both the initial process of establishing a session key, as well as the process of updating a stale session key. In either case, it involves some extra communication between the nodes, in order for them to agree on the new session key to use.

If the key update process is too inefficient, it could have too much of a negative impact on communication and communication latency. Therefore, this metric could be relevant to consider in this thesis.

3

Literature Overview

In this chapter, we summarize literature which is related to the work done in this thesis. This includes a section on related work, i.e., previously conducted research that is similar to the work done in this thesis. This is followed by a section describing the literature that was used for the purpose of eliciting which metrics to use in the performance evaluation. But first in this section we provide a description of the research methodology that was used in order to find relevant literature.

3.1 Research Methodology

As a source for finding research papers, Google Scholar was used as a search engine. Google Scholar covers all major databases and portals for scientific publications in the computer science field, including ACM Digital Library, IEEE Xplore, SpringerLink, ResearchGate and more. Therefore we considered it to be a sufficient tool for this purpose. For other types of literature, a lot was provided by Volvo, including documents about HSMs, relevant ISOs etc.

In order to increase the chance of finding every piece of literature that was relevant for this thesis, the literature study was done in a relatively structured way. The first step of the research strategy was to identify the most relevant keywords and keyword combinations and try many different permutations of these. As an example, for the metrics research this included keywords such as "hardware security module performance" and "vehicle can security performance". If these keywords did not result in finding enough relevant literature, the scope of the literature search was widened and more keywords were included.

For every keyword combination, the aim was to try and conduct the search quite exhaustively. The general approach was to continue the search until the last circa 20 results in Google Scholar either were 1): completely irrelevant, indicating that we had exhausted the relevant results of the keyword combination, or 2) contained the same (or more irrelevant) information as in previously found literature. Some combinations of terms gave no results, while some gave a subset/superset of results of previous search term combinations. When relevant papers were found, we also looked at the list of references the paper had, in order to see if any of the referenced literature could be relevant for our thesis.

In order to synchronize the literature search, each found publication was stored in a shared repository on Google Drive. Furthermore, every keyword combination that had already been searched for was noted in a document, along with how deep the search was, i.e., how many of the results had been investigated. This would help us for future reference, in case we wanted to continue the literature search at a later date.

The process of deciding which papers were considered relevant was done in multiple iterations, with both teams members participating in the process. At first, during the search process, the papers were only looked at from an overview perspective. This included quickly reading through the abstract to see if relevant keywords were mentioned, as well as scrolling through the document looking for images, diagrams and tables that seemed to contain relevant information. If a paper looked like it could be relevant, it was added to the shared repository. After the search process was done, each paper was then looked at more carefully. This included reading through the abstract and the introduction more thoroughly, in order to establish whether the paper truly was relevant or not. As a final step, some of the papers were also checked with the supervisor to get his view on the relevancy of the papers.

3.2 Related Work

This section details previously conducted research that is similar to the work done in this thesis. One particular research paper stands out in this regard, so therefore the first subsection is completely devoted to this paper. The last subsection describes other works that have some similarities with ours.

3.2.1 Wolf et al.

The most similar research to this thesis is the paper "Design, Implementation, and Evaluation of a Vehicular Hardware Security Module" by Wolf et al., from 2011 and is related to the EVITA project [14]. Their paper includes a performance analysis of an HSM with regards to throughput of cryptographic operations. More specifically, they compare the performance of the HSM with the performance of carrying out the cryptographic operations in pure software.

Our thesis differs and expands from their analysis in several ways. Firstly, our evaluation goes more in-depth when it comes to evaluating the performance. Wolf et al. mainly focus on throughput, whereas we take into account additional parameters such as CPU load and latency. Secondly, our thesis considers different combinations of algorithms for security, i.e., different modes of operation. Lastly, and perhaps most importantly, our work is more connected to real deployment in industry today, especially considering the fact that we use a market-ready solution, in contrast to Wolf et al. which implement their own HSM.

3.2.2 Other Similar Research

Seol et al. propose a secure cloud architecture with a hardware security module, which isolates cloud user data from potentially malicious privileged domains or cloud administrators [15]. As part of their work, they analyse the performance of this architecture by looking at the difference in data read/write throughput achieved with and without cryptographic operations.

In [16], Samir et al. detail their implementation of a hardware security module for IoT devices, which – depending on the available power resources – can adapt the security level provided by changing the encryption modes that are employed. Besides evaluating power usage, they also look at encryption latency and throughput of the different encryption modes used.

Hupp et al. present a hardware security module for protecting distributed energy resources on the modern electric grid [17]. As part of their evaluation of the HSM, they look at the end-to-end latency of TLS packets sent in the grid when using the HSM to perform cryptographic operations.

Lastly, in [18], Cifuentes et al. describe their implementation of a faux hardware security module to be used for handling digital signatures in the DNSSEC protocol. The HSM is not a traditional HSM, but is instead emulated on a distributed system of inexpensive commodity hardware. In their paper, they evaluate the performance of this HSM by looking at the speed by which it can generate signatures.

To summarize, these papers mostly differ from ours in that they do not put as much emphasis on doing a performance evaluation of an HSM. Rather, the performance evaluation has more of a secondary focus and deals with very few metrics. Furthermore, these papers do not base their research on commercial, market-ready solutions such as an AURIX. Instead, they more or less do their own implementation of an HSM, something which is especially true for Cifuentes et al., who base their work on a solution using commodity hardware.

3.3 Metrics Research

In order to evaluate the performance of an HSM, we needed to define how performance can be measured in this context. More specifically, which metrics that are relevant to consider for this purpose. Therefore, we decided to do a literature study in order to elicit relevant metrics for this thesis. In section 3.2 we have already detailed the research done on performance evaluations of HSMs in vehicles and in general, along with what performance aspects and metrics they touch on.

However, since the amount of literature that was found in this area was quite small, it motivated us to broaden the scope of the literature search and include other areas which could contain relevant information about performance evaluations in a security context. These areas included in-vehicle network security in general (e.g., in CAN), cryptographic accelerators and encryption algorithms in general. The

papers that were found in these areas (listed below) allowed us to not only find new relevant metrics, but also to justify metrics we already were intending to include in our performance evaluation.

3.3.1 In-Vehicle Network Security

The papers listed below provided us with additional insights to what metrics could be relevant to use in the context of security performance in vehicles, although topics may not be specifically related to HSMs.

Chen et al. do a performance evaluation of the use of encryption algorithms in in-vehicle CAN [19]. Their paper includes an analysis of the end-to-end communication latency overhead that is introduced by the use of cryptographic algorithms.

In [20], Woo et al. propose a security architecture for the lacking security in in-vehicle CAN-FD. As part of their work, they investigate the performance of this architecture by looking at aspects such as the execution time of cryptographic algorithms as well as communication response time between ECUs.

In [21], the same authors demonstrate a wireless attack performed on a vehicle using a malicious smarthpone application. As a countermeasure, they propose a security protocol for CAN. Similar to [20], they evaluate the performance of the protocol by considering aspects such as communication response time, key derivation and update time as well as varying CAN bus loads.

Lu et al. propose another security protocol for CAN with low cost and high efficiency, which they title Lightweight Encryption and Authentication Protocol (LEAP) [22]. In their paper, the performance of the protocol is evaluated, taking into consideration aspects such as key update time, encryption speed, memory usage and payload size.

Lastly, the applicability of TLS to secure in-vehicle networks is investigated by Zelle et al. [23]. As part of their analysis, the performance of applying TLS to in-vehicle networks is evaluated by looking at the execution speed and throughput of various cryptographic algorithms as well as the end-to-end latency of TLS communication.

3.3.2 Cryptographic Accelerators

Cryptographic accelerators differ from HSMs in that they (in general) do not provide safe storage for cryptographic keys. But just like HSMs, they provide hardware acceleration for cryptographic operations and should be relevant to investigate from a performance evaluation aspect. The papers listed below provided us with some relevant insights into what metrics could be used in a performance evaluation.

In [24], Zhang et al. present a common test framework oriented for cryptographic accelerators. The framework includes multiple benchmarks for evaluating the performance of cryptographic accelerators using OpenSSL. In their paper, they evaluate the performance of some commercial cryptographic accelerators, where factors such as signature generation speed, communication latency and host CPU utilization are

taken into account. They also compare the performance of the cryptographic accelerators with a baseline where the cryptographic operations are performed by the host CPU.

In [25], Hung et al. investigate the performance of the Sun Crypto Accelerator 1000, which was used as a cryptographic accelerator in Sun Microsystems servers [5]. In their paper, they look at performance aspects of this device, which include encryption throughput and host CPU utilization.

Togan et al. present an implementation of a hardware cryptographic accelerator using Field-Programmable Gate Array (FPGA) [26]. Included in their paper is an assessment of the performance of this component, which considers metrics such as cryptographic throughput with respect to different key sizes.

The IBM PCIXCC, a cryptographic co-processor for the IBM eServer, is described in a paper by Arnold et al. [27]. As part of their work, they evaluate the co-processor's performance with respect to, e.g, cryptographic throughput and key generation time.

Lastly, a lot more papers were found in this area, which consider performance aspects similar to those already mentioned. The papers are on cryptographic accelerators in various contexts, including IPsec [28], IBM processors [29, 30], OpenSSL [31], GPUs [32, 33], embedded CPUs [34], IoT [35], multi-core processors [36] and more [37] [38].

3.3.3 Encryption Algorithms

Evaluating the performance of an HSM includes assessing how well the HSM executes various encryption algorithms. We therefore looked into the topic of encryption algorithms in general, which potentially could provide us with some insight into how the performance of an HSM as a whole can be evaluated.

A lot of papers were found in this area, containing some relevant information about the performance of encryption algorithms in different contexts, including embedded systems [39–41], IoT [42–44], mobile devices [45, 46], wireless sensor networks [47, 48], smart grids [49], SSL [50], IPsec [51], cloud [52], mobile banking [53], RFID [54] and more [13, 55–58]. Although the contents of these papers had some relevancy, they did not provide any insights that previously mentioned papers had not already done.

3.3.4 Other Areas

Other areas that were looked into were IoT security and web services security (WSS). With regards to WSS in particular, there are some similarities between how web services and ECUs communicate and how this communication can be secured. Although some papers with some relevancy were found in these areas, they did not provide as relevant information as that in other, previously mentioned areas.

4

Method

In this section, we detail the practical steps that are taken in order to answer the posed research questions. Section 4.1 specifies some performance requirements set by Volvo, in order to provide us a framework to work around. Section 4.2 introduces the performance metrics that are used in this thesis, and describes the methodology for exactly how performance is measured using these metrics. Lastly, section 4.3 details how testing is performed, including what parameters are taken into account as well as a description of the testbed used in this project. It also includes a motivation as to why wolfCrypt was chosen as the software cryptographic library to use in this thesis.

4.1 Performance Requirements

In order to have a framework to work around during the performance evaluation, Volvo has provided us with some requirements they have on various performance aspects of ECUs and the communication between them.

4.1.1 CPU Load

The CPU load of an ECU cannot exceed a certain value. This is in order to fulfill real-time requirements that are set on an ECU. For example, there are some processes in an ECU that execute at certain intervals. When these processes execute, there needs to be a guarantee that necessary CPU resources will be available. Thus, for safety reasons, the maximum allowed CPU load is 80-85%. Today, it is not unusual for the CPU load on an ECU to lie between 50-60%, but it could also be higher or lower than this. A limit has been set that the use of cryptography – for, e.g., encrypting/decrypting messages – may not contribute more than an extra 5% to the CPU load. This is in order to save room for other new possible features for the ECU.

4.1.2 Bus Load

For the communication load on the CAN bus, there is not a specific limit to how much the use of cryptography may add to the bus load. Nonetheless, it should not contribute so much that the CAN bus capacity is maxed out. The maximum

bandwidth of the CAN bus used in this project is $500Kbit/s$. A rough limit is that the bus load should not exceed 70-80%. Global CAN networks are usually heavily loaded, while local subnetworks tend to have more margin.

4.1.3 Cryptographic Latency

When it comes to cryptographic latency, we need to consider that most messages have certain timing requirements. More specifically, many messages are continuously sent at specific intervals – or cycles, as they are called in the context of CAN. Different messages have different cycle times, which could range from 10ms to 5000ms. Generally, the lower the cycle time, the more important the message is. A message also has a timeout period, which denotes how long time a receiving node waits on a message before it considers the message lost. These cycles and timeouts need to be respected, especially for messages where timing is of high importance. Therefore, the use of cryptography may not result in a latency that causes a disruption of these aspects.

4.1.4 Other Requirements

For other metrics, no specific requirements have been set. In some cases it is deemed that enough resources are available, therefore setting a limit on the metric is not needed. In other cases, setting a limit is not applicable or not enough information is available to be able to set a limit.

4.2 Metrics

In this section, we present the metrics used in the performance evaluation. For each metric, we describe the method to measure performance using the metric. For a basic introduction of these metrics, refer to Chapter 2. The metrics related to research question 1, i.e., performance comparison between HSM and software, are CPU load and cryptographic latency throughput. Hence, these metrics are used to measure both HSM and software. Related to research question 2, i.e., security and performance trade-offs, are the same as for research question 1, but also bus load and TRNG latency, where the last two are only measured using the HSM.

For deciding which metrics to use, we first consulted with Volvo to elicit which metrics they would find interesting for us to investigate. Furthermore, in order to find additional metrics, we did a literature study where we looked for papers on performance in HSMs as well as performance in similar contexts. The result of this literature study can be seen in Chapter 3. From this literature, we chose metrics based on how relevant they were for the context of this thesis. We also took into account how recurring the metrics were in the examined papers. Some relevant metrics had to be excluded, as measuring them was not possible.

4.2.1 Bus load

In order to perform realistic practical measurements of the added bus load of different scenarios, testing would need to be performed in an environment where the vehicle's normal, day-to-day bus load is present. The optimal choice would be to use a real vehicle, or alternatively create an environment (testbed) representative of a real vehicle, where the messages between components in the testbed mimics that of a real vehicle. However, neither of these were an option; as testing in a real vehicle was not possible and the testbed used in this project consists of only a single node, and thus does not feature any communication.

The solution is to use a more theoretical method, where bus load is calculated by analysing the added overhead data of cryptographic operations. If used together with certain vehicle data, this can give a quite realistic approximation of the added bus load of different scenarios. The vehicle data needed are:

- Values for CAN bus capacity, i.e., CAN bus bit rate
- Tables for cycle times of different messages that are sent on the CAN bus

Method Overview

In order to calculate bus load, we must first decide on two things: 1) which (cyclic) message type we wish to secure using cryptography and 2) which cryptographic operation we want to measure the bus load for. As mentioned in section 4.1.3, most messages are cyclic. Because different message types have varying cycle times, they contribute differently to the bus load. Therefore, the added bus load depends on our choice of message type to secure.

As for the choice of cryptographic operations, the added bus load of different operations varies as well, as they contribute with different amounts of overhead data. The overhead data that gets added to a message needs therefore to be calculated. Finally, together with values for CAN bus bit rate and message cycle times, the added bus load of performing the cryptographic operation on the message type can be calculated.

Calculating Bus Load

Thus the bus load is theoretically calculated, based on some assumptions regarding the number of bits in a CAN frame and the bandwidth.

The bus transmit time for a single CAN frame can be calculated as:

$$Bus\ Transmit\ Time = \frac{Number\ of\ bits}{Bandwidth},\ (s)$$

Then, the bus load can be estimated as:

$$Bus\ Load\ Estimate = \frac{Bus\ Transmit\ Time}{Cycle\ Time},\ (Percentage, \%)$$

Before providing concrete examples of how to calculate added bus load, we first give an example of how bus load in general can be calculated using values for message cycle time. Let us say we have a message (i.e., CAN frame) of 100 bits and that the CAN bus bit rate is $500Kbit/s$. The time to transmit 1 bit on the CAN bus is $1/(bitrate) = (1/(500 * 1000))s = 2 * 10^{-6}s = 2\mu s$. The time to transmit the entire message is thus $100 * 2\mu s = 200\mu s$. If the cycle time for the message type is $100ms$, that means that for each $100ms$ time chunk, the message type will occupy the CAN bus for $200\mu s$. The resulting bus load of the message type is therefore

$$Bus\ Load = 100 \times \frac{200\mu s}{100ms} = 0.2\%$$

Now assume that a cryptographic operation increases the size of the message in the above example to 150bits, i.e., an increase in 50%. The time to transmit the message is now $150 * 2\mu s = 300\mu s$. The resulting bus load of the message type is then

$$Bus\ load = 100 \times \frac{300\mu s}{100ms} = 0.3\%$$

$$Increased\ bus\ load\ (\%)\ = 0.3 \div 0.2 = 1.5 \equiv 50\%$$

$$Added\ bus\ load\ (\%)\ = 0.3\% - 0.2\% = 0.1\%$$

Thus, the bus load resulting from this single periodic message type would be 50% higher than without cryptography. The bus load would increase with 0.1 %. Assuming the normal total bus load is 60%:

$$New\ total\ bus\ load = 60\% + 0.1\% = 60.1\%$$

$$Increased\ total\ bus\ load\ (\%) = 100 \times \frac{0.001}{0.6} = \frac{1}{6}\%$$

Note that in the examples above, we do not take into account aspects such as maximum CAN frame size and CAN frame segmentation, i.e., when a message is so large that it needs to be divided into multiple CAN frames.

In order to to calculate the resulting bus load for securing multiple types of messages (and with different cycle times), we simply do the above calculations for each message type and sum up the results.

Let us assume that we have already calculated the added bus load as 1.4% for some cryptographic operations (e.g., generating a MAC) that result in some fixed size overhead for every message type. Then, for example, for two different cycle times, 100 and 1000 ms, where the earlier is a total of 10 different message types, and the later is 20, one can calculate it as:

$$Total\ bus\ load\ estimate = 10 * 1.4 + 20 * 0.14 = 16.8\%$$

So in this example, some cryptographic overhead (e.g., a MAC) for these 30 different message types would create an additional bus load of roughly 16.8%.

4.2.2 Cryptographic Latency and Throughput

The process of performing a cryptographic operation does not only include the execution of the operation itself. Instead, in order for the operation to be carried out, some setup needs to done beforehand. This includes loading of data into buffers, preparation of state variables etc.

Depending on which parts of the process that are taken into account, the measured latency will be different. In this thesis, when it comes to the latency of performing cryptographic operations, we distinguish between three types of latency:

- Cryptographic latency
- Setup latency
- Total latency

In order to define these latencies, we need to first give a brief overview of the program flow for when a cryptographic operation is performed using the HSM, as well as how latency is measured using timestamps.

Program Flow for HSM Cryptography

As mentioned in sections 2.4.1 and 2.4.2, the AURIX board has both a main CPU (TriCore) as well a dedicated HSM CPU (ARM), both of which are involved in the process of performing a cryptographic operation. Data is communicated between them using intermediary buffers. The program flow for HSM Cryptography is shown in listing 4.1. Steps 1-3 and 7 are executed on the TriCore, whereas steps 4-6 are executed on the ARM.

```
TriCore:
1. Program start
2. Setup
3. Send request (interrupt) to ARM to perform crypto op
   (await answer from ARM)

ARM:
4. Setup
5. Hardware accelerators execute crypto op
6. Send confirmation (interrupt) to TriCore that op has completed

TriCore:
7. Confirmation received and output of op can be retrieved
```

Listing 4.1: Program flow for performing a cryptographic operation using HSM

Latency and Timestamps

Latency is measured by taking two timestamps; one initial and one final. The latency is then the difference between the timestamps. Depending on where in the code/program flow the timestamps are taken, they will represent different latencies.

How the timestamps are taken for the different latencies are shown in listing 4.2, where "TS" stands for timestamp.

```
TriCore:
1. Program start
*  TS_Initial Total latency
2. Setup
*  TS_Initial Cryptographic latency
3. Send request (interrupt) to ARM to perform crypto op
   (await answer from ARM)

ARM:
4. Setup
5. Hardware accelerators execute crypto op
6. Send confirmation (interrupt) to TriCore that op has completed

TriCore:
7. Confirmation received from ARM and output of op can be retrieved
*  TS_Final Cryptographic latency
*  TS_Final Total latency
```

Listing 4.2: Program flow for performing a cryptographic operation using HSM, including timestamps)

Difference Between Latencies

The different latencies can be defined as follows. *Cryptographic latency* is the execution of the operation at the HSM CPU (including some setup at the HSM CPU – more on this later) as seen by the TriCore (main CPU). *Setup latency* is the setup that needs to be done at the TriCore. *Total latency* accounts for the whole process, i.e, it is the sum of the cryptographic and setup latencies. Note that timestamps for setup latency are not included in listing 4.2, as setup latency simply is the subtraction of cryptographic latency from total latency.

Cryptographic latency is a measurement of how fast a cryptographic operation can be performed, and is thus the most relevant latency for this section. Total latency and setup latency are related to CPU load; this correlation will be further explained in section 4.2.4.

Cryptographic Latency Using HSM

With regards to measuring cryptographic latency using HSM, there are some important things to point out about the program flow and timestamps in listing 4.2. Firstly, there is some setup (in software) that is performed at the HSM CPU each time a cryptographic operation is performed (step 4). This can include configuration of cryptographic keys, loading of data into hardware buffers etc. We stress that this setup is not included in our defined setup latency, as setup latency is only concerned with setup at the TriCore.

Secondly, it is important to note that the cryptographic latency does not only correspond to hardware execution of the cryptographic operation. This is because the

timestamps for cryptographic latency are taken at the TriCore. As a result of this, setup time at the HSM CPU (step 4) will also be included in the calculations for cryptographic latency.

It could be argued that these timestamps instead should be taken before and after step 5. That way, the cryptographic latency would be representative of the "pure hardware" speed of executing a cryptographic operation. This was first of all not an option, as taking timestamps at the HSM CPU was deemed not possible (or at least a very complicated task). But even if that would not have been the case, the fact remains that in order for a cryptographic operation to be executed in the HSM, this setup is a prerequisite. For that reason, it was ultimately considered relevant to include this setup in the measurements for cryptographic latency, regardless of if taking timestamps at the HSM CPU would have been possible or not.

Cryptographic Latency Using Software

The program flow in listings 4.1 and 4.2 correspond to when a cryptographic operation is performed using the HSM. If, on the other hand, a cryptographic operation is performed using software, the program flow follows a similar but simpler structure. The program flow for software cryptography – including timestamps – is shown in listing 4.3. As opposed to listings 4.1 and 4.2, here each step is performed at the TriCore.

```
1. Program start
*  TS_Initial Total latency
2. Setup
*  TS_Initial Cryptographic latency
3. Execution of cryptographic operation (through function call to
   software cryptographic library)
4. Cryptographic operation finished and output can be retrieved
*  TS_Final Cryptographic latency
*  TS_Final Total latency
```

Listing 4.3: Program flow for performing a cryptographic operation using software (timestamps included)

The different latencies are defined in a similar way as for HSM cryptography. The only difference is that the cryptographic latency now corresponds to executing the cryptographic operation in the TriCore (through a function call to a software cryptographic library), without any of the setup that was performed at the HSM CPU.

Cryptographic Throughput

Cryptographic throughput is calculated using cryptographic latency. The calculation process consists of first computing the cryptographic latency of performing cryptographic operations (of the same type) on data of some size. Cryptographic throughput can then be obtained by simply dividing the size of the data with the computed cryptographic latency.

4.2.3 TRNG Latency

The process of generating random numbers – to use as cryptographic keys or IVs – is very similar to the process of performing cryptographic operations. The program flow is similar to that in listing 4.2, with the difference being that instead of performing cryptographic operations, the HSM hardware generates random numbers. Measuring TRNG latency is therefore practically analogous to measuring cryptographic latency, thus we refer to section 4.2.2 for any details about the process, program flow, taking timestamps etc.

However, as opposed to cryptographic latency, TRNG latency is only to be measured using the HSM. An equivalent measurement using software – in order to compare the performance of HSM and software – will thus not be available. Even though the used software cryptographic library offers functions for generating random numbers, there is no guarantee that the quality of random numbers generated by software and HSM are equivalent (most likely they are not), and comparing the quality would have been a difficult and overly time consuming task. Therefore, a comparison of TRNG latency for HSM and software was in the end discarded.

4.2.4 CPU Load

In this thesis, we are concerned with the CPU load that is added onto the AURIX main CPU (TriCore) when cryptographic operations are performed. Calculating this added CPU load is dependent on the time that the CPU spends on doing active work related to the cryptographic operation.

CPU Load and Latency

As can be seen in listing 4.1: when a cryptographic operation is performed using the HSM, the TriCore only does work in steps 1-3 and 7, whereas the HSM CPU (ARM) takes over in steps 4-6. This means that the TriCore only has to spend active time on the cryptographic operation process during steps 1-3 and 7, while it is free to perform other – possibly unrelated – tasks during steps 4-6. Because this active time is represented by the setup latency, setup latency is therefore used when calculating the added CPU load.

When a cryptographic operation is performed using software, the TriCore is active during the whole process, as can be seen in listing 4.3. Therefore, in this case, the total latency is used instead.

Method Overview

The method for calculating added CPU load has many similarities with the method for calculating bus load, as it also is a bit theoretical and makes use of values for message cycle times. But just like the method for calculating bus load, this method can give a quite realistic approximation of the added CPU load of different scenarios.

Just like for the bus load method, we must first decide which message type we wish

to secure using cryptography. Again, we are interested in the cycle time of the message type, as that will determine how often the CPU has to perform cryptographic operations on the message type.

We also must decide on the cryptographic operation we want to measure the added CPU load for, and calculate the latency for the operation (setup latency for HSM or total latency for SW). This is because different operations have varying latencies, which thus contribute differently to the CPU load. Finally, with the use of values for message cycle times, we can calculate the added CPU load of performing the selected cryptographic operation on the message type.

Calculating CPU Load

The formula for calculating the added CPU load from cryptographic operations is:

$$CPU\ Load = \frac{Latency}{Message\ Cycle\ Time},\ (Percentage,\ \%)$$

Note that the parameter *latency* is generally affected by the data size. In other words, it takes much longer to encrypt a longer message instead of a shorter. Thus, the CPU load estimate needs to be taken into a context of the data size. This will be covered in more detail in Chapter 5.

Assume that it takes $50\mu s$ of CPU time to encrypt and decrypt a particular message type. If the message type has a cycle time of $100ms$, that means that for each $100ms$ time chunk, the process of encrypting and decrypting the message will occupy the CPU for $50\mu s$. The resulting added CPU load is then:

$$CPU\ Load = 100 \times \frac{50\mu s}{100ms} = 0.05\%$$

Assume that the total CPU load is already 60%, then the increase would be:

$$New\ total\ CPU\ load = 60 + 0.05 = 60.05\%$$

$$Increased\ total\ CPU\ load\ (\%) = 100 \times \frac{0.05}{60} = \frac{1}{12}\%$$

Thus, in this case the added CPU load is 0.05%, whereas the relative increase from the normal total CPU load is $\frac{1}{12}$%.

Similar to the bus load example, if we want to calculate the added CPU load for securing multiple types of messages (with different cycle times), we simply do the above calculations for each message type and sum up the results.

As a concrete example, let us assume we have a set of message types, where there are two different cycle times, 100ms and 1000ms. The earlier is a total of 10 different message types, and the later is 20. Furthermore, lets assume we already have calculated the CPU load estimates (including both encryption and decryption) for the different cycle times, for some block cipher mode; 0.103 for 100ms, and 0.0103 for 1000ms (these values are taken from Table 5.2 in Chapter 5, for the mode CTR

with data size 512). The result is then:

$$Total\ CPU\ load = 10 * 0.103 + 20 * 0.0103 = 1.236\%$$

Thus, cryptography on these 30 message types would create an additional CPU load of roughly 1.24%. In other words, if the CPU load is already 60%, it would be now 61.236% due to this cryptographic functionality.

In this example, only a single block cipher mode was used. However, as a short reminder, it is possible that there can be multiple different modes used for the cryptographic functionality. In other words, all the message types does not necessarily need to be encrypted with the same block cipher mode. For example, it might be motivated to utilize GCM just for some specific message type, where authentication might be desirable. Meanwhile, the rest of the message types to be encrypted can be done with CTR.

CPU Offload

Closely related to CPU load is CPU offload, which is a measure of how much the HSM can offload the host CPU compared to when cryptographic operations are performed in software. The CPU offload will be calculated in percentages and is calculated as:

$$CPU\ offload = 100 \times (1 - \frac{HSM\ CPU\ load}{SW\ CPU\ load}),\ (Percentage, \%)$$

Thus, if the resulting percentage is positive it means that the HSM offloads the main CPU. Meanwhile, for negative percentage it is the reverse; that the software actually has better CPU load than HSM.

4.2.5 Summary of Latencies

Table 4.1 summarizes the latencies described in this section and the distinction between them. Note that even though setup and total latency can be applied to both HSM and SW cryptography, their usefulness is restricted to HSM and SW, respectively.

Latency type	Description	Usefulness
Setup	The time the CPU has to spend on setup before performing a cryptographic operation	Used to measure the time that the ECU CPU has to spend doing active work when a cryptographic operation is performed using the HSM, in order to calculate CPU load
Cryptographic	The time it takes to perform a cryptographic operation (either by HSM or SW)	Used to compare the speed by which the HSM and software perform a cryptographic operation
Total	Setup latency + cryptographic latency	Used to measure the time that the ECU CPU has to spend doing active work when a cryptographic operation is performed using SW, in order to calculate CPU load

Table 4.1: Summary of latencies.

4.3 Testing

In order to perform meaningful testing, it needs to be done in a structured and methodological way.

The general testing approach is illustrated in Figure 4.1. The idea is that there exists a blackbox for cryptographic operations, where the blackbox has input and output. This blackbox could in reality be either in an HSM (cryptographic accelerator) or a software module running on the main CPU. However, conceptually one can think that there is no significant difference between the two cases; a blackbox is going to perform some cryptographic operations on some data (e.g., the payload of a CAN frame).

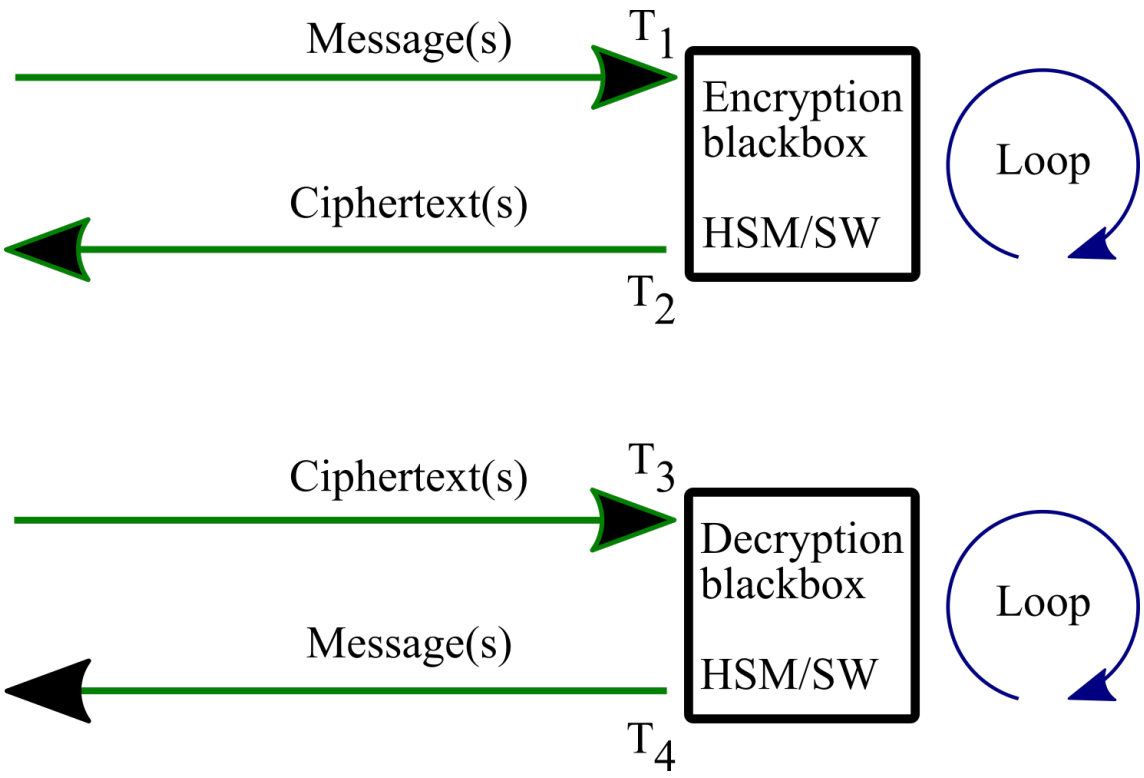

Figure 4.1: A conceptual illustration of the general testing method.

The conceptual idea is that input (either a single frame or batch of multiple frames) are sent to a cryptographic blackbox/module, which can be either in an HSM or software (SW) in a node. Then the module performs its cryptographic operations and sends out its output. Note the differentiation of encryption and decryption; they perform different cryptographic operations and thus they might have differences in performance.

So the idea is to send some arbitrary plaintext/data of some fixed size to the HSM for encryption, receive back the ciphertext, and then send the ciphertext back to the HSM for decryption. Thus, encryption and decryption is done in the same measurement. However, as described earlier, latency is measured at different points in this process.

4.3.1 Parameters and Variables

The parameters relevant to the security configuration is key length and the block cipher mode of operation (specially in this project: ECB, CBC, CTR, and GCM). Another relevant parameter is using different algorithm combinations, such as using only encryption versus using encryption and MAC together. However, this is only possible with GCM, at least when utilizing the HSM. It was decided to not perform testing for this due to the limited motivation and the time frame for this project. Note that in this thesis we are only studying AES.

Since there are several different parameters, there are several combinations/con-

figurations possible. Thus, it is vital for our work to reduce time-consuming and redundant testing.

There needs to exist some default fixed configuration, in order to reduce the combinations of configurations. For example, when testing cipher block mode of operation, it needs to be the single free variable in a default configuration. As having multiple free variables would increase testing significantly, so it needs to be avoided.

The default configuration in this project was quite easy to decide on. The cryptographic accelerator of the HSM operates in the standard block size of AES; 128-bits. Parameters such as key length and MAC (aka. tag) are implemented for that block size. In other words, the cryptographic accelerator is configured for use only with a key length that is 128-bits. Furthermore, the secure storage is implemented for keys of that size.

Thus it is not possible to compare different key lengths (128, 196, 256-bits) between the HSM and software. One could argue that this limitation of the cryptographic accelerator is a limitation in terms of security. In this case one would need to change the hardware to work with a larger key length or be flexible with key length. Meanwhile, for software it is just a algorithm or a parameter that needs to be changed.

Therefore, the only tested variables are the block cipher mode and data size. As described earlier, this is due to the setup of the cryptographic accelerator. Data size is an interesting variable to study, i.e. encrypting and decrypting several blocks in sequence. Since e.g., it might be the case that software and the HSM significantly differ in performance when it comes to different data sizes. Furthermore, it might be the case that there is different growth rate in latency in regards to data size, when it comes to different block cipher modes.

Note that it is not interesting, when it comes to testing, to look at data sizes smaller than the block size or the sum of the needed blocks. Since padding would be necessary, either by the user or automatically by the algorithm. Thus, in general (both for HSM and SW), there is no performance gain to operate with smaller data than the block size, 128 bits. Furthermore, it is redundant to perform measurements at data sizes that are not multiples of the block size. Thus, the data sizes used in testing are 16, 32, 64, 128, 256, 512, and 1024 bytes of data.

4.3.2 Testing on Testbed

Testing was quite simple to perform. The variables block cipher mode and data size was automated to change with a script, which was run through the debugger. More specifically, after the default run is done, then the script resets the code and before execution starts, the script changes the testing variables. This process can be repeated in a loop, in order to gather several measurements.

A total of ten measurements was taken for each data size and then a mean was calculated for each metric. Thus, each presented metric related to HSM and software

in Chapter 5 and Appendix C are the means. For example, when it is written just *total latency* this is actually the mean for that case.

Meanwhile, for the TRNG twenty measurements were taken for each data size. The metrics presented in Chapter 5 and Appendix C regarding TRNG are the means of these twenty measurements. Note that data in this context means the size of the true random numbers generated by the TRNG. The TRNG also operates with a block size of 128-bits. Thus, data can only be generated in multiples of 128-bits.

Data extraction can be performed in several different ways. The starting way was to manually read the variables through the debugger. Efficient data extraction was done through the script, which reads all the desired variables, after the script is done it outputs all the variables in a CSV-format as a text file. Through the CSV format of the data, it can easily be imported into Excel for further analysis.

The idea was perform testing in several iterations of a testbed. However, in this time frame, the project only finished the results of the first iteration. Thus, only the first iteration will be detailed here. The further iterations are detailed in *Future work* (see Chapter 7, Section 7.3). For example, the proposed testbed iteration II would have actually had a CAN bus. Thus, communication protocols are also covered in *Future work* (see Chapter 7, Section 7.3).

4.3.3 Testbed Iteration I

Testbed iteration I is the most basic setup for performing testing and also ensuring correctness. An illustration of this testbed is shown in Figure 4.2. This first iteration is really only a single node. A PC is connected trough a debugger (Lauterbach TRACE32, see section 2.4.3 for the debugger) to an AURIX ECU with its HSM. The PC only acts as a helping tool to control the target device and perform debugging. Meanwhile, all the cryptographic operations is performed only within the AURIX system. For example, arbitrary data is encrypted and then decrypted, but never leaves the AURIX system in a normal communication sense. But through the debugger and the PC there is read and write access to the AURIX system, which is necessary for performing efficient testing.

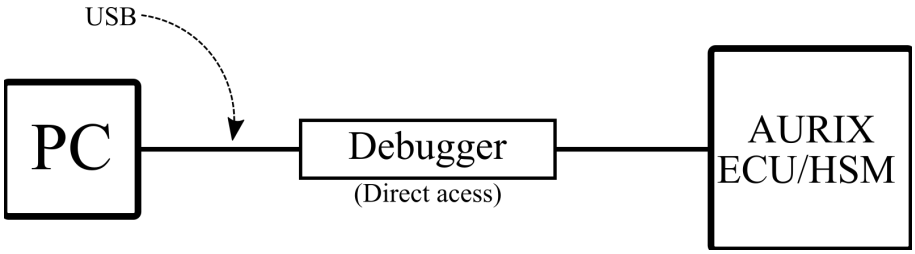

Figure 4.2: An illustration of testbed iteration I, where the communication link is a debugger. The debugger naturally also functions as a great tool in helping to ensure functionality and enable efficient testing.

4.3.4 wolfCrypt: Motivation for Choice of Software Cryptographic Library

There were a number of reasons for choosing wolfCrypt as the software cryptographic library to use in this thesis. First of all, it includes all the AES cipher modes that we wanted to use in this thesis, which we considered as the first necessary requirement in the choice of software cryptographic library to utilize.

Another important factor is that the available memory on the AURIX is quite small. This means there are limitations on the size of the program code that can be loaded into memory, which includes external libraries such as wolfCrypt. This problem is mitigated by wolfCrypt, which not only features a relatively small code size and low runtime memory usage, but also offers the possibility of customizing which features are included during compilation. For example, it is possible to only include functions for AES cryptography – which are only a fraction of the functions that wolfCrypt offers. This can in turn substantially minimize the code size that is compiled and consequently has to be loaded into the AURIX memory.

Lastly, yet another reason for choosing wolfSSL/wolfCrypt was the large amount of available documentation as well as the simplicity of the API.

5

Results

This chapter details the results of both the AURIX HSM solution (see Section 4.3.3 and Figure 4.2 for details), and the software solution, which is wolfCrypt (see Section 2.5). A comparison of the solutions is also provided, in regards to metrics such as CPU load and cryptographic latency.

In this chapter we only include the most relevant results. For further details, such as tables over the results, see Appendix C.

5.1 HSM

This section details the results of the performance evaluation of the HSM solution (testbed iteration I). Note that both encryption and decryption were done in the same testing function.

5.1.1 Latency

Figures 5.1 and 5.2 show a comparison of *encryption latency* and *decryption latency* for the block cipher modes. An easy observation is that the cryptographic latency is linear with respect to data size for all the block cipher modes. Furthermore, GCM has the highest cryptographic latency, and also the highest growth rate with data size, when it comes to *encryption latency*. Something that is harder to observe from the two figures, is that *decryption latency* is generally slightly lower then *encryption latency*.

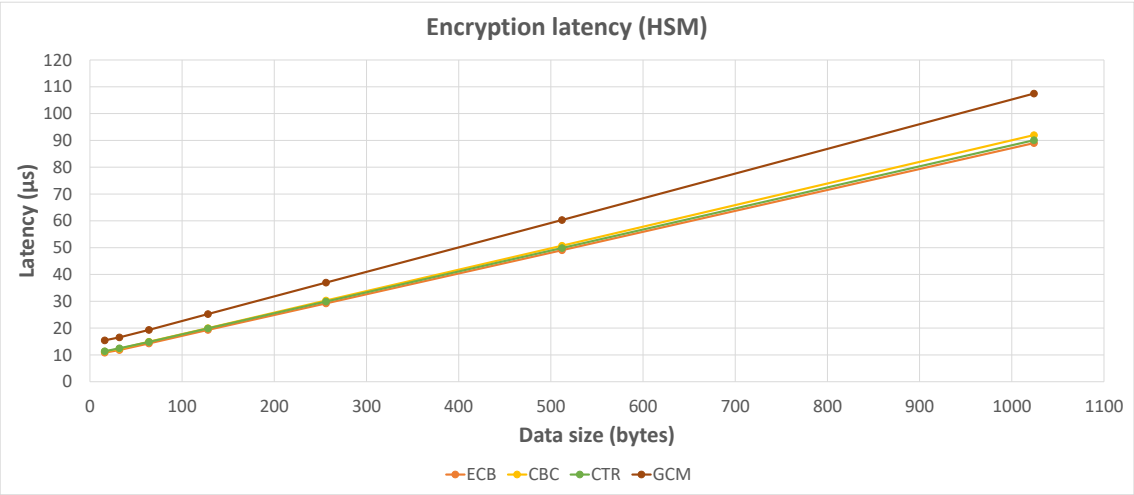

Figure 5.1: Encryption latency (HSM) vs data size for different cipher modes.

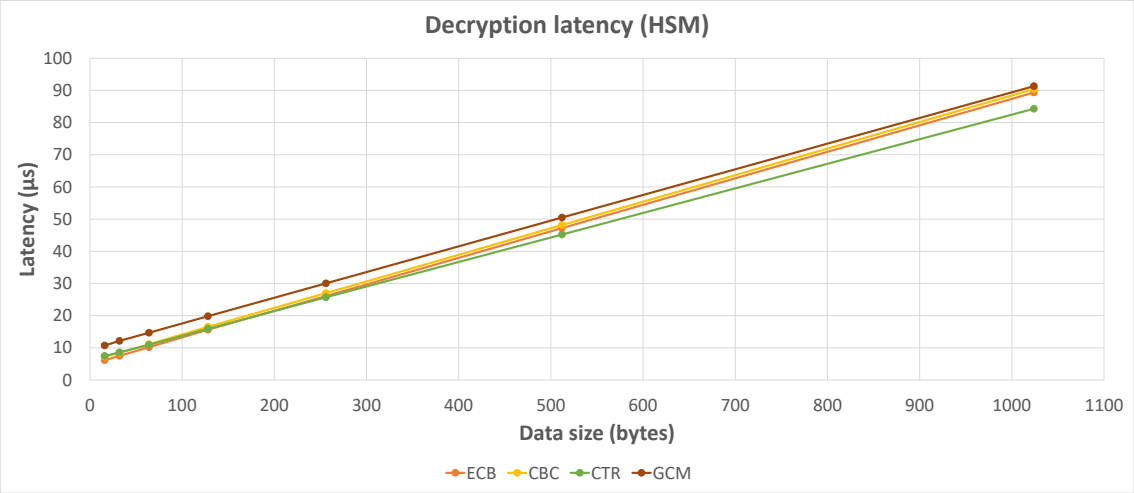

Figure 5.2: Decryption latency (HSM) vs data size for different cipher modes.

Figure 5.3 illustrates how the *setup latency* develops with the data size. Since *setup latency* is practically equal for all cipher modes, the diagram only has one curve, which represents all the cipher modes.

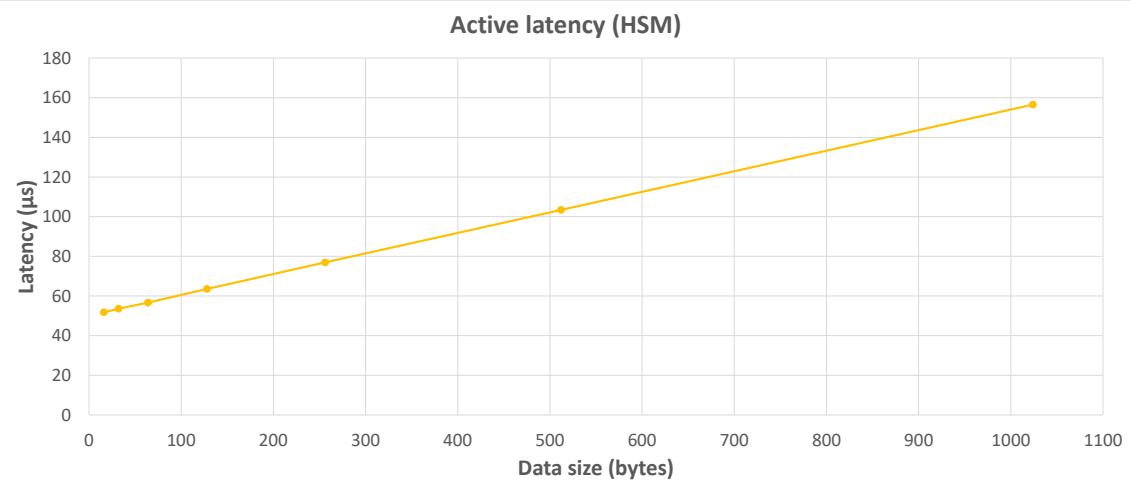

Figure 5.3: Setup latency vs data size.

5.1.2 Throughput

The throughput for encryption and decryption are shown in Figures 5.4 and 5.5. An easy observation is that *decryption throughput* is generally higher compared to *encryption throughput*. This is a logical observation based on what was observed for cryptographic latency.

GCM has generally the lowest cryptographic throughput. However, for decryption there was no significant difference for the highest data size, excluding CTR. Meanwhile, for encryption the throughput is almost the same for the modes ECB, CBC, and CTR.

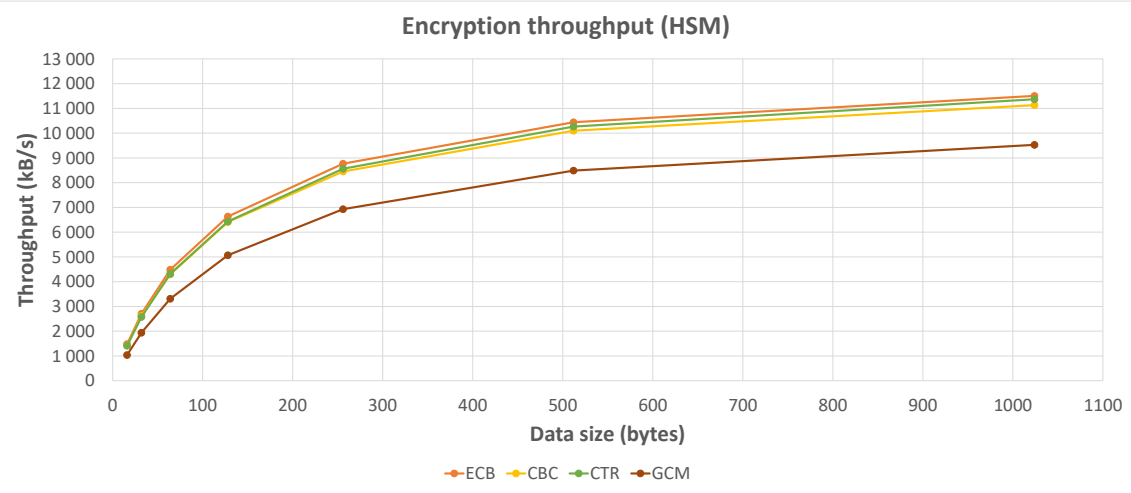

Figure 5.4: Encryption throughput (HSM) vs data size for different cipher modes.

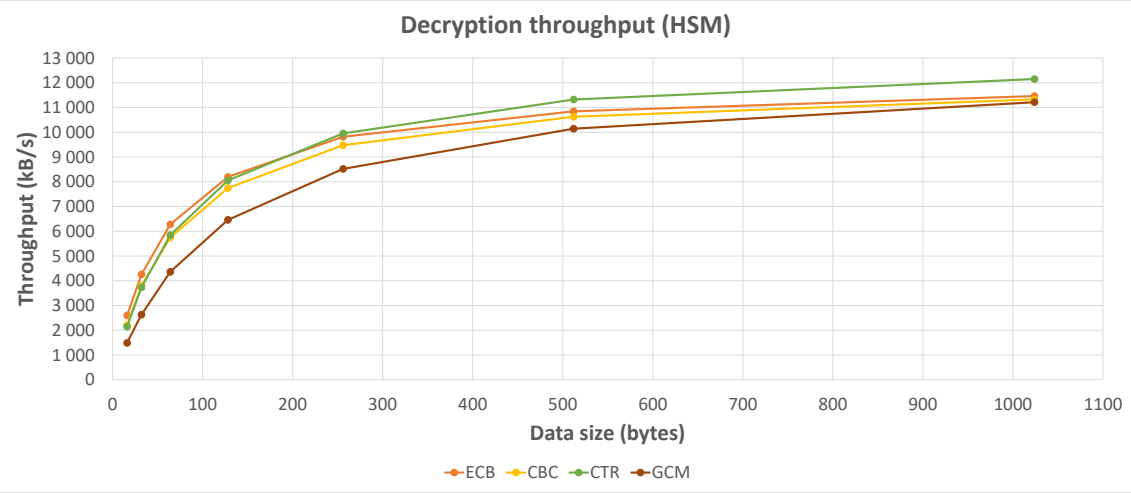

Figure 5.5: Decryption throughput (HSM) vs data size for different cipher modes.

5.1.3 TRNG

The TRNG latency is shown in Table 5.1 and Figure 5.6. One quick observation is that *Total latency* is relatively the same as *TRNG latency*. The difference is roughly 40 microseconds regardless of the generated data size. This is a logical result, since the generated data size does not affect setup latency in this case. For example, no data is needed to be sent to the HSM, in this case the TriCore only receives data.

Another easy observation, from Figure 5.6, is that the generated data size is linear with the latency. This implies that there are no relative gains in latency for generated different data sizes. Lastly, the lowest possible TRNG latency was average 300 microseconds, which is for generating 16 bytes.

Data in bytes	**Total latency** (μs)	**TRNG latency** (μs)
16	345	303
32	645	601
64	1230	1190
128	2430	2390
256	4790	4750
512	9540	9500
1024	19100	19000

Table 5.1: Table over latency for the TRNG.

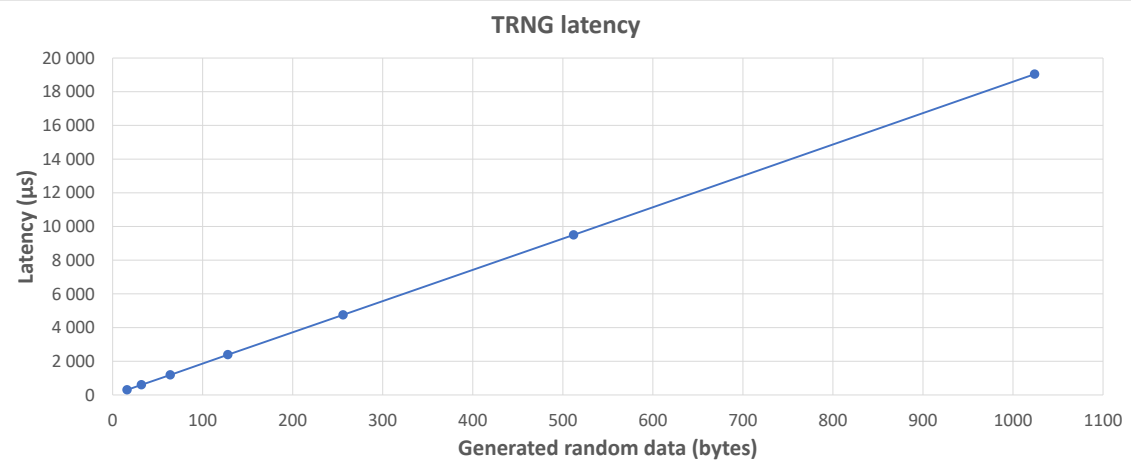

Figure 5.6: TRNG latency vs size of random data generated.

5.2 Software Cryptography

This section contains the results of the performance evaluation of the software solution. The wolfCrypt cryptography library was used as the software alternative.

5.2.1 Latency

The cryptographic latency is shown in figures 5.7 and 5.8, where the earlier is encryption and the later is decryption. An easy observation is that all the latency metrics are linear with the data size, for all the modes.

Furthermore, GCM has the highest cryptographic latency, which is significantly higher. This is due to a higher growth rate with data size. Another observation is that there is no significant difference in cryptographic latency for the other block cipher modes.

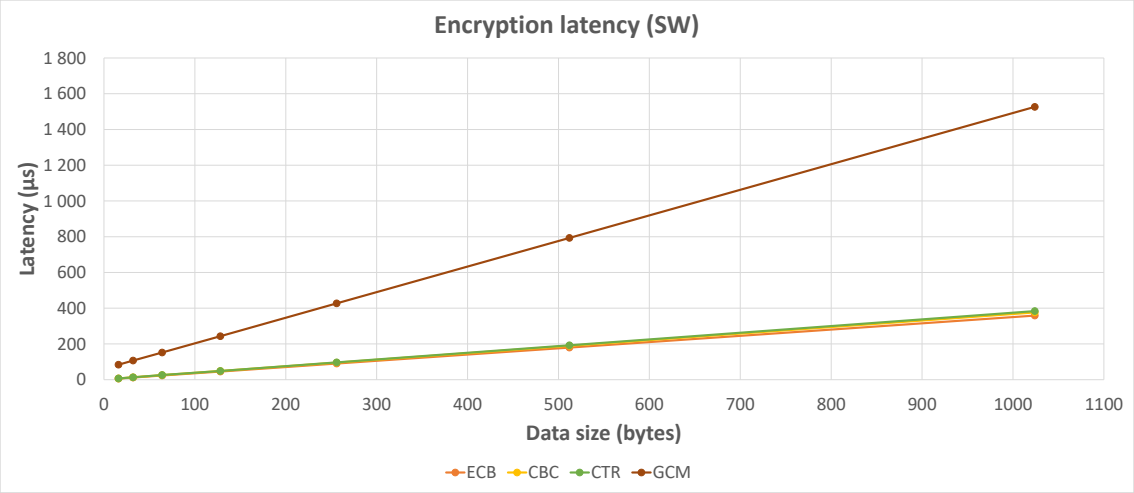

Figure 5.7: Encryption latency (SW) vs data size for different cipher modes.

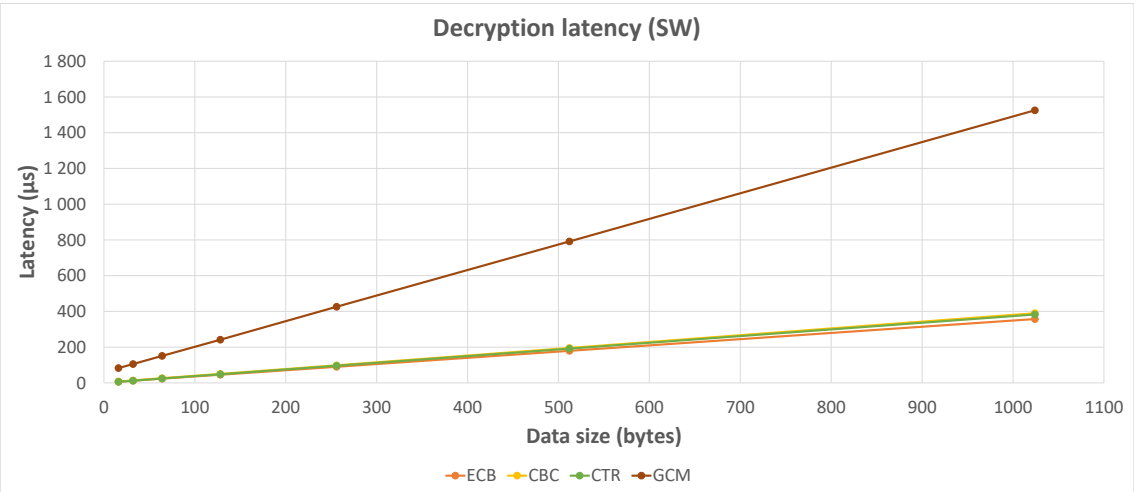

Figure 5.8: Decryption latency (SW) vs data size for different cipher modes.

Figure 5.9 illustrates how the total latency develops with the data size. It is clear that the latency of GCM grows faster and also has higher initial latency than the rest of the modes. Meanwhile, all the other modes are relatively the same This is not an absurdity, since the modes are fairly similar when it comes to the performed cryptographic operations. This can also be observed when looking at the cryptographic latency in Figures 5.7 and 5.8.

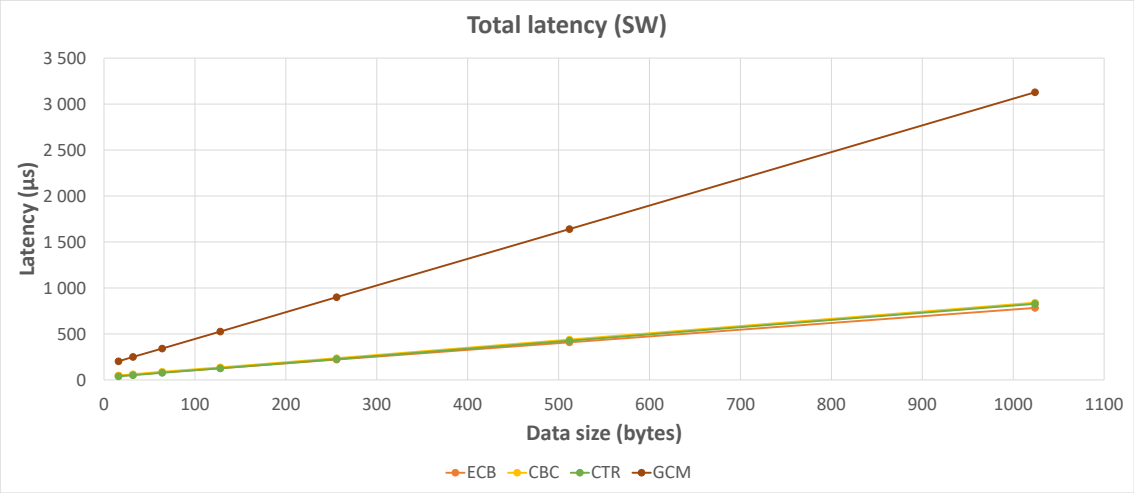

Figure 5.9: Total latency vs data size for different cipher modes.

5.2.2 Throughput

The throughput for encryption and decryption are shown in Figures 5.10 and 5.11. Clearly the throughput is roughly stable after 256-bytes in data size.

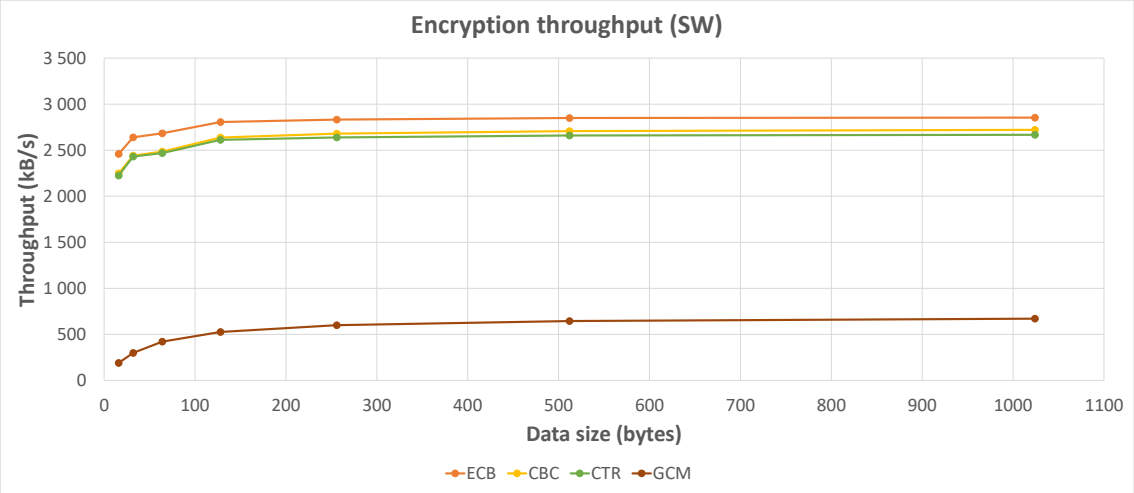

Figure 5.10: Encryption throughput (SW) vs data size for different cipher modes.

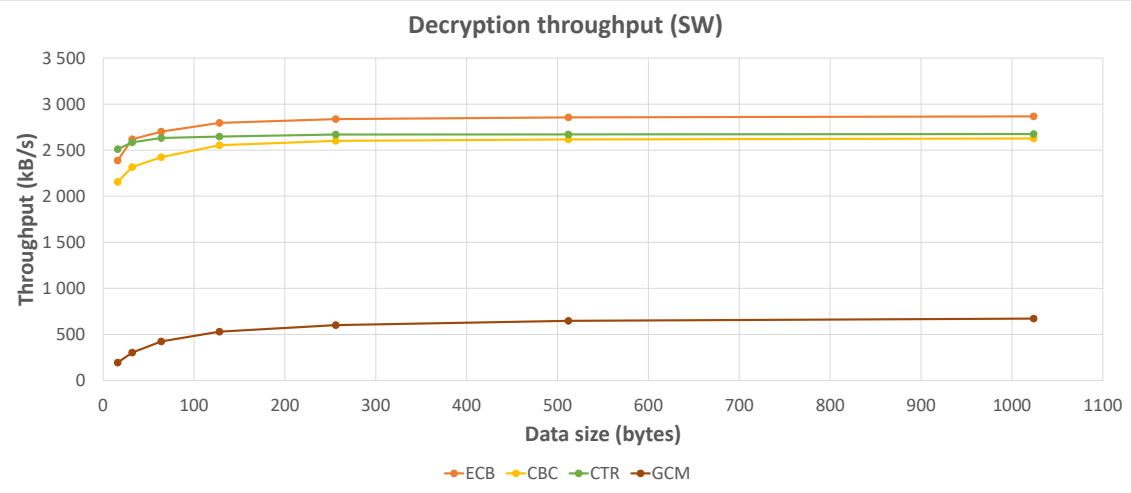

Figure 5.11: Decryption throughput (SW) vs data size for different cipher modes.

5.3 Comparison

In this section we present our main comparison metrics: CPU load, CPU offload, and difference in cryptographic latency.

5.3.1 CPU Load

In Table 5.2 there is a summary of all the estimated CPU loads, where the cycle time is ten milliseconds. A quick observation is that higher data size leads to higher CPU load, both for the HSM and software. Still the effect was more significant for the software solution, i.e., software has higher growth rate with data size. Thus, HSM is significantly better at larger data sizes.

Mode	Data in bytes	HSM CPU load (%)	SW CPU load (%)
ECB	32	0.54	0.57
ECB	512	1.04	4.08
CBC	32	0.53	0.61
CBC	512	1.03	4.38
CTR	32	0.53	0.51
CTR	512	1.03	4.26
GCM	32	0.54	2.49
GCM	512	1.03	16.4

Table 5.2: CPU load for performing cryptography on a single message type with cycle time 10ms.

An observation, is that for the lowest data size (32 bytes), both HSM and wolfcrypt CPU load are fairly equal, excluding GCM. It is clear that GCM is significantly more demanding for SW, for both small and large data sizes.

Since the *setup latency* was assumed to be relatively the same for all the block cipher modes, the HSM CPU load is based only on the results for one of the block cipher modes. In other words, here the HSM CPU load is not impacted by the block cipher mode. The focus is instead on comparing how the data size affects the CPU load estimate. Graph illustrations for CPU load can be found in Appendix C, see figures C.4 and C.5 for SW and figure C.1 for HSM.

5.3.2 CPU Offload

CPU offload is summarized in Table 5.3. It is clear that the HSM is significantly better at larger data sizes. Still, for the smallest data size of 16 bytes, it was actually the case that software is slightly better when it comes to CTR.

The biggest difference was GCM, which is clearly significantly more demanding for software. Even for the smallest data, the CPU offload was roughly 78%, and for the largest it is roughly 94%, which is quite a large CPU offload.

Mode	Data in bytes	CPU offload (%)
ECB	32	5.02
ECB	512	74.6
CBC	32	12.3
CBC	512	76.5
CTR	32	-4.4
CTR	512	75.8
GCM	32	78.5
GCM	512	93.7

Table 5.3: CPU offload between HSM and software (wolfCrypt). A positive percentage indicates that the HSM is more efficient.

5.3.3 Difference in Cryptographic Latency

Only taking CPU offloading into account does not give the whole picture, since offloading can be done through any extra peripheral with a CPU, e.g., having an extra ECU to perform cryptographic operations (SW). A significant benefit with an HSM is that it comes with a cryptographic accelerator. Thus, an HSM comes with the purpose of providing better cryptographic latency compared to software. Therefore, it is interesting to also study the difference in cryptographic latency.

The difference in cryptograhic latency between HSM and SW is calculated as:

$$Latency\ Difference = HSM\ Latency - SW\ Latency,\ (s)$$

Figure 5.12 illustrates how the difference in cryptographic latency, between HSM and software, changes with data size, for the modes CTR and GCM. Note that the lines for GCM encryption and decryption are very close to each other (the same applies for CTR encryption and decryption).

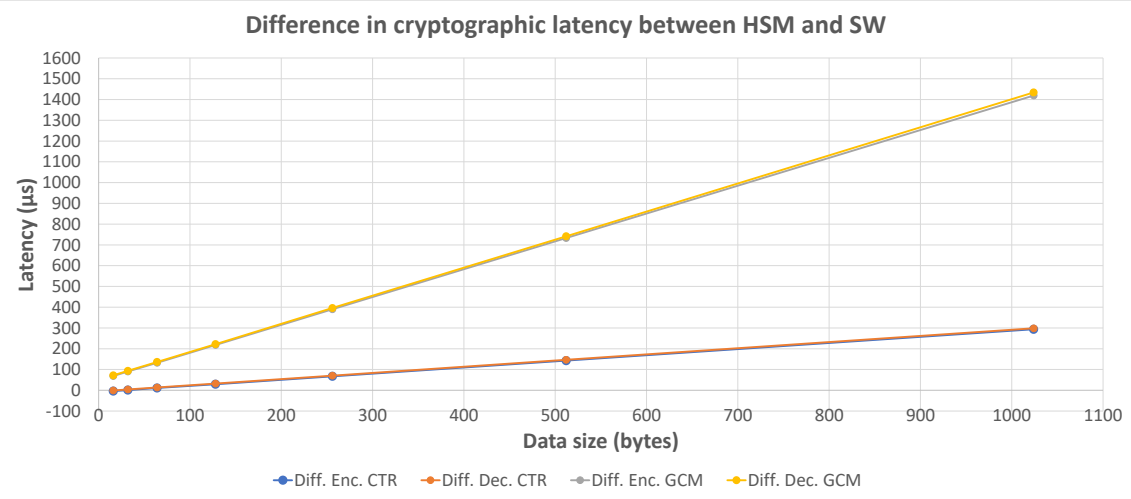

Figure 5.12: Difference in cryptographic latency between HSM and SW with respect to data size, for cipher modes GCM and CTR.

An easy observation is that the difference in cryptographic latency for GCM grows much faster, in favor of the HSM, compared to CTR. GCM also has higher difference in cryptographic latency for the smallest data size. Thus, at the largest data size of 1024 bytes, the difference between GCM and CTR is quite significant. This implies that the HSM is much faster in cryptographic latency for GCM. Furthermore, the same also is true for CTR. However, the growth rate is smaller compared to GCM.

Something that is harder to spot in this diagram is that for the lowest data size of 16 bytes, software with CTR mode is actually slightly better in cryptographic latency (the data points are a little below zero in the diagram, see Appendix C, Section C.3).

5.4 Bus Load

CAN is the most common bus protocol in vehicle infrastructure. However, it has some drawbacks. For example, the bandwidth is limited compared to other protocols. Furthermore, the protocol is small in data payload size. Thus, the CAN bus protocol will likely act as bottleneck.

To start off, naturally a CAN data payload smaller than the block size will be padded. This means that at least two CAN frames need to be sent in every transmission. So in the worst-case scenario it is a doubling in overhead. Furthermore, naturally with the use of a MAC for the data, the overhead increases. Since for the HSM the only MAC size is the block size, which leads to an additional two CAN frames (a 16-byte MAC in data payload).

Therefore, it is interesting to also look at CAN FD (see Appendix B.2.2), which is an improved version of CAN (only changes in the protocol). CAN FD comes with the improvements of flexible data payload and also flexible data-rate. Thus, in

CAN-FD it is possible to send a large message, i.e. larger than 8 bytes), in a single CAN-FD frame and also at overall higher bandwidth compared to CAN. These two aspects will likely increase performance drastically, especially for larger messages.

Bus load is estimated here in a theoretical way, and is based on some reasonable assumptions; bandwidth, number of bits in a CAN frame. In the first section, a concrete example is given regarding added bus load due to a 16 byte MAC (GCM), in a worst-case scenario when it comes to the maximum possible number of bits in a CAN frame (bit-stuffing). In the second section, the most equivalent example is given for CAN FD instead.

5.4.1 CAN

Assume that the maximum bandwidth on CAN is $500Kb/s$, this will be used as the bandwidth for calculating the bus transmit time. Note that in reality the average bandwidth could be lower depending on the circumstances. The standard CAN frame size is 125 bits, and with CAN extended (larger *arbitration field* in the CAN frame, see Appendix B.2.1) there are additional 18 bits in the header. Furthermore, there occurs bit-stuffing, where the worst-case is one extra bit for every fifth bit. Thus, $(125 + 18)/5 = 28.6$ Thus, the assumption is that in a worst-case scenario there are $125 + 18 + 28.6 \approx 171$ bits in a CAN frame. Now all the needed variables are done, so now one can calculate the bus transmit time, for a single CAN frame:

$$Bus\ Transmit\ Time = \frac{Number\ of\ bits}{Bandwidth} = \frac{171}{5E+5} = 3.42\text{E}{-4}\ s$$

As explained earlier the assumption is that there exists both some set of messages that are periodic, and some that are non-periodic. The periodic message types are sent with a cyclic time, which usually can range from one microsecond to several seconds.

Then, the bus load, with a cycle time 10ms, can be estimated as:

$$Bus\ Load = \frac{Bus\ Transmit\ Time}{Cycle\ Time} = 100 \times \frac{3.42\text{E}{-4}}{0.01} = 3.42\%$$

It is trivial that the bus load estimate is linear with the cycle time. Thus, it is preferred in a performance perspective to only increase the overhead for message types with a longer cycle time, since this would not have the same effect on the bus load.

However, this is a question that needs to be answered from message type to message type. In general a faster cycle time implies a higher priority. However, a higher priority does not necessarily mean that a message type is safety- or security-critical. Therefore, every single message type needs to be analyzed if a MAC (and encryption) is worth the performance consequences.

This bus load estimate is only for a single arbitrary message type, and also for a single CAN frame. Now the approach is that one can use this for estimating the

additional bus load for using the MAC of 16 bytes in GCM. More specifically, sending the MAC in a message would create an overhead of 16 bytes of the MAC itself (2 CAN frames), and also another two CAN frames for segmentation and padding. So, in total the overhead due to a 16-byte MAC is 4 CAN frames. Thus, the final result is that the additional bus load is $4 * 3.42 = 13.68\%$, with a cycle time of 10 ms, for a single message type.

$Bus\ load\ estimate\ MAC = 3.42 \times 4 = 13.68\%$

The general absolute worst-case scenario is where the data to be encrypted is only 1 byte. Then due to AES it will be padded to 16 bytes, meaning two CAN frames. Furthermore, with a 16-byte MAC there will be two CAN frames for the MAC itself, but also two CAN frames for segmentation and padding. Thus, the overhead from the MAC is in total four additional CAN frames. In this situation, encryption and a MAC comes with a significant performance cost.

Note that one can calculate the total additional bus load for several different message types in regards to cycle time (described earlier in Chapter 4).

For example, for two different cycle times, 100 and 1000 ms, where the earlier is a total of 5 different message types, and the later is 10:

$Total\ bus\ load = 5 * (4 * 3.42) + 10 * (4 * 0.342) \approx 82\%$

(Notice that this is specifically for a MAC of size 16 bytes, which is the size utilized in GCM)

So in this example, having the GCM MAC for these 15 different message types would create an additional bus load of roughly 82%. This is a relatively high additional load.

5.4.2 CAN FD

The way it works with CAN FD is that one can have both flexible bandwidth and data payload. This enables the possibility of sending the whole MAC (16 bytes), and higher bandwidth when sending the data payload. Thus, for this calculation one needs to split the CAN FD frame into two parts, since one needs to calculate for two different bandwidth; arbitration and CAN FD.

The arbitration speed is used to calculate the transmission time for sending bits in the *header* field, and the *footer* field. Meanwhile, a higher bandwidth (CAN FD) can be used when sending the data phase of the CAN FD frame.

Theoretical calculation example:

Note that in CAN the worst-case is that there is one bit-stuffing bit for every fifth bit in the CAN frame.

Assumptions:

- Bits with arbitration bandwidth:

  $header + footer = (17 + 18) + 12 \implies 47 + \frac{47}{5} \approx 56bits$

- Bits with CAN FD bandwidth:

  $(4 * 8$ bytes in total data payload; 16 byte MAC)

  $data\ phase = 27 + data\ payload = 27 + (4 * 64) \implies 283 + \frac{283}{5} \approx 340bits$

- Thus, in worst-case the total number of bits in this CAN FD frame is 396
- Maximum bandwidth with CAN: 0.5 Mb/s
- Maximum bandwidth with CAN FD: 1 Mb/s

$$Transmission\ time\ (CAN\ FD, normal\ bandwidth) = \frac{396}{5E+5} = 7.92\text{E}{-4}\ s$$

$$Bus\ load\ (CAN\ FD, normal) = 100 \times \frac{7.92\text{E}{-4}}{0.01} = 7.92\%$$

$$Transmission\ time\ (Header + footer) = \frac{56}{5E+5} = 1.12\text{E}{-4}\ s$$

$$Transmission\ time\ (Data\ phase) = \frac{340}{1E+6} = 3.4\text{E}{-4}\ s$$

$$Transmission\ time\ (CAN\ FD) = (1.12\text{E}{-4}) + (3.4\text{E}{-4}) = 4.52\text{E}{-4}s$$

$$Bus\ load\ (CAN\ FD) = 100 \times \frac{4.52\text{E}{-4}}{0.01} = 4.52\%$$

The result is summarized in Table 5.4. The first row is for when having CAN FD but still with the normal bandwidth of 0.5 Mb/s.

Case	**Transmission time (s)**	**Bus load estimate 4 frames (%)**
Total (normal bandwidth, with CAN FD format)	7.92E−4	7.92
Rest: Header and footer	1.12E−4	1.12
Data phase	3.40E−4	3.40
Total (CAN FD): Data phase + Rest	4.52E−4	4.52

Table 5.4: Bus load estimate with CAN FD, with the cycle time set to 10 ms. Note that this is specifically for the overhead due to a 16 byte MAC.

The second row is the calculation of the CAN FD frame that needs to sent with

the arbitration/normal bandwidth. The third is for the data phase of the CAN FD frame, which can be sent with a higher bandwidth. In this case it is assumed to be 1 Mb/s. However, it might be possible to achieve even higher bandwidth depending on some parameters. Then the final result with CAN FD is shown in the last row, which just the sum of the previous two.

This can be compared to the earlier theoretical example with CAN, which was $4 * 3.42 = 13.68\%$. Meanwhile, with CAN FD (normal bandwidth, but flexible data payload) it was 7.92%. The performance gains in percentage:

$$Performance\ increase\ (\%) = 100 \times (1 - \frac{7.92}{13.68}) = 42.11\%$$

In other words, there is a 42% better bus load with just adopting the CAN FD protocol and still maintain the arbitration bandwidth.

The full CAN FD protocol has a bus load increase of 4.52%. The performance gains in percentage:

$$Performance\ increase\ with\ CAN\ FD\ (\%) = 100 \times (1 - \frac{4.52}{13.68}) = 66.95\%$$

In other words, with regards to the overhead that comes as a result of using GMAC (16 bytes) for a single arbitrary message type with cycle time of 10 ms, the bus load is roughly 67% better with CAN FD compared to CAN. Note that we only calculated the bus load resulting from the MAC, but in reality in CAN FD you would most likely send the whole message, including the MAC, in a single CAN FD frame. Furthermore, it is likely the case that the efficiency of CAN FD compared to CAN only would increase with larger data payloads.

6

Discussion

In this chapter, we include various aspects that are relevant to discuss in this thesis. Most importantly, sections 6.1 and 6.2 relate the results in section 5 to the research questions posed in this thesis. Additionally, section 6.3 details various limitations with the testing that was performed. Lastly, section 6.4 describes metrics that could have been relevant to include in this thesis, but that for different reasons were not.

6.1 Research Question 1

The first research question was: *What is the cryptographic performance of an HSM?* The main focus was to compare the HSM solution with a software solution, where the choice was wolfCrypt. This comparison should give a fairly justified evaluation regarding the HSM solution. The layout of this discussion is structured around the relevant metrics. At the end a summary is given regarding the findings and conclusions.

6.1.1 CPU Load and Offload

Firstly, an interesting observation was that there was no significant difference in CPU load for all block cipher modes for the HSM. This was not the case for software. This is a logical observation since the CPU load for the HSM is based on the *setup latency*, which is rather fixed in our testing in the different modes. Meanwhile, the CPU load for software is based on the *total latency*, which thus includes cryptographic latency. The second observation was that higher data size leads to higher CPU load. But the growth rate was much lower for the HSM compared to software (see Table 5.2 for the CPU load estimates).

To clarify, the reason that there is a CPU load increase on the ECU with the HSM is that there still occur some operations on the ECU, e.g., initial communication with the HSM and some handling of the data before sending it to the HSM. That is why a higher data size has more impact on the CPU load.

The biggest difference in CPU load between HSM and SW was for the block cipher mode GCM. In detail it was for the HSM it was 0.54% for 32 bytes and for SW it was approximately 2.5%. This is just a difference of roughly 2%. The CPU offload for 32 bytes is roughly 78%. Meanwhile, with 512 bytes it was approximately 1%

for HSM and 16% for SW in CPU load. The CPU offload is roughly 94%.

So, for a 16 times larger data size, the GCM CPU load (SW) grows with almost seven times. But for the HSM it was roughly a doubling in CPU load. This is a logical result since as stated earlier, for software, the CPU load metric includes *cryptographic latency*, which is amplified by data size. Meanwhile for the HSM it is *setup latency*, which is primarily sending and receiving the data size.

A first conclusion is that for the CPU load with HSM, the choice of block cipher mode is not a significant factor, i.e. there is no significant difference in CPU load for all block cipher modes. This is logical since for *setup latency* there is no significant difference in latency for all block cipher modes. A second conclusion is that higher data size leads to higher CPU load, both for the HSM and software. But the growth rate is significantly higher for software. This is due to that all the cryptographic operations are performed on the TriCore for software, which is reflected in CPU load.

It is also clear that GCM is especially more demanding for software compared to the HSM. This is shown by that software has significantly higher CPU load for the smaller data sizes (compared to the other modes) and also higher growth rate.

CPU offload is summarized in Table 5.3. A conclusion is that the HSM offloads the TriCore for almost every data size here. The only exception was CTR for smaller data sizes of 32 bytes (as well as 16 bytes as shown in Appendix C). The reason for this exception is that the *setup latency* for CTR (HSM) is larger than the *total latency* for CTR (SW). This is something that could be due to insufficient testing, since the differences are so small. Otherwise, this could be due to that the difference is negligible for software and HSM in this case.

Another conclusion is that CPU offload grows with data size in favour for the HSM. Thus, the HSM is decisively preferred when it comes to larger data sizes, in the perspective of CPU load. For example, for CBC, the CPU offload is approximately 12% for 32 bytes, and for 512 bytes it is approximately 76% in CPU offload. It is reasonable to assume that there exists some maximum possible CPU offload.

This can not be explained entirely by *offloading* the TriCore. Simply looking at offloading does not explicitly give a complete sense of overall *effectiveness*. In order to achieve this it is necessary to look at difference in *cryptographic lateny* between software and the HSM.

Reflection on Performance Requirements

As mentioned earlier in Section 4.1 in Method, there exists already a CPU load on the ECU. Thus, in our work it is assumed that there is already an average CPU load of 50 to 60% on the ECU. Furthermore, there exists a maximum CPU load, due to real-time demands, which is assumed to be around 80 to 85%. This leaves a range of roughly 20-30 % for increasing the CPU load. However, this is not purely dedicated to cryptographic functionality but also to other new functions. In

conclusion, a reasonable limit for this cryptographic functionality could be in the range of increasing the CPU load with 5-10 %.

Thus, with the CPU load (HSM) being 1.03% for 512 bytes and cycle time 10ms, there is some room for having several messages types with this fast cycle time. For example, with a cycle time of 1000ms, one could encrypt and decrypt roughly 970 message types ($10\%/(0.0103\%) \approx 970$ *message types*), without generating too much in CPU load.

Meanwhile, for software the CPU load estimate was 4.26% (CTR, 512 bytes, cycle time 10 ms). The corresponding example here would be maximum roughly 235 messages types, that could be encrypted and decrypted ($10\%/(0.0426\%) \approx 235$). Thus, it is clear that the HSM is preferred when it comes to CPU load. If the goal is to perform AES cryptography on as many message types as possible, then the HSM is the obvious choice when it comes to CPU load.

6.1.2 Cryptographic Latency

The first thing that is worth to conclude is that generally for the HSM, decryption was actually slightly faster than encryption (although a rather negligible difference). This is something that was not observed for software in our results, where encryption and decryption latency was about the same. This is something that is hard to see from the figures in Chapter 5, but can be observed easily from the calculated metrics in Appendix C. The only conclusion that can be drawn here is that the HSM does have some intrinsic attribute that makes decryption faster than encryption. What that attribute might be is something that would require in-depth knowledge about the intrinsic functionality of the HSM.

Most symmetric block ciphers, including AES, will take about the same latency (within some measurement error) for encryption and decryption, when it comes to operating on a single block (e.g., 128 bits). However, when it comes to operating on multiple blocks, parallelization is possible. For example, CBC encryption must be done sequentially for the blocks (i.e., encryption can not be parallelized), while decryption can be parallelized (XOR step). However, CTR can be parallelized in both encryption and decryption.

Another interesting aspect was that GCM was the only block cipher mode that differed in cryptographic latency, both for the HSM and software. The other modes ECB, CBC, and CTR was almost the same in cryptographic latency (with some minor exceptions). It was expected that GCM would have significantly higher latency, since when writing the code for GCM (HSM) it was evident that there were many more cryptographic operations being done.

The most likely reason that the other modes has negligible difference in cryptographic latency, is that the block cipher modes are so similar in their design, i.e. their operations. This was also the case for software, but there was slightly more variance in cryptographic latency between the selected block cipher modes. In conclusion, based on the results the difference in cryptographic latency between the

modes ECB, CBC, and CTR is somewhat negligible, both for the HSM and software, but mostly for HSM.

Difference in Cryptographic Latency

The conclusion here is that HSM is more effective in the aspect of cryptographic latency, and the difference grows in favour of the HSM with data size. Thus, the HSM (cryptographic accelerator) is significantly more effective compared to software when it comes to larger data sizes. Another interesting conclusion was that the difference in cryptographic latency for GCM had much higher growth rate compared to the rest of the modes. Thus, GCM was the mode that the HSM was significantly better at.

The only exception observed was that CTR (SW) for smaller data size (16, 32 bytes) was actually faster or even to the performance of the HSM.

6.1.3 Cryptographic Throughput

Firstly, both HSM and software has an increase in throughput (encryption and decryption) for the lower data sizes. However, for software the throughput growth stagnates after 256-bytes, see figures 5.10 and 5.11. Meanwhile, for the HSM the throughput growth does not stagnate and the throughput is significantly higher when compared to software, see figures 5.4 and 5.5.

The block cipher modes ECB, CBC and CTR have roughly the same throughput, for HSM respective software. The reason for this is that they have roughly the same *cryptographic latency* in HSM or software. The reason for this has been explained earlier. Furthermore, it is clear that GCM has significantly lower throughput for software compared to the other modes. This was the case for the HSM also. This was expected since GCM is the most complex block cipher of the ones studied in our work, due to it being an authenticated encryption mode.

6.1.4 Summary

The purpose of this comparison was to give a fairly good understanding of the possible performance gains with HSM. The overall conclusion to this first research question is that the HSM provides better performance, both in CPU load and cryptographic latency. This was especially true for the block cipher mode GCM, which was significantly more demanding for software, mainly due to higher growth rate in cryptographic latency.

6.2 Research Question 2

The second research question was: *What are the security and performance trade-offs of different HSM configurations?* Here the main idea was to study how different HSM configurations of varying security levels would differ in performance. Such relevant security parameters were key length, block cipher mode, MAC and IV.

However, due to restrictions (as detailed earlier in Chapter 4), key length could not be studied. Furthermore, the parameters MAC, and IV could only be studied to a lesser extent. The layout of this discussion is also structured around relevant metrics; bus load, cryptographic latency, and TRNG latency. At the end a summary is given regarding findings and conclusions.

6.2.1 Bus load

The idea here was to study how stronger security affects bus load. For example, adding some overhead such as a MAC in a message. As detailed in Chapter 4 MACs were quite limited for symmetric cryptography in the HSM. In principle, the only thing worth studying was regarding the MAC generated in GCM, which was fixed in size (128 bits). This was done in Chapter 5, both in the context of standard CAN, but also in CAN FD. This was something that was decided to be relevant rather early in the project. The hypothesis was that a MAC of 16 bytes would be rather infeasible in practice with CAN. This is because CAN is a relatively slow bus, and with limited data payload size (leading to overhead, e.g., segmentation). Therefore, it was interesting to look at CAN FD, which has the purpose of fixing those drawbacks.

Note that we could have studied different MAC sizes since our calculations are theoretical. The reason this was not done is that there is no real motivation for this since such MAC sizes would not be feasible with the HSM.

CAN

The most important conclusion here is that having a 16 byte MAC (GCM, HSM) is rather demanding for CAN, and would not be feasible in practice. This is due to that the additional bus load is estimated to be approximately 14%, for a cycle time of 10ms. Note that this was calculated for the worst-case scenario when it comes to the number of bits in a CAN frame. So, for example, an additional bus load of 5%, would only result in having a 16 byte MAC for roughly 37 message types, with the cycle time 1000ms. Thus, in order for this to be feasible, one can only have a MAC in messages types with slow cycle time.

CAN FD

The corresponding theoretical calculations for CAN FD resulted in an additional bus load estimate of approximately 4.5%, for a 16 byte MAC, with the cycle time 10ms. The difference in percentage points is thus roughly 9% in additional bus load for the MAC. This is equivalent to roughly 67% better performance in regards to bus load.

For example, an additional bus load of 5% would result in having a 16 byte MAC for roughly 110 message types, with the cycle time 1000ms. So, in our theoretical example, CAN FD has roughly a 67% increase in the number of message types that can include a MAC of 16 bytes. Thus, adding security through a MAC would be

more feasible with CAN FD but still create some significant additional bus load, thus not entirely feasible in practice.

It would be interesting to study how the MAC size affects the bus load in practice. Note that CAN FD bandwidth was assumed to be 1Mb/s, which is double the assumed normal CAN bandwidth. However, CAN FD (and CAN) has the possibility of achieving much higher bandwidth. There is some difference where both the practical and theoretical limit on bandwidth is for CAN FD. One commonly advertised number is 5Mb/s. Cable length is a main factor for the bandwidth, because longer cables are a constraint on higher bandwidth. For example, trucks and buses utilizes fairly long cables (10-20 meters). Typical bandwidths in this context for CAN are 250 kbit/s or 500 kbit/s. This is the motivation for the assumed CAN bandwidth of 500 kbit/s. However, a feasible CAN FD bandwidth in this context is more uncertain. Therefore, a modest assumption was made that the CAN FD bandwidth would be the double of the normal CAN bandwidth.

It is interesting to study optimizations when it comes to MAC in regards to bus load. One easy thing is to do a compromise; only those message types that from a security-perspective need to have a MAC, should have a MAC. In other words, not every message type in a network needs to use GCM.

Reflection on Performance Requirements

Looking back at the performance requirements for bus load (in Section 4.1 in Method). The assumed max limit of the bus load is assumed to be around 70 to 80% (due to real-time demands). Furthermore, some global CAN links are assumed to be quite sparse in available bus load to use. But there still exists some communication links such as local subnetworks, which are deemed to have at least some room for additional bus load.

The resulting additional bus load with CAN (as detailed earlier) is not really feasible with this limitation. CAN FD would be more in line with this limitation since it has a 67% performance increase compared to CAN, when it comes to a MAC size of 16 bytes. Still, within this limitation, it is reasonable that one could have a MAC for only a limited set of message types, preferably with a slow cycle time.

Thus, the conclusion is that the security trade-off with performance is quite significant, when it comes to bus load. This is due to the high overhead resulting from the MAC (GCM). This together with the assumption of already limited available bus load, means that it is reasonably only a limited number of message types, preferably with slow cycle time, that can have a MAC. Software is likely more flexible where for example other MAC schemes are possible and MAC size is more flexible. Using both HSM and software is fully supported and feasible for real practice. However, it would increase the complexity of the security solution.

6.2.2 Cryptographic Latency and Throughput

The most important trade-off between security and performance when it comes to cryptographic latency, is the block cipher mode GCM. In our results, GCM was the only mode that had higher cryptographic latency. This is expected as GCM is the only authenticated encryption mode, meaning that GCM includes generating and verifying a MAC. The trade-off is then that GCM provides better security than the other modes but at higher performance cost.

6.2.3 TRNG Latency

In testing, the IV and the key had fixed values, i.e., not fresh values from the TRNG. In a real setting one would want to use fresh values from the TRNG or at least pseudorandom, to use for the key and IVs. It is vital to use good cryptographic material (true random numbers) in order to ensure good security.

It is important that the key and IV are unique and that the generation of them is unpredictable, i.e. randomness is required. Uniqueness is required since if a key or IV is reused or accidentally the same as another used, then an adversary could potentially break the cipher.

An IV has different security requirements than a key, and the requirements differ between some modes. In most cases, it is important that an IV is never reused under the same key, i.e. the IV needs to be a cryptographic nonce (an arbitrary number which is single use). For example, in CBC reusing an IV leaks some information about the first block of plaintext, and about any common prefix shared by the two messages. Therefore, in CBC mode, the IV must be unpredictable (either random or pseudorandom) at encryption time. In particular, for any given plaintext, it must not be possible to predict the IV for that plaintext in advance of the generation of the IV. A common example of this is the previously common practice of re-using the last ciphertext block a message as the IV for the next message. This was for example used in SSL 2.0, search TLS CBC IV attack (aka. BEAST attack) for more information.

For CTR, reusing an IV causes key bitstream re-use and this breaks security. This effectively means that the resulting bitstream, which is XORed with the plaintext, is dependent on the key and IV only.

As explained earlier, the security requirements vary depending on block cipher mode. In general it is important that the IV is unpredictable and not reused (unique) under the same key. It is recommended to review relevant IV security requirements for specific block cipher modes before practical use, e.g. NIST provide such recommendations.

The larger problem arises when looking at the possible frequent use of IVs. If one needs, for example, to call the TRNG every time an encryption occurs, then the *TRNG latency* that would be added to *total latency* would be quite significant. This could potentially have quite a significant impact on overall performance since

it would occur every encryption.

A sample test showed that the added latency to the *total latency* was roughly 380 microseconds. This sample test was simply fetching an IV from the TRNG and then perform encryption and decryption with some arbitrary block cipher mode. Then take the difference between this and the normal case where the IV is "hard coded".

From the Table 5.1, it can be concluded more exactly that it roughy takes 300 microseconds for the TRNG to generate a 128 bit random value. Note that this does not include any related operations performed on the ECU, which can be calculated as the difference between the *total latency* and the *TRNG latency*, it is roughly $40\mu s$ regardless of the size.

This is a relatively high additional latency, especially for smaller data sizes. The TRNG latency here is fixed, and will thus become less significant of the total latency for larger data sizes. Furthermore, the TRNG latency was linear with data size. Thus, there are no real performance gains for generating larger data sizes. Lastly, it is worth mentioning that the TRNG can work in parallel with the cryptographic accelerator in the HSM.

In conclusion, the security trade-off with performance could potentially be enormously significant. It depends on the frequency and how good (security implications) the cryptographic material (keys and IVs) needs to be. The most important factor is if it is necessary for security that the IV needs to be unpredictable or not, for the used block cipher mode. There are possible optimizations and compromises in this area.

A compromise could for example be that session keys should only be updated at software updates of the TriCore, or some other rather infrequent period. Meanwhile, for IVs, it could be that they are pseduorandom, which would make them more predictable, but would lead to better performance. One implementation could for example be inputting a true random number in a software algorithm that generates a new IV. This would lead to a doubling in the number of generated IVs.

Lastly, a possible optimization would be the use of a cryptographic material (true random numbers) pool. Since the TRNG can work in parallel with the cryptographic accelerator in the HSM, one could think the TRNG could continuously fill a pool of cryptographic material that could be for later use. This would lead to more efficient performance as the number of cryptographic operations that are needed when encryption is performed would be fewer. At least from the perspective of the *user* of the ECU it would be seen as a performance gain. This is the main motivation of this solution, since it would mean a decrease in the *total latency* for encryption, as now the ECU could directly fetch an IV from memory, instead of calling on the TRNG. However, this would incur a higher memory usage. Furthermore, this would also lead to lower security, as such a pool would likely need to be stored in software (since the HSM has quite limited secured storage), which would make it more vulnerable.

6.2.4 Summary

The intention of this research question was to gain an understanding of the possible trade-offs between security and performance.

The most important conclusion here is that the MAC generated in GCM (HSM) results in quite a significant additional bus load, making it infeasible in practice to be used for CAN. However, for CAN FD it is more feasible since CAN FD has significantly higher performance in this aspect with 67%. However, since the MAC size is so large, it would still only be a limited set of message types that could use a MAC, preferably those that have a slow cycle time.

The next interesting conclusion is regarding *TRNG latency*, which is relevant when it comes to security and performance regarding keys and IVs. The conclusion here is that there could potentially be quite a trade-off in performance. As for example, generating IVs for every encryption would imply an additional latency of at least 350 microseconds. This is quite large compared to the cryptographic latency, and this would occur quite frequent. Thus, having unpredictable IVs would result in significantly worse performance. The same applies for keys, but there key cycle update time and the number of session keys are the relevant parameters.

The only significant concluded trade-off between the block cipher modes is regarding GCM, which has a higher cryptographic latency. This is logical, since GCM is the only authenticated encryption mode (generates and verifies a MAC).

Another aspect worth mentioning in this research question is regarding the security possibilities of the HSM. More specifically, the fact that the HSM operates only in AES-128. Since the key length and MAC are restricted to a size of 128 bits, there is a limitation on security on the hardware level of the HSM. In other words, it is not possible with the HSM to achieve a higher degree of security by using aspects such as longer key length or larger MAC, at the cost of performance (i.e. longer computation time).

More detailed, it is substantially difficult or unpractical to make key length different to the default 128-bit key length. For example, the AES cryptographic accelerator is optimized for 128-bit keys, and the secured storage is also optimized for 128-bit keys. Thus, one could argue that this is a trade-off between performance and security, in favour of performance. This is a limitation that software is likely more flexible around, i.e. implementing AES-256 is more feasible in software. Note that AES-128 still enables relatively good security, it is just question of how far one wants to increase security.

6.3 Limitations with Testing

In this section we discuss various limitations related to the testing performed in this thesis. This includes limitations with the program code, the software cryptographic library that was used as well as the testbed that was used in this thesis.

6.3.1 HSM Code

There are some limitations with how the HSM code was implemented, as detailed below.

Data Verification

As part of the setup code for the HSM, we have included some additional code that is used to verify that the data was encrypted/decrypted correctly. This basically consists of comparing the initial (unencrypted) data with the final data that has gone through both encryption and decryption. The purpose of this code was to ensure that the cryptographic operations performed by the HSM worked as they should, something which was important in the beginning stages of testing. Ultimately though, this piece of code could be considered redundant, and as a result, adds an unnecessary extra latency to the measured *setup latency*.

The reason for why this code was not removed is the following. First of all, this code occupies just a small fraction of the setup code. Therefore, we did not consider it to contribute significantly to the *setup latency*. Secondly, we also note that this type of verification code would likely need to be present anyways in code written for real, industrial purposes. Along with other additional code that is likely to be included, the latency added from real industrial code might therefore actually be higher.

Hardcoded Data

As part of the HSM code, a lot of hardcoded data was used. For example, the same piece of data was always used for encryption – if a larger data size was needed this data was simply replicated. Additionally, cryptographic keys and IVs were fixed and always the same. As a result, this may have contributed to the measured latencies not being reflective of real-world scenarios, where more variable data would be used. In reality, these latencies would likely be higher since more setup probably would need to be performed.

Lacking Example Code and Documentation

In order to write the HSM code, we had to use example code that was provided to us from Infineon, as no real production code for the HSM was available. Because the example code was rather incomplete – for example, not all cipher modes were implemented – we had to supplement it with code from the technical specification for the AURIX. Since also this code was quite incomplete – and written in pseudocode on top of everything – we were limited in the code that we were able to write. As a result of this, there were some optimizations to the code that we were not able to do.

Firstly, when using GCM encryption in general, the possibility exists of including some plaintext data along with the encrypted data – this data is called additional authenticated data (AAD). The integrity of the AAD is thus protected by the MAC that is created during GCM encryption. An example of when this could be useful

is for message headers; the payload of the message may need to be encrypted, but the header should remain in plaintext. In this thesis though, using AAD is not that relevant as we only want to encrypt data and create a MAC for the encrypted data. However, the pesudocode that was available for GCM required that some AAD was included – more specifically, 3 blocks of 128 bits each. This thus adds some additional latency to the measured cryptographic latency for GCM. It should be noted though that when encrypting large data sizes, this added latency becomes negligible.

Another optimization that could have been made is the fact that the HSM can run in a pipelined mode. In pipelined mode, data can be loaded into the HSM while the HSM is still performing encryption/decryption, thus increasing the performance of the HSM. Again, the available documentation did not provide enough guidance as to how to properly implement pipelined mode.

We note that if enough time had been available, it may have been possible to implement these optimizations. We did however not consider it to be worth the time of trying to do this.

6.3.2 Software Cryptography

There are some limitations with how software cryptography was used in our work.

Multiple Cores

Even though the AURIX offers three CPU cores, the example code provided only allowed for the use of one core. As some cipher modes – ECB and CTR – can make use of parallel execution and multiple cores, this could have potentially increased the performance of software cryptography when using these modes. It should be noted though that it is unclear whether wolfSSL/wolfCrypt would have been able to make use of this.

MICROSAR Security

As for the choice of software cryptography solution to use in this thesis, the initial plan was to use MICROSAR security by Vector [2], which is a software cryptography solution that is commonly used in vehicles today. Due to unforeseen delays though, the prerequisite tools needed in order to work with MICROSAR could not be provided, and therefore it could not be incorporated into the thesis. Since MICROSAR security is specifically designed to be used in vehicle environments and ECUs, had it been used instead of wolfSSL/wolfCrypt we could have potentially seen an increase in the performance of software cryptography.

TRNG latency Using Software

As mentioned in section 4.2.3, TRNG latency was not measured using software due to the relative difficulty and potential time consumption of the task. Had more time been available, a measurement comparable to the one done using the HSM could

possibly have been done in software as well. That way, TRNG latency could have been included in research question 1.

6.3.3 Testbed

Initially, the plan was that testing would be performed on more advanced testbeds than the one that was ultimately used in this thesis. However, because of time constraints, this was not possible. As a result, some adjustments had to be made in this thesis.

The plan was to include communication latency as a metric in this thesis, which is basically the end-to-end latency of a message (from sender to receiver). This was not possible though, as that would require a testbed which allowed for CAN communication.

Furthermore, had a testbed with CAN communication been available, it would have been possible to measure bus load using a tool called CANalyzer. Instead, a more theoretical method for measuring bus load was used. It should be noted though that this theoretical method would have been used anyway; using a CANalyzer would just have contributed with some complementary results, which may or may not have been useful.

Refer to section 7 for illustrations on how these more advanced testbeds would have looked like.

6.4 Excluded Metrics

In this section we discuss metrics that we for different reasons (as described below) decided not to include in this thesis. These are metrics that could have been relevant in a different or broader context. Therefore, we include them as future work in section 7.

6.4.1 Communication Latency

The initial plan was to include this metric in the thesis, but this would have required a testbed consisting of multiple nodes connected by a (CAN) network. It turned out that there was not sufficient time to introduce this kind of testbed into the project. Therefore, this metric had to be excluded from the thesis in the end.

6.4.2 Memory Utilization

The intention was to include this metric as well in the thesis. It was unclear however whether it would be possible to find an effective method of measuring this metric as there were some unknown factors with regards to memory allocation, and the metric was not interesting enough to invest too much time into this. Therefore, we decided to ultimately exclude this metric.

6.4.3 Key Update Time

Since this metric has more to do with communication performance rather than the performance of the cryptographic module itself, we considered it to be out of scope for this thesis, as we are only concerned with evaluating the performance of the HSM. However, had we done a broader investigation of security in vehicles, this could have been a metric relevant to consider.

6.4.4 Heat Production and Power Consumption

It is reasonable to assume that the use of an HSM may have some effect on heat production and power consumption of a system. These are aspects that therefore could be worth considering. However, these areas were not considered to be relevant to look at. Firstly because these metrics would have been hard to measure. Secondly, the size of the AURIX is relatively small compared to other vehicle components. Since there is a wealth of both cooling and power resources available for the powertrain ECUs, heat production and power consumption would not have been an issue. If we, on the other hand, had investigated ECUs that were more power-constrained and had limited cooling, this metric could have been more relevant to include.

7

Future Work

This chapter details some work that was not realized within the timeframe and some interesting areas of possible future work.

7.1 Vector MICROSAR Security

MICROSAR is the AUTOSAR (AUTomotive Open System ARchitecture) solution from Vector, i.e., embedded software for ECUs. MICROSAR consists of its runtime environment, which is called MICROSAR.RTE, and its Basic Software Modules (BSW). They cover the entire AUTOSAR Classic standard. Furthermore, they also provide useful extensions. [2] AUTOSAR is a open and standardized software architecture for automotive ECUs. It is a global development partnership of automotive parties, with many different levels of partnership. [59] [1] [60]

It was of high interest to perform a comparison of HSM and a software solution. In this thesis the software used was wolfCrypt. The original idea was to use Vector MICROSAR Security as the software solution, with its software module Crypto (SW). The plan was that a separate team would, in parallel with our work, integrate MICROSAR Security for the TriCore. However, it got delayed and the decision was made to instead focus on some open-source solution, since the implementation work effort was deemed too large for ourselves. Still, MICROSAR Security is of interest to compare with as the software solution, thus it is left as future work. The reason MICROSAR is interesting to use as a comparison when it comes to the software solution, is because it is on an industry-standard level and would likely be the competitor to the use of an HSM.

7.2 TRNG (SW)

The TRNG of the HSM was evaluated. But a software-based TRNG was not evaluated due to the time frame. However, such a comparison is still of interest since it would enable a study of how much better hardware-based TRNG is compared to software (algorithms). Note that the performance of the software-based TRNG would be based on one particular algorithm, and that there likely exists different algorithms.

7.3 CAN

It would be interesting to study practical use of the HSM, i.e. applying encryption on some CAN data payloads. This would involve some necessary operations such as extracting the data payload, perform cryptographic operations, and then inserting it back into its CAN frame. However, this would require a more advanced testbed. Furthermore, such a testbed would enable studying communication latency, which is a metric of interest.

The idea here was to create some arbitrary testing CAN frames along with arbitrary data payloads (8 bytes). The testing CAN frames is only doing a setup of the fields in the frame, in order to ensure correct functional CAN communication. Note that it is CAN extended that would have been utilized here since it is what we have used in our theoretical work.

7.3.1 Testing - Testbed Iteration II

In this intended second iteration of the testbed there exists a CAN communication link. The design is illustrated in figure 7.1. Note that in a sense there exists two CAN nodes. However, the CAN dongle (see Appendix B, Section B.3 for more information on CAN dongle) is not a genuine CAN node, but it can still send and receive CAN frames.

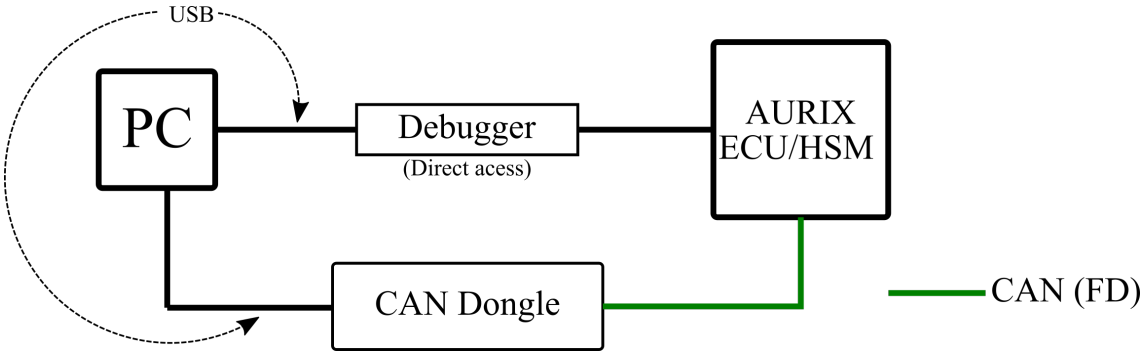

Figure 7.1: Design of testbed iteration II, where the addition is a CAN dongle.

7.3.2 Testing - Testbed Iteration III

Testbed iteration III has an addition of a second AURIX ECU with an HSM. The design is illustrated in figure 7.2. Now one can for example, try to simulate some encrypted standard communication between TCM and ECM.

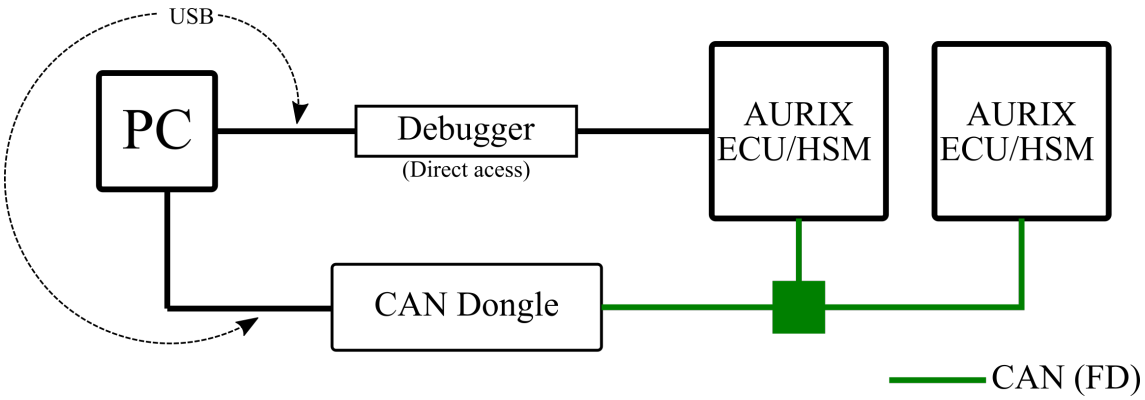

Figure 7.2: Design of testbed iteration III, which is a continuation of the previous iteration. Now there exists in a sense an actual CAN bus in the testbed, since there are two AURIX nodes.

7.3.3 CAN FD

Testing on the testbed with CAN FD would be interesting since it would give a better insight of the performance difference. The only difference between CAN and CAN FD is in software, not in the physical layer. Thus, given that there needs to be no changes in the physical layer, it should be relatively easy to integrate CAN FD in the testbed. The only requirements are that the ECUs and the CAN dongle have CAN FD support.

7.4 Asymmetric Cryptography

It would be interesting to study the asymmetric cryptographic performance of an HSM, such as the AURIX TC3xx. One reason is that with asymmetric cryptography, it is possible to perform e.g., signatures.

8

Conclusion

The aim of this work was to do a performance evaluation of an HSM from two different viewpoints: 1) comparing the performance of HSM versus SW and 2) investigating security and performance trade-offs of different HSM configurations.

When it comes to the first point, the performance of an HSM was compared to the performance of using a standard cryptographic library (wolfCrypt to be precise) on the ECU. It was concluded that using an HSM provides a considerable improvement of performance, both in terms of providing lower cryptographic latency (and thus higher throughput) as well as offloading the ECU CPU. The difference in cryptographic latency grows with data size in favour of the HSM, especially for the block cipher mode GCM. This means that the HSM has significantly higher growth rate with data size for cryptographic throughput, e.g., the HSM can achieve as much as four times higher throughput compared to SW. With regards to CPU offloading, for a data size of, e.g., 512 bytes, the HSM offloads the ECU CPU between approximately 75 to 94 % depending on the block cipher mode, with GCM being the cipher mode that benefits the most from the HSM offloading.

With regards to security and performance trade-offs of different HSM configurations, the only variable of the different configurations was the utilized block cipher mode, as it was not possible vary any other factor, e.g., key length. The conclusions vary depending on which performance aspects that are taken into consideration. If bus load is not considered, the difference in performance between the different block cipher modes are minor. GCM does for example have a lower encryption throughput than other block cipher modes, but other than that the results are quite similar. Therefore, in this case, there is no clear motivation for not utilizing GCM since it provides a significantly higher degree of security due to the authentication aspect of GCM, which the other modes do not have.

When it comes to bus load however, the trade-offs can potentially be quite significant. For example, GCM includes the use of a MAC (to be precise GMAC) of size 128 bits. The result is that for a single message CAN frame there will be an overhead of four CAN frames (where two are for the MAC itself). Although adding a MAC to the encrypted data provides message authentication, it does come with a high bus load cost. In conclusion, using GCM (i.e. incl. GMAC, otherwise it is CTR) is somewhat infeasible in many cases such as when the available bus load is limited or when any larger number of messages need to be encrypted.

Nonetheless, GCM could still be applicable if only a small fraction of messages need to be encrypted, such as messages that require higher security and preferably are sent less frequently. Additionally, if CAN-FD is utilized instead of CAN, the bus load can roughly be decreased by 67 % when compared to CAN, according to our theoretical estimates. This is because with CAN-FD, the data payload can be sent at a significantly higher bandwidth. Furthermore, segmentation is avoided since in CAN-FD the whole overhead from the addition of the MAC can be sent in a single CAN-FD frame. In conclusion, the use of CAN-FD makes it significantly more feasible to use GCM (and MAC:s in general).

Lastly, another conclusion is regarding *TRNG latency*, which is relevant when it comes to security and performance regarding unpredictable keys and IVs. The conclusion here is that there could potentially be quite a significant trade-off in performance, i.e., unpredictability of keys and IVs comes at a significant cost. The frequency of generating keys is likely significantly lower than generation of IVs. Therefore, the performance constraints likely lies on the latter. For example, having the TRNG generate IVs for every encryption would imply an additional latency of at least 350 microseconds, which is a relatively significant part of the then resulting *total latency*. Thus, having unpredictable IVs would result in significantly worse performance. If high performance is vital, then alternative methods need to be utilized that do not provide the same level of randomness.

Bibliography

[1] Vector Informatik GmbH. (2021). "Learning Module AUTOSAR," [Online]. Available: https://elearning.vector.com/mod/page/view.php?id=437 (visited on 04/11/2021).

[2] ——, (2021). "MICROSAR," [Online]. Available: https://www.vector.com/int/en/products/products-a-z/embedded-components/microsar/ (visited on 04/11/2021).

[3] EVITA. (2011). "Evita e-safety vehicle intrusion protected applications," [Online]. Available: https://www.evita-project.org/index.html.

[4] wolfSSL. (2021). "The wolfSSL embedded TLS library," [Online]. Available: https://www.wolfssl.com/ (visited on 07/08/2021).

[5] Infineon Technologies AG. (2020). "Aurix™ security solutions," [Online]. Available: https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/aurix-security-solutions/ (visited on 12/08/2020).

[6] W. Stallings, *Cryptography and Network Security: Principles and Practice*, 7th ed. Pearson Education, 2017, ISBN: 9781292158587.

[7] Infineon Technologies AG. (2019). "Aurix™ hardware security module for cyber security," [Online]. Available: https://www.infineon.com/cms/media/eLearning/Automotive/aurix-hardware-security/presentation_html5.html?lms=1 (visited on 08/12/2020).

[8] Infineon Technologies. (2019). "AURIX Training Hardware Security Module," [Online]. Available: https://www.infineon.com/dgdl/Infineon-AURIX_Hardware_Security_Module-Training-v01_01-EN.pdf?fileId=5546d46269bda8df0169ca6e34c62549 (visited on 04/22/2021).

[9] ——, (2020). "AURIX Training Hardware Security Module," [Online]. Available: %7Bhttps://www.infineon.com/dgdl/Infineon-AURIX_TC3xx_Hardware_Security_Module_Quick-Training-v01_00-EN.pdf?fileId=5546d46274cf54d50174da4ebc3f2265%7D (visited on 04/22/2021).

[10] ——, *Automotive Cyber Security Compendium*, 2019. [Online]. Available: https://www.infineon.com/cms/en/product/promopages/car-security-whitepaper/ (visited on 04/22/2021).

[11] Lauterbach. (2021). "Trace32 tricore debugger," [Online]. Available: https://www.lauterbach.com/frames.html?bdmtc.html (visited on 04/22/2021).

[12] W. Stallings and L. Brown, *Computer Security: Principles and Practice*. Pearson Education, 2011, ISBN: 9780132775069.

[13] A. Ramesh and A. Suruliandi, "Performance analysis of encryption algorithms for Information Security," in *2013 International Conference on Circuits, Power and Computing Technologies (ICCPCT)*, 2013, pp. 840–844. DOI: 10.1109/ICCPCT.2013.6528957.

[14] M. Wolf and T. Gendrullis, "Design, Implementation, and Evaluation of a Vehicular Hardware Security Module," in *Information Security and Cryptology - ICISC 2011*, H. Kim, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 302–318, ISBN: 978-3-642-31912-9. DOI: 10.1007/978-3-642-31912-9_20.

[15] J. Seol, S. Jin, D. Lee, J. Huh, and S. Maeng, "A Trusted IaaS Environment with Hardware Security Module," *IEEE Transactions on Services Computing*, vol. 9, no. 3, pp. 343–356, 2016. DOI: 10.1109/TSC.2015.2392099.

[16] N. Samir, Y. Gamal, A. N. El-Zeiny, O. Mahmoud, A. Shawky, A. Saeed, and H. Mostafa, "Energy-Adaptive Lightweight Hardware Security Module using Partial Dynamic Reconfiguration for Energy Limited Internet of Things Applications," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2019, pp. 1–4. DOI: 10.1109/ISCAS.2019.8702315.

[17] W. Hupp, A. Hasandka, R. S. de Carvalho, and D. Saleem, "Module-OT: A Hardware Security Module for Operational Technology," in *2020 IEEE Texas Power and Energy Conference (TPEC)*, 2020, pp. 1–6. DOI: 10.1109/TPEC48276.2020.9042540.

[18] F. Cifuentes, A. Hevia, F. Montoto, T. Barros, V. Ramiro, and J. Bustos-Jiménez, "Poor Man's Hardware Security Module (PmHSM): A Threshold Cryptographic Backend for DNSSEC," in *Proceedings of the 9th Latin America Networking Conference*, ser. LANC '16, Valparaiso, Chile: Association for Computing Machinery, 2016, pp. 59–64, ISBN: 9781450345910. DOI: 10.1145/2998373.2998452. [Online]. Available: https://doi.org/10.1145/2998373.2998452.

[19] H. Chen and B. Yang, "A Performance Evaluation of CAN Encryption," in *2019 First IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, 2019, pp. 140–149. DOI: 10.1109/TPS-ISA48467.2019.00025.

[20] S. Woo, H. J. Jo, I. S. Kim, and D. H. Lee, "A Practical Security Architecture for In-Vehicle CAN-FD," *IEEE Transactions on Intelligent Transportation Systems*, vol. 17, no. 8, pp. 2248–2261, 2016. DOI: 10.1109/TITS.2016.2519464.

[21] S. Woo, H. J. Jo, and D. H. Lee, "A Practical Wireless Attack on the Connected Car and Security Protocol for In-Vehicle CAN," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 2, pp. 993–1006, 2015. DOI: 10.1109/TITS.2014.2351612.

[22] Z. Lu, Q. Wang, X. Chen, G. Qu, Y. Lyu, and Z. Liu, "LEAP: A Lightweight Encryption and Authentication Protocol for In-Vehicle Communications," in *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, 2019, pp. 1158–1164. DOI: 10.1109/ITSC.2019.8917500.

[23] D. Zelle, C. Krauß, H. Strauß, and K. Schmidt, "On Using TLS to Secure In-Vehicle Networks," in *Proceedings of the 12th International Conference on*

Availability, Reliability and Security, ser. ARES '17, Reggio Calabria, Italy: Association for Computing Machinery, 2017, ISBN: 9781450352574. DOI: 10.1145/3098954.3105824. [Online]. Available: https://doi.org/10.1145/3098954.3105824.

[24] H. Zhang, X. Hu, J. Li, and H. Guan, "A comprehensive test framework for cryptographic accelerators in the cloud," *Journal of Systems Architecture*, vol. 113, p. 101 873, 2021, ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2020.101873. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762120301533.

[25] S.-H. Hung and P. Bhattacharya, "On the Delivered Performance of the Sun Crypto Accelerator 1000," in *Sun User Performance Group Conference*, Honolulu, 2002.

[26] M. Togan, A. Floarea, and G. Budariu, "Design and implementation of cryptographic modules on FPGA," in *Proceedings of the Applied Mathematics and Informatics*, 2010, pp. 149–154.

[27] T. W. Arnold and L. P. Van Doorn, "The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer," *IBM Journal of Research and Development*, vol. 48, no. 3.4, pp. 475–487, 2004. DOI: 10.1147/rd.483.0475.

[28] Y. Hasegawa, S. Abe, H. Matsutani, H. Amano, K. Anjo, and T. Awashima, "An adaptive cryptographic accelerator for IPsec on dynamically reconfigurable processor," in *Proceedings. 2005 IEEE International Conference on Field-Programmable Technology, 2005.*, 2005, pp. 163–170. DOI: 10.1109/FPT.2005.1568541.

[29] A. Krishna, T. Heil, N. Lindberg, F. Toussi, and S. VanderWiel, "Hardware Acceleration in the IBM PowerEN Processor: Architecture and Performance," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12, Minneapolis, Minnesota, USA: Association for Computing Machinery, 2012, pp. 389–400, ISBN: 9781450311823. DOI: 10.1145/2370816.2370872. [Online]. Available: https://doi.org/10.1145/2370816.2370872.

[30] B. Blaner, B. Abali, B. M. Bass, S. Chari, R. Kalla, S. Kunkel, K. Lauricella, R. Leavens, J. J. Reilly, and P. A. Sandon, "IBM POWER7+ processor on-chip accelerators for cryptography and active memory expansion," *IBM Journal of Research and Development*, vol. 57, no. 6, 3:1–3:16, 2013. DOI: 10.1147/JRD.2013.2280090.

[31] M. Khalil-Hani, V. P. Nambiar, and M. N. Marsono, "Hardware Acceleration of OpenSSL Cryptographic Functions for High-Performance Internet Security," in *2010 International Conference on Intelligent Systems, Modelling and Simulation*, 2010, pp. 374–379. DOI: 10.1109/ISMS.2010.89.

[32] W. Pan, F. Zheng, Y. Zhao, W. Zhu, and J. Jing, "An Efficient Elliptic Curve Cryptography Signature Server With GPU Acceleration," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 1, pp. 111–122, 2017. DOI: 10.1109/TIFS.2016.2603974.

[33] N. Nishikawa, K. Iwai, and T. Kurokawa, "High-Performance Symmetric Block Ciphers on CUDA," in *2011 Second International Conference on Networking and Computing*, 2011, pp. 221–227. DOI: 10.1109/ICNC.2011.40.

[34] A. Hodjat and I. Verbauwhede, "Interfacing a high speed crypto accelerator to an embedded CPU," in *Conference Record of the Thirty-Eighth Asilomar Conference on Signals, Systems and Computers, 2004.*, vol. 1, 2004, 488–492 Vol.1. DOI: 10.1109/ACSSC.2004.1399180.

[35] D. Altolini, V. Lakkundi, N. Bui, C. Tapparello, and M. Rossi, "Low power link layer security for IoT: Implementation and performance analysis," in *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2013, pp. 919–925. DOI: 10.1109/IWCMC.2013.6583680.

[36] J. Hughes, G. Morton, J. Pechanec, C. Schuba, L. Spracklen, and B. Yenduri, "Transparent multi-core cryptographic support on Niagara CMT Processors," in *2009 ICSE Workshop on Multicore Software Engineering*, 2009, pp. 81–88. DOI: 10.1109/IWMSE.2009.5071387.

[37] G. Saggese, L. Romano, N. Mazzocca, and A. Mazzeo, "A tamper resistant hardware accelerator for RSA cryptographic applications," *Journal of Systems Architecture*, vol. 50, no. 12, pp. 711–727, 2004, ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2004.04.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762104000578.

[38] A. Boudguiga, W. Klaudel, and J. D. Wesolowski, "On the Performance of Freescale i.MX6 Cryptographic Acceleration and Assurance Module," in *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, ser. RAPIDO '15, Amsterdam, Holland: Association for Computing Machinery, 2015, ISBN: 9781605586991. DOI: 10.1145/2693433.2693441. [Online]. Available: https://doi.org/10.1145/2693433.2693441.

[39] C. Manifavas, G. Hatzivasilis, K. Fysarakis, and K. Rantos, "Lightweight Cryptography for Embedded Systems – A Comparative Analysis," in *Data Privacy Management and Autonomous Spontaneous Security*, J. Garcia-Alfaro, G. Lioudakis, N. Cuppens-Boulahia, S. Foley, and W. M. Fitzgerald, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 333–349, ISBN: 978-3-642-54568-9.

[40] D. F. Pigatto, N. B. F. da Silva, and K. C. Branco, "Performance evaluation and comparison of algorithms for elliptic curve cryptography with El-Gamal based on MIRACL and RELIC libraries," *Journal of Applied Computing Research*, vol. 1, no. 2, pp. 95–103, 2011. DOI: 10.4013/jacr.2011.12.04.

[41] O. Hyncica, P. Kucera, P. Honzik, and P. Fiedler, "Performance evaluation of symmetric cryptography in embedded systems," in *Proceedings of the 6th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems*, vol. 1, 2011, pp. 277–282. DOI: 10.1109/IDAACS.2011.6072756.

[42] M. Lee, H.-S. Lim, Y. Yang, and S.-h. Kim, "A Fast AES Hardware Security Module for Internet of Things Applications," *International Journal on Advanced Science, Engineering and Information Technology*, vol. 10, pp. 1346–1352, 2020.

[43] N. Khan, N. Sakib, I. Jerin, S. Quader, and A. Chakrabarty, "Performance analysis of security algorithms for IoT devices," in *2017 IEEE Region 10*

Humanitarian Technology Conference (R10-HTC), 2017, pp. 130–133. DOI: 10.1109/R10-HTC.2017.8288923.

[44] L. E. Kane, J. J. Chen, R. Thomas, V. Liu, and M. Mckague, "Security and Performance in IoT: A Balancing Act," *IEEE Access*, vol. 8, pp. 121 969–121 986, 2020. DOI: 10.1109/ACCESS.2020.3007536.

[45] A. O. Montoya B., M. A. Muñoz G., and S. T. Kofuji, "Performance analysis of encryption algorithms on mobile devices," in *2013 47th International Carnahan Conference on Security Technology (ICCST)*, 2013, pp. 1–6. DOI: 10.1109/CCST.2013.6922058.

[46] J. Hajny, L. Malina, Z. Martinasek, and O. Tethal, "Performance Evaluation of Primitives for Privacy-Enhancing Cryptography on Current Smart-Cards and Smart-Phones," in *Data Privacy Management and Autonomous Spontaneous Security*, J. Garcia-Alfaro, G. Lioudakis, N. Cuppens-Boulahia, S. Foley, and W. M. Fitzgerald, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 17–33, ISBN: 978-3-642-54568-9.

[47] A. Kardi, R. Zagrouba, and M. Alqahtani, "Performance Evaluation of RSA and Elliptic Curve Cryptography in Wireless Sensor Networks," in *2018 21st Saudi Computer Society National Computer Conference (NCC)*, 2018, pp. 1–6. DOI: 10.1109/NCG.2018.8592963.

[48] A. Liu and P. Ning, "TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks," in *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*, 2008, pp. 245–256. DOI: 10.1109/IPSN.2008.47.

[49] O. G. Abood, M. A. Elsadd, and S. K. Guirguis, "Investigation of cryptography algorithms used for security and privacy protection in smart grid," in *2017 Nineteenth International Middle East Power Systems Conference (MEPCON)*, 2017, pp. 644–649. DOI: 10.1109/MEPCON.2017.8301249.

[50] V. Gupta, S. Gupta, S. Chang, and D. Stebila, "Performance Analysis of Elliptic Curve Cryptography for SSL," in *Proceedings of the 1st ACM Workshop on Wireless Security*, ser. WiSE '02, Atlanta, GA, USA: Association for Computing Machinery, 2002, pp. 87–94, ISBN: 1581135858. DOI: 10.1145/570681.570691. [Online]. Available: https://doi.org/10.1145/570681.570691.

[51] O. Elkeelany, M. M. Matalgah, K. P. Sheikh, M. Thaker, G. Chaudhry, D. Medhi, and J. Qaddour, "Performance analysis of IPSec protocol: encryption and authentication," in *2002 IEEE International Conference on Communications. Conference Proceedings. ICC 2002 (Cat. No.02CH37333)*, vol. 2, 2002, 1164–1168 vol.2. DOI: 10.1109/ICC.2002.997033.

[52] K. K. Chennam, L. Muddana, and R. K. Aluvalu, "Performance analysis of various encryption algorithms for usage in multistage encryption for securing data in cloud," in *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*, 2017, pp. 2030–2033. DOI: 10.1109/RTEICT.2017.8256955.

[53] C. Narendiran, S. Albert Rabara, and N. Rajendran, "Performance evaluation on end-to-end security architecture for mobile banking system," in *2008 1st IFIP Wireless Days*, 2008, pp. 1–5. DOI: 10.1109/WD.2008.4812913.

[54] M. Alizadeh, M. Salleh, M. Zamani, J. Shayan, and S. Karamizadeh, "Security and Performance Evaluation of Lightweight Cryptographic Algorithms in RFID," Jul. 2012.

[55] S. D. Rihan, A. Khalid, and S. E. F. Osman, "A performance comparison of encryption algorithms AES and DES," *International Journal of Engineering Research & Technology (IJERT)*, vol. 4, no. 12, pp. 151–154, 2015.

[56] F. Maqsood, M. Ahmed, M. M. Ali, and M. A. Shah, "Cryptography: a comparative analysis for modern techniques," *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 6, pp. 442–448, 2017.

[57] D. S. Abd Elminaam, H. M. Abdual-Kader, and M. M. Hadhoud, "Evaluating the performance of symmetric encryption algorithms.," *IJ Network Security*, vol. 10, no. 3, pp. 216–222, 2010.

[58] M. Panda, "Performance analysis of encryption algorithms for security," in *2016 International Conference on Signal Processing, Communication, Power and Embedded System (SCOPES)*, 2016, pp. 278–284. DOI: 10.1109/SCOPES.2016.7955835.

[59] AUTOSAR. (2021). "AUTOSAR," [Online]. Available: https://www.autosar.org/ (visited on 04/23/2021).

[60] T. Jordan. (2018). "How hardware security modules enable autosar," [Online]. Available: https://www.embedded.com/how-hardware-security-modules-enable-autosar/ (visited on 12/08/2020).

[61] Infineon Technologies. (2020). "Telematics control unit," [Online]. Available: https://www.infineon.com/cms/en/applications/automotive/automotive-security/telematics-control-unit/ (visited on 04/20/2021).

[62] STMicroelectronics. (2020). "Telematics and connectivity control unit," [Online]. Available: https://www.st.com/en/applications/telematics-and-networking/telematics-and-connectivity-control-unit.html (visited on 10/01/2020).

[63] NXP Semiconductors. (2021). "S32 automotive platform," [Online]. Available: https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/s32-automotive-platform:S32 (visited on 04/21/2021).

[64] Bosch, *E/E Architecture of the future*, 2021. [Online]. Available: https://www.bosch-mobility-solutions.com/en/highlights/connected-mobility/e-e-architecture/ (visited on 04/21/2021).

[65] Infineon Technologies, *Gateway*, 2020. [Online]. Available: %7Bhttps://www.infineon.com/cms/en/applications/automotive/body-electronics-and-lighting/gateway/?gclid=CjwKCAjw_NX7BRA1EiwA2dpg0l2vy5xnpfU4lls8GSZAXJ_i-vxcJeqbIPblac4XxGD3jPYQZLPX5BoC-qkQAvD_BwE&gclsrc=aw.ds%7D (visited on 04/20/2021).

[66] Bosch, *Central Gateway*, 2020. [Online]. Available: https://www.bosch-mobility-solutions.com/en/products-and-services/passenger-cars-and-light-commercial-vehicles/connectivity-solutions/central-gateway/ (visited on 04/20/2021).

[67] Vector Informatik GmbH. (2021). "Automotive ethernet," [Online]. Available: https://www.vector.com/int/en/know-how/technologies/networks/automotive-ethernet// (visited on 04/09/2021).

[68] ——, (2020). “Learning Module LIN,” [Online]. Available: https://elearning.vector.com/mod/page/view.php?id=309 (visited on 04/09/2021).
[69] CSS Electronics. (2021). “LIN Bus Explained,” [Online]. Available: https://www.csselectronics.com/screen/page/lin-bus-protocol-intro-basics/language/en (visited on 04/09/2021).
[70] Vector Informatik GmbH. (2021). “MOST,” [Online]. Available: https://www.vector.com/int/en/know-how/technologies/networks/most/ (visited on 04/09/2021).
[71] ——, (2021). “Learning Module FlexRay,” [Online]. Available: https://elearning.vector.com/mod/page/view.php?id=371 (visited on 04/09/2021).
[72] CSS Electronics. (2021). “CAN Bus Explained,” [Online]. Available: https://www.csselectronics.com/screen/page/simple-intro-to-can-bus (visited on 04/09/2021).
[73] Vector Informatik GmbH. (2021). “Learning Module CAN,” [Online]. Available: https://elearning.vector.com/mod/page/view.php?id=333 (visited on 04/09/2021).
[74] CAN in Automation. (2021). “CAN knowledge,” [Online]. Available: https://www.can-cia.org/ru/can-knowledge/ (visited on 04/11/2021).
[75] Kvaser. (2021). “The CAN Protocol Tutorial,” [Online]. Available: https://www.kvaser.com/can-protocol-tutorial/ (visited on 04/11/2021).
[76] Wikipedia contributors. (2021). “Controller area network — wikipedia, the free encyclopedia,” [Online]. Available: https://sv.wikipedia.org/w/index.php?title=Controller_Area_Network&oldid=49792333 (visited on 04/11/2021).
[77] ——, (2021). “Can bus — Wikipedia, the free encyclopedia,” [Online]. Available: https://en.wikipedia.org/w/index.php?title=CAN_bus&oldid=1056068485 (visited on 04/11/2021).
[78] CSS Electronics. (2021). “CAN FD Explained,” [Online]. Available: https://www.csselectronics.com/screen/page/can-fd-flexible-data-rate-intro (visited on 04/09/2021).
[79] Kvaser. (). “CAN Bus Bit Timing Calculator,” [Online]. Available: https://www.kvaser.com/support/calculators/bit-timing-calculator/ (visited on 04/11/2021).
[80] CSS Electronics. (2021). “CAN FD Explained - A Simple Intro: Efficiency and Average Bit Rates,” [Online]. Available: https://docs.google.com/spreadsheets/d/16XIceuoG_YBlgyFKXxjYQk2016ln3NeEC68yoMFvwkA/edit#gid=0 (visited on 04/11/2021).
[81] CAN in Automation. (2021). “CAN FD - The Basic idea,” [Online]. Available: https://www.can-cia.org/ru/can-knowledge/can/can-fd/ (visited on 04/11/2021).
[82] Kvaser. (2006). “Kvaser USBcan Professional,” [Online]. Available: https://www.kvaser.com/product/kvaser-usbcan-professional-2/ (visited on 04/20/2021).
[83] ——, (2021). “Kvaser USBcan Pro 2xHs v2,” [Online]. Available: https://www.kvaser.com/product/kvaser-usbcan-pro-2xhs/ (visited on 04/20/2021).

[84] Vector Informatik GmbH. (2021). “VN1600 - Network Interfaces with USB for CAN (FD) and LIN,” [Online]. Available: https://www.vector.com/int/en/products/products-a-z/hardware/network-interfaces/vn16xx/ (visited on 04/20/2021).

A

Extensions

There are several different possible extensions to the main work that could be done if time is available.

Here are some possible extensions in no order:

- **Study & compare key distribution schemes in regards primary to performance:** If there are key distribution schemes or similar in literature, preferably where they also provide source code, then it is possible to do a evaluation on the testbed comparing each of them. This evaluation could consist further of safety and usability for vehicles (trucks in this case), e.g., how long does it take to start the truck if you first need to exchange new session keys etc.

- **Study & compare authentication methods in regards primary to performance:** If there are more authentication schemes or similar available in literature, preferably where they also provide source code, then it is possible to do a evaluation on the testbed comparing each of them. Since it would be quite interesting to see the effect on average bus load when adding different signatures schemes. Testing could just be on a downsized testbed with e.g., three HSMs connected through CAN and/or Ethernet, and simulate some of the traffic based on the specifications about the busload and apply just a few uses cases for comparing security aspects. (better to use TC3xx in this extension)

- **Using the HSMs to implement back-end communication in the vehicle with e.g., TLS or IPsec:** E.g. It is possible increase performance in applications that for example uses SSL/TLS with HSMs. Since the RSA operations can be offloaded to the HSM instead of the host CPU. Furthermore, a HSM can support Elliptic-curve cryptography (ECC) to mitigate the problem when performance at longer key sizes is becoming increasingly important. (better to use TC3xx in this extension)

- **Comparison of HSM medium (TC2xx) and HSM full (TC3xx):** Since AURIX TC3xx has access to hardware accelerators for asymmetric cryptography and hashing, while not TC2xx has it. It could also be interesting to compare that performance of using asymmetric cryptography in hardware (TC3xx)

versus using software (TC299), i.e. how effective the use of those hardware accelerators are.

- **Testing the possibilities of using HSMs for something that involves remote communication like V2V, V2V, or V2X:** It could be interesting to study and analyse the testbed connected to some external device/network.
- **Transitioning to a real vehicle:** One could transfer the work done on the testbed to an actual real functioning vehicle, although this feels quite unrealistic with regards to the time frame of this project.

B

Background

B.1 Nodes

Telematic Control Unit (TCU)

The telematic control unit (TCU), aka. Telematic Gateway (TGW), is an on-board embedded system, which controls all external communication for the vehicle. This includes functions such as wireless tracking, remote diagnostics, software updates. In principle, all communication between the vehicle and the Original Equipment Manufacturers (OEMs) servers. But it is also responsible for other forms of external communication such as Vehicle-to-everything (V2X), which can e.g., be over a cellular network. [61] [62]

V2X is a vehicular communication system that incorporates other more specific types such as Vehicle-to-Infrastructure (V2I), Vehicle-to-Vehicle (V2V), and more. There are some main motivations for V2x, such as increasing road safety, higher traffic efficiency, energy savings, etc. In V2X there are two types of underlying communication technology; WLAN-based, and celluar-based. V2X is part of the increasing external communication and thus increasing the security threats (attack surfaces) on vehicles.

In order to provide all these functionalities a TCU provides multiple different external communication interfaces. This can include for example GSM, GPRS, WiFi, WiMax, LTE, WLAN, and etc. The TCU usually also contains a GPS unit for tracking the vehicle, in terms of its latitude and longitude coordinates.

In summary, a TCU is a vital component responsible for all external communication of the vehicle. Furthermore, it also acts as a important security device (similar to a border router/firewall). The TCU shields the in-vehicle network from malicious external communication. Usually the TCU only allows secure communication with authenticated servers. However, the degree of security on TCU can vary in regards to e.g., firewalls, intrusion-detection system (IDS), intrusion-prevention system (IPS), and more.

Gateway Unit (GW) (or Domain Control Unit)

In a vehicle it is a good practice to utilize gateways internally. This is in order to create isolation between the different systems. Thus, it creates more layers of security. These gateways can also provide functionality and perform management of their domain. In this case they are usually referred to as domain control units, as they are dedicated units with the purpose of managing their respective domain networks.

The distributed vehicle architecture, together with the increasing software size and complexity, create challenges. For example, security, hindering development, incompatibility between hardware and software, and nodes not easily upgraded or scaled. In summary, there is a need for increased scalability, flexibility and reusability.

A different electrical/electronic (E/E) architecture is part of the solution to these problems [63] [64]. This architecture is a more model-based approach that disperses the ECU landscape in the vehicle, through for example domain controllers. Furthermore, it is worth to note that one trend is to have a fewer number of high performance ECUs that are responsible for a wider range of functionality, instead of having several low performance ECUs that are responsible for a narrow range of tasks. Thus, this architecture suggests different levels of ECUs/nodes. The alleged benefits are greater network capacity, higher performance through domain architecture, increased security, increased scalabliity, and etc. [63] Thus, one possibility is to have ECUs of the same risk category contained in the same domain.

Central Gateway Module (CGW)

The Central Gateway (CGW) is a important communication security node of a vehicle. As the name implies it functions as the central gateway in the in-vehicle network. In a networking perspective one can regard the CGW as a internal router and the TCU as the border router. [65] [66]

The CGW allow secure data transmissions between the different domains (i.e., the different ECU networks) and the connection to the TCU. Furtermore, the CGW provides naturally also physical isolation. But it can also provide strong security mechanisms, such as firewall, HMS, Intrusion Detection System (IDS) to protect the in-vehicle network. With the CGW one does not need to fully rely on the TCU for providing security for external communication. For example, if malicious communication bypasses the firewall of the TCU, the CGW will act as a secondary layer of security, with its firewall.

Vehicle Master Control Unit (VMCU)

The Vehicle Master Control Unit (VMCU) in principle functions as a CGW. The VMCU acts a central node for communication, uniting all the electrical components of the vehicle. Thus, in a way the VMCU can be thought of as the electronic brain of the vehicle. However, it does not provide any strong security mechanisms.

B.2 Communication Protocols

As described in section 2.3.2, there are many different (network) systems and subsystems in the vehicle network. The IVN is a complex heterogeneous network; there are typically multiple communication links/protocols such as Ethernet, CAN, LIN, and etc. All these protocols naturally comes with their different characteristics. The most common buses in vehicles are CAN and LIN, but there is also Media Oriented System Transport (MOST) and FlexRay. However, the latter two are relatively newer and therefore they are not currently that widely deployed in practice. These four buses differ from one another in terms such as architectures, access control, bandwidth, error protection, and etc. The Table B.1 summarizes some key aspects of the different bus protocols.

Thus, each communication protocol is deployed where they are the most optimal choice. Ethernet is e.g., mostly used more back-end in the IVN, i.e., gateway to gateway communication. The bus protocols LIN, CAN, and FlexRay are mainly used for control systems. Whereas MOST is used more for telemetric applications.

Still all these different systems need to be able to communicate both internally and externally. Internally means that communication is within the system itself, e.g., the powertrain system has a dedicated communication network (usually CAN, but FlexRay is also a possibility). Externally means communication to the outside world, in this case outside for each respective system. Continuing on the powertrain system, it has indirect communication to remote servers (the Orginal Equipment Manufacturers). Furthermore, external communication also includes communication between the different systems. In other words, in order to provide e.g., functionality (could also be due to lack of communication barriers), communication can occur between the different systems.

This can be natural communication, but it can also be malicious communication. The same applies with the case with the remote servers. Note that this is generally the case for all the different systems in the vehicle, and that the isolation principle is important as always.

Bus type	LIN	CAN	FlexRay	MOST (150)
Application	Low-level Communication Systems	Soft Real-Time Systems	Hard Real-Time Systems (X-by-wire)	Multimedia, Telemetrics
Control Architecture	Single-Master	Multi-Master	Multi-Master	Multi-Master
Bus Access Control	Polling	CSMA/CA CSMA/CR	TDMA FTDMA	TDM CSMA/CA
Bandwidth	ca. 20 KBit/s	1 MBit/s	10 MBit/s	150 MBit/s
Data Bytes per Frame	0 to 8	0 to 8	0 to 254	0 to 384
Physical Layer	Electrical (single wire)	Electrical (twisted pair)	Optical, Electrical	Mainly Optical, Electrical
Error Protection	Parity Bits Checksum	Parity Bits CRC	Bus Guardian CRC	CRC System Service

Table B.1: Summary of different vehicle bus systems (information gathered from numerous sources).

Ethernet

Ethernet has many use cases in the IVN, e.g., with its high bandwidth it is excellent for transmission of large amounts of data. Thus, it is a good solution for applications such as autonomous driving, driver assistance systems (ADAS), infotainment, and data backbones. Furthermore, it is used for things such as measurements, calibration, diagnostics, etc. For more details see the Vector's page on automotive Ethernet. [67].

Local Interconnect Network (LIN)

Local Interconnect Network (LIN) is a low-speed master-slave (aka. single-master) time triggered protocol. It is (vehicle) bus on single-wire (plus ground) with a relatively slow bandwidth (aka. data rate) of up to circa 20 KBit/s. A LIN cluster consist thus of one master and can have up to 16 slave nodes. The time triggered scheduling comes with guaranteed latency time. The data length in the LIN protocol goes up 8 bytes (it can be 2, 4, or 8 bytes). Lastly, LIN supports error detection and checksums. See Table B.1 for summarized details on LIN, and for even more details see the Vector E-learning module LIN [68] and CSS Electronics quick intro [69].

Currently one can regard the LIN bus protocol as a supplement to the CAN bus. The LIN bus offers a lower performance and costs, but also still has some reliability. So if one is looking for a low-cost option, where neither speed or fault tolerance are at a critical level, then LIN is the way to go. Thus, common applications are electrical windows, door-looking, wipers, air condition, different sensors, etc. In conclusion, a LIN bus is typically used as a low-cost alternative if the full functionality of the

CAN protocol is not required.

Media Oriented System Transport (MOST)

The Media Oriented System Transport (MOST) was developed back in 1998 under the leadership of BMW and DaimlerChrysler (since 2007 it is named Daimler AG). MOST is a serial high-speed bus that is targeted toward transmitting all possible automotive multimedia applications such as audio, video, telecommunication systems, etc. The MOST bus thus features a very high bandwidth through use of fiber optic cables. See Table B.1 for summarized details on MOST, and for even more details see Vector's page on MOST [70].

FlexRay

FlexRay is a fault-tolerant and high-speed communication protocol that is targeted toward higher safety-related applications. This protocol is multi-channel and can operate either in single or dual channel mode. Each channel has a maximum bandwidth of 10 MBit/s. So by fully utilizing the dual-channel mode a FlexRay network can achieve speeds significantly faster than the maximum standard CAN bus data rate. Ideal use cases for FlexRay are e.g., data backbones, distributed control systems (e.g., powertrain), safety-critical applications (x-by-wire), i.e., timing critical applications. See Table B.1 for summarized details on FlexRay, and for even more details see the Vector E-learning module FlexRay [71].

B.2.1 Controller Area Network (CAN)

CAN is a serial communication technology and is a robust vehicle bus standard (i.e., nodes connected through a bus), that is especially known to be deployed for reliable data exchange between ECUs in vehicles (in-vehicle network) and satisfies the real-time requirements of target usage areas in vehicles. The communication is done through messages, where each message can contain 64 bits of data and some error correction code etc. As normally with bus protocols, only one node/device at a time can send data and everyone receives it. Here is a short introduction to the CAN protocol (for more information see [72–75]).

The are several advantages of CAN, both from itself and the fact that it is a bus. To begin with, the recent history of vehicles is heavily characterized by an intensive electrification. This is due to a number of reasons, but the primary one is regarded as the long growth of customer wishes for more functionality or services in a modern vehicle. In the begin of the electrification of automobiles, independently operating ECUs were sufficient to operate electronic functions. However, it became soon clear that coordination of ECUs could improve vehicle functionality significant. The data exchange between the ECUs so far was implemented by convention, which was a physical communication channel. However, for every signal to be transmitted there needed to be a physical communication channel, so it was a scalability problem, and thus also a cost problem (in terms of wires, which has in itself cost, but also

their weight and volume). This lead to intensive wiring effort just in order to enable limited data exchanges in the network.

This situation is the motivation for the vehicle bus system, i.e., serial bit exchange of data through a single communication channel, the bus. Development of the CAN bus started as early as 1983 at Robert Bosch GmbH. However the protocol became officially released a few years later and the first actual vehicle incorporating the CAN bus was released in 1991, which was the Mercedes-Benz W140. Still today, the CAN bus is widely deployed in vehicles, e.g., to enable networking between the ECUs in the powertrain.

CAN Network

The structure of the CAN network is quite straightforward. The CAN network consists of a number of nodes which are all linked via a physical transmission medium, in this case the CAN bus. The CAN network is usually based on a line topology with a linear bus, to which a a number of ECUs are each connected through their respective CAN interface.

Physical Layer

On the physical layer of the CAN bus all the nodes are connected to each other through a physically conventional (usually a Unshielded Twisted Pair - UTP) two-wire bus, which is the physical transmission medium and where the symmetrical signal transmission occurs. The two wires together communicate one bit at a time; CAN-low and CAN-high. It is the voltage between the two wires that decides whether a logical 1 or 0 is on the bus. If the voltage difference is equal to zero then it is a logical 1, and if the voltage difference is over 2V then it is a logical 0. One can think it as one has connected a multimeter between the two wires, in order to measure the value on the bus.

Furthermore, the wires are connected with resistors, so if they are left floating, the potential difference between them will be evened out to 0V (i.e., logical 1 on the bus). So if one does nothing, there will be a logical 1 on the bus (since the voltage evens out by itself). However, if one does "pull-up" CAN-high and "pull-down" CAN-low, there will be a logical 0 on the bus. In summary, the bus has only two states, it is never floating.

Architecture

If a CAN node sends a logical 0 (as the way described earlier), and a other CAN node sends a logical 1 (doing nothing), then the final result on the bus will be a logical 0. Logical 0 is the so called dominant bit, i.e., if any CAN node sends 0, then the bus value will be 0. Meanwhile, logical 1 is the so called recessive bit, i.e., only if all the CAN node "sends" 1, then the bus value will be 1.

A common thought experiment on this a number of people are sitting so that they can neither see or hear each other. However, everybody can see one electric light

that is lit. Everybody also has a button, that when pressed will turn off the electric light, as long as the button is pressed. Thus, one is able to see through the electric light if at least one person (i.e., anybody) is pressing on their button (i.e., the light is off), or if nobody has pressed (i.e., the light is on). However, one cannot conclude who or how many has pressed. The corresponding case in CAN; when the electric light is lit is equivalent to that there is a logical 1 on the bus (recessive), and when the light is off, there is a logical 0 on the bus (dominant). In conclusion, this is a functioning bus communication protocol.

This is the basic protocol. When a CAN node wants to send out a message on the bus, they first "turn off the electric light" a certain time. After this the sender can send bits out on the bus by "turning the lamp on and off" during carefully balanced times. However, there is a common problem here, what if multiple CAN nodes want access to the bus? There is clearly a risk that multiple CAN nodes try send out bits at the same time. This leads to collision on the bus, which in turn will distort the bus value, i.e., result in incorrect bus values.

Arbitration (Bus Collision) - Priority

Thus, collision handling is necessary. Here the concept is that e.g., every node has a unique 4-bit ID-number, where the more important the node is, the lower the ID-number it has is. Each CAN node will start its message with its ID-number. If a CAN node notices that another node with lower ID-number is writing, it will stop sending, since it has lower priority. In conclusion, the CAN-node with the highest priority, i.e., lowest ID-number, will get its message written on the bus.

CAN Frame

Every message on the CAN bus is packaged in a frame, see Figure B.1 and Table B.2 for details on the different fields. In (standard) CAN there is a 11-bits ID (in the "Arbitration Field") that functions exactly the same as the earlier described concept for handling bus collision (priority system) with 4-bit ID-numbers. The node with lowest ID-number will get their message written to the bus, as they have higher priority. Thus, CAN has a sophisticated system for giving priority access to the bus, so that an urgent message can be sent before routine messages.

Furthermore, it is clearly important that two nodes never try to send with the same ID-number. It is a common practice that ID-numbers is a fusion of a message type (message-ID) and a unique ID of the node (sender/node-ID). For example, if the message type is sent first, it will first decide the priority, and the sender-ID will only decide on priority if several nodes are writing with the same message type.

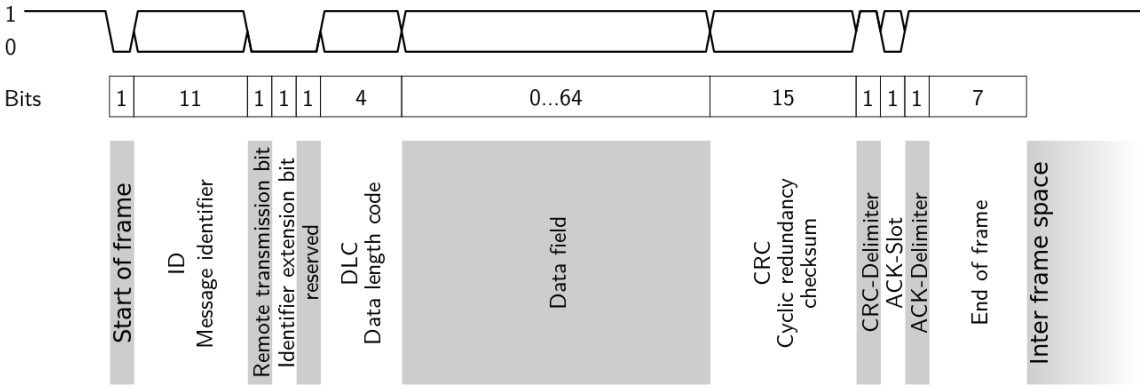

Figure B.1: The standard (data) CAN frame, with its fields, bit length, and possible bit states. (taken from [76])

Error Protection

A Cyclic Redundancy Check (CRC) is sent with each frame, so that the receivers can detect faulty bits. Each receiver is expected to write a logical 0 to the bus under the ACK-bit if the node has received a faultless message. The sender node can thus detect if nobody has successfully received its message. This handles cases such as when there is no other node on the bus (no receiver). In this case the sender node can resend the message until it is confirmed that somebody at least has received it. However, it will not detect that somebody has received and some have not received it. Back to the thought experiment, one could think of it as "press your buttons (turn off the light) if you have received the message". In this situation one can only understand if nobody has received the message.

CAN Extended

As shown earlier, in a standard CAN frame there is a 11-bit identifier field. However, there exists a so called extended CAN frame version, which has 29 identifier bits. Table B.3 represents a CAN extended frame. The lightgray cells in the second column highlights the differences compared to the standard CAN frame, which is shown in Table B.2. The most significant difference is adding a second identifier field with 18 bits. This allows a significantly larger ID space. However, the downside is that the frame becomes longer, which can affect the bus load.

Field Name	Sub-Field	Length (bits)	Purpose
Start Of Frame	SOF	1	Denotes the start of frame on CAN Bus (must be dominant, 0)
Arbitration Field	Identifier	11	A ("unique") identifier which represents the Message Priority (Arbitration) on CAN bus
Arbitration Field	RTR	1	Remote Transmission Request (RTR): Differentiate between Remote Frame (0) or Request Frame (1)
Control Field	IDE	1	Identifier Extension bit (IDE): Denotes the frame format: Standard (0) or Extended (1)
Control Field	R0	1	Reserved bit (must be dominant, 0)
Control Field	DLC	4	Data Length Code (DLC) denotes the data length on the CAN Bus (0-8 bytes)
Data Field	D0-D8	64	Data Payload
CRC Field	CRC	15	Cyclic Redundancy Check (CRC)
CRC Field	CRC Delimiter	1	Must be recessive (1)
ACK Field	ACK	1	Acknowledgement by receiving node. Transmitter sends recessive (1) and any receiver can respond with a dominant (0)
ACK Field	ACK Delimiter	1	Must be recessive (1)
End Of Frame	EOF	7	Indicates end of current frame (must be recessive, 1)

Table B.2: Table of a standard CAN Frame, with its main fields and sub-fields with bit length (here maximum frame size is 108 bits, without bit stuffing). Furthermore, a short summary of the purpose or denotation of the sub-fields. (inspired from Wikipedia [77])

Field Name	Sub-Field	Length (bits)	Purpose
Start Of Frame	SOF	1	Denotes the start of frame on CAN Bus (must be dominant, 0)
Arbitration Field	Identifier A	11	A ("unique") identifier which represents the Message Priority (Arbitration) on CAN bus
Arbitration Field	SRR	1	Always recessive, 1
Arbitration Field	IDE	1	Identifier Extension bit (IDE): Denotes the frame format: Standard (0) or Extended (1)
Arbitration Field	Identifier B	18	Decides the Message Priority (Arbitration) on CAN bus (Extended)
Arbitration Field	RTR	1	Remote Transmission Request (RTR): Differentiate between Remote Frame (0) or Request Frame (1)
Control Field	R0	1	Reserved bit (must be dominant, 0)
Control Field	R1	1	Reserved bit (must be dominant, 0)
Control Field	DLC	4	Data Length Code (DLC) denotes the data length on the CAN Bus (0-8 bytes)
Data Field	D0-D8	64	Data Payload
CRC Field	CRC	15	CRC
CRC Field	CRC Delimiter	1	Must be recessive, 1
ACK Field	ACK	1	Acknowledgement by receiving node.
ACK Field	ACK Delimiter	1	Must be recessive, 1
End Of Frame	EOF	7	Indicates end of current frame (must be recessive, 1)

Table B.3: Table over a Extended CAN frame (maximum frame size is 128, without bit stuffing). The arbitration field is extended to two identifier fields, A and B, which together form a 29-bit identifier. The lightgray color of different cells illustrates changes compared to standard CAN frame. (inspired from Wikipedia [77])

B.2.2 Controller Area Network Flexible Data-rate (CAN FD)

The CAN protocol has been around for a long time and has enjoyed wide popularity. However, with increasing challenges from modern technology the CAN protocol is pressured [78].

For example, with the increase in vehicle functionality comes a higher demand in data. But also an increasing demand on higher security in the IVN. In short, data-rate is becoming more important in the IVN. Meanwhile, the CAN network is limited to 1 Mbit/s in bandwidth, which will restrict implementations of significantly high data-producing applications. Furthermore, the CAN protocol is a much smaller bus compared to e.g., FlexRay, MOST and Ethernet. The CAN protocol has the problem of having a substantial overhead, since each CAN data frame can only have up to 8 bytes of data.

The problem with increasing the classical CAN frame, e.g., up to 64 bytes of data, is that with the slow data-rate it becomes clearly a problem that the bus might be blocked for too long. A longer message takes longer to transmit, and in CAN there is no delays or interruptions when a message is being transmitted. In other words, longer messages on a slow bus means that there is risk at delaying critical higher-priority messages. This is a situation that is important to avoid in vehicles, since there exists many different critical higher-priority messages, which can be vital for safety.

One would think that simply also increasing the bandwidth would solve the problem. However, it is not that quite simple. The problem is of the "arbitration" (bus collision handling) in CAN. When two or more nodes want to transmit data simultaneously, then arbitration will determine which node will take priority and gets to send their message, without delay. Meanwhile the other nodes will stand by during the data transmission. This is the bus collision handling, as described earlier in section 2.3.3.

During this arbitration there exists a "bit time", which provides a sufficient delay between each transmitted bit to allow every node on the network to react. This is necessary in helping to avoid any misunderstanding between the nodes during the arbitration. But in order to be certain that a bit reaches every node on the CAN network within the bit time, there exists a limit on the length of the CAN bus (i.e., the physical wire). Increasing the bandwidth during the arbitration would lead to a significantly shorter length of the CAN network. This is a problem since in many cases there is no marginal for the length. For example, trucks (which are naturally rather long) have long CAN buses (circa 10-20 meters). But with longer buses the bandwidth needs to slower. There exists online calculators (both for CAN and FD) for bit timing parameters, which are needed to ensure a reliable CAN network [79, 80].

However, after arbitration (bus collision handling) is finished, the successful higher-priority node will proceed with its data transmission. Now there is only one node "driving" the bus and here it is possible to increase the bandwidth. But before the

ACK, when multiple nodes acknowledge receiving the correct data frame, the bandwidth needs to be reduced back to the nominal data-rate. In conclusion, with the conditions it is only possible to increase the bandwidth during data transmissions.

One solution to this overall situation is CAN FD. The CAN FD protocol is relatively new and does not change the CAN physical layer. It was pre-developed by Bosch and was officially released in 2012. Then CAN FD went through standardization and is currently its own standard (ISO 11898-1).

As the names implies; the main difference between CAN and CAN FD is the Flexible Data-rate (FD). CAN FD supports dual bit rates. First it has the nominal data-rate (utilized during arbitration) limited up to 1 Mbit/s, as there is in the classical CAN protocol. And in second it has the flexible data-rate, which depends on the actual network topology and its nodes. Nominally it can go up to 10 Mbit/s, but a more realistic data-rate is lower. Still, it is a significant increase in bandwidth compared to classical CAN and can in many cases be a multiple of bandwidth gain.

CAN FD also supports up to 64 bytes of data per data frame, and as recalled classical CAN only supports 8 bytes of data. So, CAN FD enables reducing the protocol overhead and increasing efficiency.

Furthermore, CAN FD comes with an improved version of CRC and a so called protected stuff-bit counter. Thus, CAN FD comes with better reliability in terms of error protection compared to classical CAN.

Lastly, it is worth mentioning that it is possible to mix classical CAN ECUs with ECUs that use CAN FD, under certain conditions. Thus, a smooth transition to CAN FD is not that far-fetched. A gradual introduction of CAN FD nodes is fully feasible, instead of doing a full complete change of all the nodes into CAN FD.

In conclusion, the CAN FD protocol has an adjusted CAN data frame, which enables flexible larger data payloads and flexible higher data-rates, without doing any changes on the physical layer of CAN. Thus, ECUs with CAN FD can dynamically switch to different data-rates and with different messages sizes, both which can be significantly larger than classical CAN.

Furthermore, CAN FD retains the standard CAN bus arbitration, and increasing the bandwidth to a shorter bit time only when the arbitration process is finished. Then CAN FD will return the nominal (arbitration) bandwidth, i.e., the normal bit time, at the CRC delimiter, before the receiver nodes send their ACK bits.

For more details on CAN FD there are many papers and resources available online [78] [81] [73] (Chapter 6).

B.3 Hardware

CAN Dongle

A CAN dongle functions almost the same as a real basic CAN node; it can receive and send CAN traffic. In this thesis there are two CAN dongles available.

The first one is Kvaser USBCAN Professional. It provides two high speed CAN bus interfaces through a single USB connection. Thus, with the USB interfaces it can be used with any PC USB (2) port. The bus interfaces supports both CAN and the extended version. [82]

However, there exists a newer version called Kvaser USBcan Pro 2xHS v2. The main upgrade is a significantly higher bit-rate and that it supports CAN FD. [83]

The second one is Vector CANcase 1630A which is more advanced CAN dongle, compared to the Kvaser USBcan professional. For example, it has higher bit-rate and supports CAN FD. [84]

C

Results

C.1 HSM

Data in bytes	Total latency	Enc. latency	Dec. latency	Active latency
16	68.9	10.8	6.16	51.9
32	73.1	11.8	7.52	53.8
64	81.4	14.3	10.2	57.0
128	98.4	19.3	15.6	63.5
256	132	29.2	26.1	77.0
512	200	49.0	47.2	104
1024	335	89.0	89.3	157

Table C.1: ECB, μs

Data in bytes	Total latency	Enc. latency	Dec. latency	Active latency
16	70.4	11.3	7.25	51.9
32	74.2	12.3	8.43	53.5
64	82.8	14.9	11.1	56.7
128	99.9	20.0	16.5	63.4
256	134	30.3	27.0	77.0
512	202	50.7	48.2	103
1024	339	92.0	90.4	157

Table C.2: CBC, μs

Data in bytes	Total latency	Enc. latency	Dec. latency	Active latency
16	70.6	11.4	7.46	51.8
32	74.5	12.5	8.58	53.4
64	82.7	14.8	10.9	56.9
128	99.1	19.9	15.9	63.3
256	132	29.9	25.7	76.8
512	198	49.9	45.2	103
1024	331	90.1	84.3	157

Table C.3: Metrics for latency of the block cipher mode CTR (HSM). The unit for the metrics are in μs, with three decimals.

Data in bytes	Total latency	Enc. latency	Dec. latency	Active latency
16	77.9	15.4	10.7	51.8
32	82.2	16.5	12.2	53.5
64	90.7	19.3	14.7	56.7
128	109	25.2	19.8	63.6
256	144	37.0	30.1	77.0
512	214	60.3	50.5	103
1024	355	107	91.3	157

Table C.4: Metrics for latency of the block ciper mode GCM (HSM). The unit for the metrics are in μs, with three decimals.

Data in bytes	Enc. throughput	Dec. throughput
16	1480	2600
32	2700	4260
64	4480	6280
128	6630	8200
256	8770	9820
512	10400	10800
1024	11500	11500

Table C.5: ECB, kB/s

Data in bytes	Enc. throughput	Dec. throughput
16	1420	2200
32	2610	3800
64	4290	5750
128	6410	7750
256	8450	9480
512	10100	10600
1024	11100	11300

Table C.6: CBC, kB/s

Data in bytes	Enc. throughput	Dec. throughput
16	1410	2140
32	2570	3730
64	4320	5850
128	6430	8050
256	8560	9950
512	10300	11300
1024	11400	12100

Table C.7: Throughput of the block cipher mode CTR. Note that the unit is kB/s for throughput here.

Data in bytes	Enc. throughput	Dec. throughput
16	1040	1490
32	1940	2630
64	3310	4360
128	5070	6460
256	6930	8520
512	8490	10100
1024	9530	11200

Table C.8: Throughput of the block cipher mode GCM. Note that the unit is kB/s for throughput here.

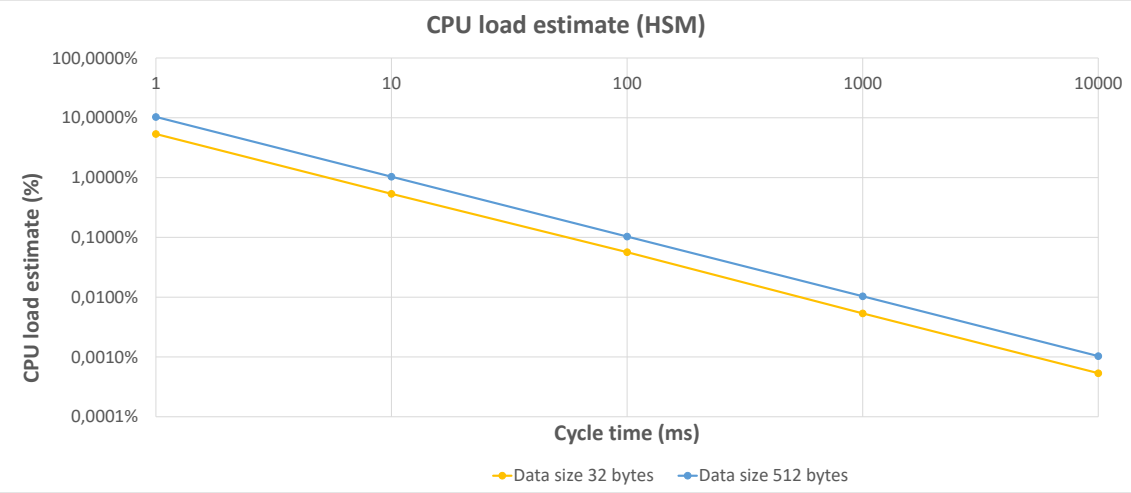

Figure C.1: CPU load vs message cycle time using HSM cryptography. The data size is 32 and 512 bytes.

TRNG - Throughput

In Table 5.1 one can also see that it is clear that throughput, based on TRNG latency, is roughly around 53 kB/s regardless of the data size. Thus, there are no performance gains for calling the TRNG to generate large amounts of random data all at once. See Figure C.2 for throughput based on *TRNG latency*. Furthermore, it is worth noting that the TRNG can work in parallel with the cryptographic accelerator.

Data in bytes	TRNG throughput
16	52.9
32	53.2
64	53.8
128	53.6
256	53.9
512	53.9
1024	53.8

Table C.9: Throughput of the TRNG, kB/s

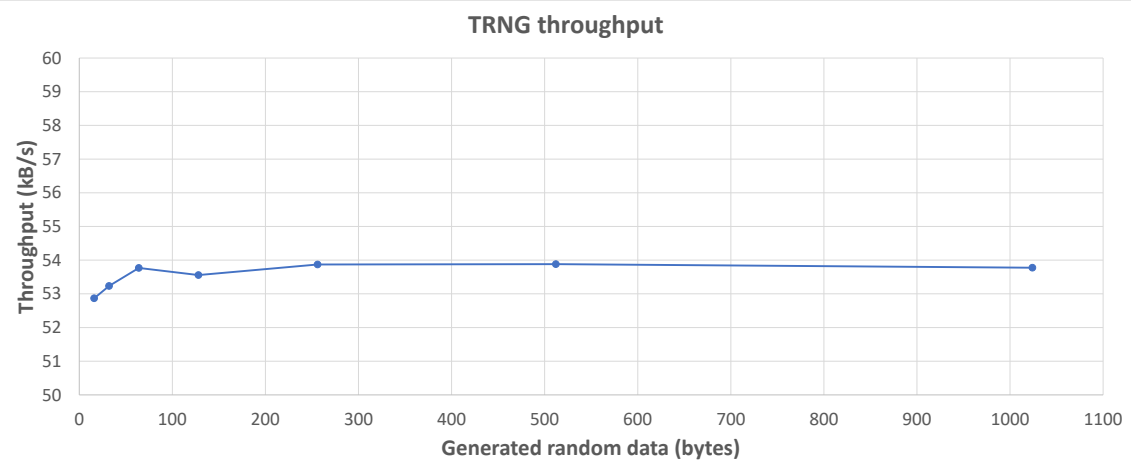

Figure C.2: Illustration of how TRNG throughput changes with the amount of random data generated. It is clear from the diagram that there is no real significant difference in throughput for different data sizes.

C.2 WolfCrypt

Data in bytes	Total latency	Enc. latency	Dec. latency	Setup latency
16	45.3	6.50	6.70	32.0
32	56.6	12.1	12.2	32.3
64	82.0	23.8	23.7	34.5
128	127	45.6	45.8	35.5
256	221	90.4	90.2	40.3
512	408	180	179	49.3
1024	783	359	357	67.3

Table C.10: ECB SW, μs

Data in bytes	Total latency	Enc. latency	Dec. latency	Setup latency
16	47.9	7.11	7.42	33.4
32	61.0	13.1	13.8	34.1
64	88.6	25.8	26.4	36.4
128	137	48.5	50.1	38.0
256	237	95.5	98.4	43.0
512	438	189	196	53.4
1024	840	376	390	73.8

Table C.11: CBC SW, μs

Data in bytes	Total latency	Enc. latency	Dec. latency	Setup latency
16	38.6	7.19	6.37	25.0
32	51.2	13.2	12.4	25.7
64	77.6	25.9	24.3	27.4
128	126	49.0	48.3	29.2
256	226	97.0	95.9	33.4
512	426	192	192	42.1
1024	827	384	383	60.5

Table C.12: Metrics for latency of the block ciper mode CTR (SW). The unit for the metrics are in μs, with three decimals

Data in bytes	Total latency	Enc. latency	Dec. latency	Setup latency
16	202	83.9	82.7	35.9
32	250	107	106	36.9
64	342	152	151	38.9
128	526	243	242	40.7
256	899	427	426	46.1
512	1640	793	792	55.8
1024	3130	1530	1530	76.4

Table C.13: Metrics for latency of the block ciper mode GCM (SW). The unit for the metrics are in μs, with three decimals

Setup latency is illustrated in Figure C.3. This is not that interesting for this project context. However, it might be worth for the bigger picture. Interestingly, there is differences between all the modes, but they all have the same development with data size. Clearly GCM has the highest *setup latency*, and CTR has the lowest.

Cycle time (ms)	CPU load (%)
1	5.34
10	0.534
100	0.0534
1000	0.00534
10000	0.000534

Table C.14: CTR, CPU load estimate (32-bytes).

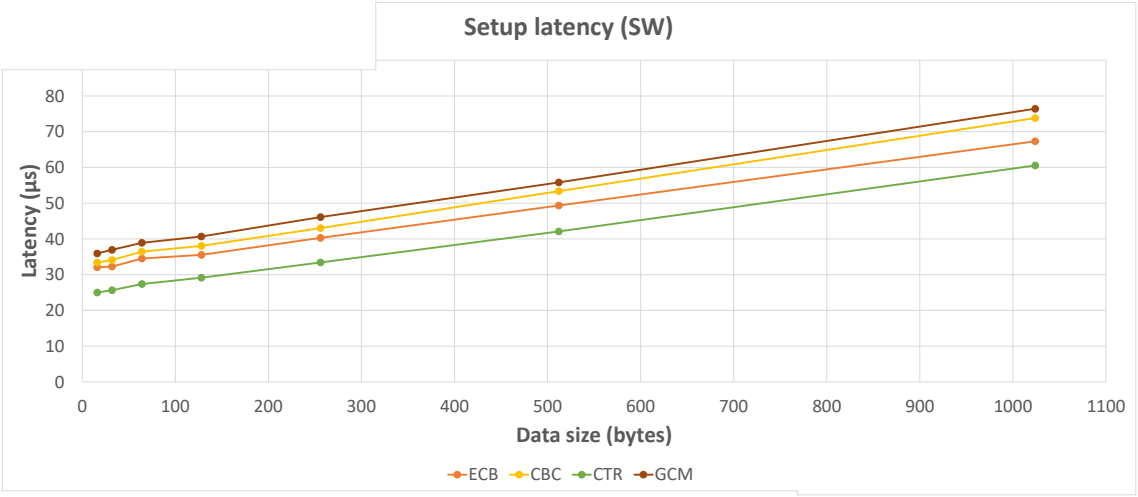

Figure C.3: Comparison of the different block cipher modes in regards to the metric *Setup latency* (SW). Clearly there are some differences in the setup latency between the modes.

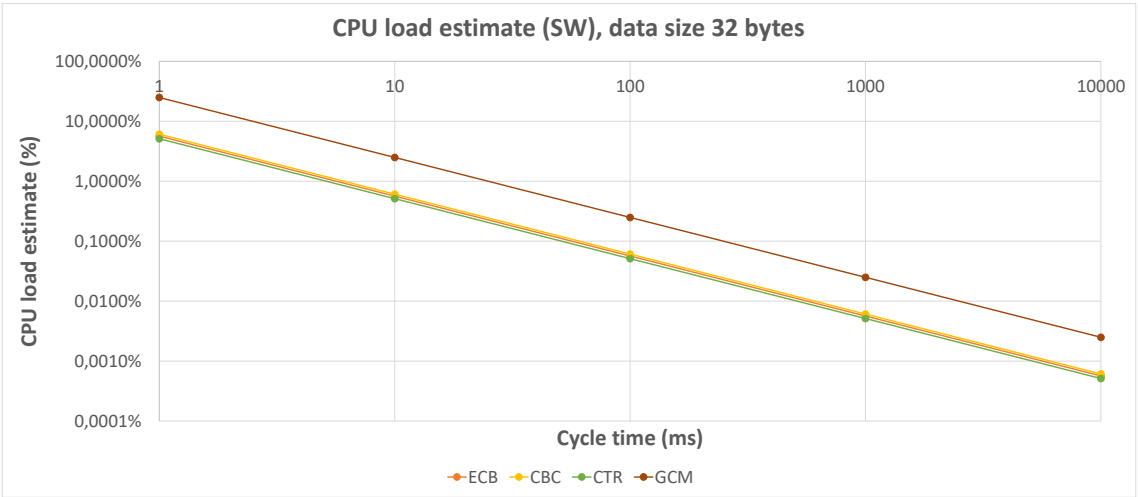

Figure C.4: CPU load vs message cycle time for different cipher modes using SW cryptography. The data size is 32 bytes.

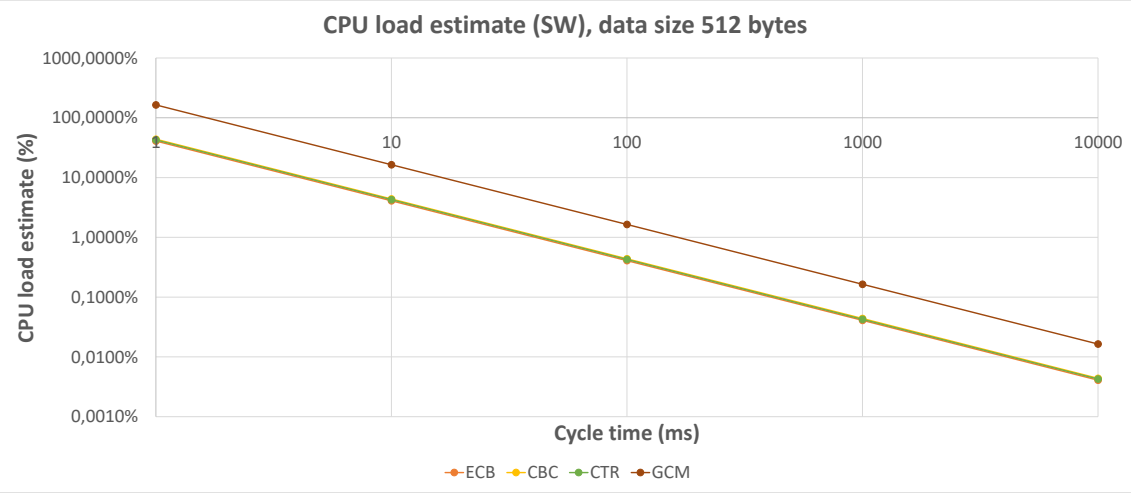

Figure C.5: CPU load vs message cycle time for different cipher modes using SW cryptography. The data size is 512 bytes.

Cycle time (ms)	Bus load 1 frame (%)	Bus load 4 frames (%)
1	34.2	137
10	3.42	13.7
100	0.342	1.37
1000	0.0342	0.137
10000	0.00342	0.0137
100000	0.000342	0.00137

Table C.15: Bus load estimate for a single arbitrary message type with the cycle time 10 *ms*. As seen here and through the formula, the bus load estimate is fully linear with the cycle time.

C.3 Difference in Cryptographic Latency

Data in bytes	HSM enc. latency	HSM dec. latency	SW enc. latency	SW dec. latency	Diff. enc. latency	Diff. Dec. latency
16	11.4	7.46	7.19	6.37	-4.16	-1.09
32	12.5	8.58	13.2	12.4	0.707	3.80
64	14.8	10.9	25.9	24.3	11.1	13.4
128	19.9	15.9	49.0	48.3	29.1	32.5
256	29.9	25.7	97.0	95.9	67.1	70.2
512	49.9	45.2	192	192	143	146
1024	90.1	84.3	384	383	294	298

Table C.16: Difference between HSM and SW in cryptographic latency, for the block cipher mode CTR. Note that the unit is μs.

Data in bytes	HSM enc. latency	HSM dec. latency	SW enc. latency	SW dec. latency	Diff. enc. latency	Diff. Dec. latency
16	15.4	10.7	83.9	82.7	68.5	71.9
32	16.5	12.2	107	106	90.4	93.5
64	19.3	14.7	152	151	133	136
128	25.2	19.8	243	242	218	222
256	37.0	30.1	427	426	390	396
512	60.3	50.5	793	792	733	741
1024	107	91.3	1530	1530	1420	1430

Table C.17: Difference between HSM and SW in cryptographic latency, for the block cipher mode GCM. Note that the unit is μs.