ASSIGNMENT OF MASTER’S THESIS

Title: Multivariate cryptography
Student: Bc. Jan Rahm
Supervisor: Ing. Jiří Buček, Ph.D.
Study Programme: Informatics
Study Branch: Computer Security
Department: Department of Information Security
Validity: Until the end of summer semester 2020/21

Instructions

Study the topic of multivariate cryptography as one of the approaches to post-quantum cryptography.
Select a specific algorithm based on multivariate cryptography such as Unbalanced Oil and Vinegar (UOV).
Create an educational implementation of the selected algorithm in Wolfram Mathematica.
Examine the reference implementation of the selected algorithm. Evaluate its time and memory complexity on a PC. Implement the algorithm on a chosen microcontroller such as ARM or ESP32 and evaluate its usability in an embedded environment.
Compare the time and memory complexity of the selected algorithm with a conventional algorithm such as RSA or ECDSA.

References

Will be provided by the supervisor.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 5, 2020

Master's thesis

Multivariate cryptography

Bc. Jan Rahm

Department of Information Security
Supervisor: Ing. Jiří Buček, Ph.D.

May 27, 2020

Acknowledgements

I would like to thank Ing. Jiří Buček, Ph.D. for the willingness, consultation and valuable advice he gave me. Also, I would like to thank my family for their support during my studies and in writing of this work.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 27, 2020 ....

Czech Technical University in Prague
Faculty of Information Technology
© 2020 Jan Rahm. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Rahm, Jan. *Multivariate cryptography.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020. Also available from: ⟨`https://github.com/rahmjan/Masters_Thesis`⟩.

Abstrakt

Diplomová práce se zabývá vybranými algoritmy multivariační kryptografie, zejména Unbalanced Oil & Vinegar a Rainbow. Cílem práce je implementace algoritmů ve Wolfram Mathematica, prozkoumání již existujících řešeních a jejich implementace na mikrokontroleru ESP32. Algoritmy jsou otestovány a změřeny vůči algoritmům RSA a ECDSA.

Klíčová slova Multivariační kryptografie, Unbalanced Oil & Vinegar, Rainbow, Wolfram Mathematica, ESP32

Abstract

This Master's thesis deals with the selected algorithms of multivariate cryptography, especially Unbalanced Oil & Vinegar and Rainbow. The aim of this work is the implementation of the algorithms in Wolfram Mathematica, the investigation of existing solutions and their implementation on the ESP32 microcontroller. The algorithms are tested and measured against the RSA and ECDSA algorithms.

Keywords Multivariate cryptography, Unbalanced Oil & Vinegar, Rainbow, Wolfram Mathematica, ESP32

Contents

Introduction **1**

1 Basic terms and definitions **3**
1.1 Basic terms 3
1.1.1 Polynomial 3
1.1.2 Degree of a polynomial 3
1.1.3 Quantum computer 3
1.1.4 Post-quantum cryptography 3
1.1.5 Finite field 4
1.1.6 Translation 4
1.1.7 Linear map 4
1.1.8 Affine map 4
1.1.9 Wolfram Mathematica 4
1.1.10 Internet of things 4
1.1.11 Valgrind 5
1.1.12 PRNG 5
1.1.13 ESP32 5
1.1.14 RSA 5
1.1.15 ECDSA 6
1.2 Multivariate cryptography 7
1.2.1 MQ problem 7
1.2.2 Public key 7
1.2.3 Encryption 8
1.2.4 Signature 8
1.3 Unbalanced Oil & Vinegar 9
1.3.1 Definition 9
1.3.2 Security 9
1.4 Rainbow 10
1.4.1 Definition 10

2 Implementation **11**
2.1 Wolfram Mathematica . . . 11
2.1.1 Unbalanced Oil & Vinegar . . . 11
2.1.1.1 Generation of instance . . . 14
2.1.2 Rainbow . . . 14
2.1.2.1 Generation of instance . . . 17
2.2 Reference implementation . . . 18
2.2.1 Unbalanced Oil & Vinegar . . . 18
2.2.1.1 Adjustments . . . 19
2.2.2 Rainbow . . . 19
2.2.2.1 Adjustments . . . 20
2.2.3 Test file . . . 20
2.3 ESP32 implementation . . . 22
2.3.1 Setup of environment . . . 23
2.3.1.1 Build & Load . . . 24
2.3.1.2 Memory . . . 24
2.3.2 Project description . . . 25
2.3.3 Lifted Unbalanced Oil & Vinegar . . . 25
2.3.3.1 Optimization . . . 26
2.3.3.2 Memory . . . 26
2.3.4 Rainbow . . . 27
2.3.4.1 Optimization . . . 27
2.3.4.2 Memory . . . 28
2.4 Conventional algorithms . . . 29
2.4.1 RSA . . . 29
2.4.2 ECDSA . . . 30

3 Testing and discussion **31**
3.1 PC . . . 31
3.1.1 Signature variants . . . 32
3.1.2 Time complexity . . . 33
3.1.3 Memory complexity . . . 35
3.1.4 Conclusion note . . . 37
3.2 ESP32 . . . 38
3.2.1 Signature variants . . . 38
3.2.2 Time complexity . . . 39
3.2.3 Memory complexity . . . 41
3.2.4 Keys & signature . . . 45
3.2.5 Conclusion note . . . 48
3.3 Conventional algorithms . . . 49

Conclusion **53**

Bibliography **55**

A Acronyms **57**

B Tables of measured values **59**

C Contents of enclosed CD **63**

List of Figures

1.1 Workflow of multivariate public key cryptosystems 8

2.1 ESP32-LyraT 22

3.1 Comparison of LUOV on PC 33
3.2 Comparison of RB on PC 34
3.3 Comparison of PC implementations 34
3.4 Memory requirement of LUOV on PC 35
3.5 Memory requirement of RB on PC 36
3.6 Comparison of PC implementations 36
3.7 Comparison of LUOV on ESP32 39
3.8 Comparison of RB on ESP32 40
3.9 Comparison of ESP32 implementations 41
3.10 Memory requirement of LUOV on ESP32 42
3.11 Memory requirement of LUOV on ESP32 42
3.12 Memory requirement of RB on ESP32 43
3.13 Memory requirement of implementations on ESP32 44
3.14 Memory requirement of my_ESP32_malloc 44
3.15 Size of signature of LUOV on ESP32 45
3.16 Size of signature of RB on ESP32 46
3.17 Comparison of LUOV with short public key and RB 47
3.18 Comparison of LUOV with short signature and RB 47
3.19 Comparison with conventional algorithms 49
3.20 Time requirement comparison with conventional algorithms by categories 50
3.21 Memory requirement comparison with conventional algorithms . . 51
3.22 Memory requirement comparison with conventional algorithms by categories 51

List of Tables

3.1 NIST security categories 32
3.2 NIST security categories of conventional algorithms 49

B.1 Time measurement in seconds on PC 59
B.2 Memory measurement in bytes on PC 60
B.3 Time measurement in seconds on ESP32 60
B.4 Memory measurement in bytes on ESP32 61
B.5 Memory measurement of *my_ESP_malloc* in bytes on ESP32 61
B.6 Size of keys and signature in bytes 62

Introduction

Cryptography is one of the most needed parts of modern informatics, because almost everyone has something they wish to stay private. But today we can see the uprise of quantum computers capable of deciphering conventional algorithms for cryptology. That is why a new category of post-quantum cryptography was created, one of its representatives being multivariate cryptography.

The objective of this work is to describe principles of multivariate cryptography for educational purposes with the creation of a simple example in Wolfram Mathematica. The focus is on Unbalanced Oil & Vinegar and Rainbow algorithms with examination of reference implementation and their possible implementation on ESP32 and possible use in IoT.

The final part deals with the comparison with the chosen conventional algorithms, namely RSA and ECDSA.

Chapter 1

Basic terms and definitions

This chapter describes the concepts and algorithms used in this thesis.

1.1 Basic terms

1.1.1 Polynomial

Polynomial p is a function of the form:

$$p(x) = \sum_{i=0}^{n} \alpha_i x^i = \alpha_0 + \alpha_1 x + \alpha_2 x^2 + ... + \alpha_n x^n,$$

where $n \in N_0$ and $\alpha_0, \alpha_1, ..., \alpha_n \in R$. Values $\alpha_0, \alpha_1, ..., \alpha_n$ are called polynomial coefficients of p.

1.1.2 Degree of a polynomial

The degree of a polynomial is the highest index $i \in N_0$ with the coefficient $\alpha_i \neq 0$. If all coefficients are equal to zero, then the degree of the polynomial is -1.

1.1.3 Quantum computer

A quantum computer is a device for performing computations, which directly uses the phenomena known from quantum mechanics as superposition or interference. In a typical computer, data are represented by bits, where each bit is either zero or one, while in a quantum computer, qubits (quantum bits) are used, which can either be zero, one, or a combination of both.

1.1.4 Post-quantum cryptography

Post-quantum cryptography refers to algorithms thought to be secure against an attack by a quantum computer.

But today this is not the case for the most used cryptographic algorithms, which are based on mathematical problems of integer factorization, discrete logarithm or elliptic-curve discrete logarithm. These problems can be solved by Shor's algorithm on a quantum computer in polynomial time.

1.1.5 Finite field

A finite field is a field with a finite set of elements. This means that multiplication, addition, subtraction and division (excluding division by zero) are defined and satisfy the rules of arithmetic known as the field axioms.

The simplest examples of finite fields are the fields of prime order: $\mathbb{F}_p$ may be constructed as the integers modulo p.

1.1.6 Translation

In Euclidean geometry, a translation is a geometric transformation that moves every point of a figure or a space by the same distance in a given direction.

1.1.7 Linear map

In mathematics, a linear map is a mapping $V \to W$ between two modules (for example, two vector spaces) that preserves the operations of addition and scalar multiplication.

1.1.8 Affine map

An affine map is the composition of two functions: a translation and a linear map. Ordinary vector algebra uses matrix multiplication to represent linear maps, and vector addition to represent translations. Formally, in the finite-dimensional case, if the linear map is represented as a multiplication by a matrix A and the translation as the addition of a vector $\vec{b}$, an affine map f acting on a vector $\vec{x}$ can be represented as:

$$\vec{y} = f(\vec{x}) = A\vec{x} + \vec{b}$$

1.1.9 Wolfram Mathematica

Wolfram Mathematica is a computer program widely used in scientific, technical and mathematical circles. The program was created by Stephen Wolfram and further developed by a team of mathematicians and programmers. It is sold by Wolfram Research headquartered in Champaign, Illinois.

1.1.10 Internet of things

Internet of Things (IoT) is a term in computer science for a network of physical devices, vehicles, household appliances, or other devices that are equipped

with electronics, software, sensors, moving parts, or network connectivity that allows these devices to connect and exchange data.

1.1.11 Valgrind

Valgrind is a computer program for Unix systems that helps in debugging and profiling programs. For example, it can be used to search for memory leaks, concurrencies, or to monitor cache usage. Valgrind is an open-source software distributed under the GPL license.

1.1.12 PRNG

The pseudo-random number generator (PRNG) is a deterministic program that generates a sequence of numbers. When using statistical tests, the sequence should be indistinguishable from a random one as much as possible.

1.1.13 ESP32

The ESP32 is a low-cost, low-power microcontroller with integrated Wi-Fi and Bluetooth. It is created and developed by Espressif Systems, a Chinese company based in Shanghai, and is manufactured by TSMC using a 40 nm process.

Its basic technical parameters are:

- CPU: Xtensa dual-core
- Memory: 520 KiB SRAM
- Wi-Fi: 802.11 b/g/n
- Bluetooth: v4.2 BR/EDR and BLE
- IEEE 802.11 standard
- Hardware acceleration: AES, SHA-2, RSA, ECC, RNG

1.1.14 RSA

RSA is an algorithm for cryptographic and signature schemes. Its name comes from the initials of its authors Rivest, Shamir and Adleman.

RSA security is based on the problem of factorization, an assumption that breaking a large composite number into a product of its prime numbers is a difficult task. From the number $n = pq$ it should be almost impossible to determine the factors p and q in a reasonable time, because there is no known

factorization algorithm working in polynomial time to the size of the binary notation of n. On the other hand, multiplying two large numbers is very easy.

The details of RSA implementation and inner workings are not subjects of this Master's thesis and will not be mentioned.

1.1.15 ECDSA

Elliptic curve digital signature protocol (ECDSA) is a variant of the DSA protocol that uses elliptic curves for digital signatures.

The security of the elliptic curve is based on the problem of discrete logarithm comprising an assumption that finding k of $Y \equiv g^k (mod\, p)$ is a difficult task. On the other hand, it is quite simple to compute Y.

The details of ECDSA implementation and inner workings are not subjects of this Master's thesis and will not be mentioned.

1.2 Multivariate cryptography

"Multivariate cryptography (MC) is the generic term for asymmetric cryptographic primitives based on multivariate polynomials over a finite field $\mathbb{F}$."[7]

This means it is a system of nonlinear polynomial equations with coefficients over a finite field: $\mathbb{F} = \mathbb{F}_q$ with q elements:

$$p^{(1)}(x_1, \ldots, x_n) = \sum_{i=1}^{n}\sum_{j=1}^{n} p_{ij}^{(1)} \cdot x_i x_j + \sum_{i=1}^{n} p_i^{(1)} \cdot x_i + p_0^{(1)}$$

$$p^{(2)}(x_1, \ldots, x_n) = \sum_{i=1}^{n}\sum_{j=1}^{n} p_{ij}^{(2)} \cdot x_i x_j + \sum_{i=1}^{n} p_i^{(2)} \cdot x_i + p_0^{(2)}$$

$$\vdots$$

$$p^{(m)}(x_1, \ldots, x_n) = \sum_{i=1}^{n}\sum_{j=1}^{n} p_{ij}^{(m)} \cdot x_i x_j + \sum_{i=1}^{n} p_i^{(m)} \cdot x_i + p_0^{(m)}$$

If the polynomials have a degree of two, they are called multivariate quadratics (MQ). Solving systems of multivariate polynomial equations is proven to be NP-hard, so-called MQ problem. This is the reason why MC is often considered to be a good candidate for post-quantum cryptography.

MC is very fast and only requires moderate computational resources, which makes it attractive for applications in low-cost devices.

1.2.1 MQ problem

Given m quadratic polynomials $p^{(1)}(x), \ldots, p^{(m)}(x)$ in the n variables $x_1, \ldots, x_n$, find a vector $\bar{x} = (\bar{x}_1, \ldots, \bar{x}_n)$ such that $p^{(1)}(\bar{x}) = \ldots = p^{(m)}(\bar{x}) = 0$.

1.2.2 Public key

The public key of MC is the system of MC polynomials. To build this system based on the MQ problem, it needs an easily invertible quadratic map $\mathcal{F} : \mathbb{F}^n \to \mathbb{F}^m$, the so-called *central map*. Because it is easily invertible, it needs to be hidden in the public key by invertible affine maps: $\mathcal{S} : \mathbb{F}^m \to \mathbb{F}^m$ and $\mathcal{T} : \mathbb{F}^n \to \mathbb{F}^n$.

The public key of this system is a composed map:

$$\mathcal{P} = \mathcal{S} \circ \mathcal{F} \circ \mathcal{T} : \mathbb{F}^n \to \mathbb{F}^m$$

and the private key consists of three maps $\mathcal{S}$, $\mathcal{F}$ and $\mathcal{T}$, also known as a *trapdoor*.

The public key should be hard to invert without the knowledge of the *trapdoor*.

$$\begin{array}{ccc} z \in \mathbb{F}^n & \xrightarrow{\mathcal{P}} & w \in \mathbb{F}^m \\ \mathcal{T}\downarrow & & \mathcal{S}\uparrow \\ y \in \mathbb{F}^n & \xrightarrow{\mathcal{F}} & x \in \mathbb{F}^m \end{array}$$

Figure 1.1: Workflow of multivariate public key cryptosystems

1.2.3 Encryption

To get a ciphertext w, a message $z \in \mathbb{F}^n$ can be easily encrypted by evaluation of the public key $\mathcal{P}$:

$$w = \mathcal{P}(z) \in \mathbb{F}^m$$

For the decryption of the ciphertext, it needs to be evaluated by the private key in three steps:

$$x = \mathcal{S}^{-1}(w) \in \mathbb{F}^m,\ y = \mathcal{F}^{-1}(x) \in \mathbb{F}^n,\ z = \mathcal{T}^{-1}(y) \in \mathbb{F}^n$$

There is a required condition $m \geq n$, this way the public key $\mathcal{P}$ will be injective and the decryption will output a unique plaintext.

1.2.4 Signature

To generate a signature for a message m, it needs to use a hash function:

$$\mathcal{H} : \{0,1\}^* \to \mathbb{F}^m$$

to compute the hash value:

$$w = \mathcal{H}(m) \in \mathbb{F}^m$$

After this step, it can be evaluated by:

$$x = \mathcal{S}^{-1}(w) \in \mathbb{F}^m,\ y = \mathcal{F}^{-1}(x) \in \mathbb{F}^n,\ z = \mathcal{T}^{-1}(y) \in \mathbb{F}^n$$

where z is the signature of the message m. As can be seen, it is similar to the decryption of a ciphertext.

The verification of the signature z is done by computing the hash value:

$$w = \mathcal{H}(m) \in \mathbb{F}^m$$

and by evaluation of the public key $\mathcal{P}$:

$$w' = \mathcal{P}(z) \in \mathbb{F}^m$$

If $w' = w$ is true, the signature is valid, otherwise not.

There is also a condition that $m \leq n$, this way the public key $\mathcal{P}$ will be surjective and the private key can sign any message.

1.3 Unbalanced Oil & Vinegar

The Unbalanced Oil & Vinegar's (UOV) name comes from the fact that the variables of the polynomials are grouped into two groups: the vinegar and the oil. These two groups are mixed in the polynomials and the unbalanced attribute refers to the ratio of the variables because there is always more vinegar than oil variables. The signature scheme was proposed by Kipnis and Patarin in 1999.

The UOV scheme is very simple, has small signatures and is fast. The main disadvantage being its public keys, which are quite large.

1.3.1 Definition

Let $\mathbb{F}$ be a finite field, $v, o \in \mathbb{N}$ and $n = v + o$, $V = \{1, \dots, v\}$, $O = \{v + 1, \dots, n\}$. The variables $x_1, \dots, x_v$ are Vinegar variables and $x_{v+1}, \dots, x_n$ are Oil variables. If $v = o$, the scheme is called balanced Oil & Vinegar (OV), for $v > o$ it is UOV.

The *central map* $\mathcal{F} : \mathbb{F}^n \to \mathbb{F}^o$ consists of o quadratic polynomials $f^{(1)}, \dots, f^{(o)}$:

$$f^{(k)} = \sum_{i,j \in V} \alpha_{ij}^{(k)} \cdot x_i x_j + \sum_{i \in V, j \in O} \beta_{ij}^{(k)} \cdot x_i x_j + \sum_{i \in V \cup O} \gamma_i^{(k)} \cdot x_i + \delta^{(k)}$$

where $\alpha_{ij}^{(k)}$, $\beta_{ij}^{(k)}$, $\gamma_i^{(k)}$, $\delta^{(k)} \in \mathbb{F}$ and $1 \leq k \leq o$.

To hide $\mathcal{F}$ in the public key, it is combined with one invertible affine map $\mathcal{T} : \mathbb{F}^n \to \mathbb{F}^n$. The public key of the scheme is in the following form:

$$\mathcal{P} = \mathcal{F} \circ \mathcal{T} : \mathbb{F}^n \to \mathbb{F}^o$$

and the private key consists of $\mathcal{F}$ and $\mathcal{T}$. The second affine map $\mathcal{S}$ is not needed for the security of UOV.

Note that in the $\mathcal{F}$ only the vinegar variables could have quadratic form and the coefficients of the polynomials could be randomly selected.

1.3.2 Security

For the UOV to be secure, it is required that $v \geq 2o$ because of the attack of Kipnis and Shamir on balanced OV.[8] Besides that, the UOV scheme resisted (for suitable parameter sets) cryptanalysis for over 20 years. Now it is one of the oldest and best-studied cryptosystems and is therefore believed to be highly secure.

1.4 Rainbow

The Rainbow (RB) is a multi-layer version of UOV. The layers are not independent from each other but there is a hierarchy which uses the results from the layer above to compute additional variables. The name comes from the link to the layers of a rainbow and the scheme was introduced by Ding and Schmid in 2005.

The main advantage when compared to UOV should be better performance, smaller key sizes and smaller signatures.

1.4.1 Definition

Let $\mathbb{F}$ be a finite field, $0 < v_1 < v_2 < \ldots < v_{u+1} = n$ be a sequence of integers and $V_i = \{1, \ldots, v_i\}$, $O_i = \{v_i + 1, \ldots, v_{i+1}\}$ and $o_i = v_{i+1} - v_i$ $(i = 1, \ldots, u)$ where o_i is a number of oil variables and v_i is a number of vinegar variables in the i layer, u is a number of UOV layers. The variables $x_1, \ldots, x_{v_i}$ are Vinegar variables and $x_{v_i+1}, \ldots, x_{v_{i+1}}$ are Oil variables.

The *central map* $\mathcal{F} : \mathbb{F}^n \to \mathbb{F}^m$ consists of $m = n - v_1$ quadratic polynomials $f^{(v_1+1)}, \ldots, f^{(n)}$:

$$f^{(k)} = \sum_{i,j \in V_l} \alpha_{ij}^{(k)} \cdot x_i x_j + \sum_{i \in V_l, j \in O_l} \beta_{ij}^{(k)} \cdot x_i x_j + \sum_{i \in V_l \cup O_l} \gamma_i^{(k)} \cdot x_i + \delta^{(k)}$$

where $l \in \{1, \ldots, u\}$ is the only integer such that $k \in O_l$ and $\alpha_{ij}^{(k)}$, $\beta_{ij}^{(k)}$, $\gamma_i^{(k)}$, $\delta^{(k)} \in \mathbb{F}$.

To hide $\mathcal{F}$ in the public key, it is combined with two invertible affine maps $\mathcal{T} : \mathbb{F}^n \to \mathbb{F}^n$ and $\mathcal{S} : \mathbb{F}^m \to \mathbb{F}^m$. The public key of the scheme is in the following form:

$$\mathcal{P} = \mathcal{S} \circ \mathcal{F} \circ \mathcal{T} : \mathbb{F}^n \to \mathbb{F}^m$$

and the private key consists of $\mathcal{S}$, $\mathcal{F}$ and $\mathcal{T}$.

Note that in the $\mathcal{F}$ only the vinegar variables could have quadratic form and the coefficients of the polynomials could be randomly selected.

Chapter 2

Implementation

This chapter describes the implementation of the algorithms on the selected platforms, namely Wolfram Mathematica, PC and microcontroller ESP32. For the last two specified, the implementation is in C++.

2.1 Wolfram Mathematica

This section describes examples in Wolfram Mathematica and step by step description of the algorithms. All of the *hard-coded* values (numbers) are randomly selected, they only have to respect the definition and parameters of the algorithm.

2.1.1 Unbalanced Oil & Vinegar

Here is a description of the signature scheme of UOV. This example of UOV is, in fact, the example of balanced OV but there is no difference to UOV.

First set up the parameters of the example: Let $\mathbb{F} = GF(7)$, $o = v = 3$. The central map $\mathcal{F} = (f^{(1)}, f^{(2)}, f^{(3)})$ is given by:

In[3]:=

```
mod=7;
F1[x1_,x2_,x3_,x4_,x5_,x6_]:=
4x1^2+4x1*x3+5x1*x4+6x1*x5+x1*x6+6x1+4x2^2+x2*x3+6x2*x4
+6x2*x5+5x2*x6+5x2+5x3^2+3x3*x4+5x3*x5+2x3*x6+5x3+6x4+3x5;
F2[x1_,x2_,x3_,x4_,x5_,x6_]:=
3x1*x3+4x1*x4+3x1*x5+4x1*x6+3x1+6x2^2+x2*x3+4x2*x4+4x2*x5
+5x2*x6+6x2+6x3^2+4x3*x4+2x3*x5+x3*x6+3x3+x4+x6+1;
F3[x1_,x2_,x3_,x4_,x5_,x6_]:=
6x1^2+6x1*x3+4x1*x5+2x1*x6+2x2^2+5x2*x3+6x2*x4+5x2*x5+
5x2*x6+6x2+3x3^2+5x3*x4+6x3*x5+x3*x6+3x3+4x4+6x5+5;
```

It will set up the value of *mod* to 7 and initialize functions of the central map. Next comes setting up the affine map $\mathcal{T}$ with the matrix A and the vector b. These two parts will later be used separately in the example.

In[7]:=

```
A=(6 5 5 5 5 4
6 6 4 5 0 6
2 5 2 1 5 0
1 1 6 2 2 3
3 6 2 2 3 0
0 5 4 6 1 5);
```

In[8]:=

```
b=(1
2
4
1
3
2);
```

In[9]:=

```
T=A.(x1
x2
x3
x4
x5
x6)+b;
```

The following block computes public key $\mathcal{P}$ by putting values of $\mathcal{T}$ inside of $\mathcal{F}$, it also simplifies the expression of $p1, p2, p3$ and finally applies modulo on the whole polynomial:

In[10]:=

```
p1 = F1 @@ T[[All]];
p2 = F2 @@ T[[All]];
p3 = F3 @@ T[[All]];
P1[x1_,x2_,x3_,x4_,x5_,x6_] = PolynomialMod[Simplify[p1],mod]
P2[x1_,x2_,x3_,x4_,x5_,x6_] = PolynomialMod[Simplify[p2],mod]
P3[x1_,x2_,x3_,x4_,x5_,x6_] = PolynomialMod[Simplify[p3],mod]
```

The results of $\mathcal{P} = \mathcal{F} \circ \mathcal{T}$ are:

Out[13]=

$$\{6+x1+5x2+4x1x2+2x2^2+3x3+x1x3+x2x3+x3^2+6x4+2x1x4+5x2x4+2x3x4+3x4^2+5x5+3x1x5+6x2x5+4x3x5+3x4x5+4x5^2+4x6+x2x6+3x3x6+2x4x6\}$$

Out[14]=

$$\{5+6x1^2+5x2+4x1x2+5x2^2+4x3+5x1x3+3x2x3+2x3^2+2x4+2x1x4+4x2x4+2x3x4+5x4^2+3x5+5x1x5+5x2x5+2x3x5+6x5^2+5x2x6+6x4x6+2x5x6+6x6^2\}$$

Out[15]=

$$\{5+5x1+4x1^2+5x2+3x1x2+5x2^2+2x3+2x1x3+x2x3+2x3^2+6x4+3x2x4+2x4^2+x5+3x1x5+6x2x5+4x3x5+2x5^2+2x6+x1x6+3x2x6+4x3x6+6x4x6+5x5x6+4x6^2\}$$

From this place on, it will only focus on the computation of the signature z for the hash w. Be aware that hash function and the message m are not used in this example because they are not needed for the example's purpose.

In[16]:=

```
w = {{3},{6},{4}};
y1 = 1;
y2 = 0;
y3 = 6;
```

It sets the hash to value $w = (3,6,4)$ and also sets the values for $y = (y1, y2, y3)$. These values for y are chosen randomly.

In[20]:=

```
f1 = PolynomialMod[F1[y1,y2,y3,y4,y5,y6],mod]
f2 = PolynomialMod[F2[y1,y2,y3,y4,y5,y6],mod]
f3 = PolynomialMod[F3[y1,y2,y3,y4,y5,y6],mod]
```

Here is the $\mathcal{F}$ after substitution, minimalizing and use of modulo:

Out[20]=

```
f1 = 6+y4+4y5+6y6
f2 = 4+y4+y5+4y6
f3 = 5+6y4+4y5+y6
```

In this result is shown the loss of quadratic variables (vinegar) by the substitution which will give linear equations.

Next two steps solve the linear system of $f^{(1)} = w_1 = 3, f^{(2)} = w_2 = 6, f^{(3)} = w_3 = 4$. Gaussian elimination can be also used for the solution:

In[21]:=

```
res=Solve[{f1==w[[1]],f2==w[[2]],f3==w[[3]]},Modulus→mod];
```

In[22]:=

```
y = {y1,y2,y3,y4,y5,y6} /. res
```

Out[22]=

```
{{1,0,6,6,3,0}}
```

It will obtain results for $(y4, y5, y6) = (6,3,0)$. After combination it is $y = (1,0,6,6,3,0)$, the so-called *pre-image* of w: $y = \mathcal{F}^{-1}(w)$. If the solution for the linear system does not exist, choose different values for $(y1, y2, y3)$ and repeat steps before.

Finally use $\mathcal{T}^{-1}$ to compute the signature z. This requires the inversion of the matrix A.

In[23]:=

```
A-1 = Inverse[A, Modulus→mod]
```

$$A_{-1} = \begin{pmatrix} 2 & 4 & 6 & 2 & 0 & 5 \\ 1 & 3 & 3 & 1 & 6 & 2 \\ 4 & 6 & 6 & 4 & 5 & 4 \\ 2 & 0 & 3 & 4 & 2 & 3 \\ 6 & 0 & 3 & 0 & 0 & 5 \\ 2 & 2 & 2 & 5 & 3 & 3 \end{pmatrix}$$

In[24]:=
```
z = Mod[A-1.(Transpose[y]-b),mod]
```

Out[24]=
```
{{4},{1},{5},{6},{3},{5}}
```

The value of the signature $z = (4, 1, 5, 6, 3, 5)$.

The last part is a check if the two hashes w and $w2$ are the same.

In[25]:=
```
w2=w;
w2={P1 @@ z[[All,1]],P2 @@ z[[All,1]],P3 @@ z[[All,1]]};
(* True? *)
Mod[w2,mod]==w
```

Out[27]=
```
True
```

The file with implementation can be found under the name *UOV.nb*.

2.1.1.1 Generation of instance

I also created a version with a generation of a random instance of UOV for selected parameters. It can be found in the *UOV-gen.nb* file. The parameters are the number of vinegar and oil variables and the value of modulus.

2.1.2 Rainbow

The description of the signature scheme of Rainbow is very similar to the description of UOV.

First set up the parameters of the example: Let $\mathbb{F} = GF(7)$, $v1 = o1 = o2 = 2$. The central map $\mathcal{F} = (f^{(3)}, f^{(4)}, f^{(5)}, f^{(6)})$ is given by:

In[30]:=
```
mod=7;
F3[x1_,x2_,x3_,x4_,x5_,x6_]:=
x1^2+3x1*x2+5x1*x3+6x1*x4+2x2^2+6x2*x3+4x2*x4+2x2+6x3+2x4+5;
F4[x1_,x2_,x3_,x4_,x5_,x6_]:=
2x1^2+x1*x2+x1*x3+3x1*x4+4x1+x2^2+x2*x3+4x2*x4+6x2+x4;
F5[x1_,x2_,x3_,x4_,x5_,x6_]:=
2x1^2+3x1*x2+3x1*x3+3x1*x4+x1*x5+3x1*x6+6x1+4x2^2+x2*x3+
4x2*x4+x2*x5+3x2*x6+3x2+3x3*x4+x3*x5+2x3*x6+2x3+3x4*x5+x5+6x6;
F6[x1_,x2_,x3_,x4_,x5_,x6_]:=
2x1^2+5x1*x2+x1*x3+5x1*x4+5x1*x6+6x1+5x2^2+3x2*x3+5x2*x5+4x2*x6+
x2+3x3^2+5x3*x4+4x3*x5+2x3*x6+4x3+x4^2+6x4*x5+3x4*x6+4x4+4x5+x6+2;
```

Next comes setting up the affine map $\mathcal{T}$ with the matrix A and the vector b which are the same as in the OV example. But with the addition of the affine map $\mathcal{S}$ with the matrix $A2$ and the vector $b2$.

In[34]:=
```
A2=(6 5 5 5
6 6 4 5
2 5 2 1
1 1 6 2);
```
In[35]:=
```
b2=(1
2
4
1);
```
In[36]:=
```
S=A2.(x1
x2
x3
x4)+b2;
```

This block computes the public key $\mathcal{P} = \mathcal{S} \circ \mathcal{F} \circ \mathcal{T}$ by putting the values of $\mathcal{T}$ inside of $\mathcal{F}$ and then it creates functions from the matrix S, which are used for the final step of computation of $\mathcal{P}$. It also simplifies the expression of $pp3, pp4, pp5, pp6$ and finally applies the modulo on the whole polynomial:

In[37]:=
```
p3 = F3 @@ T[[All]];
p4 = F4 @@ T[[All]];
p5 = F5 @@ T[[All]];
p6 = F6 @@ T[[All]];
S3[x1_,x2_,x3_,x4_] = S[[1]];
S4[x1_,x2_,x3_,x4_] = S[[2]];
S5[x1_,x2_,x3_,x4_] = S[[3]];
S6[x1_,x2_,x3_,x4_] = S[[4]];
pp3 = S3[p3,p4,p5,p6][[1]];
pp4 = S4[p3,p4,p5,p6][[1]];
pp5 = S5[p3,p4,p5,p6][[1]];
pp6 = S6[p3,p4,p5,p6][[1]];
P3[x1_,x2_,x3_,x4_,x5_,x6_]=PolynomialMod[Simplify[pp3],mod];
P4[x1_,x2_,x3_,x4_,x5_,x6_]=PolynomialMod[Simplify[pp4],mod];
P5[x1_,x2_,x3_,x4_,x5_,x6_]=PolynomialMod[Simplify[pp5],mod];
P6[x1_,x2_,x3_,x4_,x5_,x6_]=PolynomialMod[Simplify[pp6],mod];
```

Computation of x from the hash w: $x = \mathcal{T}^{-1}(w)$:

In[38]:=
```
w = {{2},{2},{3},{4}};
A2-1 = Inverse[A2, Modulus→mod]
x = Mod[A2-1.(w - b2),mod]
```

Out[38]=
```
{{6},{0},{1},{6}}
```

The result is $x = (6, 0, 1, 6)$.

Next comes computation of the *pre-image* for x and also the place where the biggest difference from the UOV scheme is (the rainbow layers). Let's

start with the first step where is setup of random values for $y1, y2$ and their substitution in the polynomials is performed:

In[39]:=
```
y1 = 0;
y2 = 1;
f3 = PolynomialMod[F3[y1,y2,x3,x4,x5,x6],mod];
f4 = PolynomialMod[F4[y1,y2,x3,x4,x5,x6],mod];
f5 = PolynomialMod[F5[y1,y2,x3,x4,x5,x6],mod];
f6 = PolynomialMod[F6[y1,y2,x3,x4,x5,x6],mod];
```

Out[39]=
```
f3 = 2+5x3+6x4
f4 = x3+5x4
f5 = 3x3+4x4+3x3x4+2x5+x3x5+3x4x5+2x6+2x3x6
f6 = 1+3x3^2+4x4+5x3x4+x4^2+2x5+4x3x5+6x4x5+5x6+2x3x6+3x4x6
```

For the second step, it can be seen from the result that $f^{(3)}$ and $f^{(4)}$ are two linear equations (first rainbow layer) with two unknown values.

In[40]:=
```
res1 = Solve[{f3==x[[1]],f4==x[[2]]},Modulus → mod];
```

It solves and with these two (new vinegar) values $(x3, x4)$, it is possible to continue the substitution and to compute the final linear system (second rainbow layer):

In[41]:=
```
f5 = PolynomialMod[F5[y1,y2,x3,x4,x5,x6]/.res1,mod];
f6 = PolynomialMod[F6[y1,y2,x3,x4,x5,x6]/.res1,mod];
```

Out[41]=
```
f5 = 2+5x5+3x6
f6 = 3+2x5+5x6
```

In[42]:=
```
res2 = Solve[{f5==x[[3]],f6==x[[4]]},Modulus → mod];
```

The *pre-image* of x is $y = (0, 1, 4, 2, 0, 2)$: $y = \mathcal{F}^{-1}(x)$.

In[43]:=
```
y = {y1,y2,x3,x4,x5,x6}/.res1/.res2;
```

Out[43]=
```
{{0,1,4,2,0,2}}
```

For the final step of the computation of z, it needs to be applied $\mathcal{T}$: $z = \mathcal{T}^{-1}(y)$:

In[44]:=

```
A₋₁ = Inverse[A, Modulus→mod]
z = Mod[A₋₁.(y - b),mod]
```

Out[44]=

```
{{3},{0},{0},{3},{1},{6}}
```

The value of the signature $z = (3, 0, 0, 3, 1, 6)$.

Last part of the Mathematica sheet is a check if two hashes w and $w2$ are the same.

In[45]:=

```
w2=w;
w2={P3 @@ z[[All,1]],P4 @@ z[[All,1]],
P5 @@ z[[All,1]],P6 @@ z[[All,1]]};
(* True? *)
Mod[w2,mod]==w
```

Out[47]=

```
True
```

By the definition of RB, section 1.4.1, in this example is used $u = 2, v_1 = 2, v_2 = 4, v_3 = 6 = n$. Because $u = 2$, it is an example of RB with two layers. The file with implementation can be found under the name *RB.nb*.

2.1.2.1 Generation of instance

I also created a version with a generation of a random instance of RB for selected parameters. It can be found in the *RB-gen.nb* file. The parameters are the value of modulus, the number of vinegar variables and the list of numbers of oil variables where each of them represents one of the layers.

2.2 Reference implementation

This section describes the reference implementation of the algorithms which are taken from the second round of submissions from NIST Post-Quantum Cryptography Standardization Process.[9] These implementations are possible candidates for the new Cryptography standard, were announced on 30th January 2019, and are written in C++ language.

The reference implementations contain several optimized solutions. But for the purpose of this Master's thesis, the only relevant solution is under the *Reference_Implementation* folder because the others are not compatible with the ESP32 microcontroller.

2.2.1 Unbalanced Oil & Vinegar

This implementation of UOV is actually an implementation of LUOV (Lifted UOV) which is an improvement of the UOV scheme that greatly reduces the size of the public keys. There are three basic modifications:

- First modification changes the key generation algorithm. By using a seed and pseudo-random number generator, its output can correspond with the part of the public key. This way one can replace a large part of the public key with a short seed for PRNG.

  This algorithm still produces the same distribution of key pairs. This means the security of the scheme is not affected, assuming that the output of the PRNG is indistinguishable from true randomness. Implementation uses SHAKE128 or Chacha8.

  Private key is also generated from seed.

- Second modification (*"Lifting"*) is that the scheme uses $\mathcal{P} : \mathbb{F}_2^n \to \mathbb{F}_2^m$ over $\mathbb{F}_2$ as a public key over an extension field $\mathbb{F}_{2^r}$. That means the public key $\mathcal{P} : \mathbb{F}_2^n \to \mathbb{F}_2^m$ is *lifted*/extended to $\mathcal{P} : \mathbb{F}_{2^r}^n \to \mathbb{F}_{2^r}^m$. This modification is where the name comes from and is the most important because the public key remains small, while solving the system $\mathcal{P}(x) = y$ for some y in $\mathbb{F}_{2^r}^m$ becomes more difficult compared to instances when y is in $\mathbb{F}_2^m$.[11]

- Third modification is having a linear map $\mathcal{P}$ in the form:

  $$\begin{pmatrix} 1_v & T \\ 0 & 1_m \end{pmatrix}$$

  where T is a v-by-m matrix. This solution makes the key generation and the signing much faster[1] while not affect the security.[10]

Source codes of version 2.1 were obtained from GitHub repository.

2.2.1.1 Adjustments

For the testing purpose, I made a few adjustments to the LUOV source files. One was to simplify *Makefile* for easy generation of testing applications and setting up the *path* to the external library. Some unnecessary files were removed (*PQCgenKAT_sign.c*) and *README.md* with building information was created. The last change was the grouping of all the different LOUV settings into one folder with the building flags:

- FIELD_SIZE - Size/degree of finite field.
- OIL_VARS - Number of oil variables.
- VINEGAR_VARS - Number of vinegar variables.
- SHAKENUM - Version of the shake XOF used.
- FIRST_PART_TARGET - Number of bytes used in recovery mode.
- PRNG_CHACHA/PRNG_KECCAK - Whether Chacha8 or SHAKE128 is used.
- MESSAGE_RECOVERY - Enable message recovery.

For a successful build on Ubuntu operating system, the library *Keccak Code Package* needs to be in the *home* folder:

```
git clone https://github.com/XKCP/XKCP.git XKCP
cd XKCP
make generic64/libkeccak.a
```

Or use the one in *src/esp* and change the *Makefile* accordingly. Additionally the tool *xsltproc*:

```
sudo apt-get install xsltproc
```

Implementation can be found in the *src/pc/luov* folder.

2.2.2 Rainbow

This implementation of Rainbow is an implementation with two layers and contains three variants:

- Classic - Typical/Classic implementation of Rainbow.
- Cyclic - This variant is motivated by Petzoldt's cyclic Rainbow scheme[12] who developed the technique of inserting a matrix into a public key and to compute a corresponding private key. It allows the creation of major parts of the public key from a seed using a PRNG. However, this variant also includes a bug causing the verification of signature to fail.

- Compressed - This variant is similar to the Cyclic variant but the private key is stored in the form of a bit seed.

2.2.2.1 Adjustments

The adjustments are very similar to the LUOV adjustments. I simplified *Makefile* for easy generation of testing applications and removed redundant files. I added the *README.md* file with building information. This implementation can now be built with the following flags:

- _RAINBOW16_32_32_32
- _RAINBOW256_68_36_36
- _RAINBOW256_92_48_48

These flags can not be used together, only one can be valid at a time. It specifies which setting will be used: finite field $\mathbb{F}_{16}$ or $\mathbb{F}_{256}$ with 32/68/92 vinegar variables and two layers of 32/36/48 oil variables.

- _RAINBOW_CLASSIC
- _RAINBOW_CYCLIC
- _RAINBOW_CYCLIC_COMPRESSED

Same as the flags above, only one can be used at a time. It specifies which variant of Rainbow is to be used.

Last change was the addition of the *test.c* file from the LOUV implementation for consistent testing purposes.

Implementation can be found in the *src/pc/rb* folder.

2.2.3 Test file

For the testing purposes, there is the *test.c* file which also serves as the main entry point of the final application. Both of the implementations have the same structure which can be described as followed:

First step is the generation of the public and private keys:

```
crypto_sign_keypair(pk,sk);
```

Second step is the generation of the signature *sm* (the signature also contains the message) of the message *m*:

```
crypto_sign(sm,&smlen,m,message_size,sk);
```

Third step is the verification of the signature *sm* and it puts the message from it to the variable $m2$:

```
crypto_sign_open(m2,&smlen,sm smlen,pk);
```

The final step is the verification, if the message m is equal to the variable $m2$.

2.3 ESP32 implementation

For the implementation of the algorithms on a microcontroller, I selected the ESP32-LyraT microcontroller, version 4.3, made by Espressif Systems.

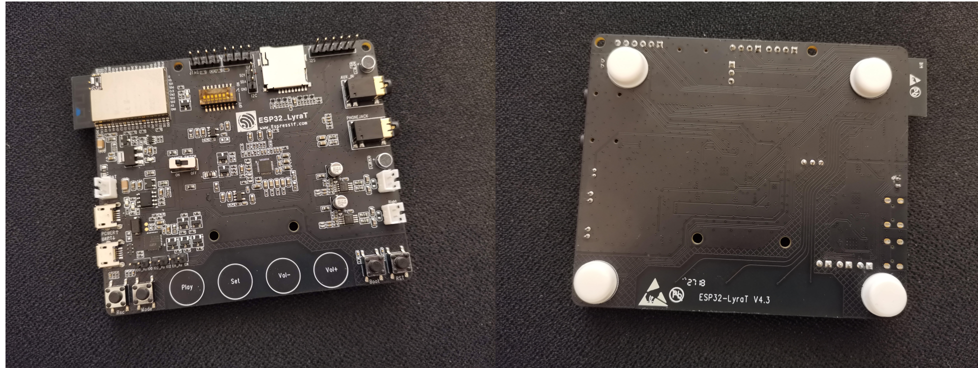

Figure 2.1: ESP32-LyraT

It is an audio development board built around ESP32 with additional hardware:

- ESP32-WROVER Module
- Audio Codec Chip
- Dual Microphones on board
- Headphone input
- 2x 3-watt Speaker output
- Dual Microphones on board
- Dual Auxiliary Input
- MicroSD Card slot
- Buttons
- JTAG
- Integrated USB-UART Bridge Chip
- Li-ion Battery-Charge Management

The main reason for selecting this platform is its inbuilt external memory with the capacity of 8MB. With this memory size, the implementation of the selected algorithms should be without any issues. However, only 4MB are able to be effectively used in the implementation (more information can

be found in section 2.3.1.2). Together with the excellent documentation at ”docs.espressif.com” it was relatively easy to choose this platform for development.

The implementations are the same as the reference implementations with my adjustments, I only ported them to the ESP32 platform.

2.3.1 Setup of environment

First step of development on ESP32 is to set up the environment. In order to set up the environment in the same way as I did (I used Ubuntu 19.4 operating system), one needs to follow these steps:

Download and install the required tools:

```
sudo apt-get install git wget libncurses-dev flex bison gperf \
python python-pip python-setuptools python-serial python-click \
python-cryptography python-future python-pyparsing \
python-pyelftools cmake ninja-build ccache libffi-dev libssl-dev
```

Add current logged user to the group *dialout* because the user needs to get read and write access to the serial port over USB and restart the PC.

```
sudo usermod -a -G dialout $USER
```

Download software libraries provided by Espressif, Espressif IoT Development Framework (*esp-idf*), to folder ”esp”. I used a release version 4.1:

```
cd $PATH_TO_ESP/esp
git clone -b release/v4.1 --recursive \
https://github.com/espressif/esp-idf.git
```

Or you can copy the whole folder ”src/esp” form this Master’s thesis to your ”$PATH_TO_ESP/esp”.

Install tools used by *esp-idf* to default directory ”$HOME/.espressif”:

```
cd $PATH_TO_ESP/esp/esp-idf
./install.sh
```

Last step is to set up environment variables in the terminal where is going to be used *esp-idf*:

```
. $PATH_TO_ESP/esp/esp-idf/export.sh
```

But I added it to the *.bashrc* file:

```
echo ". $PATH_TO_ESP/esp/esp-idf/export.sh \
&> /dev/null" >> $HOME/.bashrc
```

This way it will be added to every new shell session.

2.3.1.1 Build & Load

To load application to ESP32-LyraT, one must first build the project. For example, navigate to the project "$PATH_TO_ESP/esp/esp-idf/examples/get-started/hello_world" and in this folder, it is possible to build it:

```
idf.py -n build
```

The switch "-n" will stop treating the warnings as errors.

After successful build it is possible to load/flash application to ESP32-LyraT:

```
idf.py -p /dev/ttyUSB0 flash
```

The value of the port can differ based on which one is ESP32-LyraT connected to.

When the loading/flashing starts, it will be waiting for a connection from ESP32-LyraT. At that moment, hold the boot button and press the restart button.

To check if the application is indeed running after loading/flashing, use the monitoring system:

```
idf.py -p /dev/ttyUSB0 monitor
```

For ESP32-LyraT, when the monitoring starts, press the restart button.

It is possible to combine the previous commands together:

```
idf.py -p /dev/ttyUSB0 build flash monitor
```

2.3.1.2 Memory

Microcontroller ESP32-LyraT has 8MB of external memory (SRAM/SPI-RAM). But the processor only supports up to 4MB of the external RAM, which can be allocated using standard *malloc* calls. To use the region above the 4MB limit, it is possible to use the *himem API*.

Configuration - For enabling the external RAM, navigate to "menuconfig → Component config → ESP32-specific → Support for external, SPI-connected RAM → SPI RAM config". To access *menuconfig*:

```
idf.py menuconfig
```

Stack - It is possible to change the size of the stack for the application. The settings can be found at "menuconfig → Component config → Common ESP-related → Main task stack size".

Himem - API which enables access to the remaining memory of external RAM. However, this is done through a bank switching scheme. Configuration can be found at the same place as external RAM in *menuconfig*.

2.3.2 Project description

The basic *esp-idf* project is composed of the following:

- build - A folder where the output of the build process is stored.
- components - A folder which contains subprojects or external projects of the application.
- main - A folder which contains source codes of the application. It is also called the *main* component.
- CMakeList.txt - A global setting of the project and the starting file for *cmake*.
- sdkconfig - A setting of *menuconfig*.
- sdkconfig.default - A default setting of *menuconfig*.

For convenience's sake, I added a file to this structure:

- README.md - A file with basic information about the build.

The project can be built using *make* or *cmake*. I decided to use *cmake*, since almost every example uses it and it is also recommended in the documentation.

Main entry point of the application running on ESP32 is the *app_main* function. In provided implementations, it can be found in the *main/test.c* file.

2.3.3 Lifted Unbalanced Oil & Vinegar

To make a port of LUOV reference implementation to ESP32, I first needed to create *CMakeList.txt* in the *main* folder. This file contains build options for the *main* component.

```
idf_component_register(SRCS INCLUDE_DIRS PRIV_REQUIRES)
```

This function is for registering a project component to the internal build structure of *idf* API.

- SRCS - Files of source codes.
- INCLUDE_DIRS - Paths to header files.
- PRIV_REQUIRES - Components which have to be built before the *main* component and then linked to it.

It is necessary to set up the path to the *idf* compiler header files. Otherwise the default (in my case GCC) headers will be used, which are incompatible with ESP32. Last part of this file is the block which takes care of parsing build flags of the project in variable *B_FLAGS*.

The implementation requires the *XKCP* component for PRNG. But, how it can be seen in the *components/XKCP/CMakeList.txt* file, it only requests a few files from the *XKCP* project. This means it is not necessary to build the whole project but only the required parts. This way it will reduce the final size of the application.

One of the important required changes is the change of the default value for the size of the stack to 20000B. This size is sufficient to support all stack memory allocations. It also needs to enable the use of external memory. Both of the settings are set in the *sdkconfig.default* file.

The reference implementation has the implementation (file *rng.c*) of a random number generator (RNG) from the NIST standard. But this RNG requires the *OpenSSL* library, which is not a part of *esp-idf* by default. I found that there is a component *esp-wolfssl* which is an embedded SSL library and offers a simple API with OpenSSL compatibility layer. Unfortunately, calling the initialization function of *OpenSSL*, ESP32-LyraT will cause a segmentation fail. This means that it is impossible to use it.

What can be done (and what I also did), is the change of the RNG to the ESP32 hardware RNG. Nonetheless, there is a condition in which the Wi-Fi or the Bluetooth needs to be enabled. Otherwise, it can not be considered a truly random number generator, but only a pseudo-random number generator.

The implementation can be found in the *src/esp/luov* folder.

2.3.3.1 Optimization

Because this ported implementation of LOUV is fast and has low memory consumption (see chapter 3), I did not try to make any optimization attempts.

2.3.3.2 Memory

To be able to measure the memory allocated in ESP32, I implemented memory measurement with the help of *esp-idf* API. Implementation can be found in the *memory_measurement.c* file.

This measurement will create a new independent task which will periodically ask ESP32 about internal and external memory status. Because ESP32 is a dual-core processor, the slowdown by this task should be minimal and it

should have a minimal influence on the signing algorithm. But to be sure there is no slowdown, the speed measurement was taken without this memory measurement.

To enable this functionality, the following flag needs to be set:

- MEM_MEASUREMENT

2.3.4 Rainbow

To make a port of Rainbow reference implementation to ESP32, I proceeded in the same way as for the LUOV ESP32 implementation. That means I created *CMakeList.txt* in the *main* folder with the same formalities as *CMakeList.txt* for LUOV, see section 2.3.3. The most eye-catching difference is the set up of different kinds of *malloc*, see section 2.3.4.2 for more information.

The implementation requires the *wolfssl* component for PRNG. In this case, it builds the whole *esp-wolfssl* project which is simply added through the setting of *PRIV_REQUIRES* in *main/CMakeList.txt*. It is important to mention here that the Rainbow reference implementation contains two PRNG (see *utils_prng.c*). The first one from the NIST standard, it is the same RNG as in the LUOV reference implementation, and the second PRNG which uses the hash SHA function. Because the RNG form the NIST standard was impossible for me to run on ESP32-LyraT, I changed it to the second PRNG and deleted the implementation of the first one from the source code.

The source codes also require RNG which I changed to the hardware RNG of ESP32. This is the same change as in the LUOV ESP32 implementation (see *rng.c*).

The default value for the size of the stack I set up to 5000B because this value is sufficient to support all of the Rainbow stack allocation memory. The setting is set up in *sdkconfig.default* with enabled external memory to get access to the external 4MB RAM.

The implementation can be found in the *src/esp/rb* folder.

2.3.4.1 Optimization

I did two memory optimizations of the Rainbow implementation for ESP32:

- First, I found potentially large memory allocations and switched them from stack to heap. Candidates to the modification were the temporary helpful variables of keys (*sk_t*) in the computation of keys from seeds.

- Secondly, I created *my_ESP_malloc* and *my_ESP_free* functions, see section 2.3.4.2. With these, I found that temporary variables (for example *sk_t* tempQ*) were using a lot of memory, but in reality only needed a part of the key structure. See commit *"ESP - RB memory reduction in cyclic generation"*, hash *"1609d70c"* in this Master's thesis' Git repository for the optimization details.

With these two optimizations, there was already enough memory and I was able to run the *_RAINBOW256_92_48_48* implementation in compressed form. I was also able to run the same implementation in the *cyclic* form but there occurs the same kind of bug and it was making the segmentation fault. I suspect it is the same bug which causes the failure of signature verification.

2.3.4.2 Memory

To be able to better measure the allocated memory in ESP32, there is the same memory measurement implementation as in LUOV, see section 2.3.3.2, which can be enabled by flag:

- MEM_MEASUREMENT

Because I needed a better technique for memory allocation on the heap, I created my own allocation information table. It is a simple small array because I expected a small number of allocations at one point at a time, which holds the pointer information and its size. It can be enabled by the flag:

- MY_ESP_MALLOC

When this flag is set, it will switch all *aligned_alloc* to *my_ESP_malloc* and *free* to *my_ESP_free*, and prints its information whenever allocation or deallocation occurs. This information table helped me to identify potential places in the source code of RB for memory optimization. See the *malloc.c* file for implementation details.

The reference implementation uses the *aligned_alloc* function to allocate memory, but ESP32 with the toolchain 8.2, which I used, had an issue with this function and it was not possible to use. That is the reason why I changed it to the classic *malloc* function.

2.4 Conventional algorithms

These selected conventional algorithms were also implemented on the ESP32-LyraT microcontroller for the possibility of comparison with LOUV and RB. These two algorithms are some of the most typical and most used algorithms in computers for signature schemes. This means very good software implementations with hardware acceleration on the ESP32 chip exist.

Both of the selected algorithms are implemented using the *esp-idf* API in a similar way like previous implementations in this thesis on ESP32. That means their entry point is in the *test.c* file which has a similar structure to other *test.c* files in the previous description. The main difference is in the private and public keys which are now in the form of *context* relative to the algorithm.

2.4.1 RSA

Test implementation of RSA starts with the initialization of the *context*:

```
mbedtls_rsa_context ctx;
mbedtls_rsa_init(&ctx,MBEDTLS_RSA_PKCS_V15,0);
```

Second step is the generation of a keypair:

```
mbedtls_rsa_gen_key(&ctx,coap_prng_impl,NULL,KEY_SIZE,EXPONENT);
```

Generation of the *hash* of the message *m*:

```
mbedtls_sha256(m,message_size,hash,0);
```

Sign the *hash* and put the signature to the *sm*:

```
mbedtls_rsa_pkcs1_sign(&ctx,coap_prng_impl,NULL,
MBEDTLS_RSA_PRIVATE,MBEDTLS_MD_SHA256,
0,hash,sm);
```

Verify the signature *sm* of the *hash*:

```
mbedtls_rsa_pkcs1_verify(&ctx,coap_prng_impl,NULL,
MBEDTLS_RSA_PUBLIC,MBEDTLS_MD_SHA256,
0,hash,sm);
```

The implementation can be set with the following flags:

- KEY_SIZE - Public key size in bits.
- EXPONENT - Public exponent to use.
- MEM_MEASUREMENT - Enable memory measurement.

2.4.2 ECDSA

Test implementation of ECDSA is very similar to RSA. It starts with the initialization of the context:

```
mbedtls_ecdsa_context ctx;
mbedtls_ecdsa_init(&ctx);
```

Second step is the generation of a keypair:

```
mbedtls_ecdsa_genkey(&ctx,EC_CURVE,coap_prng_impl,NULL);
```

Generation of the *hash* of the message *m*:

```
mbedtls_sha256(m,message_size,hash,0);
```

Sign the *hash* and put the signature to the *sm*:

```
mbedtls_ecdsa_write_signature(&ctx,MBEDTLS_MD_SHA256,hash,
hash_len,sm,&smlen,coap_prng_impl,NULL);
```

Verify the signature *sm* of the *hash*:

```
mbedtls_ecdsa_read_signature(&ctx,hash,hash_len,sm,smlen);
```

The implementation can be set with the following flags:

- EC_CURVE_BITS - Number of bits for elliptic curve.
- MEM_MEASUREMENT - Enable memory measurement.

Chapter **3**

Testing and discussion

This chapter contains the algorithms measurements and testing, their comparison and discussion about their implementation. It focuses mainly on time and memory complexity of the selected algorithms and possible usability in an embedded environment.

3.1 PC

The reference implementations with my adjustments were tested on the computer with the following parameters:

- OS: Ubuntu 19.4
- CPU: Intel Core i5-7300HQ
- Clock Speed: 2.50GHz
- RAM: 8 GB
- Number of cores: 4
- Compiler: GCC 8.3.0

Every measurement was done on a message of size 10000 B and was repeated 10 times. The final results in the figures below are given as average values.

The main goal of these measurements and implementations was to have comparable data of reference implementations of LUOV and RB on a PC. Because they were measured using different processors and only in processor cycles in the reference materials. I chose to measure time (in seconds) and memory consumption (in bytes).

The solutions were compiled with the same (default) setting with the *-std=c99* flag.

3.1.1 Signature variants

For measurement, I selected the reference signature variants. The first number indicates the finite field $\mathbb{F}_n$, the second one is the number of vinegar variables, the third one is the number of oil variables and the fourth one (at RB) is the number of oil variables in the second layer. Divided by security categories there are the following:

Category I:

- Luov-47-42-182
- Luov-7-57-197
- Rb-16-32-32-32 - and its variants

Category III:

- Luov-61-60-261
- Luov-7-83-283
- Rb-256-68-36-36 - and its variants

Category V:

- Luov-79-76-341
- Luov-7-110-374
- Rb-256-92-48-48 - and its variants

Each of the categories represents a difficulty through how many hardware gates need to be used to be able to break the security variant.[15] Overview:

Category	Log_2 classical gates	Log_2 quantum gates	Ex. of alg.
I	143	130/106/74	AES-128
II	146		SHA3-256
III	207	193/169/137	AES-192
IV	210		SHA3-384
V	272	258/234/202	AES-256
VI	274		SHA3-512

Table 3.1: NIST security categories

For the quantum gates, there is also the *MAXDEPTH* parameter of 2^{40}, 2^{64}, 2^{96}, which represents fixed running time or circuit depth. The reason for this parameter is the Grover's algorithm.

An algorithm meets the requirements of a specific security category, if the best-known attack uses more resources (gates) than needed to solve the reference problem.

Concerning the LUOV parameters, a finite field of size 7 is selected because of the small size of the generated signature, but the public key size is quite large. The schemes not having the finite field of size 7 are selected because of the small public key.

3.1.2 Time complexity

The next two figures display the individual stages of the signature scheme and the time it takes to complete each of the stages. Namely generation, signing and verifying. For time measurement I used *clock_t* from the standard header *time.h*.

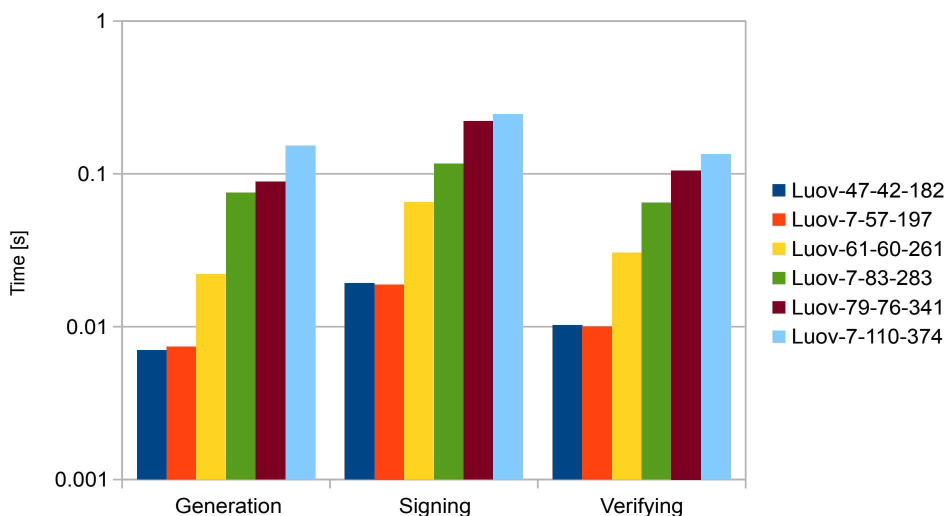

Figure 3.1: Comparison of LUOV on PC

As can be seen in 3.1, the stronger the security the more time is needed for the computation which is the expected result. Regarding the generation stage, the shorter the public key the fastest it is. The same is true for the signing and verifying stages, which are again the expected results.

In figure 3.2, it is possible to see that generating the public key (*cyc* versions) and the private key (*com* versions) from the seed is a really slow operation. It can be seen especially on the *com* variants where the signing stage can be up to 85 times slower (maybe even more) than the *classic* variants. The verifying stage is also faster than the *classic* variant because there is no generation from the seed.

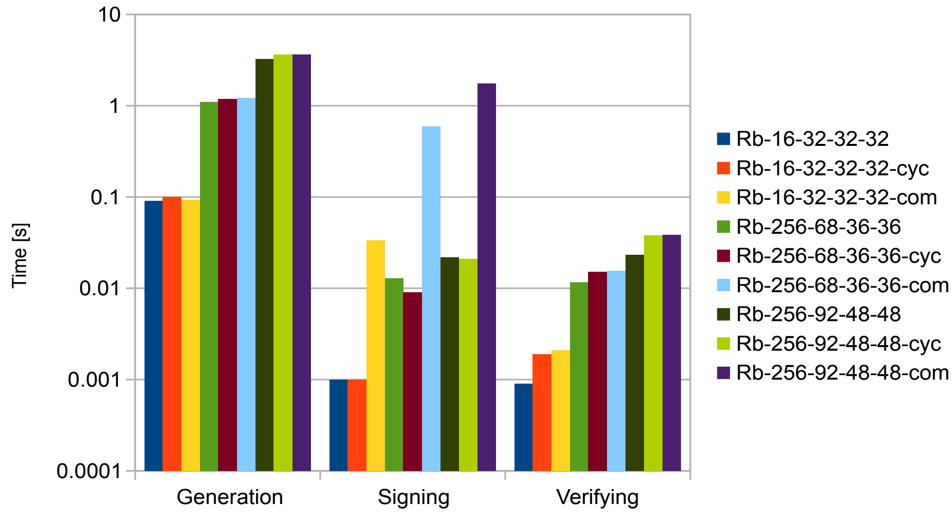

Figure 3.2: Comparison of RB on PC

The next figure 3.3 is the comparison of LUOV and RB implementation. I only compare the *com* versions of RB with LUOV with short public keys. I selected this LOUV variant because it is faster and this RB variant because it also (like LUOV) generates public and private keys from the seed. There is no generation stage because on 3.1 and 3.2 it is obvious that generation times of RB are much slower than LOUV.

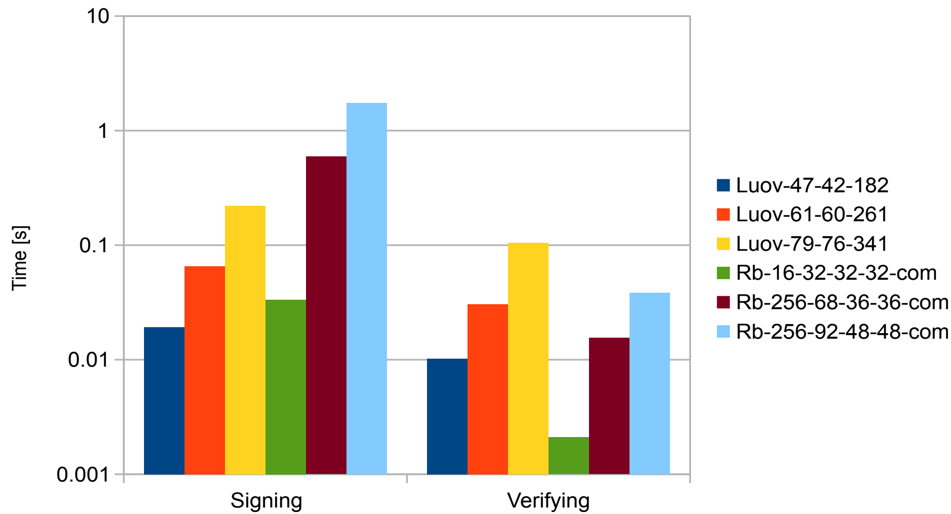

Figure 3.3: Comparison of PC implementations

How can be seen in 3.3, the LOUV implementation is faster at signing but slower at verifying.

3.1.3 Memory complexity

Memory complexity deals with the allocation of RAM on a heap for the signature schemes. I did not measure the required memory on a stack because it is negligible in comparison to the heap (in the magnitude of a few kilobytes). I measured the PC implementations with the help of Valgrind. Specifically, with the help of tool *Massif* which is a heap profiler:

```
valgrind --tool=massif ./test
```

To get a human-readable output it needs to be used with the following:

```
ms_print massif.out.*
```

From its output, I created the next few figures which show the maximum memory allocation (in bytes) over the whole run of the test application.

In figure 3.4, the memory allocation of LUOV is visible. It shows the expected result which is the following: with higher security category more memory is needed. It is also visible that LOUV variants with a shorter public key need less resources.

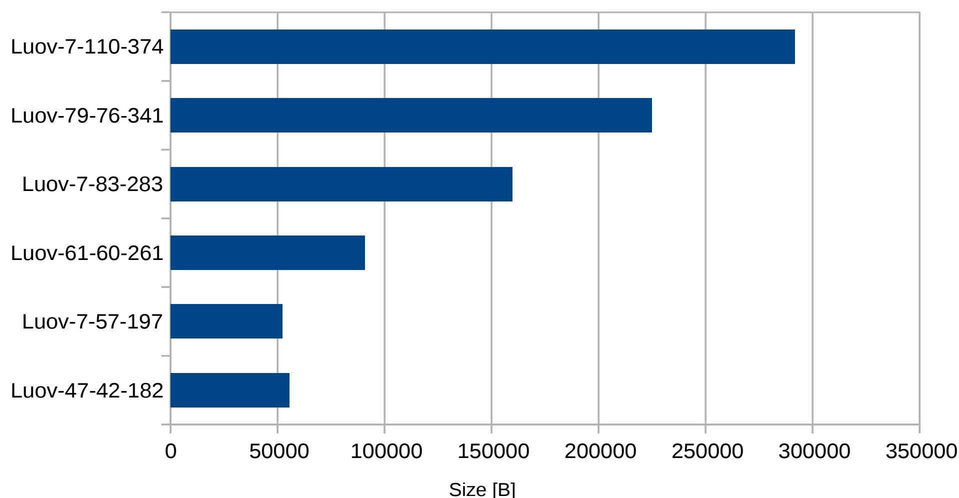

Figure 3.4: Memory requirement of LUOV on PC

Next is the figure 3.5 illustrating RB memory allocation. It is shown again that the higher security category the more resources it needs, which is the expected result. But to my surprise, the *com* variants of Rainbow need less RAM allocation compared to the *classic* and *cyc* variants. The only explanation I can think of, which can explain this behaviour, is the generation of a private key from the seed. That is the main difference compared to other variants.

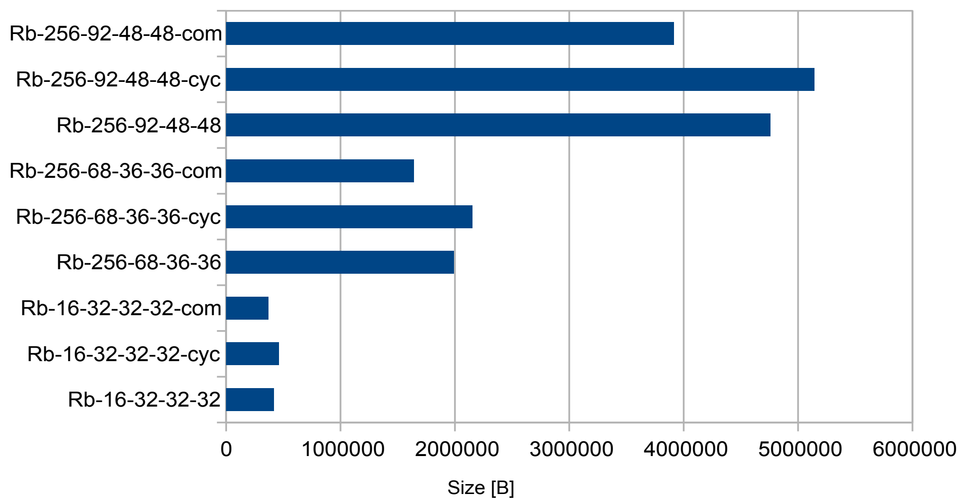

Figure 3.5: Memory requirement of RB on PC

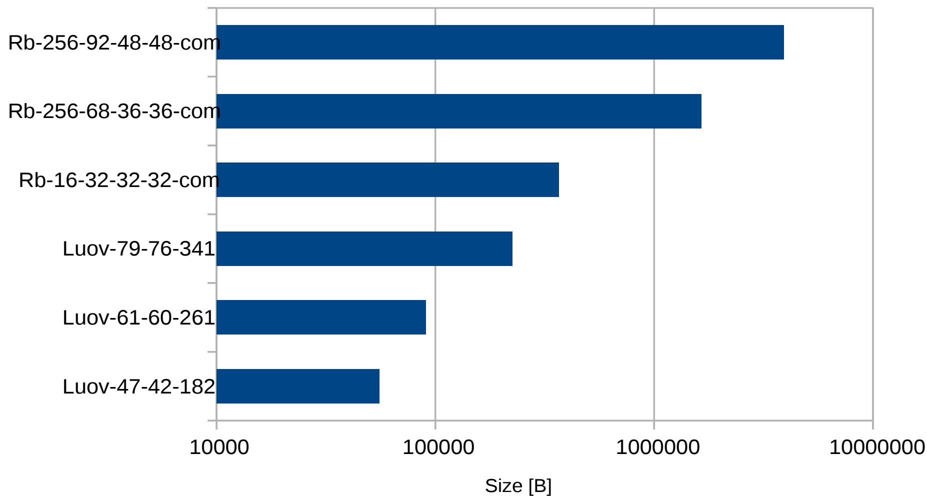

Figure 3.6: Comparison of PC implementations

The last figure of the memory complexity section is figure 3.6 illustrating the comparison of LUOV and Rainbow. The figure shows that LUOV requires much less memory than RB. This behaviour I attribute to the implementation specifics (code programming, optimization and memory handling), which are much better than in the Rainbow implementation.

3.1.4 Conclusion note

The measured values show that the reference implementation of LOUV is almost better at everything. It is faster, has a smaller memory imprint and has a simpler implementation.

The only thing in which Rainbow is better is signature verification (how can be seen in figure 3.3), but that is an implication from the signature size, which is shorter than LUOV. For more details, see section 3.2.4.

From the point of usability, the Rainbow implementation is only optimal if a lot of signature verification needs to be done or if there are a lot of very short messages. Then the shorter signature can speed up the computational means, for example, the network communication.

But I do not recommend this implementation of RB because there seems to be some kind of an implementation bug. It can be seen in the *cyc* variant of RB where the verification of the signed message reports that it is an invalid signature.

If I were faced with choosing one of these implementations for general use, I would select the LUOV implementation because, as was already mentioned, it is faster, smaller and it does not contain any bugs (at least I did not find any).

3.2 ESP32

ESP32 implementations were tested and ported to the ESP32-LyraT microcontroller, specification of the microcontroller can be found in section 2.3. The reason for the port is to test the behaviour of the selected algorithms in the IoT environment or at least in an environment which is as similar as possible to it.

Every measurement was done on a message of size of 1000B and was repeated 10 times. The final results in the figures below are given as average values.

3.2.1 Signature variants

The same variants of signature schemes were selected as in the PC section 3.1.1. There were, however, some changes. Dividing by security categories they are the following:

Category I:

- Luov-47-42-182
- Luov-7-57-197
- Rb-16-32-32-32 - and its variants

Category III:

- Luov-61-60-261
- Luov-7-83-283
- Rb-256-68-36-36 - and its variants

Category V:

- Luov-79-76-341
- Luov-7-110-374
- Rb-256-92-48-48-com

In the category V, there is only the *com* variant of RB scheme because I was not able to run the other variants (*classic* and *cyc*) due to the limited RAM.

I did a few memory optimizations in the RB scheme which can be found in the 2.3.4.1 section. These optimizations reduced a considerable amount of memory but when I tried to run the *Rb-256-92-48-48-cyc* variant I received a segmentation fault. I attributed this behaviour to a bug in the implementation where the signature is failing.

3.2.2 Time complexity

As in the PC measurement before, section 3.1.2, following figures display individual stages of the signature scheme and the time it takes to complete each of the stages. For the time measurement, I also used *clock_t* from the standard header.

In figure 3.7, the comparison of LUOV times which it took for each of the stages is shown. It can be seen that the higher the security category, the more time is needed for the completion of the stage. That is the expected result as in the PC implementations, but some of the times were in the periods around 10 seconds. Especially Luov-7-110-374 or generally the whole security category V is unusable with these periods in the moment when a lot of messages need to be signed or verified.

Also, category III took a lot of time but the LUOV variant *Luov-61-60-261* seems to be fast enough on ESP32-LyraT and can be used for common signature schemes. As a category III, it can provide very high security for IoT devices but it has one disadvantage which is a large signature.

The *Luov-47-42-182* and *Luov-7-57-197* variants of security category I can be fast on embedded devices putting them at the forefront of the other LUOV signature variants.

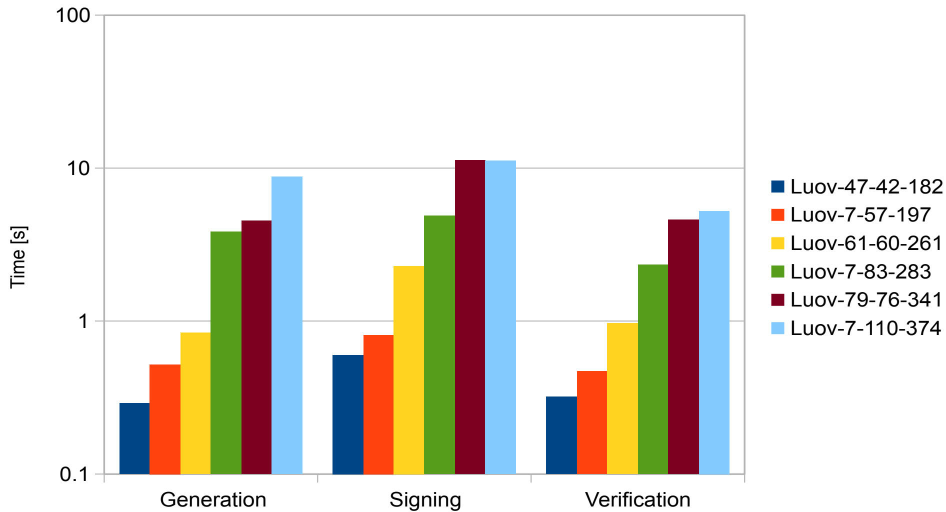

Figure 3.7: Comparison of LUOV on ESP32

Figure 3.8 is the comparison of times of Rainbow scheme. There it can be seen that the generation of public and private keys is very slow but in the

practical (mostly) use cases this step needs to be only done once, therefore it has little influence.

It can also be seen that signing and verifying stages do not belong into fast operations. Especially the *com* versions which are really slow. It is very well visible that generating public and private keys from the seed is not a good idea from a time point of view. And if there was no generation, then there is a large speedup, as can be seen in the comparison of *Rb-256-68-36-36* and *Rb-256-68-36-36-com*.

As in the LOUV scheme before, in the RB scheme the best variant is from security category I, the variant *Rb-16-32-32-32*. It has more than enough security for the common communication on IoT devices.

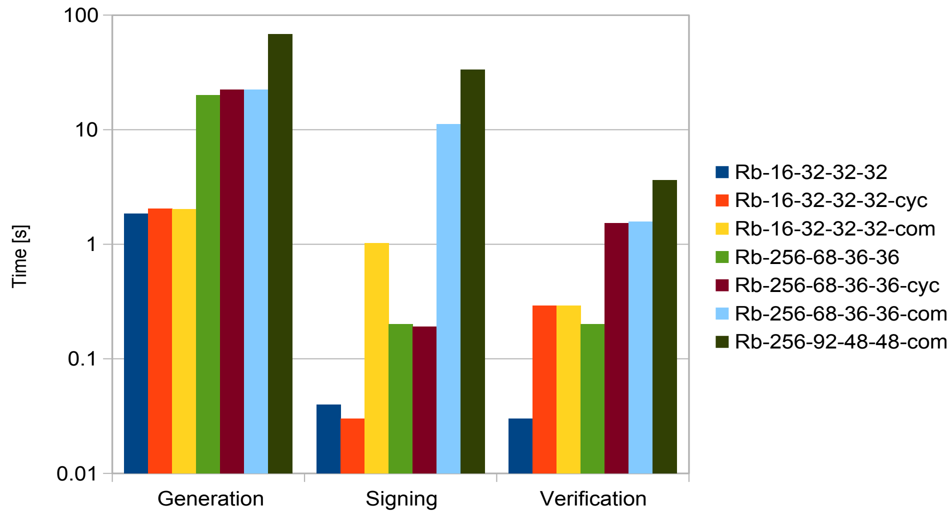

Figure 3.8: Comparison of RB on ESP32

Last figure 3.9 of this section represents the comparison between LUOV and RB. It is the only comparison between the LOUV variants with a shorter public key and the RB *com* variants. The reasons for this selection of signature schemes variants are given in section 3.1.2.

In this figure, it can be seen that the LUOV variants are faster in signing and the RB variants are faster in the verification process, except for the *Rb-256-68-36-36-com* which got slowed down. There is no comparison of the generation stage because LOUV is clearly better in every security category.

If we compare the ESP32 figures of the time results with the PC implementations, we can see similar shapes. This means there are no large mistakes

in my port of implementation on ESP32. It is also noticeable that the verifying stage of all the variants of RB got slowed down compared to the PC implementations.

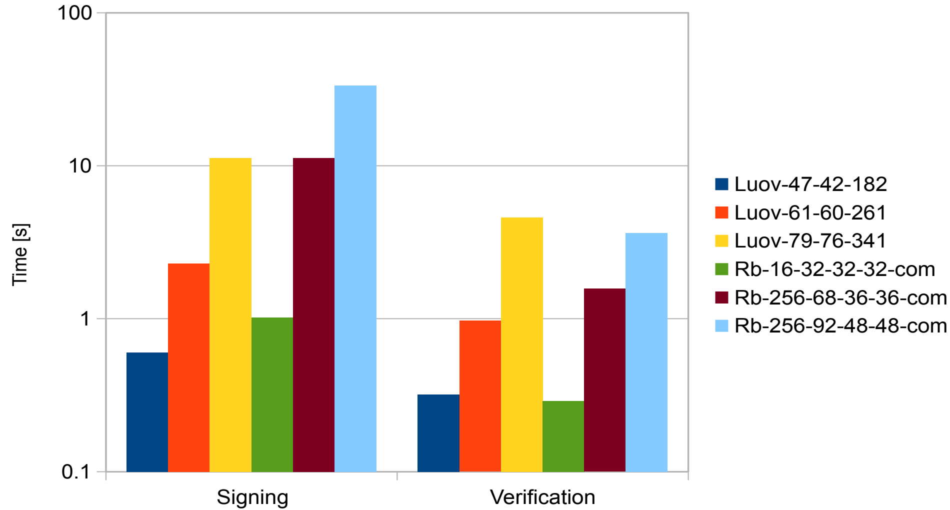

Figure 3.9: Comparison of ESP32 implementations

3.2.3 Memory complexity

Memory complexity was measured through the *MEM_MEASUREMENT* flag, which is described in more detail in the 2.3.3.2 section. It measures an allocation on the heap which is extended by the external memory. The required memory on the stack was not measured because it is negligible in comparison to the heap (in the magnitude of a few kilobytes) and it was set to a maximum of 20000B in LUOV implementation and 5000B in RB implementation.

From the flag output, I created the next few figures showing the maximum memory allocation (in bytes) over the whole test application run.

The first figure 3.10 is an exception from the previous statement because there was a small number of output data from the flag and it was possible to align them such that I was able to create a figure visualizing memory allocation of LUOV over the whole period of generation (left peak), signing (middle) and verification (right peak). The peaks represent the moments when the implementation needs to compute the public or private key from the seed. It then uses the key and discards it afterwards.

It is possible to see that more memory is needed with higher security. Also, the LOUV variants with a shorter public key need fewer resources. This could also

be seen in figure 3.11 which only shows the maximum allocation of memory. This is the same conclusion as in the PC memory measurement, section 3.1.3.

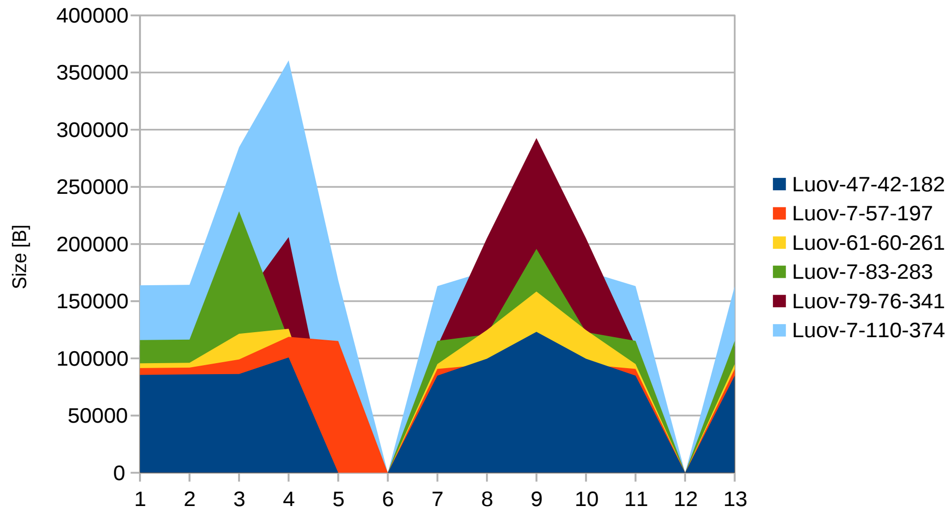

Figure 3.10: Memory requirement of LUOV on ESP32

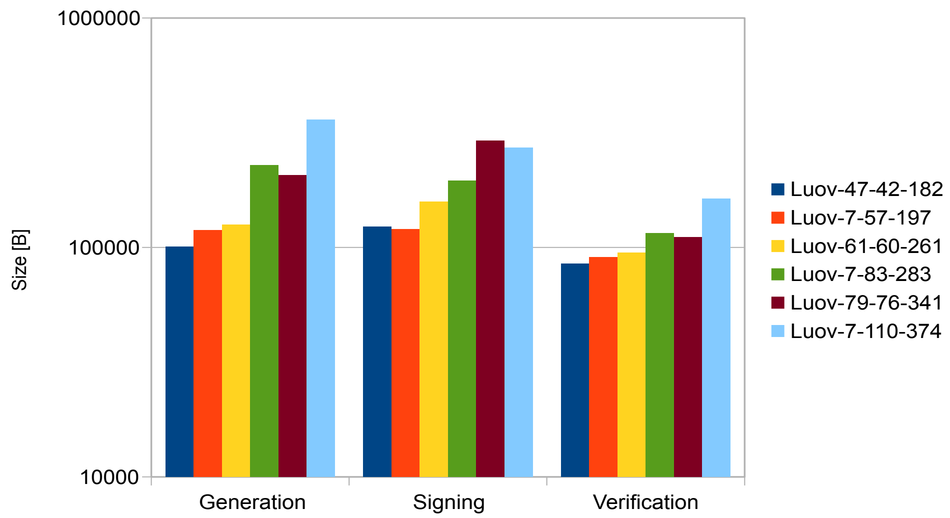

Figure 3.11: Memory requirement of LUOV on ESP32

In the next figure 3.12, memory allocation of the Rainbow scheme is displayed. Again, it shows that with higher security category the implementation needs more memory. However, some of the Rainbow variants suspiciously need the same amount of memory, around 4MB, for example, *Rb-256-68-36-36-com* or *Rb-256-92-48-48-com*. However, this same allocation of memory can be seen on the RB variant *Rb-16-32-32-32-cyc* in the verification stage.

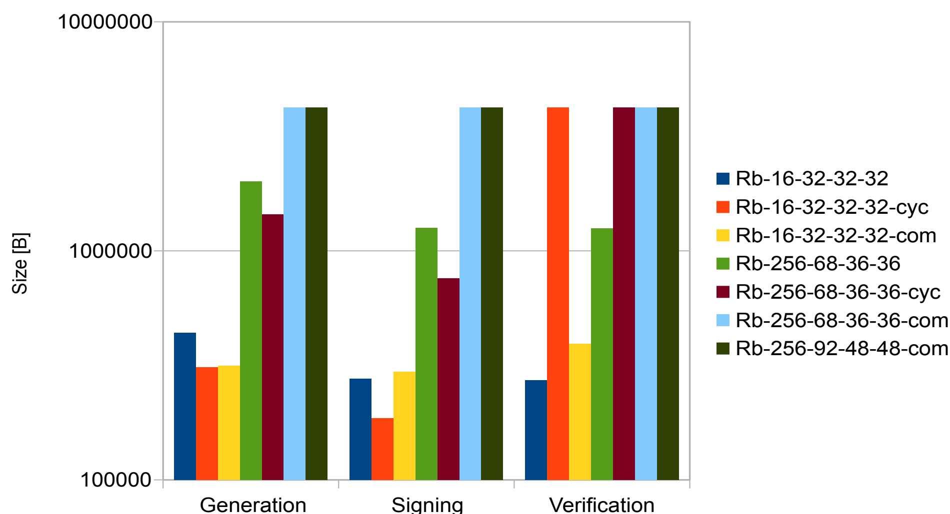

Figure 3.12: Memory requirement of RB on ESP32

I can only think of two possibilities of this result:

- Error in measurement - In my implementation of memory measurement on ESP32 is some kind of error which inputs this result. But it is only a few lines of C++ code directly using the *esp* API which is why I am inclined to the second possibility.
- Wrong allocation - ESP32-LyraT only allocates the rest of free memory in external RAM. This can happen when there is not enough time after the free operation and the allocation table is not updated, therefore it uses the next free block. Additionally, the segmentation of memory may not be optimal and the needed memory block may not be allocated in previous blocks because it requires a bigger capacity.

From these results, I can assume that a minimum of 4MB RAM is needed to safely run the Rainbow implementation on ESP32-LyraT. It is also shown that having a public key in the form of seed saves some of the memory (comparison of the *cyc* and *classic* variants).

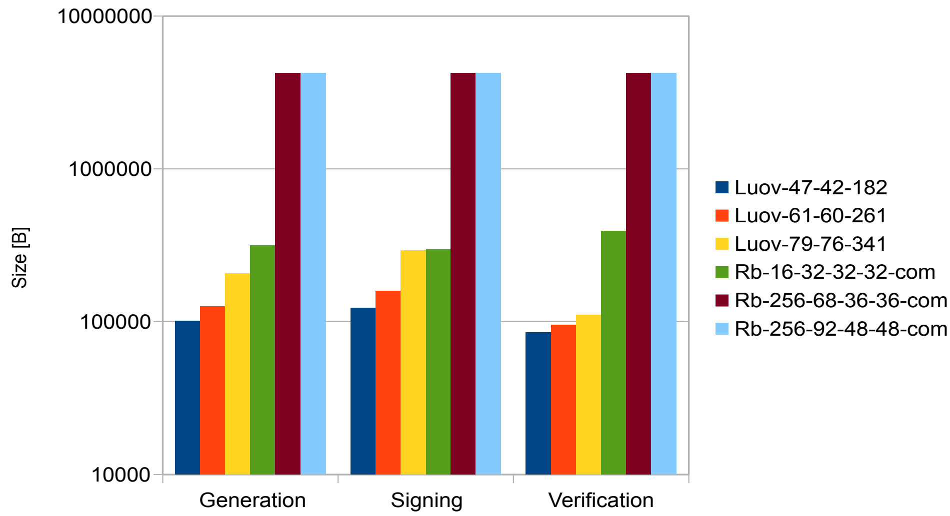

Figure 3.13: Memory requirement of implementations on ESP32

Figure 3.13 shows the comparison of LUOV and RB schemes in memory requirements. LUOV is again a better implementation in memory management. But because of this strange memory allocation in RB cases I created the *MY_ESP_MALLOC* flag, see 2.3.4.2. With this flag, I received information about every *malloc* and *free* in the implementation and was able to create figure 3.14.

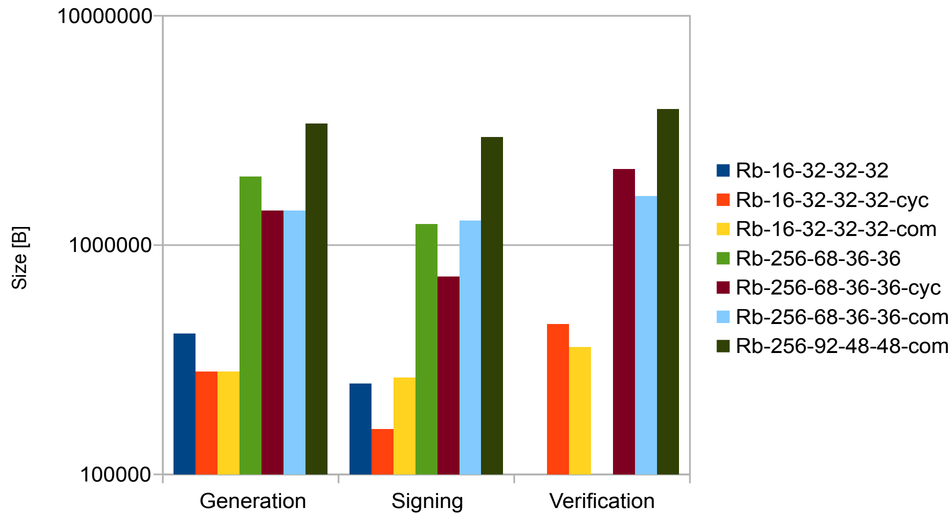

Figure 3.14: Memory requirement of my_ESP32_malloc

Now, compared to figure 3.12, more meaningful results of memory allocation can be seen. It must be noted that there are heap allocations of bare imple-

mentation without system libraries.

Figure 3.14 shows that the *cyc* variants of RB are using the seed for a public key because the memory needs are lower in the signature stage but higher in the verification stage when compared to the *classic* variants. It also beautifully shows the use of the seed for a private key in the *com* variants. In the signature stage, it needs more memory than the *cyc* variant because of the generation of a private key from the seed but in the verification stage, it saves the memory because the private key is only in the form of a small seed.

3.2.4 Keys & signature

This section covers the comparison of sizes of keys for the different variants of the signature algorithms. It must be said that the sizes of keys of the ESP32 implementations are the same as the PC implementations. I noted them from the measurement of the ESP32 implementations.

Figure 3.15 shows LOUV sizes of public key, private (secret) key and signature. It is clear that the LUOV implementation uses the same length of the seed for its secret key. The variants with shorter signatures (*Luov-7*) which can be up to 10 times shorter compared to their LUOV equivalents in the NIST security category can also be seen.

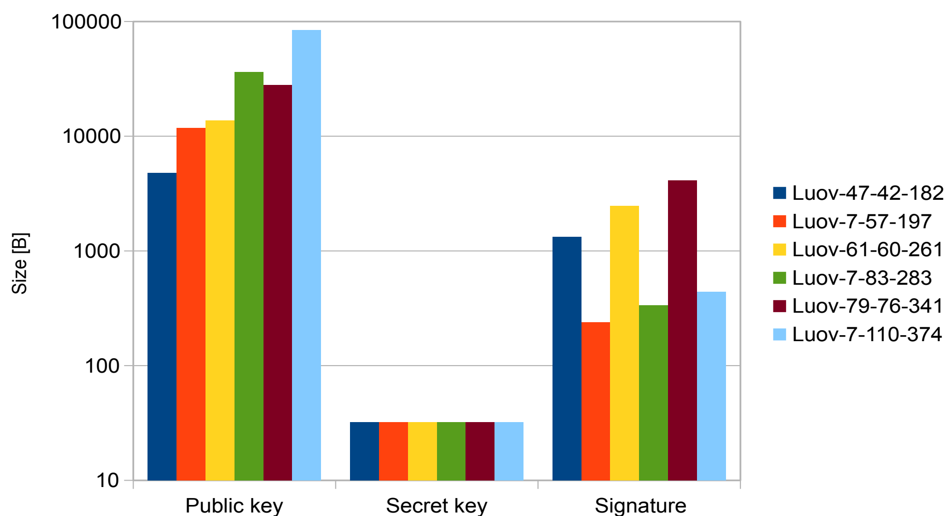

Figure 3.15: Size of signature of LUOV on ESP32

On the other hand, the same figure shows that with shorter signatures the variants need much more space for public keys, which can be up to 3 times longer if I compare *Luov-7-110-374* and *Luov-79-76-341*.

The variants of LUOV show a relatively large difference in these two values. This means it depends on the situation which of the variants will be more advantageous to use.

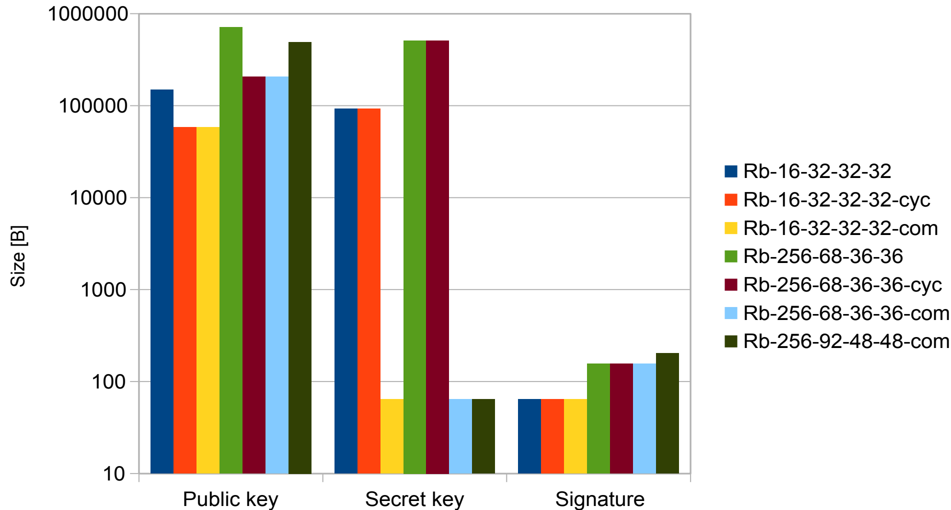

Figure 3.16: Size of signature of RB on ESP32

Figure 3.16 displays the size of the keys and signature of the Rainbow variants. It shows that when the seed is used for the secret key in the *com* variants, it saves a lot of space (the seed is only the size of 64B). For the public key in the *cyc* variants the memory size was reduced 2.5 to 3.5 times. The signature is longer (larger) with the stronger security category. However, for the *Rb-256-92-48-48-com* variant it is only 204B. This amounts to almost nothing if we consider that only the NIST security category V is taken into account.

Next two figures 3.17 and 3.18 show the comparison between LOUV and RB. The first one is the comparison between the LUOV variants with shorter public key and RB, and the second one is the comparison between the LOUV variants with shorter signature and RB. I split it into two figures for a better comparison, especially because of the signature.

In both of the figures, it is shown that LOUV use shorter seeds for the secret key. In numbers it is 32B for LUOV and 64B for RB. However, the comparison of the signatures also shows that the RB implementation has shorter ones in both figures. The size of RB signature is 204B, LOUV with shorter signature has 440B and LUOV with shorter public key the signature has 4134B in security category V.

The public key of LUOV is considerably shorter in both of the figures. In the first figure, it is up to 17 times shorter and in the second figure, it is up to 6 times shorter.

From these results in this section, it is evident that the only benefit of Rainbow implementation is short signature which can maybe be very beneficial, for example in some congested networks.

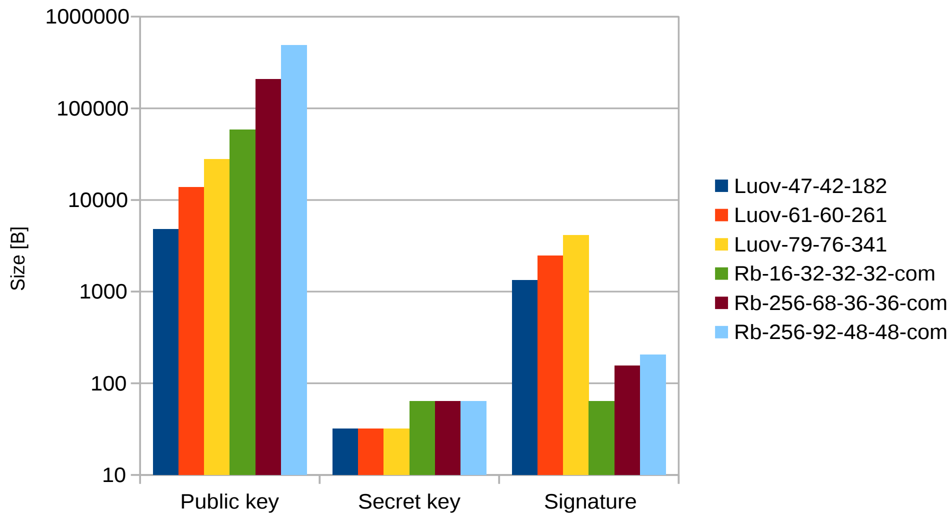

Figure 3.17: Comparison of LUOV with short public key and RB

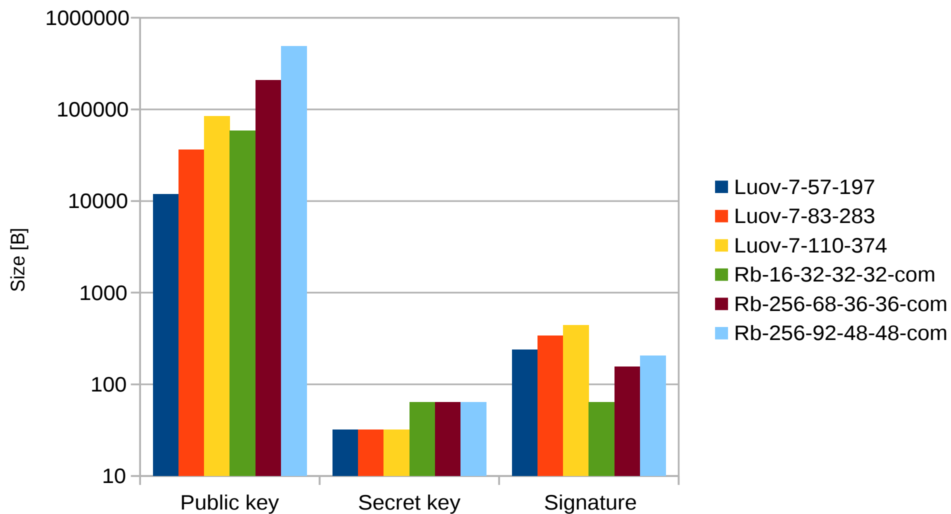

Figure 3.18: Comparison of LUOV with short signature and RB

3.2.5 Conclusion note

The measured values show that the implementation of LOUV is almost better in everything compared to RB. And also, the results of time measurements of the implementations on ESP32 are similar (if I do not count the general slowdown due to slower processor) to the implementations on PC, as they should be.

Rainbow has shorter signature length, see figures 3.17 and 3.18, but brings no other advantages. It needs much more memory for running (in some cases it allocates all available memory) and it is slow when we compare the *cyc* variants with LUOV.

The variants without key generation from the seed are fast, even much faster than the LUOV variants. On the other hand, if the LUOV was not using the seed for the keys it would also be faster. It would be interesting to make the comparison of these two variants but the LUOV implementation does not contain this implementation.

Interestingly, with higher NIST security categories the differences between two variants of the same category grow. The reason being that these variants use more resources which when compared have a larger difference.

3.3 Conventional algorithms

One of the important criteria in usability in an embedded environment is the comparison of algorithm complexity with conventional algorithms. For this comparison, I selected a signature scheme of RSA and ECDSA. All of them were measured on ESP32-LyraT.

Table 3.2 below lists the variants and the corresponding NIST security categories for possible comparison. [16] I was not able to measure the *RSA-7680* variant because the signature verification was failing (the reason for it could be the maximum support of 4096 bits for hardware "big number" accelerator), but it is good for illustration of the difference in the size of public keys between categories I and III and their counterparts in ECDSA.

Alg.	Category	Bit security
RSA-2048	N/A	112
RSA-3072	I	128
RSA-4096	N/A	140
RSA-7680	III	192
ECDSA-256	I	128
ECDSA-384	III	192
ECDSA-521	V	256

Table 3.2: NIST security categories of conventional algorithms

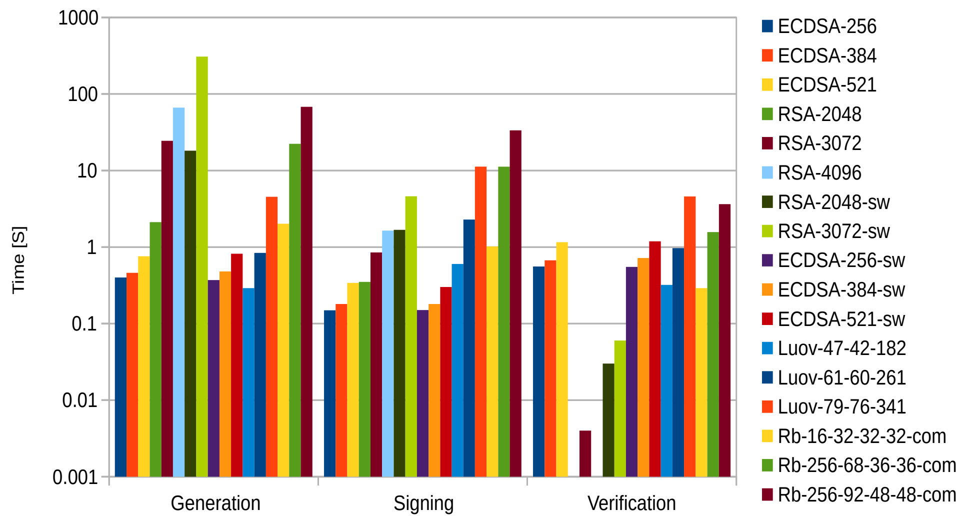

Figure 3.19: Comparison with conventional algorithms

First two figures 3.19 and 3.20 are the comparisons of time with the algorithms

selected for this thesis. They show that ECDSA is faster than RSA except for the verification. The LUOV variants show that they are not much slower than ECDSA, *Luov-47-42-182* is actually faster in the generation and verification stages.

RB variants have similar generation times with RSA. In both cases, these are rather long and the other stages are also slower. On the other hand, the variant not using the seed is really fast, even faster than the conventional algorithms.

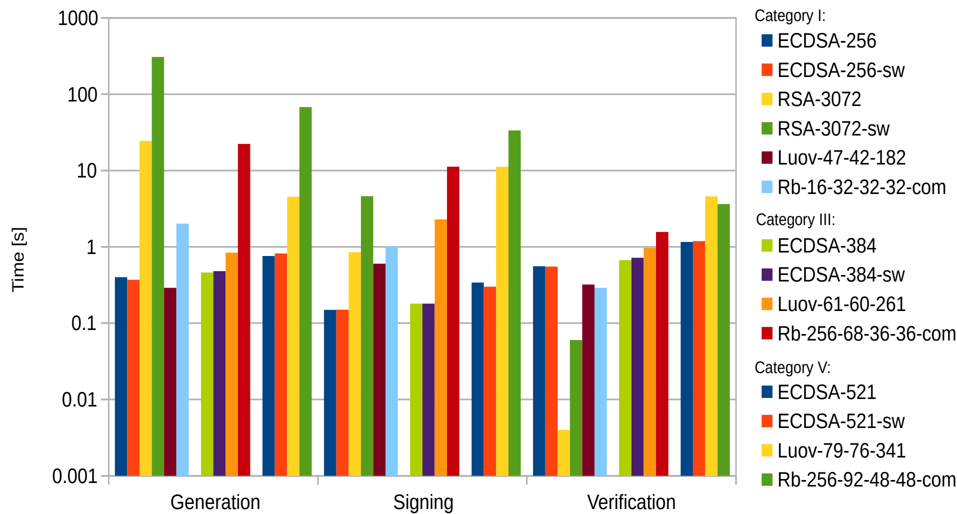

Figure 3.20: Time requirement comparison with conventional algorithms by categories

The ESP32 supports hardware acceleration of RSA and ECDSA in terms of big number multiplication for up to 4096 bits. I think with this disadvantage the implementation of LUOV and RB are not lagging too much behind and can be used in an embedded environment from the time point of view. But for comparison, I measured RSA and ECDSA without the hardware acceleration. These variants are with suffix *sw*. From the results, it is visible that RSA is heavily depending on the HW support, the *RSA-4096-sw* variant took so long that I decided to skip it, and ECDSA only has minimal to negligible slowdown.

Last two figures 3.21 and 3.22 illustrate the comparisons of memory requirements with the algorithms selected for this thesis. They show that most of the implementations need a similar amount (in KB) of memory for them to work, with the exception of the Rainbow.

From the comparison with the conventional algorithms, it can be seen that LOUV implementation is as good as conventional algorithms and maybe even

a little bit better because of the hardware acceleration, high security for embedded devices and small signature.

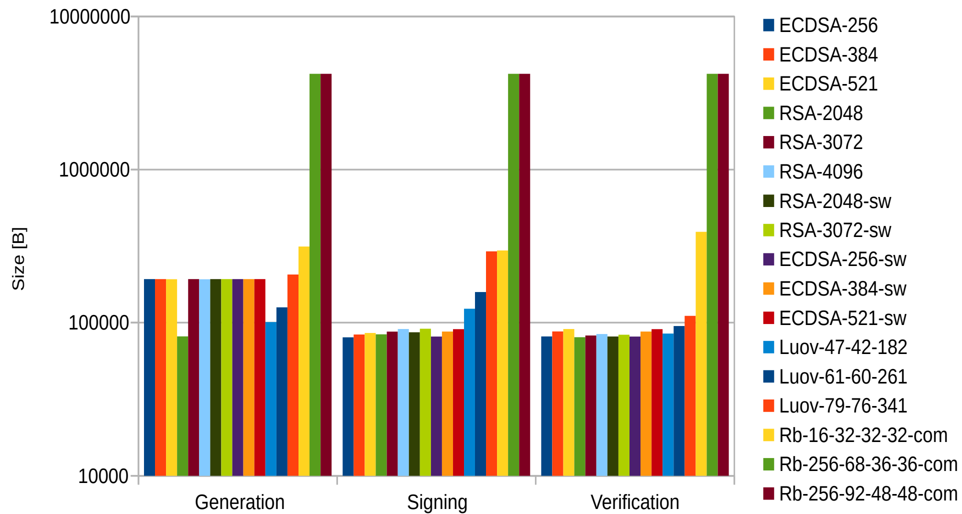

Figure 3.21: Memory requirement comparison with conventional algorithms

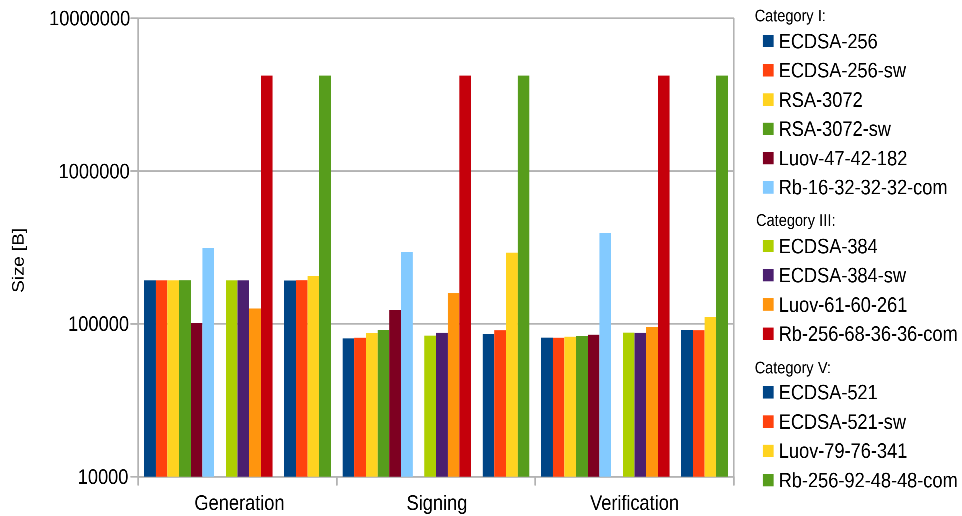

Figure 3.22: Memory requirement comparison with conventional algorithms by categories

Conclusion

The goal of this thesis was to describe multivariate cryptography and create a Wolfram Mathematica example for the educational purpose of the selected algorithms, specifically: Unbalanced Oil & Vinegar and Rainbow. It also deals with the implementation of the algorithms on PC and ESP32 and evaluates their memory and time complexity. Finally, it compares the implementations with the conventional algorithms, RSA and ECDSA. The goal of this Master's thesis was fulfilled.

In the first chapter, the terms used in the thesis are described and defined, which is followed by a description of multivariate cryptography and algorithms.

The second chapter deals with the referenced implementation of the algorithms, step by step examples in Wolfram Mathematica and description of ESP32 and its implementation of the selected algorithms.

In the third and last chapter is a description of the testing environment, including measuring and testing of the implementations on PC and ESP32. The algorithms were then compared with each other and with the conventional signature scheme implementations.

Bibliography

[1] CZYPEK, P.: *Implementing Multivariate Quadratic Public Key Signature Schemes on Embedded Devices.* Ruhr-Universität Bochum, 2012.

[2] PETZOLDT, A.: *Multivariate Cryptography Part 1: Basics* [online]. 2017, [cit. 2020-04-1]. Available at: `https://2017.pqcrypto.org/school/slides/1-Basics.pdf`

[3] PETZOLDT, A.: *Multivariate Cryptography Part 2: UOV and Rainbow* [online]. 2017, [cit. 2020-04-1]. At: `https://2017.pqcrypto.org/school/slides/2-UOV+Rainbow.pdf`

[4] GEOVANDRO, C.C.F.P.: *Introduction to Multivariate Public Key Cryptography* [online]. 2013, [cit. 2020-04-1]. Available at: `http://www.ic.unicamp.br/ascrypto2013/slides/ascrypto2013_geovandropereira.pdf`

[5] GOUBIN, L.; PATARIN, J.; YANG, BY.: *Multivariate Cryptography.* In: van Tilborg H.C.A., Jajodia S. *Encyclopedia of Cryptography and Security.* 2011, Springer, Boston, MA

[6] DING, J.; PETZOLDT, A.: *Current State of Multivariate Cryptography.* In: *IEEE Security & Privacy.*, vol. 15, no. 4, pp. 28-36, 2017.

[7] *Multivariate cryptography* [online]. 2020, [cit. 2020-04-1]. Available at: `https://en.wikipedia.org/wiki/Multivariate_cryptography`

[8] KIPNIS, A.; SHAMIR, A.: *Cryptanalysis of the oil and vinegar signature scheme.* In *CRYPTO 1998*, LNCS vol. 1462, pp. 257–266, Springer, 1998.

[9] *NIST - Post-Quantum Cryptography, Round 2 Submissions* [online]. 2020, [cit. 2020-04-1]. Available at: `https://csrc.nist.gov/Projects/post-quantum-cryptography/round-2-submissions`

[10] WOLF, CH.; PRENEEL, B.: *Equivalent keys in multivariate quadratic public key systems.* In *Journal of Mathematical Cryptology*, pp. 375–415, 2011.

[11] BEULLENS, W.; PRENEEL, B.: *Field lifting for smaller UOV public keys.* In *Progress in Cryptology INDOCRYPT 2017: 18th International Conference on Cryptology in India*, Springer, 2017.

[12] PETZOLDT, A.; BULYGIN, S.; BUCHMANN, J.: *Multivariate Signature Scheme with a Partially Cyclic Public Key.* In: *INDOCRYPT.* 2010, vol. 6498, pp. 33 - 48. Springer, 2010.

[13] CZYPEK, P.: *LUOV. Signature Scheme proposal for NIST PQC Project (Round 2 version).* imec-COSIC KU Leuven, Belgium, 2019.

[14] DING, J.: *Rainbow - Algorithm Specification and Documentation. The 2nd Round Proposal.* University of Cincinnati, USA, 2019.

[15] *NIST: Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process.* [online]. 2016, [cit. 2020-04-1]. Available at: `https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf`

[16] BARKER, E.: *NIST Special Publication 800-57 Part 1 Revision 4.* NIST, U.S. Department of Commerce, 2016. Available at: `https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf`

Appendix A

Acronyms

ECDSA Elliptic Curve Digital Signature Algorithm

IoT Internet of Things

LUOV Lifted Unbalanced Oil & Vinegar

MC Multivariate cryptography

MQ Multivariate quadratics

NIST National Institute of Standards and Technology

OV Oil and Vinegar

PRNG Pseudo-random number generator

RNG Random number generator

UOV Unbalanced Oil & Vinegar

Appendix B

Tables of measured values

	Generation	Signing	Verifying
Luov-47-42-182	0,0070	0,0192	0,0102
Luov-7-57-197	0,0074	0,0189	0,0100
Luov-61-60-261	0,0220	0,0652	0,0305
Luov-7-83-283	0,0753	0,1164	0,0647
Luov-79-76-341	0,0890	0,2208	0,1053
Luov-7-110-374	0,1530	0,2455	0,1347
Rb-16-32-32-32	0,0907	0,0010	0,0009
Rb-16-32-32-32-cyc	0,1004	0,0010	0,0019
Rb-16-32-32-32-com	0,0932	0,0335	0,0021
Rb-256-68-36-36	1,1021	0,0129	0,0116
Rb-256-68-36-36-cyc	1,1929	0,0090	0,0151
Rb-256-68-36-36-com	1,2262	0,5935	0,0155
Rb-256-92-48-48	3,2330	0,0219	0,0232
Rb-256-92-48-48-cyc	3,6522	0,0210	0,0380
Rb-256-92-48-48-com	3,6334	1,7513	0,0384

Table B.1: Time measurement in seconds on PC

Luov-47-42-182	55497	Rb-16-32-32-32	418448
Luov-7-57-197	52339	Rb-16-32-32-32-cyc	460640
Luov-61-60-261	90701	Rb-16-32-32-32-com	367744
Luov-7-83-283	159809	Rb-256-68-36-36	1990596
Luov-79-76-341	224939	Rb-256-68-36-36-cyc	2151116
Luov-7-110-374	291712	Rb-256-68-36-36-com	1639732
		Rb-256-92-48-48	4760028
		Rb-256-92-48-48-cyc	5141804
		Rb-256-92-48-48-com	3914764

Table B.2: Memory measurement in bytes on PC

	Generation	Signing	Verification
Luov-47-42-182	0,29	0,600	0,320
Luov-7-57-197	0,52	0,810	0,470
Luov-61-60-261	0,84	2,290	0,970
Luov-7-83-283	3,84	4,880	2,330
Luov-79-76-341	4,54	11,240	4,590
Luov-7-110-374	8,77	11,150	5,240
Rb-16-32-32-32	1,86	0,040	0,030
Rb-16-32-32-32-cyc	2,04	0,030	0,290
Rb-16-32-32-32-com	2,02	1,020	0,290
Rb-256-68-36-36	20,06	0,200	0,200
Rb-256-68-36-36-cyc	22,31	0,190	1,530
Rb-256-68-36-36-com	22,33	11,230	1,570
Rb-256-92-48-48-com	67,90	33,490	3,640
ECDSA-256	0,40	0,149	0,556
ECDSA-384	0,46	0,180	0,670
ECDSA-521	0,76	0,340	1,160
RSA-2048	2,12	0,350	0,001
RSA-3072	24,46	0,852	0,004
RSA-4096	66,26	1,641	0,001
RSA-2048-sw	18,16	1,680	0,030
RSA-3072-sw	307,82	4,610	0,060
ECDSA-256-sw	0,37	0,150	0,550
ECDSA-384-sw	0,48	0,180	0,720
ECDSA-521-sw	0,82	0,300	1,190

Table B.3: Time measurement in seconds on ESP32

	Generation	Signing	Verification
Luov-47-42-182	100790	123214	84870
Luov-7-57-197	118854	120170	90814
Luov-61-60-261	125882	158418	94986
Luov-7-83-283	228654	195778	115302
Luov-79-76-341	206106	292658	110730
Luov-7-110-374	360554	272514	163178
Rb-16-32-32-32	439439	276247	272895
Rb-16-32-32-32-cyc	309431	185399	4219643
Rb-16-32-32-32-com	314499	296531	392435
Rb-256-68-36-36	2011587	1260919	1253123
Rb-256-68-36-36-cyc	1440523	757023	4219643
Rb-256-68-36-36-com	4220771	4219643	4219643
Rb-256-92-48-48-com	4220771	4219643	4219643
ECDSA-256	192354	80198	81230
ECDSA-384	192322	83666	87554
ECDSA-521	191994	85630	90826
RSA-2048	81358	83898	80358
RSA-3072	192430	87354	82390
RSA-4096	191918	90818	84246
RSA-2048-sw	192354	86522	81154
RSA-3072-sw	192482	91254	83394
ECDSA-256-sw	192430	81122	81122
ECDSA-384-sw	192426	87442	87442
ECDSA-521-sw	192438	90666	90666

Table B.4: Memory measurement in bytes on ESP32

	Generation	Signing	Verification
Rb-16-32-32-32	410424	248088	0
Rb-16-32-32-32-cyc	280408	157240	452184
Rb-16-32-32-32-com	280472	263640	359288
Rb-256-68-36-36	1982572	1233020	0
Rb-256-68-36-36-cyc	1411500	729124	2142660
Rb-256-68-36-36-com	1411564	1279580	1631276
Rb-256-92-48-48-com	3377764	2949444	3906308

Table B.5: Memory measurement of *my_ESP_malloc* in bytes on ESP32

	Public key	Secret key	Signature
Luov-47-42-182	4773	32	1332
Luov-7-57-197	11810	32	239
Luov-61-60-261	13757	32	2464
Luov-7-83-283	36200	32	337
Luov-79-76-341	27829	32	4134
Luov-7-110-374	83976	32	440
Rb-16-32-32-32	148992	92960	64
Rb-16-32-32-32-cyc	58144	92960	64
Rb-16-32-32-32-com	58144	64	64
Rb-256-68-36-36	710640	511448	156
Rb-256-68-36-36-cyc	206744	511448	156
Rb-256-68-36-36-com	206744	64	156
Rb-256-92-48-48-com	491936	64	204

Table B.6: Size of keys and signature in bytes

Appendix C

Contents of enclosed CD

```
README.md ........................ the file with CD contents description
src ...................................... the directory of source codes
  esp .......................... the implementations for esp32 platform
  mathematica ................... the implementations in Mathematica
  offline ............................... the offline reference materials
  pc ........................... the implementations for PC platform
  thesis .............. the directory of LaTeX source codes of the thesis
text .......................................... the thesis text directory
  thesis.pdf ........................... the thesis text in PDF format
```