Master's Thesis

Post-quantum Secure Communication on a Low Performance IoT Platform

Rickard Johansson
Thomas Strahl

Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, 2016.

Post-quantum Secure Communication on a Low Performance IoT Platform

Rickard Johansson, Thomas Strahl
dat11rjo@student.lu.se, dat11tst@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Paul Stankovski

June 21, 2016

Printed in Sweden
E-huset, Lund, 2016

Abstract

A recent trend in the IT world is the term Internet of Things (IoT). As more and more devices get connected to the internet, and with companies trying to lower production costs in order to stay competitive, security can easily be neglected.

In this report traditional Transport Layer Security (TLS) implementations and post-quantum based TLS were evaluated and performance measurements were conducted. The initial attempt was to run post-quantum secure algorithms on an IoT device in order to see if an IoT device theoretically would be able to withstand an attack from a quantum computer. Due to memory constraints it was not possible to run the modified version of the cryptography library, PolarSSL, on the intended IoT device. For that reason we switched to another platform, namely a Raspberry Pi. The measurements were conducted on that platform and in-depth analysis was performed to determine if current implementations of post-quantum algorithms can be suitable for IoT devices or not.

The conclusion was that post-quantum algorithms are more time-consuming than traditional algorithms used today. One of the experiments in this report shows that using a post-quantum algorithm for the key exchange is 2.5 times slower and needs 10 times as much RAM memory than a traditional solution with the same security level. There is therefore no apparent need to start using post-quantum algorithms today in terms of security level, when considering the duration and RAM usage for the key exchange. With time and optimization some of the algorithms evaluated in this report, or similar algorithms, could be good candidates the day large quantum computers are produced.

Acknowledgments

We would like to thank our supervisors Paul Stankovski, Einar Vading and Marcus Johansson. Paul who helped us to manage this project and report, Einar who guided us and made sure that our evaluation board worked with external RAM and Marcus with his great knowledge about microcontrollers from his professional career and hobby projects. Their help and continuous feedback has been very valuable.

We would also like to give many thanks to Axis for letting us do this project and for giving us access to the office, different hardware platforms and the very valuable breakfast and table tennis breaks.

Preface

This thesis was carried out as a part of a Master in Computer Science and Engineering at Lund University, Faculty of Engineering. The thesis was done in collaboration with Axis Communications, situated in Lund.

The implementation and the measurements have mostly been carried out together by the thesis workers. Pair programming has often been applied, but individual implementation and debugging has also been carried out. Rickard had the main responsibility for the pre-study of the post-quantum algorithms and the client application, while Thomas was responsible for setting up the development environment and the server application. The thesis workers have also focused on different areas of the report.

All pictures presented in the report have been created by the authors or are public domain.

Acronyms

- SSL - Secure Sockets Layer
- TLS - Transport Layer Security
- TCP - Transmission Control Protocol
- IP - Internet Protocol
- IoT - Internet of Things
- DH - Diffie-Hellman
- DHE - Diffie-Hellman Ephemeral
- ECDH - Elliptic Curve Diffie-Hellman
- ECDSA - Elliptic Curve Digital Signing Algorithm
- GCM - Galois/Counter Mode
- CA - Certificate Authority
- MAC - Message Authentication Code
- AES - Advanced Encryption Standard
- LWE - Learning With Errors
- RLWE - Ring Learning With Errors
- SIDH - Supersingular Isogeny Diffie-Hellman
- NTRU - N-th degree truncated polynomial ring
- MDPC - Moderate Density Parity Check
- LDPC - Low Density Parity Check
- TTS - Tame Transformation Signatures
- SHA - Secure Hash Algorithm
- LwIP - Lightweight Internet Protocol
- RTOS - Real Time Operating System

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Axis communications is a company situated in Lund with their main focus on manufacturing network cameras. They are however also developing other innovative products which make use of network connectivity in order to communicate. Axis has several self-developed hardware platforms which their current products are running on. Most of these platforms have relatively high performance and are often developed to process high resolution video. They are therefore not optimal for products with low performance requirements or products not requiring video encoding.

Axis is interested in using a new hardware platform for future products to explore new areas. Two important constraints for this platform are price and power consumption, but it also needs to be able to communicate securely.

1.1 Motivation

Finding secure solutions for small hardware platforms is a very up-to-date matter since the growth of Internet of Things (IoT). As more devices get connected to the Internet, the importance of security grows. Today's society relies on these connected devices and they are continuously increasing in number. To stay up to date in a fast growing field, where large quantum computers may soon be accessible, our focus will be on post-quantum cryptography.

An existing platform based on an *ARM cortex m4 microcontroller* will be used to investigate performance differences between today's Transport Layer Security (TLS) solutions and post-quantum TLS solutions. The reader should be observant of the fact that TLS is the successor to Secure Sockets Layer (SSL) but is still sometimes referred to as SSL. This is the fact with for example the PolarSSL which actually supports TLS.

The post-quantum algorithm used in in this project is based on the problem Ring Learning With Errors (RLWE) [4] and the TLS-library is a modified version of PolarSSL [5][6].

1.2 Project Aims and Main Challenges

The aim with this Master's thesis work is to investigate how the performance of the key exchange during a TLS handshake is affected by using post-quantum

cryptography in a low performance environment. The measurements from the post-quantum solutions will be compared to a standard TLS implementation and in-depth analysis will be performed regarding suitability for IoT devices today and in the future.

One of the main challenges is to modify an existing implementation to fit the hardware constraints, e.g. memory and CPU speed, on the microcontroller. The aim is to end up with a platform that is secure enough to withstand an attack from a quantum computer, but lightweight enough to run on a microcontroller.

Goals:

- Get standard TLS, e.g. PolarSSL, running on the microcontroller
- Get post-quantum based TLS running on the microcontroller
- Measure performance for the key exchange on the standard implementation and post-quantum implementation
- Evaluate the measurements
- Choose a suitable candidate in terms of performance versus security

1.3 Background

This section provides a short background of the different areas for this Master's thesis project. It will begin with describing the term Internet of Things and then discuss the different post-quantum algorithms which were evaluated. The cryptographic libraries WolfSSL and PolarSSL were also evaluated before implementation.

1.3.1 Internet of Things

One interpretation of the term Internet Of Things (IoT) is small embedded systems integrated in everyday objects [7]. These devices often consist of sensors and other components which can be monitored and controlled over networks and the Internet. IoT devices are often used to create so called "smart" systems, for example *smart homes* where the electrical devices in the home are interconnected and can be monitored and controlled remotely.

The term IoT is very broad. Some IoT-devices have extremely low performance and very harsh performance constraints because they run on batteries for example. As a consequence some can not even implement a full TCP/IP stack. Another example of IoT devices can be devices that have not been connected to the internet earlier, that offer higher performance e.g. a network camera.

IoT devices with lower performance often have constraints in terms of memory usage and power consumption to make the cost and size as small as possible. Due to these constraints some of the security mechanisms used in regular computers are not possible to implement on these devices. At the same time, security on IoT devices is very important since a successful attack on an IoT device can let an intruder get full access to a complete system.

1.3.2 Post-quantum algorithms

Post-quantum algorithms are thought to be secure against attacks using quantum computers. This is not the case for today's public-key algorithms which probably will be broken, more about this will be described under Section 2.3. Research in this area is important to stay one step ahead of tomorrow's quantum computers. This is probably the reason why there already are many different post-quantum algorithms available, although there are no publicly known large-scale quantum computers yet.

This section will shortly describe some of the post-quantum algorithms that were considered early on in this project for key exchange in a TLS handshake. The algorithms were evaluated to find an algorithm which was both lightweight enough to run on an IoT device while also proven and well-documented.

Ring Learning With Errors

Ring Learning With Errors (RLWE) is based on the problem Learning With Errors (LWE), but designed for polynomial rings. The greatest advantage of RLWE over LWE is that the key size is much smaller for the same level of security. Having a small key size can be very important for keeping the implementation efficient with low overhead. Parameters suggested to provide 128 bits of security result in public keys of size 6595 bits and private key of 14000 bits [8]. This can seem large compared with today's key size, but compared to LWE they are very small. The security for both learning with error problems lie in the fact that it is as hard to solve as the lattice problem *Shortest Vector Problem*, which is NP-hard [9]. RLWE can be used for key exchange, signing and encryption. RLWE key exchange will be explained further in Section 2.4.2.

N-th degree truncated polynomial ring

This algorithm is often abbreviated NTRU and is another contender due to the fact that the public key size is approximately the same size as for RLWE, but with a smaller private key, see Table 1.1. NTRU is also lattice-based like RLWE. At the time of writing there are two main implementations of this algorithm. One is patented and the other is open source. The open source version is licensed under GPL, meaning that any software using it must also be made public. NTRU has been added to WolfSSL and according to some benchmarks [10] NTRU performs better compared to for example RSA.

Goppa-based McEliece

Goppa-based McEliece is built on the problem of error correcting codes, and in this version the type of codes used are algebraic geometric codes, often referred to as *Goppa codes*. The key size for this version of McEliece is very large even compared to other post-quantum algorithms. For 128 bits of security, a public key of 8373911 bits and a private key of 92027 bits are required [11]. The complexity of the algorithm is lower than for some of the schemes used today, but the large key size might be impractical, especially for small devices with constrained resources.

Algorithm	Public key size	Private key size
Ring Learning With Errors	6595	14000
N-th degree truncated polynomial ring	6130	6743
Goppa-based McEliece	8,373,911	92,027
MDPC-based McEliece	65542	4384
Supersingular isogeny DH	6144	6144

Table 1.1: A summary of the key sizes for the algorithms [1].

MDPC-based McEliece

This version of McEliece is also based on error correcting codes, but in this case Moderate Density Parity Check (MDPC) codes. MDPC codes are higher-density Low Density Parity Check (LDPC) codes which are used in telecommunication [12]. The greatest advantage with this version of McEliece is that the key size is much smaller compared to the Goppa-based version, but still retains its security properties. The size of the keys are 65542 bits and 4384 bits for the public and private keys respectively. The relatively small private key size is good for storage space, which also is limited on a small device, but the rather large public key size will limit network throughput.

Supersingular Isogeny Diffie-Hellman key exchange

Supersingular Isogeny Diffie-Hellman key exchange (SIDH) is an algorithm based on the difficulty of calculating so called isogenies between elliptic curves [13]. As the name implies it is used for key exchange similar to the popular *Diffie-Hellman key exchange*, which will be described in Section 2.2.1, more specifically elliptic curve Diffie-Hellman. SIDH is similar to elliptic curve Diffie-Hellman in terms of computations and does provide *forward secrecy*, i.e. if a long term key is compromised previous sessions are still private. This is an advantage compared to other post-quantum algorithms such as McEliece and NTRU which do not provide forward secrecy. The transmission overhead for SIDH is similar to many of the public-key systems used today.

1.4 Evaluation and selection

In the end PolarSSL using RLWE as key exchange method was chosen. The most important reason for this choice was the fact that there existed a post-quantum modified version of PolarSSL that was accessible to the public. This was important since it was not in the scope of the project to develop an implementation from scratch. It is furthermore desirable to have the same base library when performing measurements so only the part being analyzed differs between the different algorithms, in this case the key exchange during a TLS handshake. This could be done with the modified version of PolarSSL since it supports both classical and post-quantum cryptography.

RLWE was however chosen as the key exchange algorithm for several reasons. It has reasonable key sizes compared to the other evaluated algorithms as can be seen in Table 1.1, it has been used in similar projects with other low performance devices [4][3] and it is thought to be secure [14]. The other algorithms could be excluded with the following reasons; SIDH had very little support and was only researched by a small number of people. NTRU was a good candidate, but could mean some legal issues for the company due to it being patented. McEliece was not suitable for an IoT device, because of the large keys, and there were also very few existing implementations available.

Note that there are a lot of other algorithms that are not evaluated in this project which possibly could be better choices. Which is the best algorithm to use is also very specific to what project it is meant for since criteria can be prioritized in different ways. When selecting an algorithm for a specific project it is important to weigh all factors such as, key size, memory usage, existing code and so on. In this project PolarSSL and RLWE were selected for the reasons stated above, but might not be suited in another project.

Chapter 2
Theory

This chapter will give the reader some background theory to understand the rest of the project. Note that some basic prior knowledge is required in computer science, computer security and mathematics to fully understand the report.

2.1 TLS

Transport Layer Security (TLS) is a cryptographic protocol used to ensure security on the Internet and computer networks. TLS provides security for the underlying transport layer in the *OSI model [15]*. The transport protocol Transmission Control Protocol (TCP) is often used together with TLS. TCP maintains a reliable stream of bytes between two communicating computers but needs TLS to provide authentication, data integrity and confidentiality during the communication [16], how that is achieved will be described later.

One big advantage with TLS is that it can be used with many different methods for encryption, authentication and key exchange. This makes it possible to adapt the TLS implementation depending on what security level is needed or supported by the parties setting up the connection [17].

2.1.1 Confidentiality

Communication between two computers achieves confidentiality when the connection is private and an eavesdropper can not understand the transferred data. To maintain confidentiality TLS encrypts the data to be sent.

There are two kinds of encryption techniques; symmetric and asymmetric [18]. Symmetric cryptography refers to algorithms only using one secret key. This key is used for encryption and decryption by both parties. The security in this technique relies on the fact that the key is secret, the two communication parties must be the only ones knowing the shared key.

In asymmetric cryptography each party has a private and a public key. The private key is only known to the owner and the public key can be known to everyone. The public key is used for encryption and the private key for decryption. The sender uses the recipient's public key to encrypt the message and it is only the corresponding private key that can be used to decrypt the message.

In order to use symmetric cryptography the private key needs to be shared securely between the communicating parties. This is the main disadvantage compared with asymmetric cryptography where the public key can be known by everyone [19]. On the contrary, asymmetric key sizes need to be rather large, which makes asymmetric algorithms generally very time consuming. Therefore, asymmetric cryptography is usually used for key exchange and symmetric cryptography used during the actual message transmission. This is the way symmetric and asymmetric cryptography is used in TLS.

2.1.2 Authentication

Confidentiality makes the transmission private between two communicating computers but it is also important to ensure that the communicating party is in fact who it claims to be.

Authentication is often obtained by using public key cryptography together with certificates. A certificate is a digitally signed document that binds a subject, which is the owner of the certificate, to its public key. One way to make this binding reliable is to let a trusted Certificate Authority (CA), also called the issuer, perform the binding. When browsing web sites on the Internet certificates are used to verify that the web site is trusted and secure. Whether an issuer is trusted or not is specified by a list of trusted CAs stored in the *Trusted root CA store* in the browser.

Normally only the server is authenticated when browsing web sites, this is the case in the example under Section 2.1.4. In some systems though, it is important for the server to authenticate the client as well. Some servers have this as a requirement and will not connect to a client that is not authenticated. Client authentication can be conducted in the following way [20][21]:

1. The server sends a certificate request message.
2. The client creates a *digital signature [22]* by making a hash of randomly generated data from the handshake and encrypts it with its private key. The hash and a copy of the clients certificate is then sent to the server.
3. The server checks if the certificate is trusted and valid. The server can then verify that the public key in the certificate actually corresponds to the private key used to create the digital signature. If this is the case, the client is considered as authenticated and the connection can be set up.

2.1.3 Integrity

To ensure integrity over the connection TLS uses checks on the messages being received by using a Message Authentication Code (MAC) [23]. The MAC is used for message authentication, but also to ensure the integrity of the message which means that it has not been modified during the transmission.

A MAC function, or a keyed hash function, is similar to a hash function but also needs a secret key to generate the digest. This makes it possible to not only ensure integrity of the message, but also authenticate the sender of the message due to the fact that the secret key only is known by the sender and recipient.

Message integrity and authentication using MAC between a sender named Alice, and a recipient named Bob, can be managed as explained below.

1. Alice uses the secret key and a keyed hash function to generate a MAC of the message to be sent. She then appends the MAC to the original message and sends it to Bob.

2. Bob receives the message and uses the same keyed hash function and secret key as Alice used, to generate a new MAC from the received message. If the generated MAC is the same as the one received from Alice, he can be sure that the sender is in fact Alice and the message was not modified during the transmission.

2.1.4 Handshake

The TLS handshake is used to let a client and a server that intend to set up a session agree on cryptographic parameters. The handshake is performed with a series of messages that are sent between the client and the server. Figure 2.1 shows an example of how a handshake is performed when the client authenticates the server [16].

The client sends a *ClientHello* to the server to initiate the session. The message also contains a random number, a list of cipher suits in order of preference and possibly a list of algorithms for compression which can also be set to none.

The server selects the best common cipher suite from the list the client sent and a compression algorithm, if they have at least one in common. If the server and client do not have a common cipher suite the connection will be aborted. If they have at least one common cipher suite the server sends a *ServerHello* to the client followed by a certificate. In this example RSA key exchange will be used. Finally the server sends a *ServerHelloDone* to indicate that the *ServerHello* is finished.

The client verifies the certificate to see if it is trusted and valid. If this is the case, a *pre-master secret* is generated. The client then sends the *pre-master secret* encrypted with the server's public key. The *pre-master secret* is used to generate the *master secret*, the client and the server calculates the master secret to let them share a common secret key. In order for the server to access the *pre-master secret*, it decrypts the message using its private key.

The client sends a *ChangeCipherSpec* message to notify that the following messages will use the recently negotiated keys and ciphers. The server also responds with a *ChangeCipherSpec*. A *ChangeCipherSpec* is always followed by a *Finished* message indicating that the key exchange and authentication processes were successful, as illustrated in Figure 2.1. The *Finished* message includes all handshake messages up to now encrypted with the negotiated secret. It is up to the receiver to verify that the content of the message is correct. After this a secure connection is established and both parties can send data protected by the parameters for this session. When the data has been sent the connection is terminated. If one wishes to send more data after that, a new connection must be established.

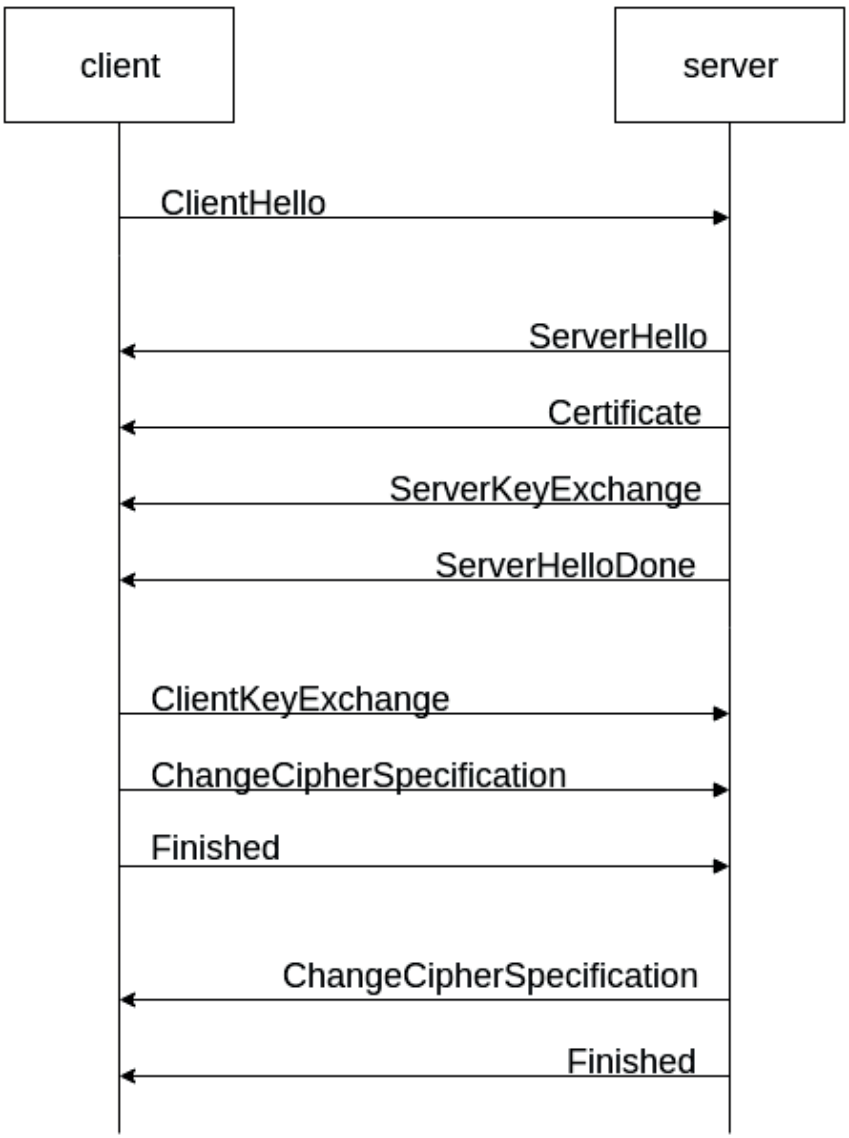

Figure 2.1: A generic TLS handshake.

2.2 Classical cryptography

Classical cryptography refers to the cryptographic algorithms that are used today, but are regarded as vulnerable to attacks from large quantum computers using e.g. *Shor's algorithm [24]*. Integer factorization and Discrete logarithm are problems believed to be hard to solve for digital computers, but can be solved in polynomial time with Shor's algorithm on quantum computers. The majority of the public-key algorithms used at present time are based on these two problems. Examples of classical cryptography algorithms are described below.

2.2.1 Diffie-Hellman

Diffie-Hellman key exchange is a way for two parties to exchange cryptographic keys in a secure way. It is most commonly used to establish a shared secret between the parties, for example a symmetric key to be used for encryption. To provide forward secrecy the so called *ephemeral Diffie-Hellman* is often used [25]. Ephemeral Diffie-Hellman, among other achieve forward secrecy by using temporary keys which are different in each session, this is not the case for *static Diffie-Hellman* where the same key is always used between two parties. Below is an explanation of the algorithm and its parameters [26]. The public and private keys in the context below shall not be confused with RSA public and private keys. For a simplified illustration of the algorithm, see Figure 2.2.

- p - a large prime
- g - a number that is mathematically linked to p.
- a - Alice's private key.
- b - Bob's private key.
- A - Alice's public key.
- B - Bob's public key.

The numbers p and g are known to everyone, even potential attackers. The selection of p and g is generally done beforehand.

The following demonstration illustrates how the algorithm works for Alice and Bob to calculate the shared secret s:

1. Alice selects a random number a, and sends $A = g^a \mod (p)$ to Bob.
2. Bob selects a random number b and sends $B = g^b \mod (p)$ to Alice.
3. Alice computes $B^a \mod (p) = s$
4. Bob computes $A^b \mod (p) = s$

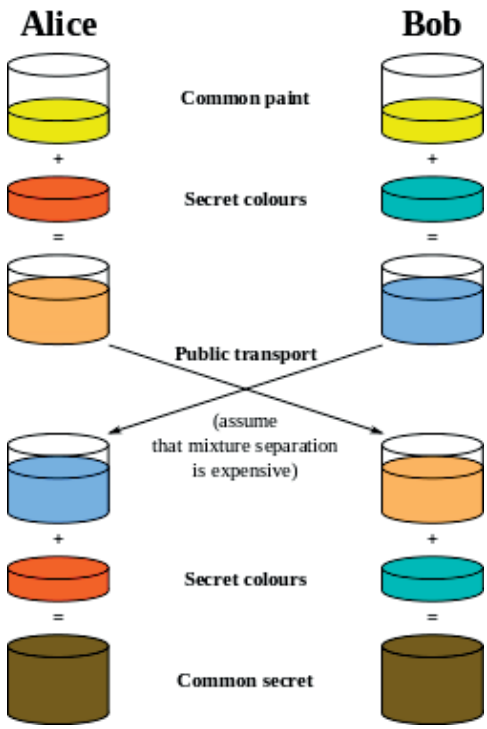

Figure 2.2: Illustrating the Diffie-Hellman key exchange with colors instead of variables, to get a general understanding of how the algorithm works.

2.2.2 Elliptic Curve Cryptography

Elliptic curve cryptography (ECC) is a public-key method using the algebraic structure of elliptic curves. It uses points on an elliptic curve to decide a public key that is mathematically linked to a private key [27].

One of the greatest advantages with ECC is that is has a much smaller key size compared to equal security with non-ECC cryptography, for example RSA.

Elliptic Curve Diffie-Hellman

Elliptic Curve Diffie-Hellman (ECDH) is a type of Diffie-Hellman key exchange, as the name indicates. The hardness of both the algorithms are based on discrete logarithms but in this case more specifically elliptic curves over finite fields. The sequence of sending and receiving data is the same as Diffie-Hellman as described above and illustrated in Figure 2.2, but the calculations and the actual data being sent differs. Below is a short explanation of how Alice can compute a shared secret with Bob and the parameters being used [28]:

- T - elliptic curve domain parameters
- d_a - Alice's elliptic curve private key
- d_b - Bob's elliptic curve private key
- Q_a - Alice's elliptic curve public key
- Q_b - Bob's elliptic curve public key

where d_a, d_b, Q_a and Q_b are associated with T. Alice can then compute the shared secret s by:

1. Calculate the elliptic curve point $P = (x_P, y_P) = d_a Q_b$
2. Check if $P \neq 0$ and if P=0 output "invalid" and stop
3. The shared secret is given by $s = x_P$

Bob can in a similar way receive s by doing the same calculations using his private key d_b and Alice's public key Q_a.

Elliptic Curve Digital Signing Algorithm

Elliptic Curve Digital Signing Algorithm (ECDSA) is an asymmetric signing algorithm that uses the hardness of elliptic curves as security [29]. As the name implies it is a variant of the algorithm *Digital Signing Algorithm*, proposed by the National Institute of Standards and Technology (NIST). The same keys as mentioned in the section above are used to sign and verify data to provide authenticity during communication.

2.2.3 RSA

RSA can be used in four different ways; key generation, key distribution, encryption and decryption. It is the same core algorithm, but the way it is used differs, which is clarified below.

- `Key generation:` Is how key-pairs are created, one public key and one private key. The keys are linked in a way where only the private key can decrypt a message encrypted with the public key. The procedure for generating the key-pair is conducted in the following way [26]:
  - Select two primes p and q.
  - Compute $n = pq$
  - Compute $\phi(n) = \phi(p)\phi(q) = (p-1)(q-1) = n - (p+q-1)$, where $\phi(n)$ is Euler's totient function [30].
  - Select an integer e where $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$, e is the public key.
  - The private key d is computed as follows:

$$ed = 1 \mod (\phi(n)) \tag{2.1}$$

- `Key distribution:` Distributing keys can be done by Alice sending her public key to Bob, and keeping her private key secret. Bob can now send messages to Alice that only she can decrypt. Bob can in the same way send his public key to Alice, to make it possible for her to send encrypted certificates to Bob.
- `Encryption:` When Bob intends to send a message M to Alice, M is first converted to the integer representation m of the message M, where $0 \leq m \leq n$ and $\gcd(m, n) = 1$, the transformation of the message M to m is given by a protocol. The cipher text c is then computed in the following way: $c \equiv m^e \mod (n)$.
- `Decryption:` When Bob has sent a message M to Alice she can decrypt it using her private key exponent d in the following way:

$$c^d \equiv (m^e)^d \equiv m \mod (n) \tag{2.2}$$

  which reveals the private message Bob sent to Alice.

2.2.4 AES

Advanced Encryption Standard (AES) is a block cipher algorithm based on symmetric keys [31]. The block size is fixed at 128 bits, but the key size can be one of the following; 128, 192 or 256 bits. The plaintext that is to be encrypted is transformed to several state matrices that are 4x4 matrices of 16 bytes, how a state matrix is transformed can be seen in Figures 2.3, 2.4, 2.5 and 2.6.

The algorithm consists of four transformation steps, explained below:

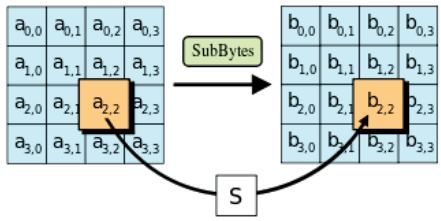

Figure 2.3: How the subByte step is performed.

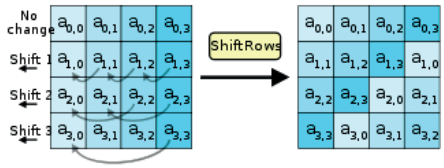

Figure 2.4: How the rows are shifted in the state matrix.

- `SubBytes`: In this step a substitution box (S-box) [32] is applied to each byte of the state matrix. This is the only nonlinear operation in the AES algorithm. Passing each byte through the S-box results in a new matrix as shown in Figure 2.3. The S-box is composed of two invertible mappings.
- `ShiftRows`: This step shifts the bytes in each row cyclically, each row is shifted with a different offset. In AES the offset is 0 for the first row, 1 for the second row, 2 for the third row and 3 for the last row. A visual explanation can be seen in Figure 2.4. The reason that the ShiftRow step is performed is to ensure that the columns are linearly dependent. If the columns are linearly independent then AES degrades to four independent ciphers.
- `MixColumns`: In this step each column in the state matrix is multiplied with a fixed polynomial $c(x) = 3x^3 + x^2 + x + 2$, the coefficients are written in hexadecimal representation. How this step is performed can be seen in Figure 2.5.
- `AddRoundKey`: In this step the round key, which is derived from the cipher key, is added by using bitwise XOR on every byte in the state matrix. The

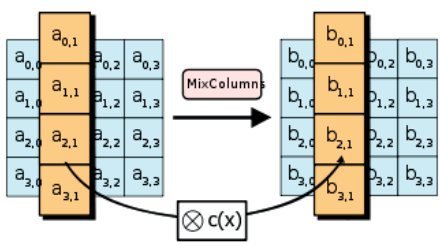

Figure 2.5: How the mixColumn step is performed.

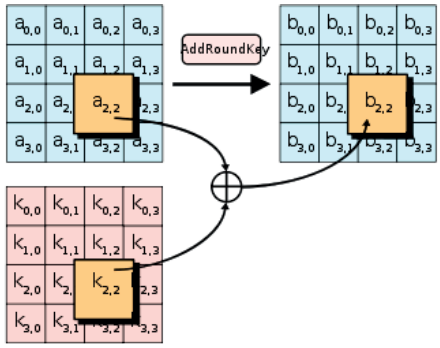

Figure 2.6: How the round key k is added to each byte in the state matrix.

length of the round key is equal to the length of the block, which is 128 bits. See Figure 2.6 for further details.

The structure of the algorithm and how the transformation steps are used can be seen in the pseudocode in Listing A.1. The encryption and decryption algorithms are not identical for AES, but the structure is the same.

2.3 Quantum computing

Quantum mechanics is very different from classical physics and can therefore require some extra thought for the inexperienced reader. In the section below, quantum computers and post-quantum cryptography is described in a simplified way. For the interested reader there are suggestions on possible further reading in this section.

2.3.1 Quantum computers

Quantum computers are systems making use of quantum mechanics in order to perform computations on data. Quantum computing is non-deterministic and every computation has a probability to produce the correct answer. A quantum bit, or qubit, is the quantum computers' equivalent to the digital computers' bit [33]. Where a regular bit can be 0 or 1 a qubit can be 0, 1 or both. One often refers to a qubit being 0,1 or both, but it is actually the state of an atom which can be ground, excited or both depending on energy levels. Ground and excited can easily be viewed as 0 and 1 for simplicity.

Imagine a computer with 2 bits resulting in four possible states the computer can be in; 00, 01, 10 and 11. The difference between a digital computer and a quantum computer is that the digital computer can only be in exactly one state of the four states above, but a quantum computer can be in all four states simultaneously. Every state has a probability, and the most probable state will be the final state of the quantum computer when it is done with the computation. To clarify, if the four states above are assigned a probability; A, B, C and D

respectively, then that means that the probability of 00 being the current state is A, the probability of 01 being the current state is B etc. The quality explained above is called superposition and is one of the properties that makes quantum computers powerful.

When performing a computation on a quantum computer it can at first, before the state of the computer is observed, be in all states simultaneously. Later on when the state is observed to get the result, all the states but the "correct" one will collapse and in a sense become a regular bit register where the result can be read.

One of the main differences between classical digital computers and quantum computers is that in quantum computers the accuracy of the result can increase polynomially with the input size. This characteristic makes quantum computers tremendously more powerful compared with ordinary computers [24].

The greatest obstacles to overcome in order to create large fully functional quantum computers is imprecision and decoherence [24]. Imprecision has to do with what precision the quantum gates can have in order to still have a reasonable probability to compute the correct answer. Decoherence happens when the state of the machine is not coherent any more, in other words, when the states are not coupled. Quantum computing relies on the fact that the states are coherent and evolve in an undisturbed way. If not, they do not possess their ability to be in superposition [33]. If a quantum computer has decohered it can not compute any answers any more. The time it takes for a quantum computer to decohere is actually the same amount of time the computer has to compute the correct answer. After that point the computation has failed if no answer is produced.

2.3.2 Shor's Algorithm

There are two versions of Shor's algorithm, one for solving discrete logarithms and one for prime factorization. The building blocks of the algorithms are gates, much like AND, OR and NOT in digital computers. The gates manipulate the states and help them end up in the correct state with a high probability. Both versions of the algorithm start in a superposition state and gradually increase the probability to get the correct answer when the state is observed. Both algorithms pose threats to today's public key encryption standards. The prime factorization algorithm would, for example, break RSA on a quantum computer, and the discrete logarithm algorithm would for example break Diffie-Hellman and Elliptic curve cryptography. The two algorithms are briefly described below. For a more mathematical explanation the reader is welcome to read the following paper [24].

Prime factorization

The algorithm does not try to factor n directly, where $n = pq$, p and q are large primes. Instead, it tries to find the integer $\lceil r \rceil$ that satisfies $x^r \equiv 1 \mod (n)$. Because p and q are odd numbers, n must also be odd. To find one of the factors of an odd integer n, the following operations can be performed. Given that there exists a method to compute the order r [24]:

1. Choose a random $x \mod (n)$

2. Find the order r of x
3. Calculate $\gcd(x^{r/2}-1, n)$

Because

$$(x^{r/2}-1)(x^{r/2}+1) = x^r - 1 \equiv 0 \mod (n),$$

$\gcd(x^{r/2}-1, n)$ and $\gcd(x^{r/2}+1, n)$ must be two factors of n. This algorithm can fail under some circumstances with low probability, the circumstances for when the algorithms fails are when r is odd or if $x^{r/2} \equiv -1 \mod (n)$. Shor's algorithm performs the computations in polynomial time on a quantum computer, with the ordo notation $\mathcal{O}((\log n)^2 \log\log n)$ in complexity. The most crucial step that actually makes this method work, is to calculate the order r of x. Calculating r efficiently has to be be done on a quantum computer in order to perform the algorithm in polynomial time. The computation is performed with the quantum gates that were previously mentioned.

Discrete logarithm

The approach here is approximately the same as for the prime factorization problem, the difference is that it involves solving a different equation, namely $g^r \equiv x \mod (p)$, where r is an integer and $0 \leq r \leq p-1$. Here g is a generator for $x \mod (p)$, x is a discrete logarithm and p is a prime. A generator g for $x \mod (p)$ is the cyclic sequence of numbers $[1, g, g^2, ..., g^{p-2}]$ where they represent the remainder of $\mod (p)$. To solve the discrete logarithm equation on a classic computer using the most efficient known algorithm still has the complexity $\mathcal{O}(e^{(\log p)^{1/3}(\log\log p)^{2/3}})$. This is why these calculations are not feasible in reasonable time for large numbers which makes cryptography based on discrete logarithms believed to be safe. Shor's discrete logarithm algorithm runs in polynomial time on a quantum computer and can calculate r correctly with a non-negligible probability of it being correct. Running the algorithm multiple times will increase the probability that r is the correct answer.

2.3.3 Grover's Algorithm

Grover's Algorithm is another algorithm that utilizes quantum mechanics to efficiently solve mathematical problems. The problem the algorithm solves is guessing a key that "unlocks" a black-box. Classical digital computers can find the key to a black-box in $\mathcal{O}(N)$ where N is the number of combinations in the domain. That complexity is with a brute force attack that tries all possible combinations and the worst case is that the last number is the correct number. With Grover's Algorithm one can find the key in $\mathcal{O}(N^{1/2})$ that is a considerable increase in speed if N is large. In order to be protected against these attacks it is suggested to double the length of the keys used today for symmetric encryption. The algorithm works in the following way:

Given that $|s\rangle = 1/\sqrt{N}\sum_{x=0}^{N-1}|x\rangle$, where $|s\rangle$ is the uniform superposition of all states.

Perform the following "Grover iteration" $r(N)$ times. The function $r(N)$, that has asymptotic complexity $\mathcal{O}(N)$, is described below.

- Apply U_ω, where $U\_\omega = I - 2|\omega\rangle\langle\omega|$.
- Apply U_s, where $U\_s = 2|s\rangle\langle s| - I$.
- Perform the measurement Ω. The result will be $\lambda\omega$ with probability very close to 1 for $N \gg 1$.

The reader is encouraged to read more about Grover's algorithm [34].

2.4 Post-quantum cryptography

2.4.1 Lattice problem

A *lattice problem* is a mathematical problem thought to be hard to solve for both classical and quantum computers. For use in cryptography, a lattice over $\mathcal{Z}^n$ is often considered. $\mathcal{Z}^n$ is a set of points, with integers as coordinates, in the n-dimensional real time space $\mathcal{R}^n$ [14]. A lattice, L, spans $\mathcal{Z}^n$ which consists of a set of n linearly independent vectors b:

$$L = \sum_{i=1}^{d} x_i b_i | x \in \mathcal{Z}$$

where d is the rank, if $d = n$ the lattice has full rank.

One of the so-called *hard lattice problems* is the *shortest vector problem* which basically is to find the length of the shortest non-zero vector in a lattice. This problem is *NP-hard* which essentially means there is no effective way of knowing that a vector is actually the shortest vector without going through all the vectors in the lattice and comparing if the given vector is shorter then all the other vectors. For NP-hard problems there is no known algorithm that can solve the problem in polynomial time, neither for regular nor quantum computers. This makes the problem not feasible to solve and therefore often considered secure. These properties makes the lattice problem very convenient to use in cryptographic algorithms. One algorithm that make use of the lattice problem is *Ring Learning With Errors*.

2.4.2 Ring Learning With Errors Key Exchange

Ring Learning With Errors (RLWE) is a computational problem that is used in post-quantum cryptographic algorithms [35]. RLWE is derived from the problem Learning With Errors (LWE) but is a larger problem designed for so-called *polynomial rings*. LWE is a problem in machine learning to distinguish noisy linear equations from random ones. RLWE can be reduced to the lattice problem *shortest vector problem* and is, due to its NP-hardness considered to be secure against quantum computers.

One suitable use for RLWE is in key exchange methods, which is used in *Ring Learning With Errors Key Exchange*. The RLWE key exchange method is, similar

to Diffie-Hellman, used to agree on a shared secret securely. Below is a simplified explanation of the parameters and the overall steps for the method [8][36]:

- a - a fixed polynomial
- s_A and e_A - Alice's private key
- s_B and e_B - Bob's private key

The fixed polynomial a is known to everyone and the private keys are only known to the owner. The private keys consist of two small polynomials where e is an error term. To create and agree on a shared secret Alice and Bob perform the following steps:

1. Alice creates her public key $A = a * s_A + e_A$ and sends it to Bob.
2. Bob creates his public key $B = a * s_B + e_B$ and sends it to Alice.
3. Alice computes the shared secret $s_A * B = s_A * a * s_B + s_A * e_B$
4. Bob computes the shared secret $s_B * A = s_B * a * s_A + s_B * e_A$

Since $s_A * e_B - s_B * e_A$ is small, it is possible to assume that $s_A * B \approx s_B * A$ which means that Alice and Bob have agreed on an *approximate* shared secret. To get an *exact* shared secret, Bob creates a string of "masking bits" which are sent to Alice. The masking bits give an extra hint of Bob's approximate secret that are fed to a so-called reconciliation technique. The reconciliation technique together with the masking bits provides Alice with enough information to let both parties derive an exact shared secret with a very high probability.

2.4.3 Multivariate Signature Schemes

A multivariate public-key cryptosystem is an algorithm based on multivariate polynomials which are polynomials consisting of more than one variable. This is the main approach to provide a secure communication when large quantum computers exist [37]. Some studies [38][3] further show that multivariate signature schemes are theoretically more efficient than classical schemes like for example RSA or ECDSA.

A suitable use for these cryptosystems are for signature schemes. Below is a simple explanation of the parameters and the idea of a multivariate public-key cryptosystem:

- F - a multivariate system of quadratic polynomials which is easy to invert
- S and T - two affine linear invertible functions
- $P = S * F * T$

The functions S and T are used to hide the structure of F by creating the public key as $P = S * F * T$ which is hard to invert. The private key consists of S, F and T and it is only when these parameters are known you can invert P.

Below, two post-quantum secure signing schemes using multivariate public key cryptography will be described. These can be used to provide authenticity between communicating parties.

TTS

Tame Transformation Signatures (TTS) was first introduced in 2002 and is based on T. Moh's theory for digital signature using Tame Transformation [38]. It is an extension of the first multivariate public-key cryptosystem named C^* which was broken.

There are different version of TTS and some of them are flawed which affect the security of the cryptosystem [38][39]. Some articles even claim that TTS is broken [40]. New versions of TTS which are gradually proposed are however said to be secure.

Rainbow

The Rainbow signature scheme was first introduced in 2005 by J. Ding and D. Smith. It is one of the most promising candidates to provide authenticity and like other multivariate schemes it is very efficient and has a very fast signature generation and verification [37].

The security of Rainbow has in the last years been investigated and many attacks have been proposed [41]. These attacks have however not yet succeeded to break the algorithm. This is partly due to the properties of Rainbow algorithm, but also due to the complexity of the attacks resulting in them being infeasible to run in polynomial time. For that reason Rainbow is considered secure at the time of writing this report, even when considering large quantum computers.

Chapter 3
System Description

This chapter will go through the hardware and software that was used during this project. It will give the reader relevant information regarding the system to be able to understand how it is configured. Furthermore it touches on the problems encountered and how they were solved.

3.1 Hardware

The initial plan for this project was to do all the implementation and measurements on a micro-controller. Due to memory problems and lack of RAM, the measurements were in the end performed on a Raspberry Pi, more about the problems that were encountered will be discussed in Section 6. This section will describe the two hardware platforms, their specifications and their performance differences.

3.1.1 STM3241G-EVAL

The STM3241G-EVAL [42] is an evaluation board with a STM32F417IG [43] microcontroller unit. It consists of an ARM Cortex-M4 core, memories and other hardware units. Below is a more detailed list of the hardware components on the microcontroller and the evaluation board. In Figure 3.1 a picture of the evaluation board can be seen.

- A 168 MHz single-core Arm Cortex-M4 processor
- 1 Mbyte internal flash memory
- 192 Kbytes internal SRAM including 64 Kbyte of CCM (Core Coupled Memory)
- 2 Mbyte additional external RAM
- 1 Gbyte or more MicroSD card
- Embedded ST-LINK/V2 debugger

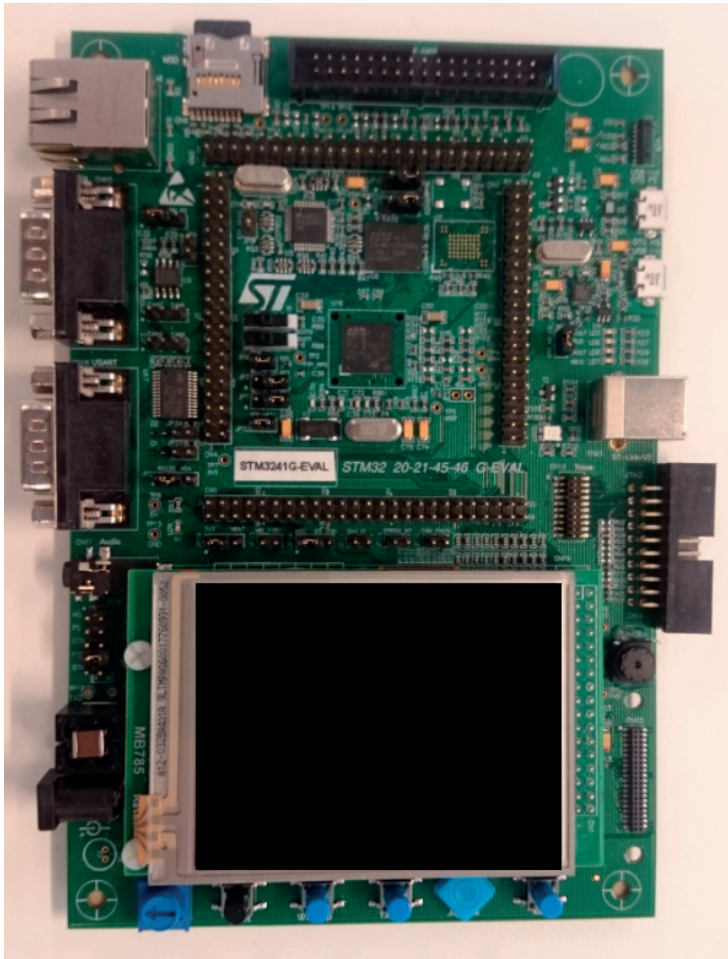

Figure 3.1: A picture of a STM3241G-EVAL.

3.1.2 Raspberry Pi 2 Model B

The Raspberry Pi 2 Model B [44] is a single-board computer with support for Linux and other operating systems such as Raspbian [45], which is used in this project. See Figure 3.2 for a picture of the board. It has the following hardware specifications:

- A 900 MHz quad-core ARM Cortex-A7 processor
- 1 Gbyte of SDRAM
- 1 Gbyte or more MicroSD card

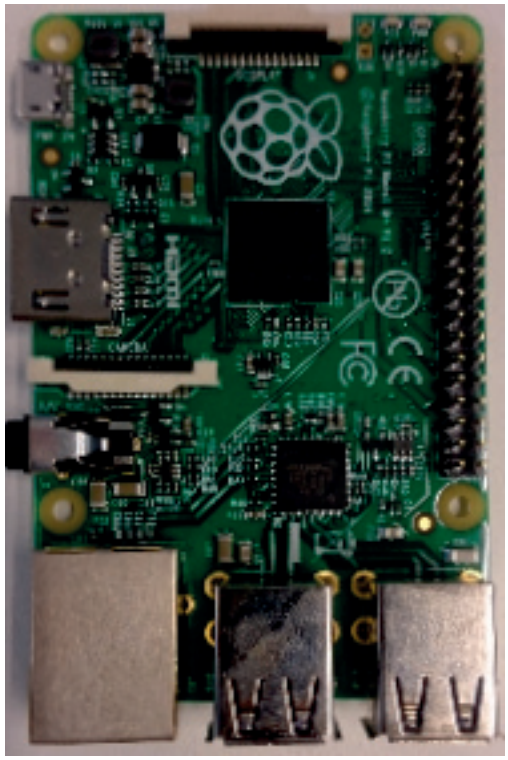

Figure 3.2: A picture of a Raspberry Pi 2 Model B.

3.1.3 Performance differences

The main differences between the Raspberry Pi and the STM32 evaluation board is that the Raspberry Pi has 4 cores, cache memory, higher clock speed and more RAM. The code is not written to utilize more then one core at a time, to minimize performance gains over the evaluation board. Even though the Raspberry Pi has more RAM than the evaluation board, it has memory of a different kind. The evaluation board has SRAM and the Raspberry Pi has SDRAM. SDRAM is a slower and cheaper kind of RAM memory. The processor clock frequencies are very different as well. To minimize speedup between the Raspberry Pi and the evaluation board, the clock speed has been reduced to 700MHz which is slightly more than 4 times faster. The clock speed was lowered in the operating system Raspbian that was running on the Raspberry Pi. The cache memory is one of the greatest problems when it comes to differences between the platforms. The code gets a lot of speed up from data prefetching and cache memory because it contains many loops and arrays that are written to and read from in order. To minimize the effects mentioned above and make the measurements more applicable on an IoT device the clock speed has been lowered and all parallel execution has been turned off. Unfortunately most differences can not be levelled which is why the Raspberry Pi will perform better than the evaluation board would.

3.2 Software

In this project several software packages/libraries have been used to form the final system. The individual software packages run on different levels in the system and cover areas like operating system and TCP/IP stack. More detailed descriptions about the individual software packages can be found below. Only the software running on the platform is described here. The software running on the Raspberry Pi was only the modified version of PolarSSL, apart from the operating system. Other software such as tools etc. are left out.

3.2.1 FreeRTOS

FreeRTOS is an event driven Real Time Operating System (RTOS) that switches processes or so-called tasks depending on priority. As most real time operating systems FreeRTOS is also designed to be lightweight and have a small footprint to save RAM and ROM. The exact amount of RAM and ROM used by the RTOS is hard to say, because it is very dependent on the applications that FreeRTOS runs. As a guideline it uses somewhere from 10KB of ROM and approximately 1KB of RAM. This operating system was chosen partly because it had good support and many users, but also because it was licensed under the GPL license which was convenient for the company.

3.2.2 PolarSSL

PolarSSL is a cryptography library that for example can be used to ensure secure communication over a network. In this project it was used to setup a TLS connection between two communicating parties.

Two versions of PolarSSL were used in this project. One was a standard version of PolarSSL and the other was a modified version with support for RLWE key exchange and multivariate signing algorithms. The modified version was the outcome of a research project that published their code and results [3]. The modified version was developed for Linux systems with a lot of RAM and computational power. Meaning that it is not ideal for an embedded system.

PolarSSL is now called mbedTLS. The reason why it is still called PolarSSL in this report is because at the time it was modified to be post-quantum secure it still bore the old name.

3.2.3 LwIP

LwIP is a lightweight TCP/IP stack that is intended for embedded systems. One of the advantages with LwIP is that it keeps resource usage low, but at the same time implements a full TCP/IP stack which can be very useful if the device is to communicate with other devices in a network. It can also be used to implement 802.1x certificate solutions.

Chapter 4
Methodology

In this chapter the workflow and methodology is described. The aim of the project was to run the measurements on the evaluation board, but as explained later that was not possible.

4.1 Implementation on a STM3241G Evaluation Board

The following section describes the methodology during the project on the evaluation board.

4.1.1 Workflow

The first step in the implementation part of the project was to set up a regular lightweight version of PolarSSL with FreeRTOS and LwIP. When all components worked together and the TLS connection had been verified, it was time to switch cryptography library to the modified PolarSSL with support for post-quantum algorithms. This is illustrated in Figure 4.1, where the modified version of PolarSSL is called PQ-PolarSSL.

The new version of PolarSSL was included into the project and the first task was to get classical cryptography working. The project seemed however to need more RAM than was available, even for the classical cryptography.

To be able to run the new setup the external RAM on the evaluation board was needed. A lot of work was needed to get the RAM working properly but it made it possible to run the application with a post-quantum cipher suite, at least to a certain point in the TLS handshake where the application crashed due to some memory problem. Since it seemed to be a problem with FreeRTOS it was updated to a newer version. This together with a change of memory allocator in FreeRTOS did unfortunately not fix the problem. A more detailed description of the problems and solutions will be described in Section 6.5.2. In the end the measurements were executed on the Raspberry Pi, due to the fact that the bugs and problems with external RAM and FreeRTOS were not solved in time. A description of that can be found in Section 4.2.

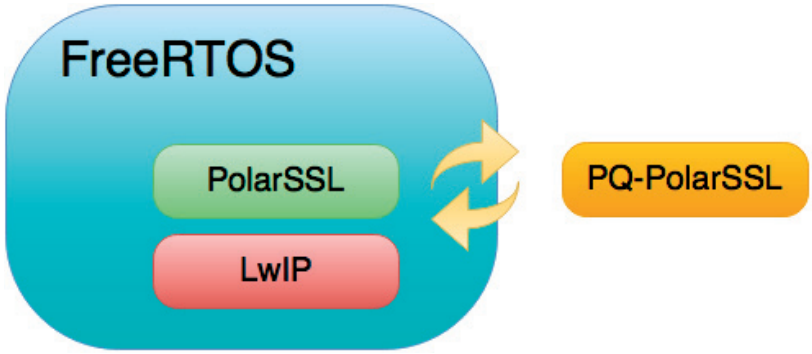

Figure 4.1: Shows what part of the software was changed and what was left the same.

4.1.2 Test and debug

To test and debug the application during the implementation process different techniques were used. When classical cryptography was running, a simple web-server displaying a small website was set up on the evaluation board which could be accessed via a web browser running on a computer connected to the same network. Traffic could then be analyzed with the network analyzer *Wireshark* [46] to verify that everything was working as expected.

Since normal web browsers do not support post-quantum cryptography another technique to debug the modified version of PolarSSL was needed. Two evaluation boards were connected to a switch, the two ports that the evaluation boards were connected to mirrored their traffic to a third port. The third port was connected to a computer that monitored the communication with Wireshark to ensure that the TLS handshake was performed in a correct manner. This setup is graphically shown in Figure 4.2

4.2 Implementation on a Raspberry Pi

4.2.1 Workflow

Since the modified version of PolarSSL was originally implemented to run on a Linux machine it was also directly compatible with Raspbian running on the Raspberry Pi. It could easily be compiled and built with a simple Make-command, and then executed.

For simplicity both the server and the client were started on the same Raspberry Pi communicating locally over localhost. Since only one party is operating at a time and the focus of the thesis is not to measure network utilization this should not affect the final result. The clock frequency for the processor was configured to 700 MHz to at least be a little more similar to an IoT device.

When the setup was verified to be correct, the server and the client were configured to run with different cipher suites to compare the key exchange methods.

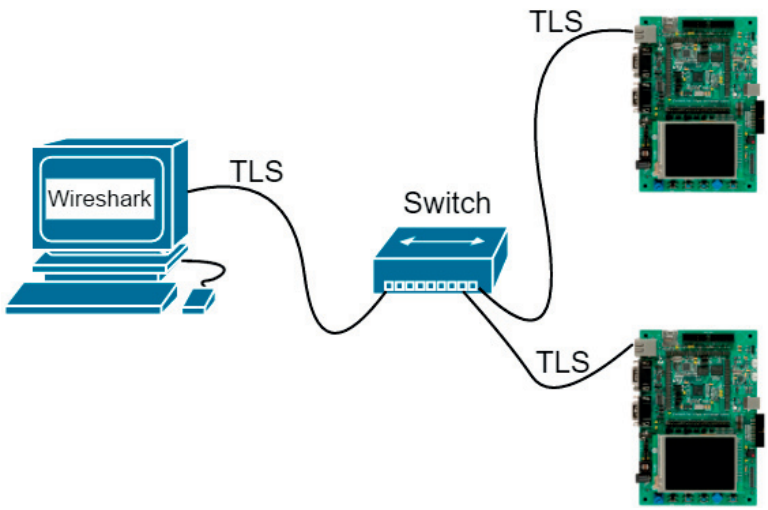

Figure 4.2: Setup of how the evaluation boards were connected to the computer running Wireshark.

4.2.2 Measurements

The measurements were conducted by using the clock in the operating system and calculate the difference before and after the key exchange. Other techniques like counting the number of clockcycles for a process were considered but since the interest is in the total amount of execution time, both for the server and the client, the clock was used.

The clock has a resolution of microseconds and to make the result more accurate each key exchange method was executed 100 times and a mean value was calculated.

For the LATTICEE (RLWE) based suites there are two sets of parameters that offer different levels of security. Param1 offers 80 bits of security and param3 offers 128 bits of security [3]. For a more specific description of the different parameters, see [47].

The cipher suites that were used during the tests were the following:

- TLS-DHE-RSA-WITH-AES-128-GCM-SHA256
- TLS-ECDHE-ECDSA-WITH-AES-128-GCM-SHA256
- TLS-LATTICEE-RAINBOW-WITH-AES-128-GCM-SHA256 with param1
- TLS-LATTICEE-RAINBOW-WITH-AES-128-GCM-SHA256 with param3
- TLS-LATTICEE-TTS-WITH-AES-128-GCM-SHA256 with param1
- TLS-LATTICEE-TTS-WITH-AES-128-GCM-SHA256 with param3

The RSA that was used in this project was 2048-bit RSA, and for ECC, 256-bit ECC was used. The reason that DHE-RSA and ECDHE-ECDSA were selected was that they are common key exchange methods at the present time and the cipher

suite offered the same symmetric encryption and MAC as the post-quantum secure cipher suites used. The other signing algorithms used were TTS and Rainbow, where TTS version one was used. There are several versions of TTS and version one is believed to be insecure at present time. Rainbow on the other hand is not regarded as broken.

The creators of the modified version of PolarSSL named the RLWE-based key exchange method LATTICEE as you can see in the list above. In this report it will be called RLWE.

As can be seen in the list above, the same MAC and encryption algorithm were used for all the suites. Only the key exchange and signing algorithm differs. That is to keep the measurements as accurate as possible, even though only the key exchange part of the communication is timed.

Chapter 5
Results

In this section the main results will be presented. See Table 5.1 for information about how long time the different key exchanges took. In Table 5.3 and Table 5.4 one can see the amount of RAM the client and the server application required to execute respectively, analyzed with Valgrind [48] using the tool massif. In Table 5.2 performance information is displayed for both platforms. Dhrystone Million Instructions Per Second (DMIPS) is a benchmark metric which can be used to compare processor performance [49].

The key exchange methods using classical cryptography are faster than the ones using post-quantum cryptography, the duration is however very dependent on the parameters being used as can be seen in Table 5.1.

Cipher suite	Time (s)
DHE-RSA	0.9141
ECDH-ECDSA	1.3716
RLWE-RAINBOW-param1	1.9293
RLWE-RAINBOW-param3	3.4594
RLWE-TTS-param1	1.7993
RLWE-TTS-param3	3.4216

Table 5.1: Duration of the key-exchange for different cipher suites running on the Raspberry Pi.

Platform	DMIPS/MHz	DMIPS	Speed (MHz)
Raspberry Pi 2 Model b	1.9	1330	700
STM3241G-EVAL	1.25	210	168

Table 5.2: Performance figures for the hardware platforms.

The standard implementation of PolarSSL uses less RAM than showed in Table 5.3 and Table 5.4, and was possible to run in internal memory on the evaluation board. Since the initial project was developed for an evaluation board it could not run on a Linux machine and it was therefore not possible to evaluate with Valgrind. What can be said is that the standard implementation requires less RAM

than the 192KB of internal RAM that the evaluation board has. The memory usage measurements shown in the tables are only for PolarSSL, in order to run the application on the evaluation board it requires LwIP and FreeRTOS which also require some RAM and ROM even if it is a small amount in comparison to PolarSSL.

The server and client applications with the modified implementation of PolarSSL could not be run on the evaluation board in internal RAM regardless of key exchange method. The application that required the least amount of RAM was the server running DHE-RSA, which required at least 133KB more RAM compared to the 192KB of internal RAM that the evaluation board had. From this one can see that it is hard to implement post-quantum secure TLS on the evaluation board due to memory constraints. Several of the key exchanges could be run on the evaluation board with external RAM, that gives an indication that it is possible, but will be slow in terms of performance.

Cipher suite	RAM usage (kB)
DHE-RSA	342
ECDH-ECDSA	349
RLWE-RAINBOW-param1	1721
RLWE-RAINBOW-param3	2939
RLWE-TTS-param1	1683
RLWE-TTS-param3	2872

Table 5.3: RAM usage for the client application in PQ-PolarSSL.

Cipher suite	RAM usage (kB)
DHE-RSA	325
ECDH-ECDSA	328
RLWE-RAINBOW-param1	1822
RLWE-RAINBOW-param3	2985
RLWE-TTS-param1	1717
RLWE-TTS-param3	2858

Table 5.4: RAM usage for the server application in PQ-PolarSSL.

The measurements performed on the code are from compiling the code with GCC using optimization level 2. All other optimizations that can be turned on in PolarSSL have been turned off, such as specific optimizations for algorithms or other optimizations for using less RAM or ROM. Optimization for size has not been used either when compiling the project.

Chapter 6
Analysis

In this project we have evaluated different key exchange methods and signing algorithms to see if post-quantum secure cryptography could be used today. Furthermore is it possible to use in a low performance IoT platform?

The measurements have been performed on a Raspberry Pi, even if it was not our original idea of an IoT-device. One can however easily view it as an IoT-device because the term IoT is very broad. What we refer to as an IoT device in our report is a device with performance similar to our evaluation board. The analysis will still cover IoT devices, even though a Raspberry Pi has fairly high performance. Our theory about the future of IoT devices from a security standpoint will also be presented.

In this thesis we have mainly focused on the actual performance i.e. how fast a key exchange can be performed. Other factors that can be of equal importance is footprint, memory usage, security, power consumption and so on. It is very important to keep in mind that a recommendation in this report is not necessarily the correct recommendation for another application running in another environment. The importance of the aspects mentioned above can differ a lot and might result in a different key exchange method or higher respectively lower security being selected.

It is not impossible that low performance IoT devices will have the computational power of a Raspberry Pi in the future. Because the cryptography analysis in this project also is intended for the future, we still think it is relevant even though no measurements have been performed on what we refer to as an IoT device in this report. In this chapter these topics will be discussed and answered.

6.1 Measurements

As can be seen in Table 5.1 the classical algorithms have the fastest key exchange. Thereafter comes the RLWE key exchange method with param1 using TTS as authentication algorithm closely followed by Rainbow. The difference is however only 0.4 seconds when comparing ECDH-ECDSA with RLWE-TTS using param1 which makes RLWE-TTS the best candidate for a post-quantum key exchange method when considering speed. Looking at Table 5.3 and 5.4 one can see that RLWE-TTS is also the method that uses the least amount of RAM, using param1.

If looking at the same cipher suites but for param3 the durations are around

1.5 seconds larger than the measurements for param1. Also with these parameters the TTS authentication algorithm is slightly faster than Rainbow. These results are considerably longer than the ones with the classical algorithms, being almost 4 times slower than DHE-RSA and 2.5 times slower than ECDH-ECDSA.

According to our measurements the signing algorithm does not affect the performance to a large extent. TTS is marginally faster than Rainbow according to the result, which is to be expected since they are based on the same problem but TTS has smaller private keys. For the same reason TTS uses less memory compared to Rainbow. It is the key exchange itself that is the most time consuming part.

Previously stated, in Section 4, 100 measurements were done for each cipher suite to make the result more accurate. When looking at these measurements the values were varying somewhat between the readings. This is probably in part due to the complexity of the algorithms. The polynomials and parameters being used for the key exchange are randomly generated and can therefore be of different complexity. They are always the same length but can be harder or easier to process by the algorithm and can therefore give some different results among the runs. One must also keep in mind that the algorithm is non-deterministic, meaning that the result can differ even for the same input. Another thing that could affect the preciseness of the result is the fact that the application is not executed on a real-time operating system. The risk with this is that another process with higher priority could interrupt the execution of our measurement.

To get an indication about how many times faster the Raspberry Pi is compared to the evaluation board one can look in Table 5.3, where DMIPS is presented for both platforms. According to DMIPS is the processor used on the Raspberry Pi over 6 times faster than the processor on the evaluation board.

6.2 Parameter choice

We have not tested all possible parameters for the algorithms and recently there have been some suggestions for better parameters, some offering up to 70% key reduction [50]. That would probably speed up the key exchange considerably. On the other hand the mathematical problem could possibly increase in complexity and therefore be more time consuming to compute. One must keep in mind that it is not only the key size and overhead that is limiting the performance, it is also the computation of the polynomials that takes up a lot of time. To save time one could have the polynomials pre-generated, but that would not be a secure solution even if it would save a lot of time.

The parameters are specific for the key exchange method RLWE. In the future it is possible that not only new parameters will exist for RLWE, however it is also possible that new lattice based cryptography algorithms are present. Because the field of computer security and cryptography is always changing it is hard to give a recommendation about parameters or even algorithms for that matter. The parameters and algorithms selected at the time of the thesis where the ones best suited for our needs at the time.

6.3 Suitable for IoT devices and security

Even in the future when the general performance has increased, there will still be a need for cheap low performance devices. This is a big problem, when security is neglected in order to produce cheap and innovative devices connected to the internet. Companies will not start producing secure IoT devices until customers start demanding high security and customers will not start demanding security until they understand the risk they are at.

If one compares ECDH-ECDSA with RLWE-TTS-param1 or RLWE-RAINBOW-param1, the performance penalty is not too large. On the other hand, they only offer 80 bits of security, as can be seen in Table 6.1, which is a rather low security level compared to the elliptic curve with 128 bits security. From this we can conclude that it is pointless to use these cipher suites today because they are slower and have lower security. But the day large quantum computers are around DHE-RSA and ECDH-ECDSA will be broken and then 80 bits will be better than none. The security level of 80 bits is however too low to be classed as secure and preferably the corresponding cipher suites with parameter set 3 would be used to provide 128 bits of security. On the other hand if memory usage is very important it could be very hard to adopt one of the RLWE based key exchange methods using parameter set 3, as they require almost 3MB of RAM each.

Cipher suite	bits of security
DHE-RSA	112
ECDH-ECDSA	128
RLWE-RAINBOW-param1	80
RLWE-RAINBOW-param3	128
RLWE-TTS-param1	80
RLWE-TTS-param3	128

Table 6.1: Bits of security for different cipher suites. [2][3]

The durations for the key exchanges with RLWE-TTS and RLWE-RAINBOW with param3 are a lot longer than the duration for the classical ones. We therefore think that they are too slow for today's IoT devices, maybe even for a relatively high performance platform as a Raspberry Pi, where it take over 3 seconds to perform the key exchange. For many applications that is too long and it would create a bad experience for the user of the system. Small hardware platforms like IoT devices are however continuously increasing in performance and in a couple of years they might have sufficient performance to run these algorithms in reasonable time. If not, one can hope for faster post-quantum algorithms or better optimization of existing post-quantum algorithms. Hopefully this will happen before large quantum computers exist.

6.4 Optimization

The RLWE key exchange method has not been optimized in the same way as the algorithms used today have been. Many other key exchange methods have been used for many years and these implementations have continuously been optimized to be as efficient as possible, yet retaining their security. RLWE is a rather new technique compared with for example DH and optimization of the implementation could probably bring up the speed a lot. RLWE-TTS-param1 could possibly be optimized down to a figure as low or lower than ECDH-ECDSA. This project did not have the time to do a correct security analysis and we did therefore not try to do any optimization of our own because of the risk of destroying the security of the algorithm.

The modified version of PolarSSL that was used had been developed on a powerful Linux machine. It was therefore not suitable for this kind of low performance hardware platform and little or no attention had been paid to how much RAM the applications needed.

6.5 Problems

As mentioned in Section 4.1, many problems were encountered when trying to implement the modified version of PolarSSL on the evaluation board. This section will go through the problems and solutions more thoroughly.

6.5.1 Development environment

One of the most frequent problems during the thesis project was the development environment. Instead of using Linux machines and make files, we used Windows and an Eclipse-based development tool. This tool was easy to start using and getting a project working quickly, but every time one wanted to change a file or update something in the project we ran into trouble, due to linking problems and other limitations in the development tool. Changing a file or updating a library in a project is a very common task and should be easy to have a good workflow. We realize now that using make and GNU tools would have been much easier in the long run, even if it had meant a learning curve in the beginning. At the point we realized that we had made a mistake in the choice of development environment it was too late to change. We had already invested a lot of time in to it and changing environment would only mean more setbacks. Therefore we stuck with the Eclipse based environment throughout the length of the project.

6.5.2 Upgrading to post-quantum algorithms

The first challenge was to include the new version of PolarSSL into the existing workspace. Even if the architecture was the same, the modified version had a fair amount of changes made to it. The modified version was also considerably larger in size compared with the first lightweight version. We changed the configuration file to optimize for low RAM and ROM and optimized the code by decreasing unnecessarily large allocations. We realized quite fast that more RAM memory

was needed to be able to run the modified version of PolarSSL and especially to be able to run the post quantum algorithms. After all one of the polynomials required up to 100KB of RAM and the certificates ranged from 70KB to 250KB depending on algorithms. Running that on an evaluation board with 192KB of RAM would be very hard or even impossible.

The implementation of the external RAM was very troublesome. It proved to be very hard to use the combination of FreeRTOS and external RAM because when using FreeRTOS with its own memory allocator the stack is placed in the heap. This meant that the system had multiple heaps and stacks growing in different directions. After a lot of debugging and different setups for how memory was allocated and located in RAM the system worked.

After the external RAM was correctly configured with FreeRTOS the first step was to run classical cryptography, which we now got working. When switching to RLWE as key exchange method everything seemed to work until the server was going to generate its public key. After many more hours of debugging we figured out why. At one point in time during the key exchange the server had some overflow issues where the memory allocator had very strange values for a variable that tracked how much memory the system had left. Just before the variable got corrupt values it had approximately 1MB of RAM left. We spent a tremendous amount of time to try and fix this issue, with help from our supervisor, but nothing seemed to help.

We found a bug in FreeRTOS update history that displayed similar behavior as our application and decided to update our operating system to a newer version. According to their update history the bug should have been corrected in the version we where running but this was a safety precaution. Unfortunately the update did not help, not even after changing the memory allocator. Due to lack of time we decided to run the measurements on a more powerful platform. The company had a Raspberry Pi at their disposal and even if it is not very low performance we decided to use it instead of using the evaluation board.

After evaluating the server and client application on the Raspberry Pi, with Valgrind, it is however evident that it would not be possible to run the applications on the evaluation board without modifying it further. According to the memory profiling tool massif, with information about stack enabled, the server application uses 2858 kilobyte of RAM at its peak and the client uses 2872 kilobyte of RAM at its peak, as can be seen in Table 5.3 and Table 5.4. In order to run the applications on the evaluation board one would have to rework the whole code and that was not possible in this 20 week long project.

6.6 Conclusions

Security is very important, not only for IoT devices. In the future when RSA, elliptic curve and Diffie-Hellman will be broken we suggest to use one of the post-quantum secure key exchange methods that were evaluated in this project. However if better suited algorithms have been invented in the future they should be used instead. In other words we think that at the time of the project, lattice based key exchange methods offer good qualities in terms of footprint, memory

usage, speed and security. Furthermore we think that multivariate signing algorithms such as Rainbow and TTS also are promising candidates for the future. The day large quantum computers are produced the performance of regular computers and embedded systems may be significantly higher, meaning that a more time consuming algorithm today may have reasonable speed in the future.

We see no need to change the key exchange methods used today if speed for the key exchange is important, because it is hard to justify halved speed for equal security level. If security is very important and the system has higher performance than an IoT device then maybe it could be worth it. Again it is very dependent on the application. If time was invested in the algorithms and it was possible to achieve performance equal to today's public key cryptography then it would be wise to change algorithms and be protected before the current standards have been broken.

TTS has been claimed to be broken at different points in time and several versions of TTS have been developed to cope with the vulnerabilities. For that reason we suggest that Rainbow should be used instead as a signing algorithm because it is not vulnerable to known attacks.

If large quantum computers are produced earlier than expected and IoT devices have not yet caught up in performance then we think that the price of one or two extra seconds during the key exchange is low compared to the security gain it offers. We would gladly wait that time, and start using post-quantum secure algorithms like RLWE, TTS, Rainbow and double key size for symmetric encryption provided that the devices have enough RAM to run the algorithms.

Below are the main conclusions listed:

- It is possible to set up a secure TLS connection on the evaluation board using classical cryptography
- It is hard to implement post-quantum secure TLS on the evaluation board due to memory constraints
- It is not possible to run the modified version of PolarSSL being used in this project on the evaluation board without major rework.
- The RLWE key exchange method with small parameters can be used on a Raspberry Pi for key agreement with reasonable time, it however provides only 80 bits of security.
- The RLWE key exchange method with large parameters can be used on a Raspberry Pi to provide 128 bits of security, it takes however over three seconds for the key exchange. In the future when the performance of IoT devices has been increased, this configuration may after all be a good solution.
- Lattice-based cryptography seems like a good candidate for key exchange in the future, at the time of the project.
- Multivariate-based cryptography seems like a good candidate for signing in the future, at the time of the project

Chapter 7
Future work

If time allowed it would be very interesting to further develop the platform and add more support for several different key exchange methods that are post-quantum secure. During the project some ideas have come up that would be fun to elaborate further, that we did not have time to do or was out of the scope of this project. As we said before there is only so much that one can do in 20 weeks.

Because we struggled a lot with the development environment we would like to migrate the whole project to Linux instead so that for example GNU tools could be used instead and compilation could be done with make. This would also be valuable knowledge to have for the future.

Examples of algorithms that would be interesting to implement and evaluate further are NTRU, SIDH and possibly some McEliece algorithms. In this project we only had time for one key exchange algorithm and therefore it is hard to conclude if that is the best suited algorithm for our application. It is also hard to say if it is a candidate at all when no measurements have been made in between different post-quantum algorithms. For that reason it would be very interesting to see how the other algorithms perform, and how much memory they require to run. We would be especially interested in seeing how NTRU would perform compared to for example RSA, Diffie-Hellman, but also to other post-quantum algorithms. It would be interesting for the simple reason that according to the companies performance figures, NTRU is supposed to be superior compared to other algorithms. Doing some measurements on all the algorithms would reveal if NTRU is fast in general or only in specific circumstances.

Instead of running everything on a Raspberry Pi it would be fun to run it on one of Axis platforms and see how it performed there. To take it even further it would be interesting to try and integrate our platform with several of the companies products, and make it work well with their platforms. Another hardware related idea that we would like to explore further is exactly how small hardware platform one can use for this project? What is the least amount of ROM and RAM that the system can have? Could we potentially rewrite some parts of the code to save some extra memory?

An interesting project would be to optimize the RLWE algorithm so that RAM and ROM were used in the most efficient way. All constant variables could be stored in flash and loaded piece by piece so that RAM memory was saved, after all flash memory is much cheaper then SRAM. But that would require a lot of time and in the end the algorithm would have to be analyzed to see if it still was

secure.

Lastly, but not least we would like to figure out a good way to debug our memory bugs and shed some light on what actually went wrong. It is still an unanswered question why we could not use all of the external RAM, after all some of the post-quantum secure algorithms used less then 2MB of RAM and should have been able to run in external memory on the evaluation board.

References

[1] Wikipedia. Post-quantum algorithms key sizes. https://en.wikipedia.org/wiki/Post-quantum_cryptography#Key_size_table, last accessed 2016-05-04.

[2] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management - part 1: General (revision 3). http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf, last accessed 2016-05-10, 2016.

[3] Yun-An Chang, Ming-Shing Chen, Jong-Shian Wu, and Bo-Yin Yang. Postquantum ssl/tls for embedded systems. In *Proceedings of the 2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*, SOCA '14, pages 266–270, Washington, DC, USA, 2014. IEEE Computer Society.

[4] Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Efficient software implementation of ring-lwe encryption. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 339–344, San Jose, CA, USA, 2015. EDA Consortium.

[5] fast-crypto lab. Pq-polarssl. https://github.com/fast-crypto-lab/PQ-polarssl, last accessed 2016-01-25.

[6] ARM. mbed tls (polarssl). https://tls.mbed.org/, last accessed 2016-01-22.

[7] Hauke Petersen, Emmanuel Baccelli, and Matthias Wählisch. Interoperable Services on Constrained Devices in the Internet of Things. In W3C, editor, *W3C Workshop on the Web of Things*, Berlin, Germany, June 2014.

[8] Vikram Singh. A practical key exchange for the internet using lattice cryptography. Cryptology ePrint Archive, Report 2015/138, 2015. http://eprint.iacr.org/.

[9] Chris Peikert. Lattice cryptography for the internet. Cryptology ePrint Archive, Report 2014/070, 2014. http://eprint.iacr.org/.

[10] wolfSSL. Cyassl+ntru - high-performance ssl. https://www.wolfssl.com/files/flyers/cyassl_ntru.pdf, last accessed 2016-02-29.

[11] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. Mcbits: fast constant-time code-based cryptography. Cryptology ePrint Archive, Report 2015/610, 2015. http://eprint.iacr.org/.

[12] Rafael Misoczki, Jean-Pierre Tillich, Nicolas Sendrier, and Paulo S. L. M. Barreto. Mdpc-mceliece: New mceliece variants from moderate density parity-check codes. Cryptology ePrint Archive, Report 2012/409, 2012. http://eprint.iacr.org/.

[13] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *Proceedings of the 4th International Conference on Post-Quantum Cryptography*, PQCrypto'11, pages 19–34, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] Thijs Laarhoven, Joop van de Pol, and Benne de Weger. Solving hard lattice problems and the security of lattice-based cryptosystems. *IACR Cryptology ePrint Archive*.

[15] Behrouz A. Forouzan. Data communications and networking. pages 29–42, New York, NY, USA, 2007. McGraw-Hill, Inc.

[16] Dieter Gollmann. Computer security. pages 310–313, New York, NY, USA, 2011. John Wiley & Sons, Inc.

[17] P. Karlton A. Freier. https://tools.ietf.org/html/rfc6101, last accessed 2016-02-03.

[18] Dieter Gollmann. Computer security. pages 264–270, New York, NY, USA, 2011. John Wiley & Sons, Inc.

[19] Michael Cobb. What are the differences between symmetric and asymmetric encryption algorithms? http://searchsecurity.techtarget.com/answer/What-are-the-differences-between-symmetric-and-asymmetric-encryption-algorithm last accessed 2016-02-18.

[20] Oracle. Client authentication during ssl handshake. https://docs.oracle.com/cd/E19424-01/820-4811/aakhe/index.html, last accessed 2016-02-19.

[21] Clemens Heinrich. *Encyclopedia of Cryptography and Security*, chapter Secure Socket Layer (SSL), pages 1135–1139. Springer US, Boston, MA, 2011.

[22] Dieter Gollmann. Computer security. pages 260–264, New York, NY, USA, 2011. John Wiley & Sons, Inc.

[23] Behrouz A. Forouzan. Data communications and networking. pages 969–970, New York, NY, USA, 2007. McGraw-Hill, Inc.

[24] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, October 1997.

[25] ARM. Why use ephemeral diffie-hellman. https://tls.mbed.org/kb/cryptography/ephemeral-diffie-hellman, last accessed 2016-01-23.

[26] Behrouz A. Forouzan. Data communications and networking. pages 949–956, New York, NY, USA, 2007. McGraw-Hill, Inc.

[27] Nick Sullivan. A (relatively easy to understand) primer on elliptic curve cryptography. http://arstechnica.com/security/2013/10/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/2/, last accessed 2016-05-10.

[28] Certicom Research. Standards for efficient cryptography, SEC 1: Elliptic curve cryptography, September 2000. Version 1.0.

[29] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International Journal of Information Security*, 1(1):36–63, 2001.

[30] Burt Kaliski. Euler's totient function. In *Encyclopedia of Cryptography and Security (2nd Ed.)*, page 430. Springer, 2011.

[31] Joan Daemen and Vincent Rijmen. Rijndael. In Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security, 2nd Ed.*, pages 1046–1049. Springer, 2011.

[32] Wikipedia. S-box. https://en.wikipedia.org/wiki/S-box, last accessed 2016-02-18.

[33] Ari Ben-Menahem. *Historical Encyclopedia of Natural and Mathematical Sciences*. Springer, Berlin, 2009.

[34] C. Lavor, L.R.U. Manssur, and R. Portugal. Grover's algorithm: Quantum database search. arXiv.org, 2008. http://arxiv.org/abs/quant-ph/0301079.

[35] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. *Advances in Cryptology – EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 – June 3, 2010. Proceedings*, chapter On Ideal Lattices and Learning with Errors over Rings, pages 1–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[36] Vikram Singh and Arjun Chopra. Even more practical key exchanges for the internet using lattice cryptography. Cryptology ePrint Archive, Report 2015/1120, 2015. http://eprint.iacr.org/.

[37] Albrecht Petzoldt, Stanislav Bulygin, and Johannes Buchmann. *Post-Quantum Cryptography: Third International Workshop, PQCrypto 2010, Darmstadt, Germany, May 25-28, 2010. Proceedings*, chapter Selecting Parameters for the Rainbow Signature Scheme, pages 218–240. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[38] Jiun-Ming Chen and Bo-Yin Yang. *Information Security and Cryptology - ICISC 2003: 6th International Conference, Seoul, Korea, November 27-28, 2003. Revised Papers*, chapter A More Secure and Efficacious TTS Signature Scheme, pages 320–338. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[39] Jiun-Ming Chen Bo-Yin Yang. Tts: Rank attacks in tame-like multivariate pkcs. https://eprint.iacr.org/2004/061.pdf, last accessed 2016-05-06.

[40] Jintai Ding, Dieter Schmidt, and Zhijun Yin. Cryptanalysis of the new tts scheme in ches 2004. *International Journal of Information Security*, 5(4):231–240, 2006.

[41] Jintai Ding and Dieter Schmidt. *Applied Cryptography and Network Security: Third International Conference, ACNS 2005, New York, NY, USA, June 7-10, 2005. Proceedings*, chapter Rainbow, a New Multivariable Polynomial Signature Scheme, pages 164–175. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[42] STMicroelectronics. Stm3241g-eval. http://www2.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-eval-tools/stm32-mcu-eval-tools/stm32-mcu-eval-boards/stm3241g-eval.html, last accessed 2016-05-04.

[43] STMicroelectronics. Stm32f417ig. http://www2.st.com/content/st_com/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32f4-series/stm32f407-417/stm32f417ig.html, last accessed 2016-05-04.

[44] Raspberry Pi Foundation. Raspberry pi 2 model b. https://www.raspberrypi.org/products/raspberry-pi-2-model-b/, last accessed 2016-05-04.

[45] Raspberry Pi Foundation. Raspbian. https://www.raspberrypi.org/downloads/raspbian/, last accessed 2016-05-04.

[46] The network analyzer wireshark. Wireshark. https://www.wireshark.org/, last accessed 2016-05-09.

[47] Jiang Zhang, Zhenfeng Zhang, Jintai Ding, Michael Snook, and Özgür Dagdelen. *Advances in Cryptology - EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, chapter Authenticated Key Exchange from Ideal Lattices, pages 719–751. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[48] The Valgrind Developers. Valgrind. http://valgrind.org/, last accessed 2016-05-20.

[49] Wikipedia. Dhrystone. https://en.wikipedia.org/wiki/Dhrystone, last accessed 2016-06-07.

[50] Erdem Alkim, Leo Ducas, Thomas Poppelmann, and Peter Schwabe. Post-quantum key exchange - a new hope. Cryptology ePrint Archive, Report 2015/1092, 2015. http://eprint.iacr.org/.

Appendix A

Appendix

Listing A.1: Pseudo code describing the steps for encryption in the AES/Rijndael algorithm. Nr denotes the number of rounds and the expanded_key is derived from the cipher key.

```
rijndael(state, cipher_key)
{
        key_expansion(cipher_key, expanded_key);
        add_round_key(state, expanded_key[0]);
        for(i = 1; i < Nr ; i++)
                round(state, expanded_key[i]);
        final_round(state, expanded_key[Nr]);
}

round(state, expanded_key[i])
{
        sub_bytes(state);
        shift_rows(state);
        mix_columns(state);
        add_round_key(state, expanded_key[i]);
}

final_round(state, expanded_key[Nr])
{
        sub_bytes(state);
        shift_rows(state);
        add_round_key(state, expanded_key[Nr]);
}
```

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2016-518

http://www.eit.lth.se