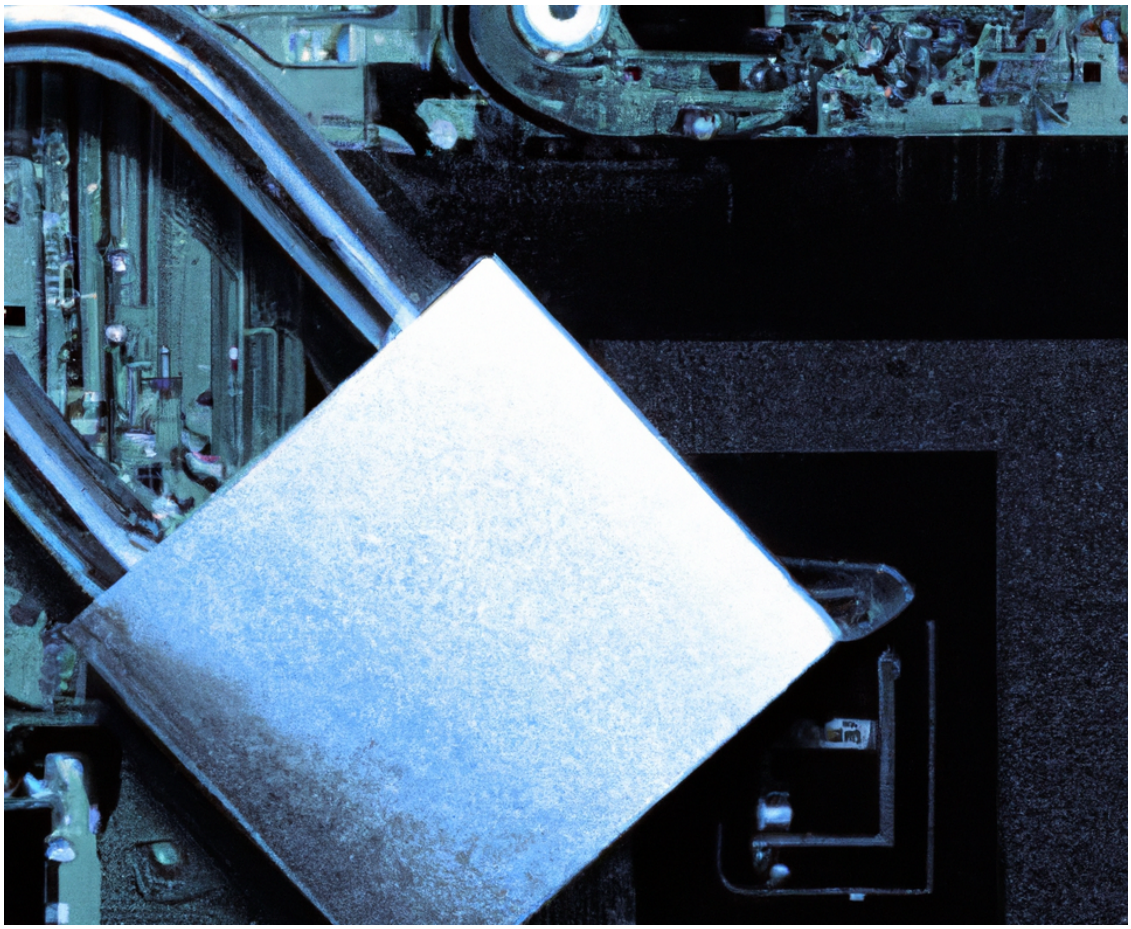


ACRSES

Advanced Cryptography on a Resource Scarce Embedded System

Degree Project Report in Mechatronics Engineering Mechatronics

Jonatan Svensson, Anders Karlsson

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023
www.chalmers.se

Degree project report 2023

Advanced Cryptography on a Resource Scarce Embedded System

Using a Raspberry Pi Pico as a model system for a scarce resource environment in order to study the effects of AES on the performance of the system.

Jonatan Svensson, Anders Karlsson

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computers Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Advanced Cryptography on a Resource Scarce Embedded System
Using a Raspberry Pi Pico as a model system for a scarce resource environment to study the effects of AES on the performance of the system.
Jonatan Svensson Anders Karlsson

Supervisor: Henrik Fagrell, Diadrom
Examiner: Lars Svensson, Department of Computer Science and Engineering

Degree project report 2023
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Cover: Picture generated with DALL-e with the prompt "Advanced cryptography applied on a resource scarce embedded system". Aims to abstractly depict cybersecurity on a circuit board or embedded system.

Typeset in LaTeX
Gothenburg, Sweden 2023

Execution time analysis of AES cryptography on a Raspberry Pi Pico
Using a Raspberry Pi Pico as a model system for a scarce resource environment to study the effects of AES on the performance of the system.

Jonatan Svensson
Anders Karlsson
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Abstract

This degree project aims to explore how a advanced cryptography can be applied on a resource scarce system. Specifically, the targeted system is a Raspberry Pi Pico and the advanced cryptography method is AES. Open-source AES implementations were used to achieve encryption, decryption and CMAC verification. Various tests cases were constructed to investigate the Raspberry Pi Pico's performance under these test cases. The performance of the system was analysed by extracting data about the execution time.
The data conclusively described a linear and predictable behaviour both when encrypting and decrypting 16 bytes thousands of times; when encrypting 4096 bytes; and when the encryption functions were interrupted by a load case. Encryption and decryption should theoretically take the same amount of time, however, the data showed decryption taking significantly longer; a problem that gets exacerbated with larger data sizes. Furthermore, the findings indicates that the performance of encryption and decryption functions remains unaffected by the available RAM.

Keywords: Cryptograhpy, Cybersecurity, Embedded Sytems, AES, Microcontroller, Raspberry Pi Pico

Acknowledgements

We would like to thank our families and friends for encouraging us to take on new challenges in life. However, a special thanks to our mothers and fathers who have dedicated years of their lives to raise us. Without them, needless to say, we would never be here. We would like mention our dear friend Anton Lundh for supporting us with discussions, knowledge and friendliness. Thanks to Wilda and Mina who has always supported us in our endeavours. Special thanks to Pedro Petersen Moura Trancoso who was encouraging and accomodating. Thanks to Henrik Fagrell for all the ideas and guidance.

Jonatan Svensson, Anders Karlsson Gothenburg, 06 2023

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

AES	Advanced Encryption Standard
CMAC	Cipher-based Message authentication code
DES	Data Encryption Standard
ECC	Elliptic Curve Cryptography
FPD	Floating Point Division
MAC	Message authentication code
RSA	Rivest–Shamir–Adleman

Contents

List of Figures

List of Tables

1
Introduction

1.1 Background

Cybersecurity is becoming increasingly important in the world and not least in the automotive industry. The industry demand for embedded systems is increasing. From a safety perspective, basic features requiring embedded systems, such as ABS brakes and airbags, are mandatory and widespread with more advanced safety features driving the demand up across the world. More demand for these types of systems comes from technological strides that solve more problems and implement new features with smart systems [1]. This development of products and features occurs faster than the cybersecurity of the systems that handle the computations. This has led to many embedded systems being vulnerable to attacks attempting to exploit this lack of cybersecurity for malicious purposes [2].

In the year 2024, a new cybersecurity ISO standard (SS-ISO/SAE 21434:2021) [3] will become mandatory for the Swedish automotive industry. The goal of the standard is threefold: define cybersecurity policies and processes; manage cybersecurity risks; and foster a cybersecurity culture in order to keep up with evolving attacks and technology. It is a comprehensive approach to solving the problem of embedded system security acknowledging the rapidly developing industry. With more integrated systems requiring security protocols, issues such as time-delays develop and act as an obstacle for the embedded system. The latter part, is generally a processing power issue that more advanced computers solve with their luxury of immense computing power. The effect of advanced cybersecurity running continuously is not as well documented on weaker embedded systems as it is on their more capable counterparts [4]. Their resource scarce environment attempts to control multiple processes at the same time. This has led to the thesis question:

How can advanced cryptography be used in a resource-scarce environment?

1.2 Aim

The project aims to collect performance data about an embedded system under the stress of an advanced encryption solution that would be sufficient to deter any reasonable attempt to break the encryption directly. Information collected from the system provides insight into the possibility of applying advanced cryptography on any embedded system.

1.3 Goals

The goal of the project is to have a complete software solution for cryptographic functions and testing, applied on an embedded system. Regarding cryptography, it should be able to use encryption and decryption such that the embedded system can perform: authentication of its source and to protect data. The testing goal is to be able to apply stress loads upon the embedded system and gather performance data.

1.4 Limitations

There are a multitude of ways to attack an embedded system and dealing with the different approaches requires different defenses. It is not within this project's scope to deal with the entire chain of the message, nor with the full range of attacks. This degree project will deal with the core of cryptography and only apply a cryptography solution to the embedded system to study its effect on the perfromance of the system.

1.5 Ethical considerations

Any work with cryptography usually means handling sensitive information and or ensuring that vital information is authentic. If the data is not handled correctly, it can have dramatic negative consequences. Therefore, as a consideration, the project will not handle any real-world data that can generate such consequences. Moreover, the project has the purpose of research and thus, any application into real-world settings should be taken with great precaution and further researched by experts in the field.

Ecological considerations are not handled due to their perceived irrelevancy to this project. The project is software focused and will not be applied to any areas within ecological interests. Industries providing the chips, materials and or other ecologically encroaching industries are not within the scope of this project and therefore, not considered.

Societal considerations, apart from the sensitive information consideration mentioned above, are not dealt with. This project is not expecting its result to encroach upon anything at a societal, communal or group level. If such a problem arises, it will be discussed in this degree project report.

2
Theory

2.1 Cryptography

This section will deal with the background theory of cryptography that is relevant for this report.

2.1.1 Symmetric and Asymmetric cryptography

In cryptography, a key is used to encrypt a message that person A wants to send to person B. The way this key is shared is the difference between asymmetric and symmetric encryption models.

Symmetric encryption relies on a shared key that is kept secure and secret [5]. This key is both used for encrypting a message and decrypting messages, meaning that if person B wants to send a message to person A, both persons needs to have the same key. Person B encrypts a message using the key and sends the encrypted message to person A who can then decrypt it using the same shared key. Thus, distributing the key is the major weak point of symmetric encryption. The encryption can be powerful enough, however, if a third party has intercepted the key, then the security is void. The focus for symmetric cryptography is both creating a encryption method and distributing the keys safely to the intended parties. Figure 2.1 illustrates how symmetric encryption works.

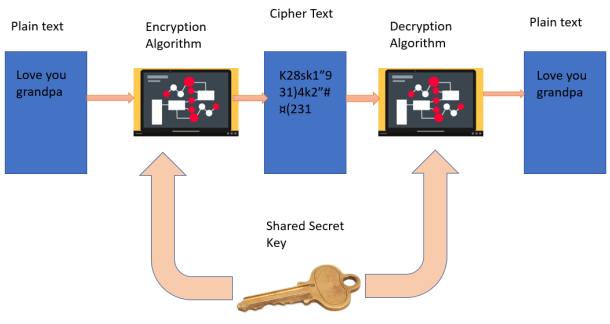


Figure 2.1: Simple illustration of how symmetric encryption works

Asymmetric encryption solves the distribution problem from symmetric encryption by creating a pair of keys. One key is private, held securely by the creator, and one key is public and published for everyone [5]. Person B can send a message using person A's public key to person A. Person A can then use the private key to decrypt the message. No matter who has access to the public key, the message cannot be decrypted without the private key. The pair of keys are created through a one-way function. This means that if you have the initial conditions of the function, deriving the keys is easy. However, with only the result of the one-way function, deriving the initial conditions are for practical purposes impossible and thus, the private key is secure. Figure 2.2 illustrates how asymmetric encryption works.

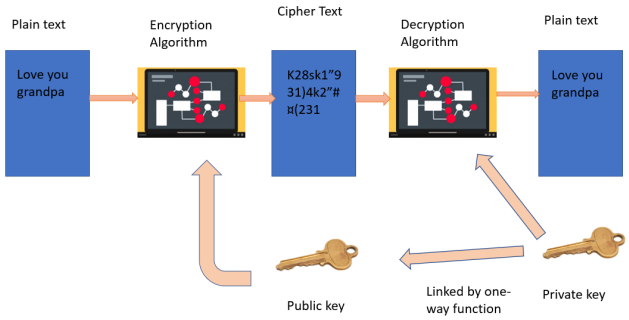


Figure 2.2: Simple illustration of how asymmetric encryption works

2.1.2 Advanced Encryption Standard

AES stands for Advanced Encryption Standard and is a symmetric block cipher [6]. Block cipher refers to the data being treated in blocks of predetermined sizes. For AES, the block size is 128 bits which are treated in several rounds of the algorithm. There are three versions of AES: AES-128, AES-192 and AES-256. The number is referring to the size of the key used. Other than key size, the number of rounds of the encryption algorithm applied varies between the three versions, with AES-256 doing the most and AES-128 doing the least. The encryption algorithm utilises bitwise xor, bit shifting, byte substitution and matrix multiplication to encrypt a given block. The byte substitution is performed with a substitution box, also known as an s-box. From the original key, new keys are generated that are used in the rounds. However, since AES is a symmetric encryption method both the encryptor and decryptor needs to have the same key. This could mean a security risk if the key is not transferred over secure channels.

2.1.3 Blowfish

Blowfish is a symmetric block cipher with a variable key length [7], from 32 bits to 448 bits and a block size of 64 bits. It consists of two parts, a key expansion part

and a data encryption part. The key is expanded up to 4168 bits. The encryption consists of 16 rounds and is treated in blocks of 32 bits each, All operations are xor or addition with the exception of a few byte substitutions.

2.1.4 RSA

RSA is an asymmetric encryption method. It utilises big prime numbers often over 100 digits per prime number to achieve a one way function [8]. Both the public and private keys are based on the result of the multiplication of the two prime numbers. Both keys consist of an exponent and a modulus. The modulus m is the result of the prime number multiplication, commonly around 1024 bits, and is the same in both keys. The public exponent is normally set to 65537 which is a prime number that is not too large. The private exponent is derived from the other parameters. A message is encrypted by raising it to the power of the receiver's public exponent and taking the modulus of the resulting number with the value of m. The private exponent is used for decryption. Since both keys are very big the resulting calculations will be massive in size. However, none of the parties needs the other ones private key to do the encryption. The security won't be compromised because no secret keys need to be exchanged.

2.1.5 Elliptic curve cryptography

Elliptic curve cryptography or ECC for short is an asymmetric encryption method [9]. It utilises the mathematics of elliptic curves to generate the key pair. One strength is that it uses smaller keys than RSA while still obtaining the same security level. However, ECC is difficult to securly implement which is its biggest drawback. Its main uses are for secure key exchange and authentication protocols and it is best used in conjunction with other methods.

2.1.6 Message authentication code

A message authentication code or MAC for short is a cryptographic technique used to verify the authenticity and integrity of a message [10]. It can also be use It is a short piece of information, often referred to as a tag. The MAC is generated with a secret key by the sender and is verified by the receiver by generating a MAC with the same key. If these are the same the sender is authenticated. MACs can also be used as a way of signing messages to verify the senders identity. There are several different methods to generate a MAC and the encryption method used decides which MAC version is suitable. As an example, CMAC is a method used to generate a MAC and is appropriate for block ciphers

2.2 Target system

The target system aims to mimic an embedded system. An embedded system is a computer system that uses processors, memory and input/output peripherals to achieve a or several different functions [11] within a larger system or by itself. It

combines hardware, software and smart control to provide an array of functions such as intelligence and automation. Commonly, microcontrollers are part of embedded systems to perform operations and send control signals. Common places where embedded systems are used are in cars, medical equipment, and industrial plants. A microcontroller is a small computer on a single integrated circuit chip [12]. It typically contains a processor core, memory, and programmable input/output peripherals. Microcontrollers are commonly used in embedded systems and are designed for specific tasks that require a high level of control and automation. Arduino Uno and Raspberry Pi Pico are current examples.

3
Implementations

The implementation of the project followed the high level requirements (HLR). These criteria are based on an initial literature study in order to direct the efforts and align the degree project with the employer. The HLR are as follows:

- HLR01: The system should have a microcontroller that is able to encrypt a message between 1-64-bit on its own using an advanced encryption method.
- HLR02: The system should use one of the encryption methods: Blowfish, AES or ECC.
- HLR03: The system should be able to generate a MAC in order to respond to a request for authentication.
- HLR04: The system should be able to verify the MAC sent by a similar system.
- HLR05: The system should have the ability to measure the performance of itself under loads.
- HLR06: The system should be able to introduce performance disturbances such as severely limited memory.
- HLR07: The system should be able to introduce performance disturbances such as calculation.
- HLR08: The system should be able to produce information about execution time.
- HLR09: The system should generate performance data to be the foundation for a security level-performance balance.
- HLR10: The system should handle unacceptable runtime lengths and response times gracefully.

Based on these HLRs revolved around three key points for the project which fell naturally in the following order:

1. Find a microcontroller that represents a part of an embedded system.
2. Use a cryptographic solution and apply it to the microcontroller.
3. Extract performance data and run various tests on the microcontroller.

3.1 Choice of microcontroller

An NXP S32K148 development board was considered as the targeted microcontroller since it was provided by the company. Although it is appropriate for an embedded system, it had several issues. For example, the start-up resources linked pages that had been removed and used software that had modern successors. These conditions resulted in connection issues and prevented uploading code to the microcontroller. Without progress in solving the problems, the decision was taken to discard the NXP board and search for a new microcontroller.

The search was largely based on microcontrollers with an appropriate clock frequency. By researching three major processor and embedded system solution providers and with reference to the previously considered NXP development board's processor, a few criteria were created.

1. The processor's clock frequency should be between 40-150MHz.
2. The processor should be 32-Bit.
3. The microcontroller should be able to serially communicate with a terminal.
4. The microcontroller should have support for start-up processes such as, code upload and serial communication.
5. The microcontroller should have programmable IO ports with support for hardware and software interrupts.

The Raspberry Pi family has a large community, and plenty of documentation and thus, was a natural place to explore. The Raspberry Pi Pico (Pico), seen in figure 3.1 fulfilled the criteria excellently and allowed for the swift implementation of ideas. Relevant specifications are: Dual-core Arm Cortex-M0 processor, 133 MHz, and 264 KB on-chip SRAM [13].

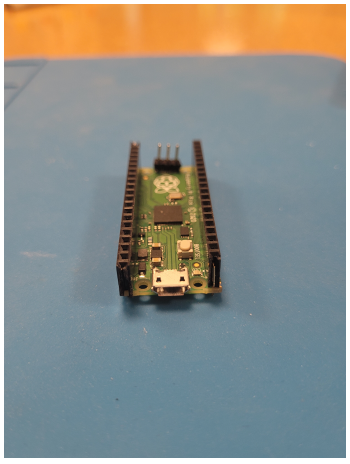


Figure 3.1: Raspberry Pi Pico

3.1.1 Working with the Raspberry Pi Pico

To work with the Pico using C/C++ code, it required setting up libraries, compilers and configurations. The guide by V. Hunter Adams [14] eases this process by providing steps to set up Visual Studio Code such that one can generate UF2 files from C code. UF2 is the file format that is needed to transfer the code to the Pico. The preferred programming language of the Pico is MicroPython [15]. However, C is supported as well within the Pico setup manual. The guide by V. Hunter Adams assists with this in a symbiotic way. The set-up with Visual Studio Code provided a suitable environment to run C code with the addition of Pico functions provided by a library.

3.2 Cryptographic solution

With the literature study as the basis, four candidates for the cryptographic solution were found to be relevant. All four candidates provide ample protection for a microcontroller in an embedded system. The candidates were investigated and evaluated jointly.

3.2.1 Exploring the asymmetric cryptography candidates

The candidates for asymmetric encryption were RSA and ECC (their specifications are described in the theory chapter). One side-effect of RSA encryption is that it handles extraordinarily large numbers [8]. With a quick first implementation, it was clear that this broke the compilation or in the best cases handicapped the computation. Consequently, RSA was abandoned as a possibility.

ECC has an important role in security. Its strengths lay in authentication protocols, key sharing methods and as a tool for generating a secure s-box (used in AES and Blowfish). It fulfils the HLR03-04 by generating authentication however, it does not provide a solution to encrypting entire messages on its own [9]. However, it could be used in conjunction with a symmetric encryption method as a way to securely transfer the key between two microcontrollers. This is however out of scope for this project. In addition, finding an applicable solution proved difficult and consequently, the decision was made to reject the ECC.

3.2.2 Deciding which symmetric cryptography solution

The candidates for symmetric cryptography were AES and Blowfish. From a computer science perspective, certain mathematical operations are more resource efficient for processors. In broad terms, bit-wise functions are relatively simple for processors to compute. For example, XOR and left/right shifting are straightforward computations for a processor due to their underlying architecture. Other operations such as division are slower because of the architecture. AES and Blowfish are block ciphers and these use resource efficient operations. This point argues for the suitability of their application on a microcontroller that has limited resources.

AES is the adopted encryption standard by the United State National Institute of Standards and Technology (NIST) [6] and is considered unbreakable from brute-force attacks . It is widely adopted in the world to protect data and has seen widespread use. One of the reasons for this is that it is an open standard which means anyone is free to use it for non-commercial and commercial use. Several open-source codes make the cryptographic solution easily accessible and implementable.
With AES fulfilling the HLRs, it was elected as the cryptographic solution which the thesis project would deploy on the Raspberry Pi Pico.

3.3 Implementation of AES encryption on the Raspberry Pi Pico.

For starters, a ready-made implementation of the AES encryption needed to be found, preferably implemented with at least the two key sizes 128 and 256. A lightweight implementation of AES was found [16] that fulfilled the requirements for the cryptographic solution and as per HLR02. As a bonus, it was verified with test vectors from NIST and was chosen as the AES code. However, this implementation did not have MAC generation which is required by HLR03. Therefore, a second implementation that had both an AES and CMAC solution was searched for and discovered [17]. The AES of this second implementation was not verified and therefore, it was decided to take the CMAC and apply with the first AES implementation. This combines the strength of the first implementation's AES with the capability of the second implementation's CMAC generation.

To join the first implementation's AES and the second implementation's CMAC, some changes in the code were required since they were working with two different libraries. The main changes were in the form of changing function calls and variable type declarations. Therefore an understanding of how each implementation worked was needed such that functions and variables could be used in a proper manner.

This conjoining of implementations was firstly done in a normal C environment for debugging purposes and then transferred over to the adapted C environment for the Pico. No issues were found in that transition. With this completed, HLR01, HLR02, HLR03 and HLR04 were achieved.

The resulting library has both the key itself and the plain text implemented as an array. To change the key size only one value needs to be changed in the code. The encryption and decryption function takes pointers to the key and plain text to be used as arguments. If a larger text than the block size is to be treated, the pointer to the text needs to be moved 16 positions after each function call. This way it effectively steps through the array since the cryptographic functions take blocks of 16 bytes.

3.4 Performance data: Execution time

The aim of the project is reflected in HLR08 which requires performance information about the system. There are several different performance indicators for systems; however, some require special tools that are bound to the respective systems they are developed for. In our case, the Raspberry Pi Pico has access to a way to calculate execution time which is a parameter that is of interest and fulfills HLR08. Other tools for performance data were not found and thus, not considered.

To measure the execution time of select parts of the code the builtin function time_us_64() of the Raspberry Pi Pico was used. It returns a 64-bit timestamp in microseconds which can be used to calculate the execution time by calling the timestamp function at the start of the part to be measured, saving the return value, and doing the same at the end. The difference between these values will be the execution time. With this functionality, HLR05 is fulfilled.

3.5 Compiler issues

With the encryption up and running and a method to obtain its execution time was implemented, some basic testing was conducted. The first rounds of encryption were promising in terms of performance. However, the obtained values were unrealistically fast. Since both the key and the text were hard-coded during these first tests, the compiler did all the calculations during compilation. This was circumvented by declaring said variables as volatile. For good measure, a code snippet that allows a user input of plain text to be used was implemented. With this two-fold safeguard against the optimization of the compiler, more reasonable test values were obtained.

3.6 Testing data and load cases

To simulate an embedded system, a way to give the microcontroller arbitrary tasks had to be established. Two possibilities were considered, a simple approach with interrupts or a more advanced approach with a real-time operating system. The former was chosen since it can fulfil HLR06 and HLR07 while still being fairly simple to implement. Since the Pico supports both software and hardware interrupts, periodic and sporadic tasks with high priority can be simulated.

3.6.1 Load case: Floating point division

A function that performs floating point division (FPD) was created to simulate a resource demanding task. The function is Quite simple in its implementation with the only input argument being how many iterations of floating point division are to be done. This allows for a flexible way to stress the system to various degrees.

3.6.2 Load case: Memory claim

Another interesting aspect to test was the system's performance when most of its RAM has been claimed. Once again this was a function with quite a simple implementation. The input argument decides how big of an array of the type integer is going to be allocated. Some quick tests found that 15821 integers were the maximum amount of memory that could be claimed without the Pico crashing. Therefore, to test the limit 15820 integers were claimed and then normal encryption/decryption tests of 16 bytes were run. A sample of 100 data points was extracted as described below.

3.6.3 Test arrays and data extraction

With all of the necessary components implemented, a proper test structure was needed to make all the tests run as smoothly as possible. Four buttons were set up as hardware interrupts. Three of them change test parameters and the last runs the test. This made it possible to run several different tests without the need to re-flash the Pico. However, to change the key size re-flashing was necessary. The test data was saved in two arrays, one for how many iterations, or which iteration, and one for execution time. The results were reported via USB to a computer in a structured list, thus, fulfilling HLR08 and HLR09. To test the functionalities of the constructed system, a couple of test cases were designed. These are described in the sections below.

3.6.4 Test case: Encryption and decryption of 16 bytes

This test case was designed to gain data about encryption and decryption execution time. Moreover, the test case will be used to investigate whether if there is a difference in the execution time between individually called encryptions/decryptions and when they are called en-masse. For this test, a plain text of 16 bytes was used, encrypted or decrypted a single time or several times with both the implemented key sizes. The number of iterations for the mass calls ranged from 0 to 3100.

3.6.5 Test case: Encrypting of 4096 bytes

This case will test the capability of encrypting larger amounts of data. A plain text of 4096 bytes is encrypted or decrypted with both the key sizes. 4096 bytes were chosen since prior to 2016, 4095 bytes were the biggest payload that could be sent according to ISO 15765-2 [18]. The standard's payload has since increased however, 4095 is justified by the company requesting it and the aim of applying this implementation to older embedded systems. Systems that would be operating according to the older standard. As a side note, the reason for 4096 bytes is the AES's block requirement of being a multiple of 16. Since AES needs messages in multiples of 16 bytes one extra byte was added.

3.6.6 Test case: Encryption with interrupts

This case was designed to test how the execution time of encryptions or decryptions is affected when being interrupted. A plaintext of 16 bytes was encrypted or decrypted 100 times with both key sizes. However, during this test, a periodic interruption every 200µs was triggered which ran 60 operations of floating point divisions. 200 µs were chosen since early testing showed that a single encryption of 16 bytes has an execution time of around 100 µs. This ensures that at least every other encryption gets interrupted. 60 FPDs has an execution time of around 100 µs. A smaller number of divisions would not impact the execution time enough to be properly detectable. A larger number of division would just make the testing take a longer time.

3.6.7 Test case: Encryption with severely limited memory.

This case was designed to test how the execution time of encryptions and decryptions is affected when most of the microcontroller's memory has been claimed. The same parameters as the previous test were used. However, this time the load case of memory claim as described in its section was applied.

3.6.8 Test case: Message authentication

This case was designed to test the execution time of the implemented MAC, more specifically CMAC. The case will only test the verification of the CMAC since the only difference between generation and verification is a simple comparison operation. The test is run with a sample size of 100.

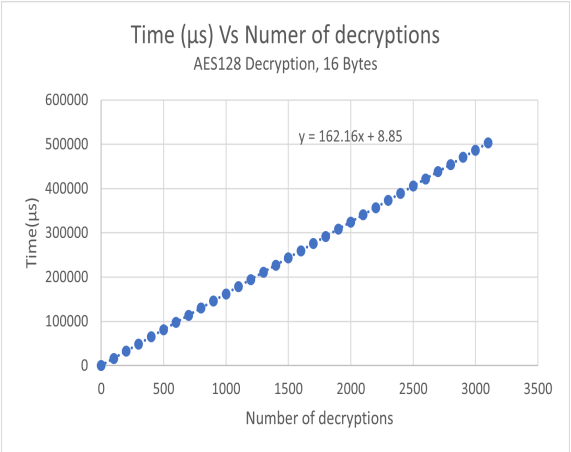
4

Results

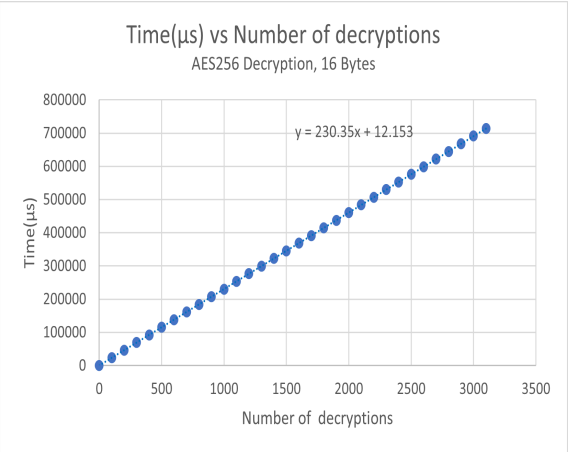
This chapter will handle all of the results yielded by the test run on the project. It will present the data and discuss it. Full source of raw data is in appendix 1.

4.1 Mass-call test of encryption and decryption of 16 bytes

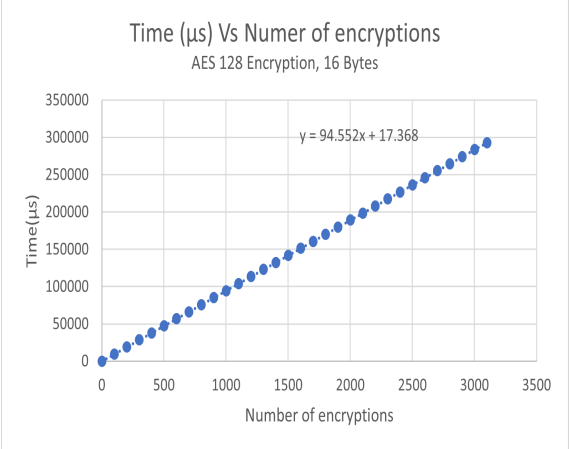
Figure 4.1 shows four causal relationships that are completely linear. There is no decrease or increase in performance when running 10 or 3500 encryptions. Each encryption takes the same amount of time. In addition to this, each try provided the exact same time data or within a couple of microseconds.



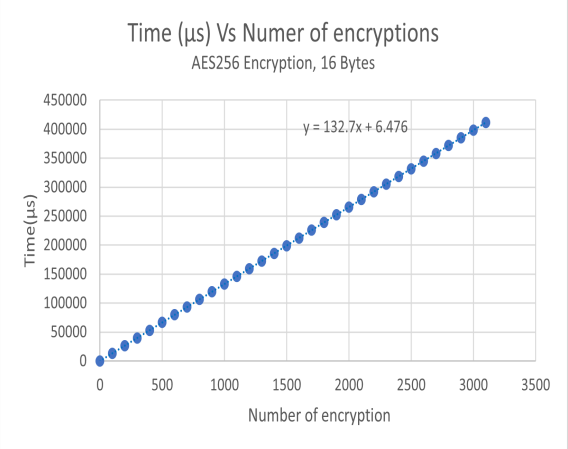
(a) Key-size 128, decryption of 16 bytes



(b) Key-size 256, decryption of 16 bytes



(c) Key-size 128, encryption of 16 bytes



(d) Key-size 256, encryption of 16 bytes

Figure 4.1: Four graphs of the results from a mass-call test that encrypts or decrypts 16 Bytes 3500 times with varying key-size 128 and 256.

At '0' encryptions and decryptions, the average time was between 0.4 µs and 0.7 µs. This is the expected result as not running any process should only make the clock start and then turn off immediately. The trend lines y-intercept, however, is ranging between 6-17 µs when it should be '0'. One hypothesis is that it is time taken for the processor calling functions. This is further supported by table 4.1, where the data shows that when calling encryption/decryption functions individually the average time is increased across the board. Moreover, it is consistent within decryption,

Table 4.1: Table over average times when calling the encryptions/decryptions individually or by mass-call.

	Average time: 100 mass-call	Average time(µs): 100 individually called
AES128 decryption	163.26	175.04
AES256 decryption	231.47	243.27
AES128 encryption	94.94	98.05
AES256 encryption	133.12	136.66

which has 12 µs increase, and encryption, which has a 3 µs increase.

Decryption is about 70 µs slower than its encryption counterpart. Theoretically, decryption and encryption should have the same execution time since decryption is doing the same functions in reverse. This increase in time is traced back to a difference in the implementation of the code. For the full code see appendix 1. The encryption function uses the function "MixColumns()" and decryption calls "InvMixColumns()". Their implementations are different and results in "InvMixColumns()" taking 70 µs longer compared to "MixColumns()". This is possibly an optimization flaw in this AES implementation. Nevertheless, it is not a security flaw as the decryption function stills works as intended. A deeper analysis is beyond this project's scope however, it nevertheless is important to take into consideration. This difference is even further exacerbated when encrypting larger messages as the following section's data will illustrate.

4.2 Encrypting 4096 Bytes

Figure 4.2 shows the execution times of encrypting or decrypting a plain text with a size of 4096 bytes, with both AES-256 and AES-128. The time was calculated as an average of 10 test results which all were within 30 µs of each other.

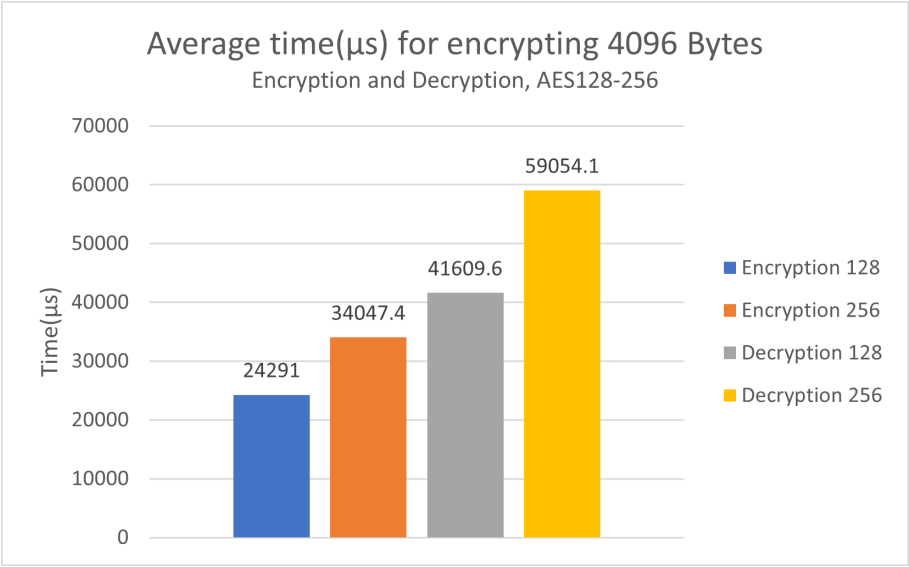


Figure 4.2: Bar graph that displays the average times for encrypting/decrypting 4096 bytes with AES128-256.

By studying the results, one can see that the execution time is almost exactly 256 times longer than the execution time for the corresponding encryption or decryption of 16 bytes. The results are calculable using the trend lines gradients, for example AES256 encryption: $133.12 \cdot 256 = 34078.72$ µs.

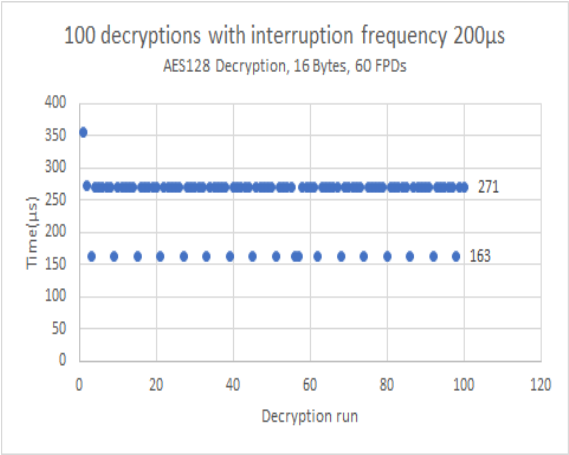
Furthermore, it is interesting to note that if using the average time for individually called functions, as shown in table 4.1, the calculated value would be off. It is illustrated with the calculation example as above, AES256 encryption: $136.66 \cdot 256 = 34984.96$ µs. The first example is exceeds the value in figure 4.2 by 31 µs and the second with 937.56 µs. Thus, when encrypting larger data one can consider it as a mass-call of encryption rather than several individually called encryptions.

From figure 4.2, the execution time should be further highlighted. For encryption key-size 128, it takes 24.29 ms; for encryption key-size 256 it takes 34.05 ms; for decryption key-size 128 it takes 41.61 ms; for decryption key-size 256 it takes 59.05 ms. The relevant magnitude is no longer in µs and thus, have higher implications on applications.

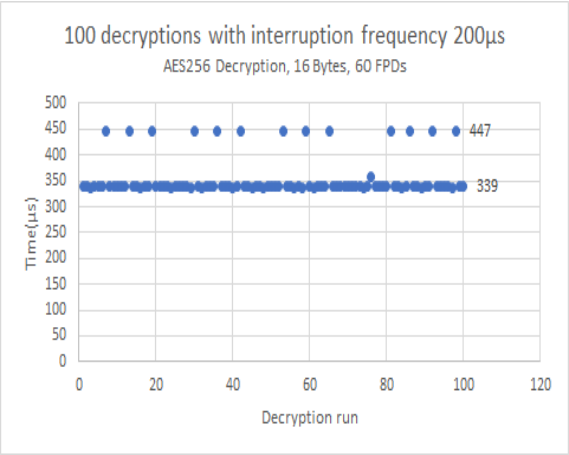
4.3 Loads and interruptions

This section will present and discuss the results from various tests under loads.

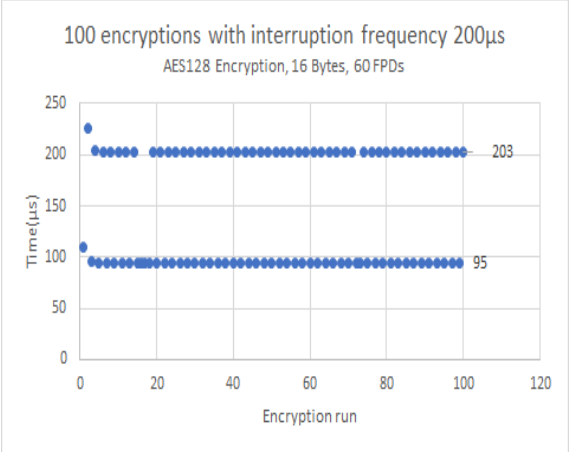
4.3.1 Load case: Periodic floating point interruptions



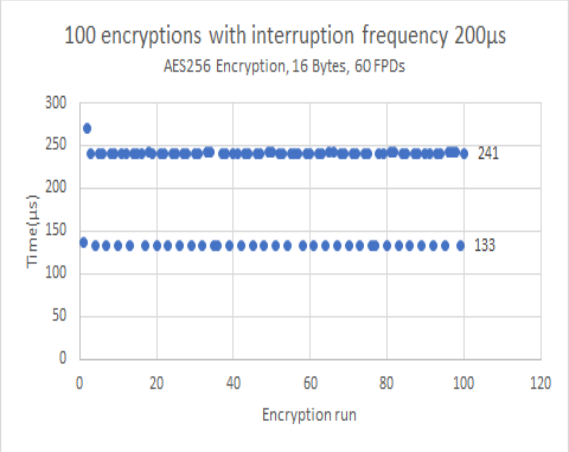
(a) Key-size 128, decryption of 16-Bytes with interruptions every 200 µs



(b) Key-size 256, decryption of 16-Bytes with interruptions every 200 µs



(c) Key-size 128, encryption of 16-Bytes with interruptions every 200 µs



(d) Key-size 256, encryption of 16-Bytes with interruptions every 200 µs

Figure 4.3: Four graphs of the results from a mass-call test that encrypts or decrypts 16 Bytes 100 times and gets interrupted every 200 µs

Figure 4.3 is another collection showing a mass-call of 100 encryption and decryption functions being interrupted every 200 µs by 60 FPD calculations.

All graphs are aligned with the data in Figure 4.1 and table 4.1; meaning the time for encryption/decryption is the same when taking into account the delay caused by the FPD interruptions. This supports the robustness of the data. Moreover, the graphs shows a linear correlation again. The increase in execution time is 108 µs for

all graphs in Figure 4.3 with the exception of graph b. Graph b is decryption of 16 bytes with AES256, which in figure 4.1 and table 4.1 shows, takes more than 200 µs. Therefore, the decryption gets interrupted twice which doubles the delay. Nevertheless, the delay is highly predictable and the encryption/decryption functions, despite being interrupted, does not show a decrease in their own performance.

Moreover, the delay of FPDs are predictable as one can see in the following figure:

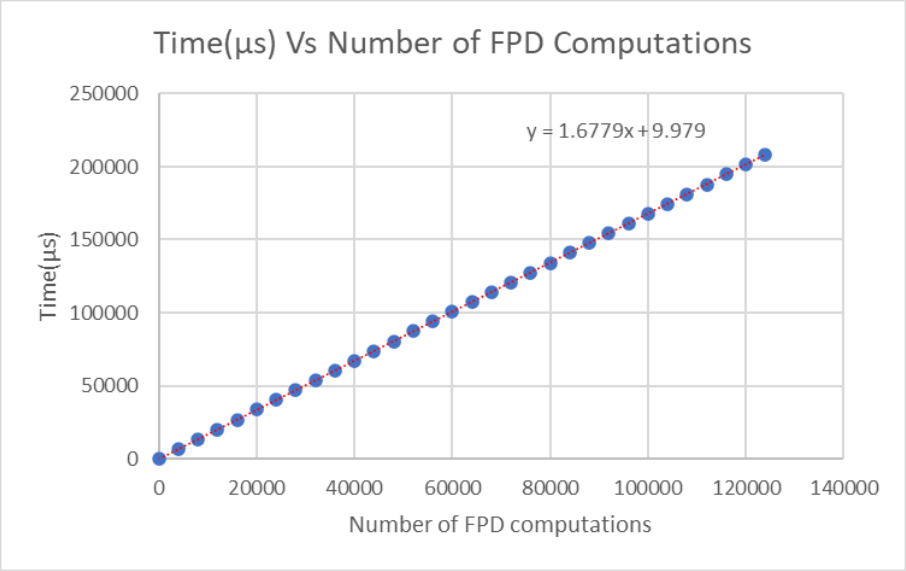


Figure 4.4: Graph showing time against FPD computations.

Figure 4.4 describes a predictable and linear correlation between the number of FPD computations and time. The trend line gradient provides the time per computation which is about 1.67 µs. One can connect this to the interruptions delay of 60 FPDs and find it calculable: $60 \cdot 1.6679 \approx 100.7$ µs. The execution time for calling the FPD function is the y-intercept as argued in previous chapters. Adding this to the total time gives the result: $100.074 + 9.979 \approx 110.05$ µs. Comparing it to the actual delay 108 µs, the calculable result is supported.

4.3.2 Load case: Memory claim

Table 4.2: Table of average time from the memory claim load case present adjacent to part of table 4.1

	Average time(µs) of 100 mass-called encryption/decryption	
	Under memory claim load	Under no load
AES128 decryption	163.35	163.26
AES256 decryption	231.54	231.47
AES128 encryption	95.23	94.94
AES256 encryption	133.63	133.12

This second load case attempts to mimic the case of an abritrary process that claims chunks of memory and the effect on the encryption/decryption functions. In this process, the test dynamically allocates 15820 integers which is one integer away from causing a system failure. The results in column 1 of Table 4.2 describes the effect on the encryption/decryption functions after this memory claim had been done. Column 2 is taken from Table 4.1 and acts as comparison to the case without a memory claim load.

Table 4.2 illustrates that the amount of dynamically allocated memory does not affect the Pico's performance in terms of the encryption/decryption functions' execution time. The influence of the load case is neglible with the largest difference being 0.5 µs for AES256 encryption. The AES operates seemingly without the need of allocating new data.

The Pico system expectedly fails once there is no more memory to allocate. This is not considered an effect that affects the execution time. It is a limitation of the memory of the microcontroller.

4.4 CMAC

Table 4.3: Table of the average time of a CMAC verification.

	CMAC, Average time(µs)	Standard Deviation(µs)	Standard deviation in %
AES128	280.3	17.060	6%
AES256	378.74	15.827	4%

Table 4.3 illustrates the average time taken for a CMAC verification test as described in the implementation chapter. The sample was 100 and the average was taken. However, in this data set, compared to previous data sets, the data samples did vary. On AES128 and AES256, there was a standard deviation of 17.06 µs and 15.82 µs respectively. Although not an immense deviation, it is a 6% and 4% discrepancy on average between values.

Despite this, the CMAC verification is quite consistent and the discrepancies are not necessarily alarming. The CMAC could still be considered as a constant delay and the deviation could be accounted for by always considering the worst case execution time. This would be a practical way to still handle the CMAC as an invariable delay.

4.5 Evaluation and conclusion of the results

There are two main conclusions from this data. Firstly, the data conclusively shows that the applied AES encryption consistently behaves linearly, both when being forced to encrypt the same message size thousands of times and when encrypting large data sizes such as 4096 Bytes. Moreover, the CMAC results were relatively predictable with only slight discrepancies. Because of this, one can extrapolate highly predictable and consistent results when running the various cryptographic functions. Secondly, under the two load cases of periodic FPD interruptions and memory claims causing memory scarcity, the results are still predictable. The FPD process acted as a delay and did not affect the performance of the cryptographic functions. The memory claim had no effect at all and the cryptographic functions operated normally in extreme memory scarcity. Overall the AES implementation is robust and reliable, however, there are still several aspects that should be investigated.

4.5.1 Implementation on other microcontrollers

For further research, it would be important to investigate the same implementation on different kinds of microcontrollers. Primarily, this would investigate if the execution times scale linearly with the clock frequency of the processor and determine if other factors affect the performance of the AES implementation. In addition, the impact of a different processor architecture than that of ARM would be of interest as well. The results of this investigation would be a more holistic view that could provide general advice and recommendations.

4.5.2 Issues with decryption taking longer than encryption

As stated previously, theoretically, decryption and encryption in AES should take the same amount of time. However, in this AES implementation, this was not the case. The cause was traced to the function "InvMixColumns()" which on average was 70 µs slower. This has dramatic effects as it accounts for about 43% and 30% of the execution time for AES128 and AES256 decryption of 16 bytes. It is further exasperated for even larger data sizes. The benefits of optimizing that function, if possible, are immense.

4.5.3 More load cases

The thesis project attempted to create two possible load cases that could occur in an embedded system. However, they are severely limited in their type and structure. Therefore, a further research case would be to expand the tests. The expansion could be into live systems or simply more complex tests. Nevertheless, investigating the effects of more tests would again provide a more holistic understanding of this AES implementation.

4.5.4 Memory claim load case possible effects

The results of the memory claim are promising showing that the AES implementation can operate without dynamically allocating data. However, it should not be overlooked as unimportant. If a message being received cannot be stored then it cannot be used. This should be taken into consideration when applying this or any other cryptographic solution. It is not an issue with the AES implementation itself however, it is the thesis argument that further research would be beneficial. The result could be guidelines for how an effective operating space should be constructed for a cryptographic solution.

5

General conclusion

The research question of the thesis project is: How can advanced cryptography be used in a resource scarce environment? The thesis project concludes that for encrypting messages and authentication, AES does excellently in a resource scarce environment. It is reliable and predictable which are both wanted qualities. Several avenues of research are still required for a better understanding of the AES implementation as discussed above. Nevertheless, for companies attempting to follow the new ISO standard (S-ISO/SAE 21434:2021) AES is a solid cryptographic candidate. Moreover, it seems entirely possible based on the result that advanced cryptography in the form of AES can be applied effectively in embedded systems resources scarce environments. With the option to choose between three key-sizes and message length, in terms of the number of blocks, a good balance between security and performance can be achieved.

Bibliography

[1] M. Sonia, "Embedded Systems in Automobiles Market by Product (Passenger Vehicle, Two-Wheelers, Commercial Vehicle), by Type (Hardware, Software), by Components (Sensors, MCU, Transceivers, Memory Devices) and by Application (Infotainment Telematics, Body Electronics, Powertrain Chassis Control, Safety Security): Global Opportunity Analysis and Industry Forecast, 2023-2032", alliedmarketresearch.com, Accessed: May. 25, 2023. [Online.], Available: https://www.alliedmarketresearch.com/embedded-systems-in-automobiles-market-A12177

[2] B. Boldt, 'AUTOMOTIVE SECURITY in a CAN: With car safety issues extrapolating due to the rapid increase in electronics, the automotive security market has been forced to immediately transition from effectively no security to robust security implementations.', *Electronic Design*, vol. 65, no. 12, p. 37, 2017.

[3] *Road vehicles — Cybersecurity engineering* ISO/SAE 21434:2021 June 2021 [Online]. Available: https://www.sis.se/en/produkter/road-vehicles-engineering/road-vehicle-systems/car-informatics-on-board-computer-systems/ss-isosae-214342021/

[4] M. Thompson, 'UDS Security Access for Constrained ECUs', in WCX SAE World Congress Experience, SAE International, Mar. 2022. doi: https://doi.org/10.4271/2022-01-0132.

[5] "What is asymmetric encryption?" cloudflare.com, Accessed: May. 25, 2023. [Online.], Available: https://www.cloudflare.com/learning/ssl/what-is-asymmetric-encryption/

[6] M. Dworkin et al., 'Advanced Encryption Standard (AES)'. Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, Nov. 26, 2001. doi: https://doi.org/10.6028/NIST.FIPS.197.

[7] B. Schneier, 'Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)', in Fast Software Encryption, Cambridge Security Workshop, Berlin, Heidelberg: Springer-Verlag, 1993, pp. 191–204. doi: https://doi.org/10.1007/3-540-58108-1_24.

[8] R. L. Rivest, A. Shamir, and L. Adleman, 'A Method for Obtaining Digital Signatures and Public-Key Cryptosystems', *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978, doi: 10.1145/359340.359342.

[9] V. S. Miller, 'Use of Elliptic Curves in Cryptography', in *Advances in Cryptology — CRYPTO '85 Proceedings*, H. C. Williams, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 417–426.

[10] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. Bosa Roca, UNITED STATES: CRC Press LLC, 2014. [Online]. Available: http://ebookcentral.proquest.com/lib/chalmers/detail.action?docID=1619504

[11] K. Iniewski, *Embedded Systems: Hardware, Design and Implementation*. Somerset, UNITED STATES: John Wiley & Sons, Incorporated, 2012. [Online]. Available: http://ebookcentral.proquest.com/lib/chalmers/detail.action?docID=1051822

[12] B. Lutkevich "microcontroller (MCU)" *techtarget.com* [Online.], Available: https://www.techtarget.com/iotagenda/definition/microcontroller Accessed: May. 25, 2023.

[13] Raspberry Pi Ltd, *Raspberry Pi Pico Datasheet*, July 2022, Available: https://datasheets.raspberrypi.com/pico/pico-datasheet.pdf Accessed: May. 25, 2023.

[14] Hunter, Adams, V. "Setting up the Raspberry Pi Pico for C/C++ Development on Windows" *vanhunteradams.com*, https://vanhunteradams.com/Pico/Setup/PicoSetup.html#V.-Hunter-Adams Accessed: May. 25, 2023,

[15] L. Pounder, "Raspberry Pi Pico Review: 'Pi Silicon' Debuts on $4 Microcontroller", *tomshardware.com*, [Online.], Available: https://www.tomshardware.com/reviews/raspberry-pi-pico-review Accessed: May. 25, 2023.

[16] kokke (2019) tiny-AES-c [Source Code], https://github.com/kokke/tiny-AES-c Accessed: May. 25, 2023.

[17] R. Savrulin, (2015) AES-CMAC-RFC [Source Code] https://github.com/flexibity-team/AES-CMAC-RFC/blob/master/aes-cbc-cmac.c Accessed: May. 25, 2023.

[18] *Road vehicles - Diagnostic communication over Controller Area Network (DoCAN) - Part 2: Transport protocol and network layer services* ISO 15765-2:2016, Apr 2016 [Online]. Available: https://www.sis.se/produkter/fordonsteknik/fordonssystem/fordonsinformatik-datasystem-i-fordon/ssiso1576522016/

A

Appendix 1

For all data and source code see: https://github.com/VargEnberg/ACRSES—Advanced-Cryptography-on-Resource-Scarce-Embedded-Systems

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY