FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

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

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Multivariate cryptography

Bc. Jan Rahm

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

April 9, 2020

Acknowledgements

I would like to thank Ing. Jiří Buček, Ph.D. for the willingness, consultation and valuable advice he gave me.

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 April 9, 2020 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

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/Diploma_Thesis`⟩.

Abstrakt

Diplomová práce se zabývá vybranými algoritmy multivariační kryptografie, zejména Unbalanced Oil & Vintage 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 & Vintage, Rainbow, Wolfram Mathematica, ESP32

Abstract

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

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

Contents

List of Figures

List of Tables

Introduction

Cryptography is one of the most needed part of modern informatics because almost everyone has something they wish to stay private. But today we can see uprise of the quantum computers which are able to decipher the conventional algorithms for cryptology. That is why a new category of post-quantum cryptography was created and one of its candidates is multivariate cryptography.

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

The final part belongs to comparison with conventional algorithms which are RSA and ECDSA.

Chapter 1

Basic terms and definitions

The chapter describes concepts and algorithms used in the thesis.

1.1 Basic terms

1.1.1 Polynomial

Polynomial p is function to which applies

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

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

1.1.2 Degree of a polynomial

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

1.1.3 Post-quantum cryptography

It refers to algorithms that are thought to be secure against an attack by a quantum computer.

But today it is not true 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 quantum computer.

1.1.4 Finite field

A finite field is a finite set which is a field. 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.

1.1.5 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.

1.1.6 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.

1.1.7 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:

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

1.1.8 Wolfram Mathematica

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

1.1.9 IoT

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.

1.1.10 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.

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

1.1.11 RSA

1.1.12 ECDSA

1.2 Multivariate cryptography

1.2.1 Definition

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

It means it is system of nonlinear polynomial equations with coefficients over a finite filed $\mathbb{F} = \mathbb{F}_q$ 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 are of degree two, they are called multivariate quadratics (MQ). Solving systems of multivariate polynomial equations is proven to be NP hard, so called MQ Problem. That is the reason why MC is often considered to be good candidate for post-quantum cryptography.

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

1.2.2 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 $p^{(1)}(\bar{x}) = \ldots = p^{(m)}(\bar{x}) = 0$.

1.2.3 Public key

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

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

1.2.4 Encryption

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

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

For decryption of ciphertext, it needs to be evaluated by private key in tree 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 condition that requires to be $m \geq n$, this way the public key $\mathcal{P}$ will be injective and decryption will output a unique plaintext.

1.2.5 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 message m. As can be seen, it is similar to description of ciphertext.

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

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

and by evaluation of 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 condition that requires to be $m \leq n$, this way the public key $\mathcal{P}$ will be surjective and private key can sign any message.

1.3 UOV

The Unbalanced Oil and Vinegar's 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, where is always more vinegar than oil variables. The signature scheme was proposed by Kipnis and Patarin in 1999.

1.3.1 Definition

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

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

$$f^{(k)} = \sum_{i=1}^{v}\sum_{j=1}^{v} \alpha_{ij}^{(k)} \cdot x_i x_j + \sum_{i=1}^{v}\sum_{j=v+1}^{n} \beta_{ij}^{(k)} \cdot x_i x_j + \sum_{i=1}^{n} \gamma_i^{(k)} \cdot x_i + \delta^{(k)}$$

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 form:

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

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

1.3.2 Security

For the security of UOV is required $v \geq 2o$ because of the attack of Kipnis and Shamir on balanced OV.[8] Besides of 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 of high security.

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

1.4 Rainbow

The Rainbow 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 compared to UOV should be in 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 number of oil variables and u is number of UOV instances.

The *central map* $\mathcal{F} : \mathbb{F}^n \to \mathbb{F}^m$ consist 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 form:

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

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

Chapter 2

Realization

Chapter describes implementation of algorithms on selected platforms which are Wolfram Mathematica, PC and microcontroller ESP32. For the last two specified the implementation is in language C++.

2.1 Wolfram Mathematica

This section describes examples in Wolfram Mathematica and step by step description of algorithms.

2.1.1 UOV

Here is description of signature scheme of UOV. This example of UOV is in fact example of balanced OV because it simplify the explanation of the algorithm.

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 is setting up of the affine map $\mathcal{T}$ with matrix A and vector b. These two parts will be later 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;
```

This block computes public key $\mathcal{P}$ by putting values of $\mathcal{T}$ inside of $\mathcal{F}$, it also simplify the expression of $p1, p2, p3$ and finally applies modulo on 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 computation of signature z for message w. Be aware that in this example is not used hash function for the message because for the example purpose it is not needed.

In[16]:=
```
w  = {{3},{6},{4}};
y1 = 1;
y2 = 0;
y3 = 6;
```

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

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 are the results after minimalizing and use of modulo:

Out[20]=
```
f1 = 6+y4+4y5+6y6
f2 = 4+y4+y5+4y6
f3 = 5+6y4+4y5+y6
```

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

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 $y = (1, 0, 6, 6, 3, 0)$, so called *pre-image* of w: $y = \mathcal{F}^{-1}(w)$. If the solution for linear system do not exist, choose different values for $(y1, y2, y3)$ and repeat steps before.

Finally use $\mathcal{T}^{-1}$ to compute signature z. For that is needed inversion of 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 $z = (4, 1, 5, 6, 3, 5)$.

Last part is check if two hashes (in this example two messages w) 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 by find under name "UOV.nb".

2.1.2 Rainbow

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

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 is setting up of the affine map $\mathcal{T}$ with matrix A and vector b which are the same as in OV example. But with addition of affine map $\mathcal{S}$ with matrix $A2$ and 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 public key $\mathcal{P}$ by putting values of $\mathcal{T}$ inside of $\mathcal{F}$ and after it makes from matrix S functions which are used for final step of computation of $\mathcal{P}$, it also simplify the expression of $pp3, pp4, pp5, pp6$ and finally applies modulo on 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 x from message 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 is computation of *pre-image* for x and also the place where is the biggest difference from OV scheme, let's start with the first step where is set up of random values for $y1, y2$ and makes their substitution in the polynomials:

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 is visible from the result that $f^{(3)}$ and $f^{(4)}$ are two equations with two unknown values.

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

It solves and with these two values $(x3, x4)$, it is possible to continue the substitution and to compute the final linear system (repeat the first and second steps):

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 computation of z, it needs to by applied $\mathcal{T}$: $z = \mathcal{T}^{-1}(y)$:

In[44]:=
```
T_1 = Inverse[A, Modulus→mod]
z = Mod[T_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 check if two hashes of the message w (in this example two messages) 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, in this example is used $u = 2, v_1 = 2, v_2 = 4, v_3 = 6 = n$. Because $u = 2$, it is example of RB with two layers. The file with implementation can by find under name "RB.nb".

2.2 Reference implementation

The section describes reference implementation of the algorithms which are selected 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 30. 1. 2019, and are written in language C or C++.

The reference implementation contains several optimized solutions. But for purpose of this Diploma thesis is only relevant solution in folder *Reference_Implementation*.

2.2.1 UOV

This implementation of UOV is in reality implementation of LUOV (Lifted UOV) which is a improvement of the UOV scheme that greatly reduces the size of the public keys. There are tree basic modification:

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

  This algorithm still produces the same distribution of key pairs, that 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 a 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 where y is in $\mathbb{F}_2^m$.[11]

- Third modification is having 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] but it does not affect the security.[10]

Source codes of version 2.1 were obtained from GitHub repository.

2.2.1.1 Adjustments

For testing purpose, I did few adjustments to source files of the LOUV. One is to simplify *Makefile* for easy generation of testing application and setting up of *path* to external library. Removed of some unnecessary files (*PQCgenKAT_sign.c*) and created *README.md* with building information. The last change is grouping of the all different settings of LOUV into one folder with 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 that is used.
- FIRST_PART_TARGET - Number of bytes used in recovery mode.
- PRNG_CHACHA/PRNG_KECCAK - If use Chacha8 or SHAKE128.
- MESSAGE_RECOVERY - Enable message recovery.

For successful build on Ubuntu operating system is needed library *Keccak Code Package* in *home* folder:

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

Additionaly tool *xsltproc*:

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

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

2.2.2 Rainbow

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

- Classic - Typical/Classic implementation of Rainbow.
- Cyclic - The variant is motivated by Petzoldt's cyclic Rainbow scheme[12] who developed technique to insert a matrix into public key and to compute a corresponding private key. It allows to create major parts of the public key from a seed using a PRNG. But this variant includes some kind of bug where the verification of signature fails.
- Compressed - This variant is similar to Cyclic variant but the private key is stored in the form of bit seed.

2.2.2.1 Adjustments

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

- _RAINBOW16_32_32_32
- _RAINBOW256_68_36_36
- _RAINBOW256_92_48_48

These flags can not be used together, only one at the time can be valid. It specify which setting will be used: finite filed $\mathbb{F}_{16}$ 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 the time. It specify which variant of Rainbow to be used.

Last change is adding a *test.c* file from LOUV implementation for consistent testing purpose.

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

2.2.3 Test file

For testing purpose there is file *test.c* which also serves as main entry point of the final application. Both of the implementation has the same structure and it can by basically described as followed:

First step is generation of public and private keys:

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

Generation signature *sm* (it also contains the message) of message m:

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

Verify signature *sm* and put the message from it to $m2$:

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

Final step is verification if the message m is equal to $m2$.

2.3 ESP32 implementation

For the implementation of algorithms on microcontroller was selected ESP32-LyraT of version 4.3 from company Espressif Systems.

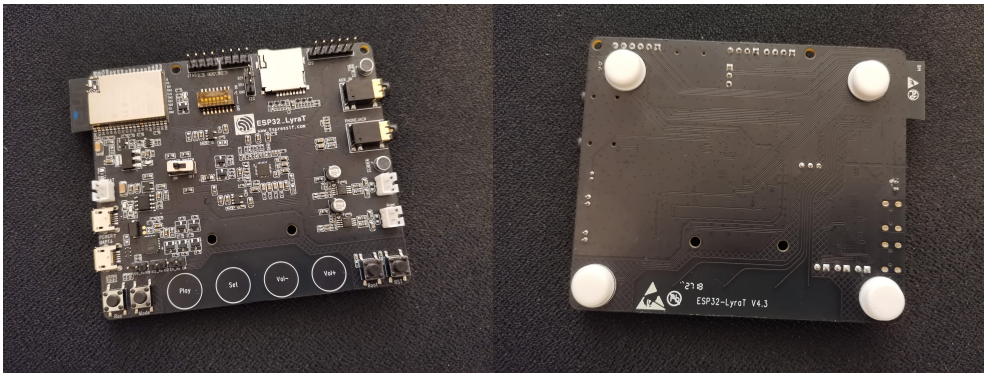

Figure 2.1: ESP32-LyraT

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 inbuild external memory of the capacity 8MB. With this size of memory, it should be without any problems to implement selected algorithms. But in reality, only 4MB are able to be used in the implementation, more information can be found in section

2.3.1.2. Added with excellent documentation at "docs.espressif.com" it was relatively easy to choose this platform for development.

The implementations are the same as reference implementations with my adjustments. What I did is that I tried to port them on ESP32 platform.

2.3.1 Setup of environment

First step of development on ESP32 is to set up of environment. This setting I did on Ubuntu 19.10 operating system, starting with downloading of 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 group *dialout* because the user needs to get read and write access to the serial port over USB.

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

Download software libraries provided by Espressif, Espressif IoT Development Framework (esp-idf), to folder "$HOME/esp". I used release version 4.1:

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

Be **aware** that all of the setting is related to absolute path "$HOME/esp".

Install tools used by "esp-idf" to directory "$HOME/.espressif":

```
cd \$HOME/esp/esp-idf
./install.sh
```

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

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

But I added it to file *.bashrc*:

```
echo ". $HOME/esp/esp-idf/export.sh" >> $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, copy the project "src/esp/luov" (in the Diploma thesis sources) to "$HOME/esp/luov". In this folder it is possible to build it:

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

The switch "-n" will stop treat 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 be different it depends on to which is ESP32-LyraT connected to.

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

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

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

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

It is possible to combine 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 it is 32-bit processor that means, if external RAM is enabled, only 4MB 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 size of the stack for the application. Setting 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 in menuconfig as external RAM.

2.3.2 Project description

Base project of esp-idf is composed of the:

- build - A folder where the output of 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 starting file for *cmake*.
- sdkconfig - A setting of *menuconfig*.
- sdkconfig.default - A default setting of *menuconfig*.

But for good convenience I added to this structure a file:

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

The project can be build by using *make* or *cmake*. For simplicity and recommendation in documentation, also almost every example is written in it, I decided to use *cmake*.

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

2.3.3 LUOV

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

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

Function is for registering project component to internal build structure of *idf* API.

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

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

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

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

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

What is possible to do and what I also did, is to change the RNG to hardware RNG of ESP32. But there is condition in which Wi-Fi or Bluetooth needs to be enabled otherwise it can not be considered a true random number generator but only pseudo-random number generator.

Implementation can be found in folder *src/esp/luov*.

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

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

should have minimal influence on signing algorithm. But to by 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 folder *main* with the same formalities as *CMakeList.txt* for LUOV, see section 2.3.3. The most eye catching difference is set up of different kinds of *malloc*, see section 2.3.4.2 for more information.

The implementation requires component *wolfssl* for PRNG. In this case is build the whole *esp-wolfssl* project which is simply added through *PRIV_REQUIRES* in *main/CMakeList.txt*. Here is important to mention that Rainbow reference implementation contains two PRNG (see *utils_prng.c*). One from NIST standard, it is the same RNG as in LUOV reference implementation, and second PRNG which use hash SHA function. Because the RNG form NIST standard was not possible for me to run on ESP32-LyraT, I change it to second PRNG and I deleted from source codes the implementation of the first.

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

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

Implementation can be found in folder *src/esp/rb*.

2.3.4.1 Optimization

I did two memory optimizations of Rainbow implementation for ESP32:

- First I found potential big allocation of memory and switch them from stack to heap. Candidates to change were temporary helpful variables of keys (*sk_t*) in computation of keys from seeds.
- Second I created *my_ESP_malloc* and *my_ESP_free* functions, see 2.3.4.2. With these I found that temporary variables (for example *sk_t* tempQ*) were using 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"* for details of optimization.

With these two optimizations there was already enough memory and I was able to run *_RAINBOW256_92_48_48* implementation in compressed form. I was also able to run the same implementation in cyclic form but there is same kind of bug and it was making segmentation fault. I am suspecting that it is the same bug which causes 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 ESP32 2.3.3.2, which can be enabled by flag:

- MEM_MEASUREMENT

Because I needed better technique for memory allocation on heap, I created my own allocation information table. It is simple small array, I expect small number of allocation at one point at time, which holds information of pointer and its size. It can be enabled by 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 every time when there is allocation od deallocation it will print its information. This information table helped me to identify potential places in source code of RB for memory optimization. For implementation details see file *malloc.c*.

The reference implementation use *aligned_alloc* function for allocation of memory but ESP32 with toolchain 8.2 which I am using has issue with this function and it is not possible to use for now. That is the reason why I change it to classic *malloc* function.

2.4 Conventional algorithms

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

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

2.4.1 RSA

Test implementation of RSA starts with initialization of context:

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

Second step is generation of keypair:

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

Generation *hash* of message *m*:

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

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

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

Verify signature *sm* of *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 flags:

- KEY_SIZE - Size of public key 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 initialization of context:

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

Second step is generation of keypair:

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

Generation *hash* of message *m*:

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

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

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

Verify signature *sm* of *hash*:

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

The implementation can be set with flags:

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

Chapter 3

Testing and discussion

This chapter contains measurement and testing of algorithms, their comparison and discussion about their implementation. It mainly focus 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 computer with parameters:

- OS: Ubuntu 19.10
- CPU: Intel Core i5-7300HQ
- Clock Speed: 2.50GHz
- RAM: 8 GB
- Number of cores: 4
- Compilator: GCC 8.3.0

Main goal of this measurements and implementations is to have comparable data of reference implementations of LUOV and RB on PC because in reference materials there are measured on different processors and only in number of processor cycles. I chose to make the measurement of time (in seconds) and memory consumption (in bytes).

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

3.1.1 Signature schemes

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

Category II:

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

Category IV:

- 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 represent a difficulty through how many hardware gates needs to be used to be able to break the security scheme.[15] Overview:

Category	$\log_2$ classical gates	$\log_2$ quantum gates
I	143	130/106/74
II	146	
III	207	193/169/137
IV	210	
V	272	258/234/202
VI	274	

Table 3.1: NIST security categories

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

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

3.1.2 Time complexity

The next two graphs display individual stages of the signature scheme and its time it takes to complete each of the stages. Namely generation, signing and verification.

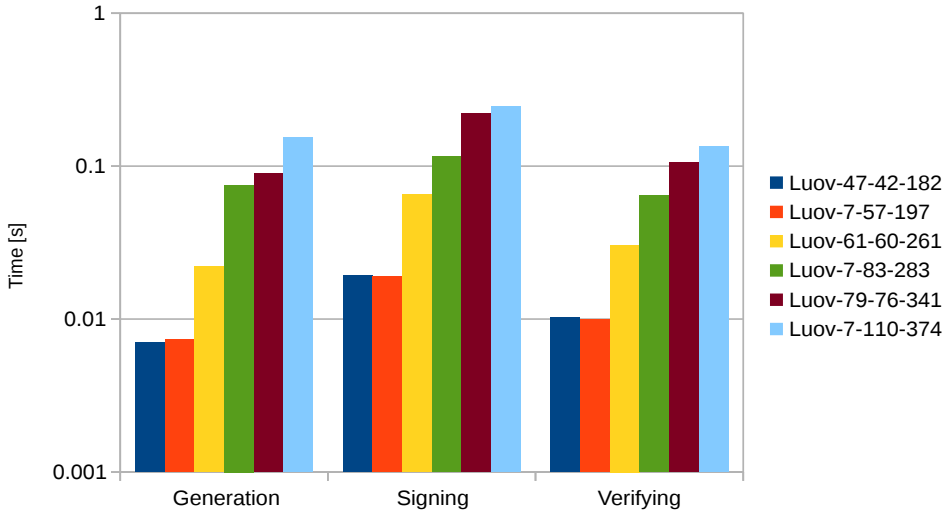

Figure 3.1: Comparison of LUOV on PC

How it can be seen on 3.1 the stronger the security

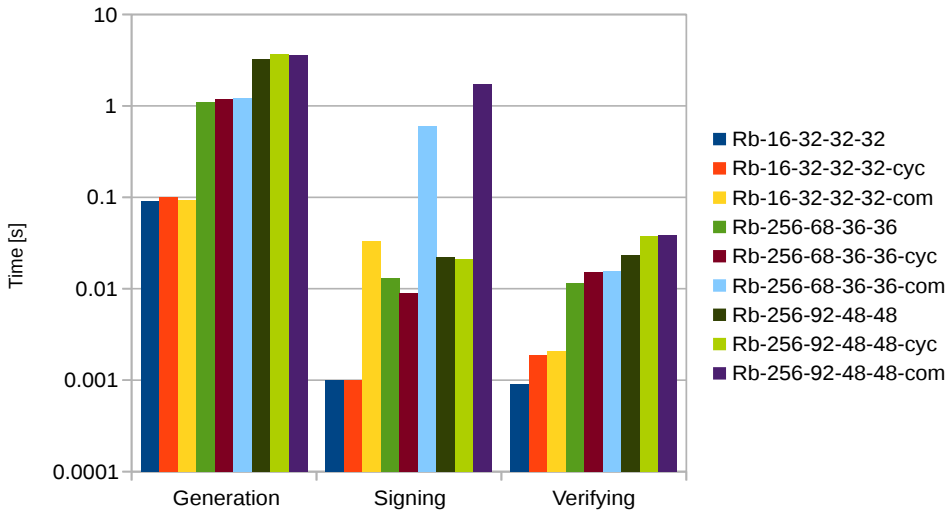

Figure 3.2: Comparison of RB on PC

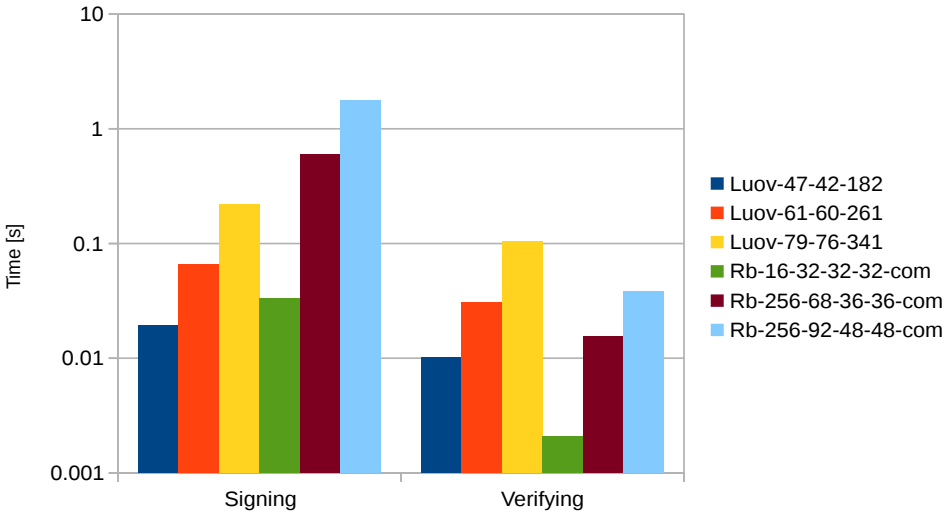

Figure 3.3: Comparison of PC implementations

3.1.3 Memory complexity

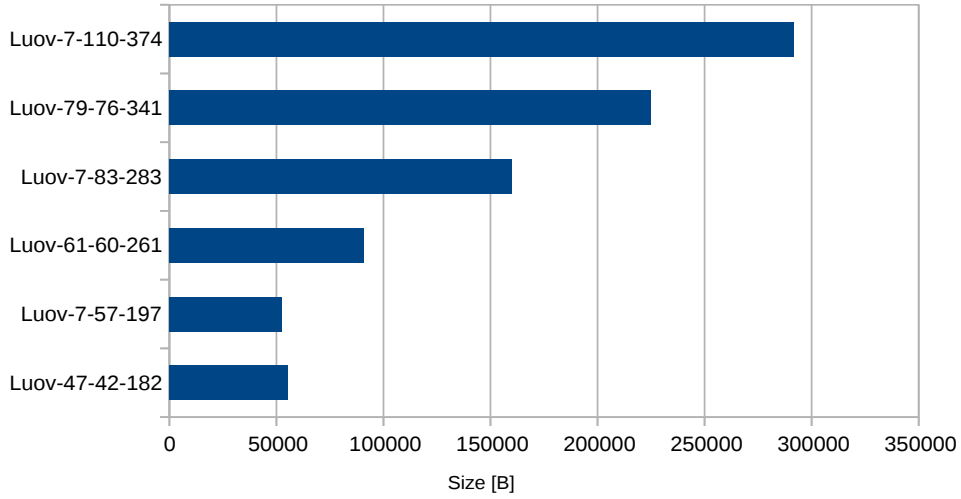

Figure 3.4: Memory requirement of LUOV on PC

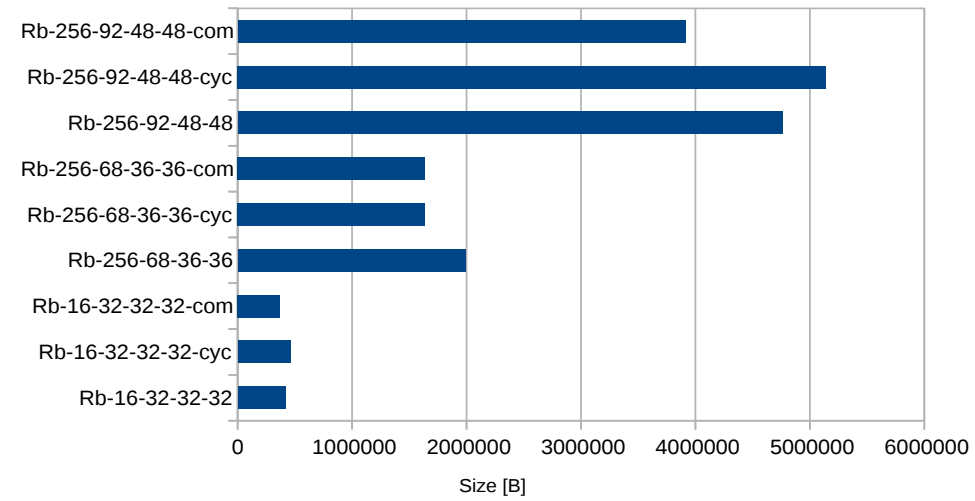

Figure 3.5: Memory requirement of RB on PC

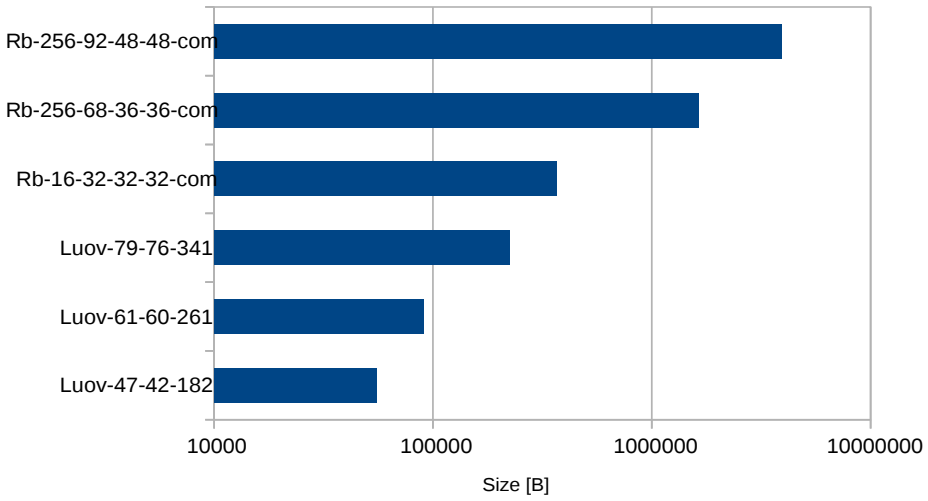

Figure 3.6: Comparison of PC implementations

3.2 ESP32

3.2.1 Time complexity

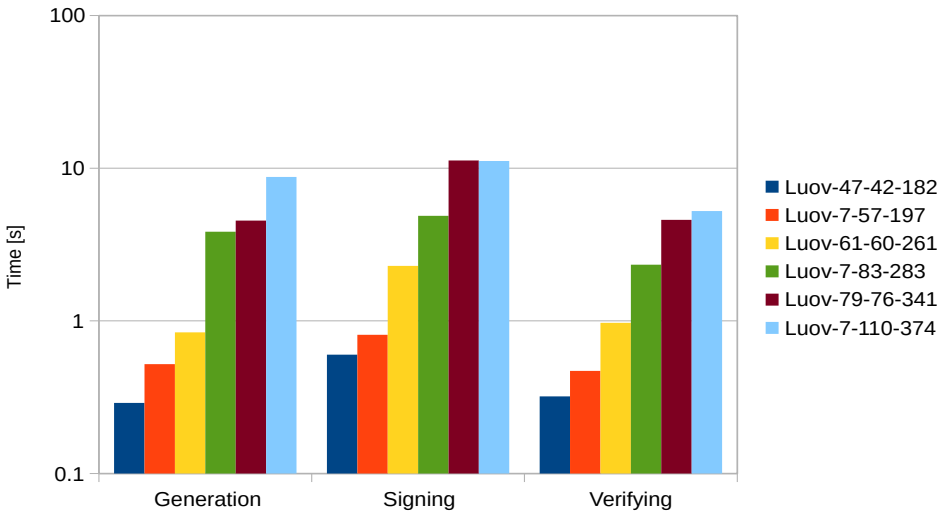

Figure 3.7: Comparison of LUOV on ESP32

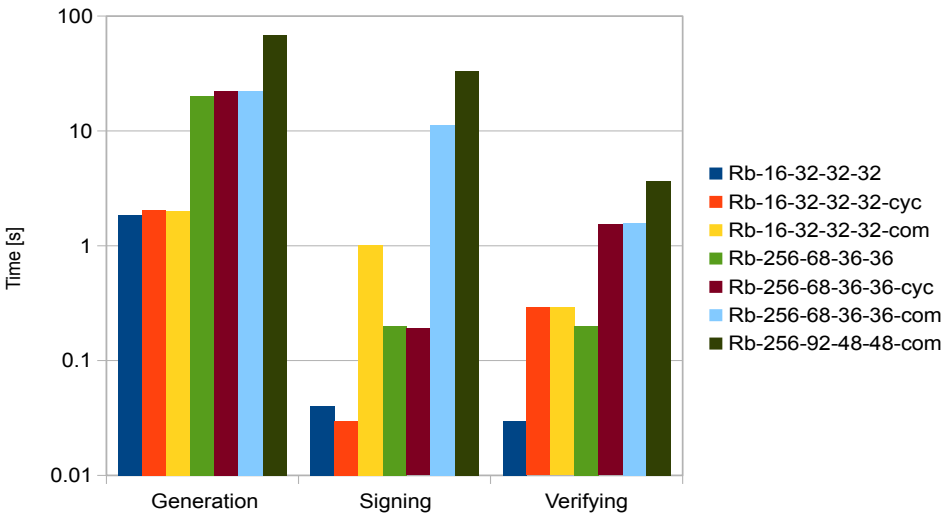

Figure 3.8: Comparison of RB on ESP32

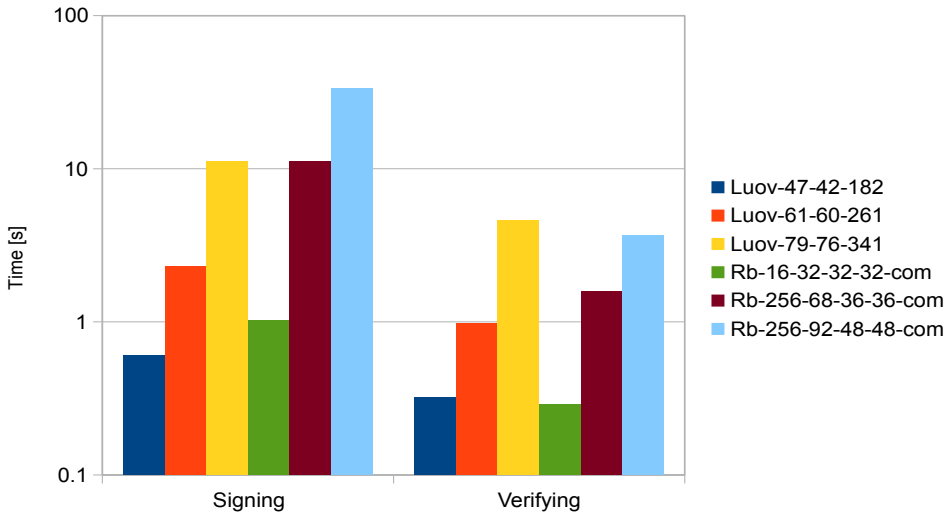

Figure 3.9: Comparison of ESP32 implementations

3.2.2 Memory complexity

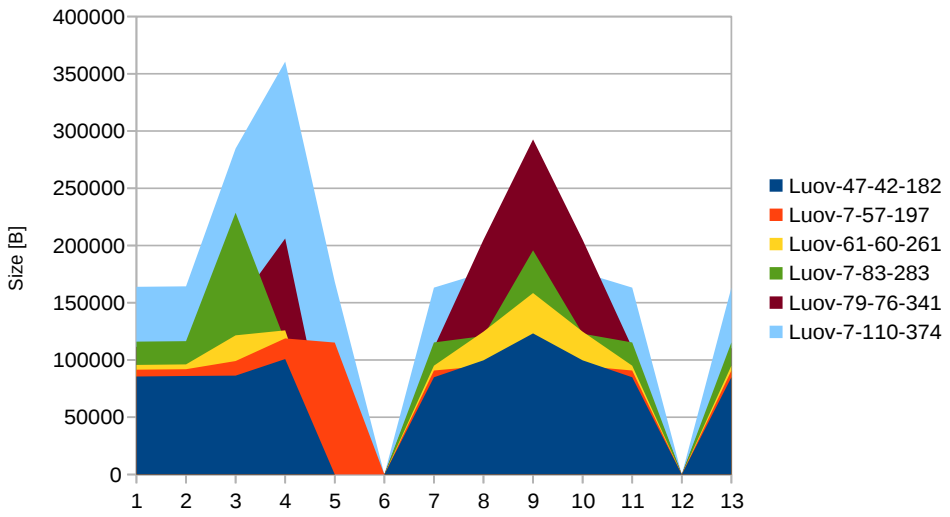

Figure 3.10: Memory requirement of LUOV on ESP32

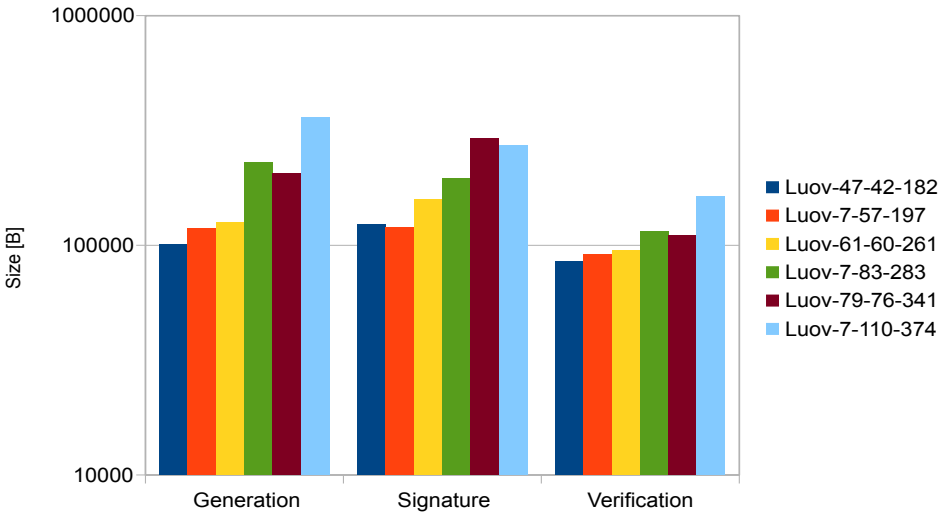

Figure 3.11: Memory requirement of LUOV on ESP32

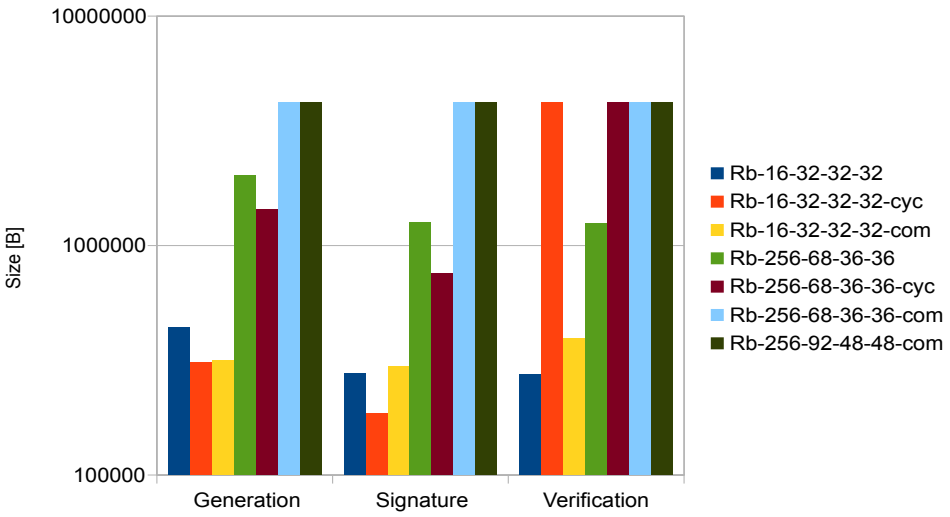

Figure 3.12: Memory requirement of RB on ESP32

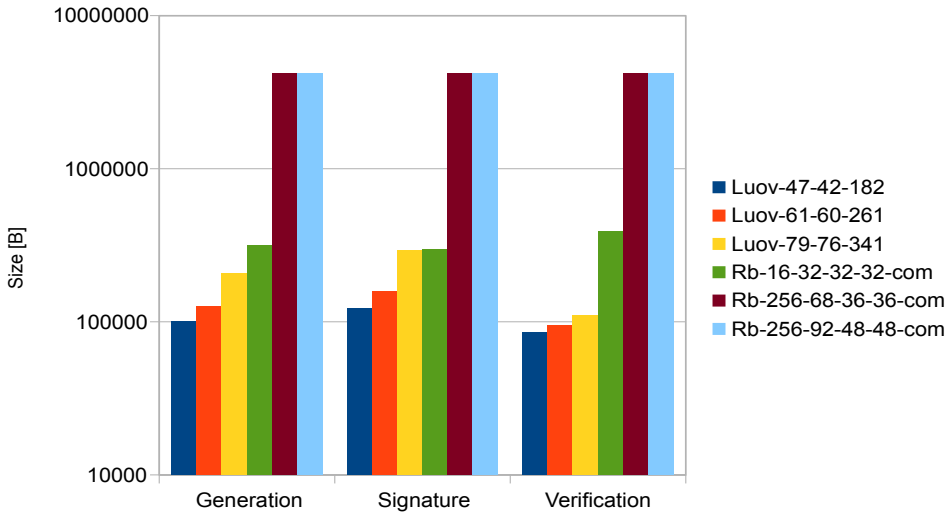

Figure 3.13: Memory requirement of implementations on ESP32

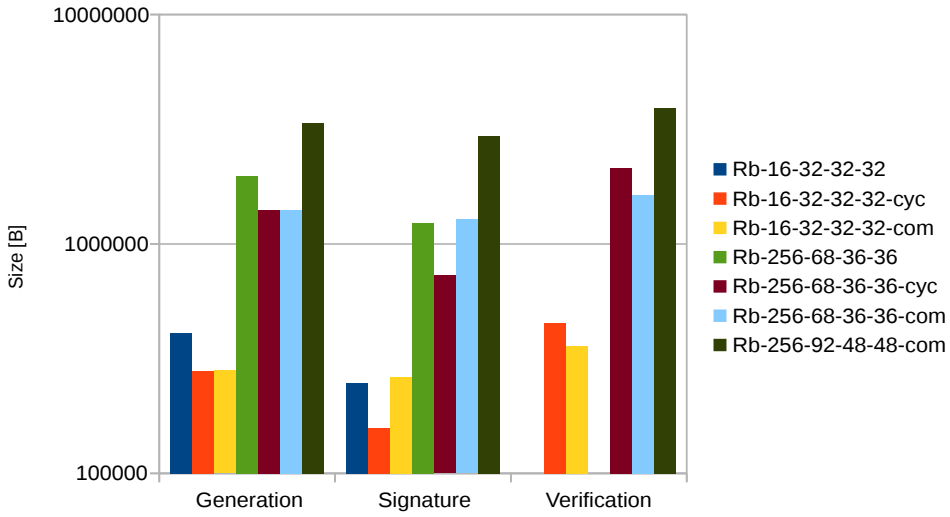

Figure 3.14: Memory requirement of my_ESP32_malloc

3.2.3 Keys & signature

In this section is comparison of size of signature for different settings of algorithms. Must be noted that the size of signature of ESP32 implementations are the same as PC implementation. I just noted them from the ESP32 implementations.

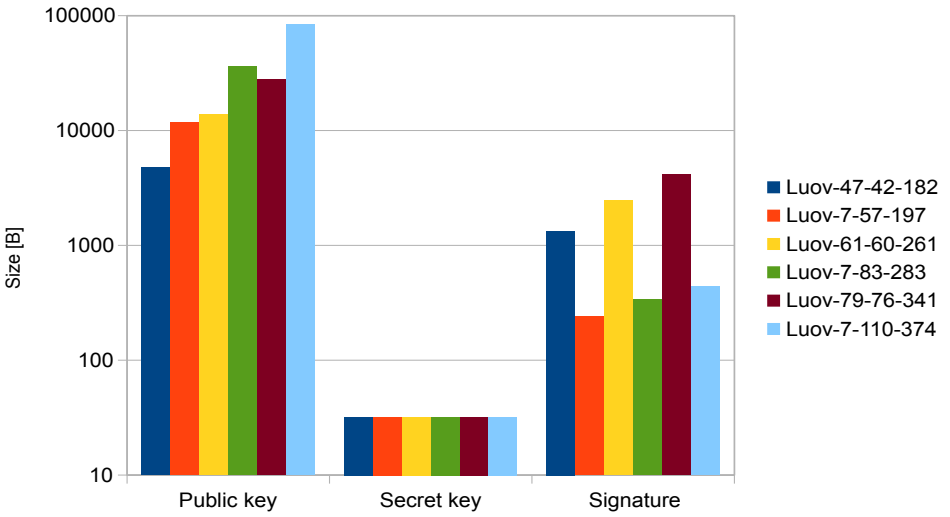

Figure 3.15: Size of signature of LUOV on ESP32

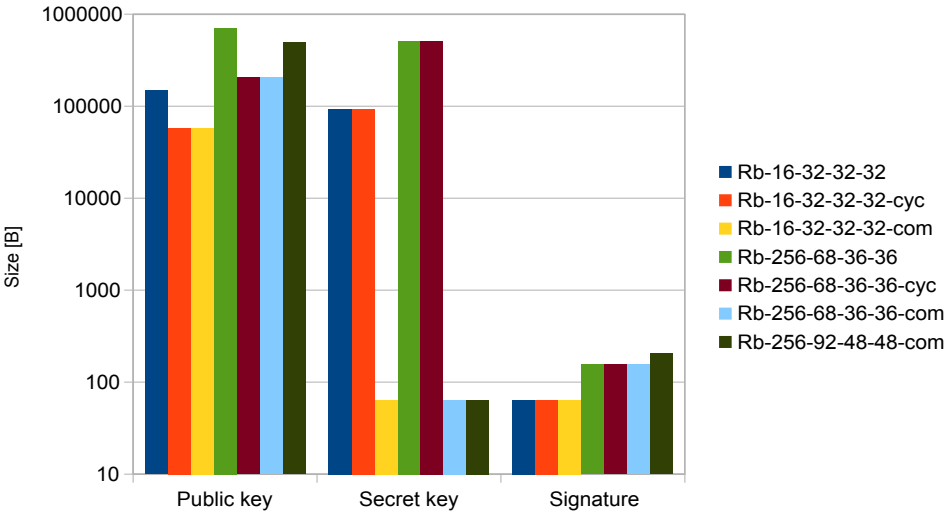

Figure 3.16: Size of signature of RB on ESP32

3.3 Conventional algorithms

RSA, ECDSA

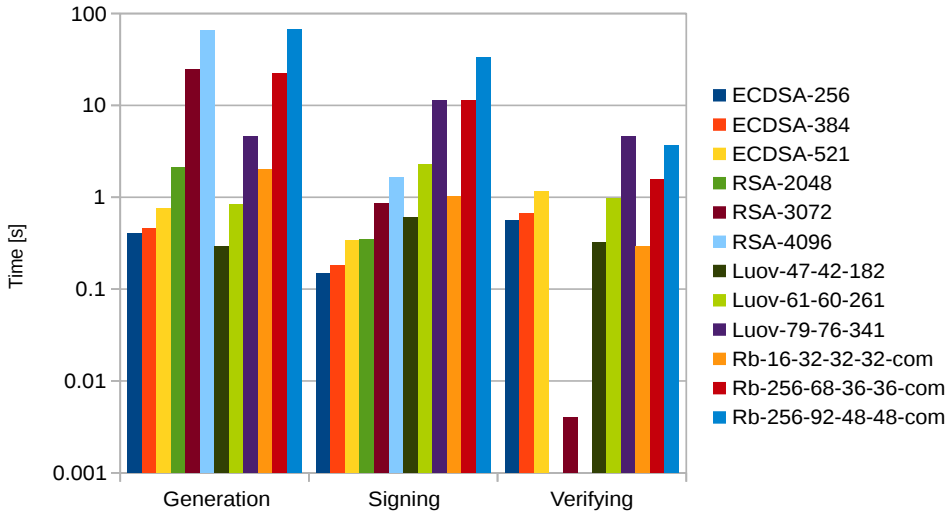

Figure 3.17: Comparison with conventional algorithms

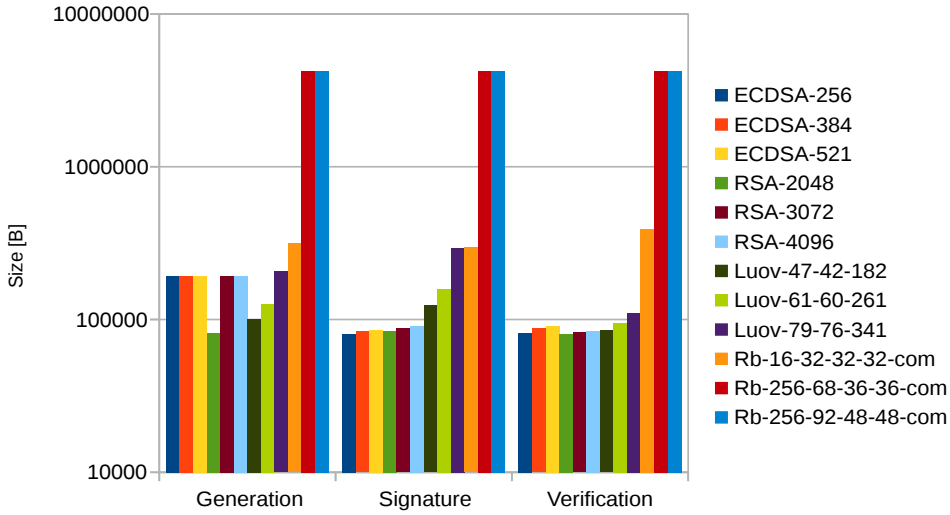

Figure 3.18: Memory requirement comparison with conventional algorithms

Conclusion

How good I was...; LOUV is much more better...

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]. 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]. 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]. 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]. 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]. 2020, [cit. 2020-04-1]. At: `https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf`

Appendix A

Acronyms

ECDSA Elliptic Curve Digital Signature Algorithm

IoT Internet of Things

LUOV Lifted Unbalanced Oil and 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 and Vinegar

Appendix B

Contents of enclosed CD

- `README.md` ..... the file with CD contents description
- `exe` ..... the directory with executables
- `src` ..... the directory of source codes
  - `esp` ..... implementation for esp32 platform
  - `mathematica` ..... implementation in Mathematica
  - `offline` ..... offline reference materials
  - `esp` ..... implementation 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