Hardware Accelerated Post Quantum Cryptography

Master Thesis Fall 2022

Master of Science in Engineering
Electrical Engineering

Author
Michael Schmid
Supervisor
Prof. Dr. Paul Zbinden
University of Applied Sciences of Eastern Switzerland
Advisor
Prof. Dr. Tao Wei
University of Rhode Island

University of Rhode Island
January 27, 2023

This document is typeset with LaTeX and Ti*k*Z

The layout is based on the lecture notes of Prof. Dr. Andreas Müller' *Partial Differential Equations*

Abstract

Introduction In the last decade, there has been a lot of research on quantum computers. Quantum computers are using quantum mechanical phenomena to solve various mathematical problems that are infeasible for traditional computers. In theory, those quantum computers could break many of the standard public key cryptosystems we are using today. This would compromise the confidentiality and integrity of nearly every form of digital communication. Post quantum cryptography is the term used for cryptographic algorithms which are resistant against attacks for both quantum and conventional computers. Small quantum computers are working in a non-commercial research environment, but to successfully attack cryptosystems, much larger quantum computers will need to be built. It is nearly impossible to predict the arrival of such large scale quantum computers, but the time of arrival is more or less certain [1].

The National Institute of Standards and Technology of the United States of America (NIST) began the process of standardizing post-quantum cryptography in 2017. After four rounds, the first algorithms were selected for standardization in July 2022. The hardware acceleration with Field Programmable Gate Arrays (FPGAs) has been done for all algorithms but one, namely Fast-Fourier Lattice-based Compact Signatures over NTRU (Falcon). For Falcon, only the signature verification has been accelerated. Due to the use of floating point operation and recursive functions the acceleration of the key generation and signature generation is still missing.

Approach After a deeper understanding of the theory of the Falcon algorithm, the implementation on an FPGA could be started. For the implementation, the High Level Synthesis (HLS) was preferred over a traditional Hardware Description Language (HDL) like Verilog or Very High Speed Integrated Circuit Hardware Description Language (VHDL). The reference implementation of the algorithm was written in the programming language *C*. HLS takes C code and translates into a Register Transfer Level (RTL) hardware description, therefore the reference implementation is directly used in HLS. Many minor modifications to the code had to be done in order to be compliant with HLS. All recursive functions had to be rewritten into an iterative version.

Conclusion All important parts of the Falcon algorithm could be implemented successfully on the FPGA. To the best of our knowledge, key generation and signature generation were implemented on an FPGA for the first time in this work. Further optimizations are necessary in terms of improving both hardware utilization and performance.

Acknowledgement

I would like to express my great appreciation to my supervisor, Prof. Dr. Paul Zbinden, for his valuable and constructive support. A special thanks also goes to Prof. Dr. Tao Wei, the co-examiner of this project. They have helped me in many ways throughout the project with their extensive experience.

Prof. Dr. Wei generously sponsored me to write my master's thesis at the University of Rhode Island. The collaboration with Dr. Wei is based on the carefully maintained friendship and cooperation between Dr. Zbinden and Dr. Wei. I was warmly welcomed and immediately felt respected and integrated into Dr. Wei's team. Another mention goes to Zhenyu Xu and Miaoxiang Yu, who advised me on professional and personal issues, as well as my seatmates James Morris and McKensie Sherlock.

Undoubtedly, the support from my friends from the living and learning community of the International Engineering Program at the University of Rhode Island was immense. ⊢

Finally, I have to thank my girlfriend and my parents who not only hosted me during the last weeks of this project, but also kept me spirited throughout my stay abroad.

Contents

Abbreviations

VHDL Very High Speed Integrated Circuit Hardware Description Language

HDL Hardware Description Language

CPU Central Processing Unit

FPGA Field Programmable Gate Array

DSP Digital Signal Processor

RTL Register Transfer Level

Falcon Fast-Fourier Lattice-based Compact Signatures over NTRU

HLS High Level Synthesis

AES Advanced Encryption Standard

PQC Post-Quantum Cryptography

WSL Windows Subsystem for Linux

TCL Tool Command Language

GUI Graphical User Interface

FPU Floating Point Unit

FFT Fast Fourier Transform

KAT Known Answer Test

NIST National Institute of Standards and Technology of the United States of America

ASIC Application-Specific Integrated Circuit

Chapter 1

Introduction

Background *In recent years, there has been a substantial amount of research on quantum computers - machines that exploit quantum mechanical phenomena to solve mathematical problems that are difficult or intractable for conventional computers. If large-scale quantum computers are ever built, they will be able to break many of the public-key cryptosystems currently in use. This would seriously compromise the confidentiality and integrity of digital communications on the Internet and elsewhere. The goal of post-quantum cryptography (also called quantum-resistant cryptography) is to develop cryptographic systems that are secure against both quantum and classical computers, and can interoperate with existing communications protocols and networks. The question of when a large-scale quantum computer will be built is a complicated one. While in the past it was less clear that large quantum computers are a physical possibility, many scientists now believe it to be merely a significant engineering challenge. Some engineers even predict that within the next twenty or so years sufficiently large quantum computers will be built to break essentially all public key schemes currently in use. Historically, it has taken almost two decades to deploy our modern public key cryptography infrastructure. Therefore, regardless of whether we can estimate the exact time of the arrival of the quantum computing era, we must begin now to prepare our information security systems to be able to resist quantum computing.* [1]

Motivation When this new generation of cryptographic algorithms is not only standardized but also used in daily applications, an efficient, fast and cost-effective implementation is of great importance. An implementation of these algorithms on FPGAs is an important step towards achieving these criteria. If a sufficient RTL version exists, it can be further used for Application-Specific Integrated Circuits (ASICs) development.

The NIST began the process of post-quantum cryptography standardizing in 2017. After four rounds, the first algorithms were selected for standardization in July 2022. The hardware acceleration with FPGAs has been done for all algorithms except Falcon. For Falcon, only the signature verification has been accelerated. Due to the use of floating point operation and recursive function calls, the implementation of the key and signature generation is still missing.

A working FPGA implementation would be an important step for Falcon in order not to lose significance to the other algorithms.

Content Chapter 2 presents fundamentals about common cryptography and an insight about Post-Quantum Cryptography (PQC). In chapter 3, the methods on how to implement Falcon on

an FPGA with HLS is shown. Chapter 4 states the results and chapter 5 presents the conclusion of the project.

Chapter 2

Fundamentals

2.1 Cryptography

This section is heavily inspired by [2].

Basically, cryptography has the goal of secure communication with four main objectives.

- *Confidentiality* describes the property to keep information secure from all but those authorized to access the data. From physical protection of the data to encrypting a message with mathematical algorithms to make them unreadable without further knowledge.
- *Data integrity* describes the unauthorized manipulation of information. To get data integrity one must be able to detect data manipulation.
- *Authentication* is related to identification, it applies to both entities and information itself. Two individuals communicating with each other should be able to identify themselves. The information needs to be authenticated as the origin with its properties such as date of origin, data, time sent etc.
- *Non-Repudiation* describes that an entity cannot deny the actions performed in a secure communication. For example, an entity sends authentication to delete some data from a server and later denies that authentication was granted. Non-Repudiation provides a system in which this is not possible.

There are a lot of cryptographic tools to provide above described goals. There are encryption schemes, hash functions and digital signature schemes. These schemes have to be evaluated with various criteria:

- *Level of security* is the most important property but also rather difficult to quantify. Normally, the number of operations needed to break the cryptographic function by the best method currently known is given.
- *Functionality* needs to be given by the combination of different schemes to get the needed level of security.
- *Performance* describes the efficiency of a scheme depending on its functionality. As an example, how many bits can an encryption schemes can encrypt per second.

- *Implementation*. A scheme should be relatively easy to implement both in software and hardware.

The importance of all criteria is depending heavily on the application.

2.1.1 Functions

Definition 2.1. A function is defined by two sets, the domain X and codomain Y, and the rule f, which assigns to each element in X precisely one element in Y. f: $X \longrightarrow Y$, if $y \in Y$ and $x \in X$ then $y = f(x)$. The image of x is the element in Y where $y = f(x)$. If $y \in Y$, then a preimage of y is an element of $x \in X$ for which $f(x) = y$. The set of all elements in Y which have at least one preimage is called the image of f donated $\mathrm{Im}(f)$.

Example. Consider $X = \{a, b, c\}$, $Y = \{1, 2, 3, 4\}$ and $f(a) = 2$, $f(b) = 3$, $f(c) = 1$. The preimage of 2 is a and the image of f is $\{1, 2, 4\}$ ○

Definition 2.2. A function is one-to-one if each element in Y is the image of at most one element in X

Definition 2.3. If a function $f : X \longrightarrow Y$ is one-to-one and $\mathrm{Im}(f) = Y$, then f is called a bijection

Definition 2.4. If f is a bijection from $X \longrightarrow Y$, a bijection g $Y \longrightarrow X$ can be found. For each $y \in Y$ define $g(y) = x$ where $x \in X$ and $f(x) = y$. The function g is called inverse function of f and is denoted by $g = f^{-1}$

2.1.1.1 One Way Functions

One way functions are of major importance in cryptography.

Definition 2.5. $f : X \longrightarrow Y$ is called a one way function if for all $x \in X$, $f(x)$ is easy to compute but for most $y \in Im(f)$ is computationally infeasible to find any $x \in X$ such that $f(x) = y$.

Example. $X = \{1, 2, 3, \ldots, 16\}$, $f(x) = r_x \, \forall \, x \in X$, where r_x is the remainder when 3^x is divided by 17.

x	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
f(x)	3	9	10	13	5	15	11	16	14	8	7	4	12	2	6	1

With $f(x)$ it is easy to find y. But without the table it is quite hard to find x given $f(x) = y$. But not in all cases, if $y = 3$, it is clear that $x = 1$. ○

2.1.1.2 Trapdoor One Way Functions

A trapdoor function is a one-way function whose inverse can be found with additional information in a reasonable amount of time.

Example. Primes $p = 48611, q = 53993$ from $n = pq = 2624653723$ and let $X = \{1, 2, 3 \ldots, n-1\}$, $f(x) = r_x \, \forall \, x \in X$, where r_x is the remainder when 3^x is divided by 17. For instance $f(2489991) = 1981394241$ since $2489991^3 = 5881949859 \cdot n + 1981394214$ Same as in the previous example, the direct way is easy but to find x for a given y is very hard. But if the factors p and q of n are known, there is an efficient algorithm to compute the inverse. This is called the integer factorization problem. ○

These one way and trapdoor functions are the foundation of public-key cryptography.

2.1.2 Basic Terminology and Concepts

Encryption domains and codomains

- $\mathcal{A}$ the alphabet is a finite set. For example $\mathcal{A} = \{0, 1\}$, the binary alphabet.
- $\mathcal{M}$ denotes the message space. $\mathcal{M}$ consists of a string of symbols from $\mathcal{A}$, an element of $\mathcal{M}$ is called the plaintext.
- C denotes the ciphertext space, an element of C is called ciphertext.

Encryption and decryption functions

- $\mathcal{K}$ is the key space, an element of $\mathcal{K}$ is a key.
- Each element $e \in \mathcal{K}$, determines a bijection from $\mathcal{M}$ to C donated by E_e. E_e is the encryption function.
- Each element $d \in \mathcal{K}$, determines a bijection from C to $\mathcal{M}$ donated by D_d. D_d is the decryption function.
- An encryption scheme consists of $\{E_e : e \in \mathcal{K}\}$ of the encryption functions and $\{D_d : d \in \mathcal{K}\}$ of decryption functions.
- e and d is a key pair.

Example. $\mathcal{M} = \{m_1, m_2, m_3\}, C = \{c_1, c_2, c_3\}$. There are $3! = 6$ bijections from $\mathcal{M}$ to C with $\mathcal{K} = \{1, 2, 3, 4, 5, 6\}$, the six encryption functions are denoted by E_i, $1 \leq i \leq 6$. Fig. 2.1 illustrates the functions. ○

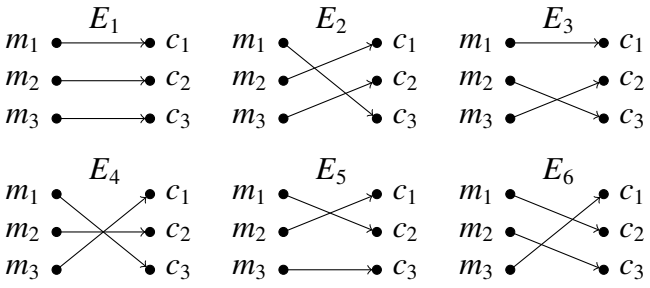


Figure 2.1: Simple example for six encryption functions

Communication participants The common terminology in a cryptographic communication (illustrated in Fig. 2.2):

- An entity is someone which sends, receives or manipulates.
- A sender is the legitimate transmitter of information and is called Alice
- A receiver is the intended receiver of information and is called Bob
- An adversary is an entity which tries to defeat the information security between Alice and Bob and is called Eve

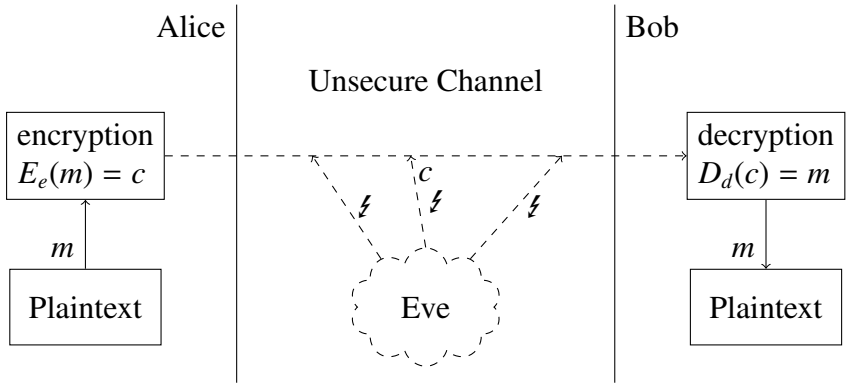


Figure 2.2: Two party communication with the common terminology

Security Fundamental in cryptography is that the sets $\mathcal{M}, \mathcal{C}, \mathcal{K}, \{E_e : e \in \mathcal{K}\}, \{D_d : d \in \mathcal{K}\}$ are public knowledge. In a two-way communication the only thing to keep secret is a specific key pair (e, d).

2.1.3 Symmetric Key Encryption

Definition 2.6. An encryption scheme with $\{E_e : e \in \mathcal{K}\}, \{D_d : d \in \mathcal{K}\}$ is called symmetric-key if for each key pair (e, d) it is easy to determine d knowing e.

In most symmetric key encryption schemes $e = d$, the basics are shown in Fig. 2.3.

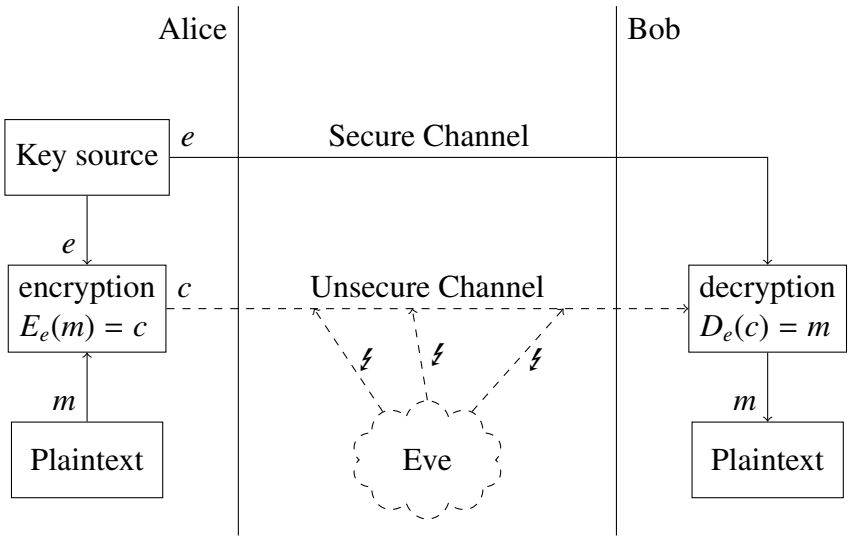


Figure 2.3: Symmetric key encryption

2.1.4 Digital Signatures

Digital signatures are imperative for authentication, authorization, and non-repudiation.

- $\mathcal{M}$ is a set of messages to be signed
- $\mathcal{S}$ is a set called signatures

- S_A is a transformation from $\mathcal{M}$ to $\mathcal{S}$ by Alice. $\mathcal{S}_A$ is kept secret by Alice
- V_A is the verification transformation from the sets $\mathcal{M} \times \mathcal{S}$ to the set $\{true, false\}$ and is publicly known. It is used by Bob to verify the signatures created by Alice

Signing Procedure Alice crates a signature for $m \in \mathcal{M}$ by:

- compute $s = S_A(m)$
- transmit (m, s)

Verification Procedure Bob verifies the pair (m, s) by:

- Obtain the verification function V_A from Alice
- Compute $u = V_A(m, s)$
- Accept the message if $u = true$ reject otherwise

2.1.5 Public Key Cryptography

Paradoxically, for secure communication, it is essential that secure channels between Alice and Bob are not required.

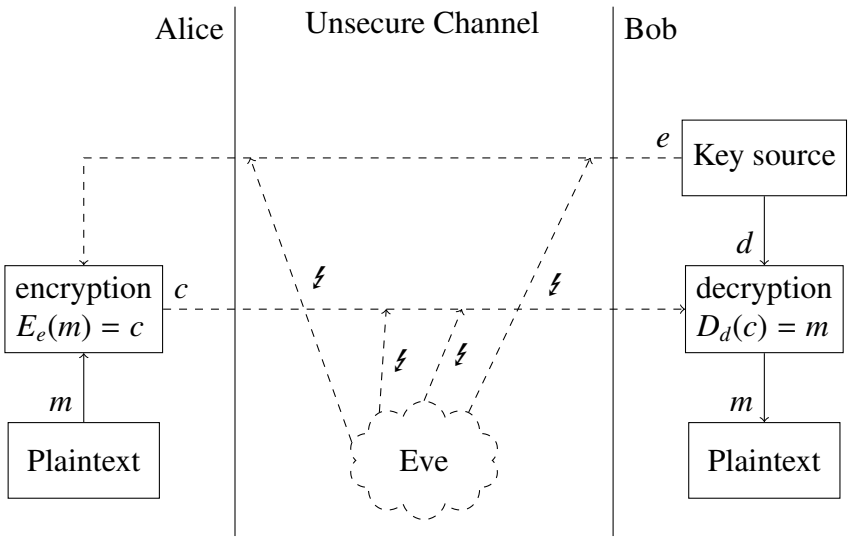


Figure 2.4: Public key cryptography

In public key cryptography, the key e in the pair (e, d) is made public. The schemes must provide that it is infeasible to compute d from e.

Fig. 2.5 illustrates the need for authentication between the two communicating entities. An attacker, in this case Eve, can easily gain access to the message and even manipulate it. By impersonating Bob, Eve sends a public key e' to Alice, which she assumes is Bob's key. Alice then sends the message to Eve, and Eve re-encrypts it with Bob's real public key and sends the message to Bob. With this attack, Alice and Bob communicate with each other, but do not know that Eve is eavesdropping.

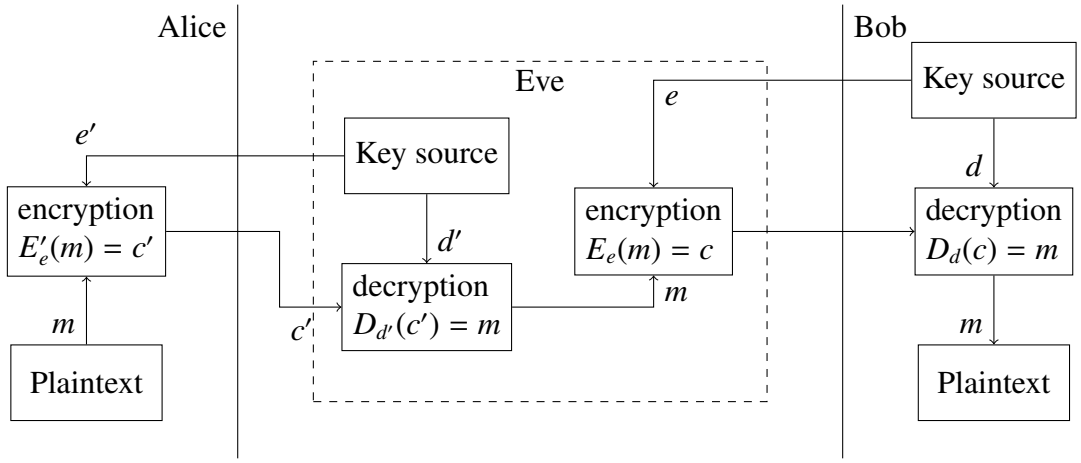


Figure 2.5: Impersonation attack in which Eve pretends to be Bob in the eyes of Alice, and Eve pretends to be Alice in the eyes of Bob

2.1.6 Mathematical Background

The following is some mathematical background commonly used in cryptography:

The set of integer is donated by $\mathbb{Z}$.

Definition 2.7. Let $a, b \in \mathbb{Z}$, a divides b if there is a $c \in \mathbb{Z}$ such that $b = ac$. If a divides b, then $a|b$.

Definition 2.8. $c \in \mathbb{Z}$ is a common divisor of a and b if $c|a$ and $c|b$.

Definition 2.9. A positive integer d is the greatest common divisor of integers a and $bd = gcd(a, b)$, if

- d is a common divisor of a and b
- whenever $c|a$ and $c|b$ then $d|c$

Definition 2.10. $a, b \in \mathbb{Z}$ are relatively prime if $gcd(a, c) = 1$

Definition 2.11. For $n \geq 1$, let $\phi(n)$ donate the number of integers in the interval $[1, n]$ which are relatively prime to n. This function is called Euler-ϕ function.

Definition 2.12. If $a, b \in \mathbb{Z}$ then a is said to be congruent to b modulo n, written $a \equiv b \ (mod\, n)$, if n divides $(a - b)$.

Definition 2.13. The integers modulo n, denoted $\mathbb{Z}_n$, is the set of integers $\{0, 1, 2, \ldots, n - 1\}$. Addition, subtraction and multiplication in $\mathbb{Z}_n$ are performed modulo n.

Definition 2.14. $a \in \mathbb{Z}_n$, the multiplicative inverse of a modulo n is an integer $x \in \mathbb{Z}_n$ such that $ax \equiv 1 \ (mod\, n)$. If such an x exists, then it is unique and a is said to be invertible. The inverse of a is denoted a^{-1}.

Definition 2.15. $a, b \in \mathbb{Z}_n$, the division of a by b modulo n is the product of a and b^{-1} modulo n and is only defined if b is invertible modulo n.

Example. The invertible elements in $\mathbb{Z}_9$ are $\{1, 2, 4, 5, 7, 8\}$. For example, $4^{-1} = 7$ because $4 \cdot 7 \equiv 1 \pmod 9$ ○

Groups

Definition 2.16. A binary operator $*$ on a set S is a mapping from $S \times S$ to S. That is, $*$ is a rule which assigns to each ordered pair of elements from S an element of S.

Definition 2.17. A *group* $(G, *)$ consists of a set G with a binary operator $*$ on G satisfying:

1. The group operation is associative. $a * (b * c) = (a * b) * c \, \forall \, a, b, c \in G$
2. There is an element $1 \in G$, called the identity element, such that $a * 1 = 1 * a = a \, \forall \, a \in G$
3. For all $a \in G$ there exists an element $a^{-1} \in G$, called the inverse of a, such that $a * a^{-1} = a^{-1} * a = 1$

A group G is abelian if $a * b = b * a \, \forall \, a, b \in G$

Example. The set of integers $\mathbb{Z}$ with the operator of addition forms a group. The identity element is 0 and the inverse of an integer a is $-a$. ◯

Rings

Definition 2.18. A *ring* $(R, +, \times)$ consists of a set R with two binary operations arbitrarily denoted + (addition) and $\times$ (multiplication) on R, satisfying:

1. $(R, +)$ is an abelian group with identity 0
2. The operation $\times$ is associative. That is, $a \times (b \times c) = (a \times b) \times c \, \forall \, a, b, c \, in R$
3. There is a multiplicative identity denoted 1, with $1 \neq 0$, such that $1 \times a = a \times 1 = a \, \forall \, a \in R$
4. The operation $\times$ is distributive over $+$. That is, $a \times (b + c) = (a \times b) + (a \times c)$ and $(b + c) \times a = (b \times a) + (c \times a) \, \forall \, a, b, c \in R$

The ring is a commutative ring if $a \times b = b \times a \, \forall \, a, b \in R$

Example. The set of integers $\mathbb{Z}$ with addition and multiplication is a commutative ring. ◯

Example. The set of integers $\mathbb{Z}_n$ with addition and multiplication performed modulo n is a commutative ring. ◯

Fields

Definition 2.19. A *field* is a commutative ring in which all non-zero elements have multiplicative inverse

Polynomial Rings

Definition 2.20. If R is a commutative ring, then a *polynomial* in the indeterminate x over the ring R is an expression in the form

$$f(x) = a_n x^n + \cdots + a_2 x^2 + a_1 x + a_0$$

where each coefficient $a_i \in R$ of x^i and $n \geq 0$.

Definition 2.21. If R is a commutative ring, the polynomial ring $R[x]$ is the ring formed by the set of all polynomials in the indeterminate x having coefficients from R. The two operations are the polynomial addition and multiplication with coefficient arithmetic performed in R.

Example. Let $f(x) = x^3 + x + 1$ and $g(x) = x^2 + x$ be elements of the polynomial ring $\mathbb{Z}_2[x]$. Working in $\mathbb{Z}_2$,

$$f(x) = g(x) = x^3 + x^2 + 1$$

and

$$f(x) \cdot g(x) = x^5 + x^4 + x^3 + x$$

○

Definition 2.22. Let $f(x) \in F[x]$ be a polynomial of degree at least 1. Then $f(x)$ is said to be irreducible over F if it cannot be written as the product of two polynomials in $F[x]$.

Definition 2.23. If $g(x), h(x) \in F[x]$, then the polynomial division of $g(x)$ by $h(x)$ yields polynomials $q(x)$ and $r(x) \in F[x]$ such that

$$g(x) = q(x)h(x) + r(x)$$

Example. Let $g(x) = x^6 + x^5 + x^3 + x^2 + x + 1$ and $h(x) = x^4 + x^3 + 1$ in $\mathbb{Z}_2$. Since,

$$g(x) = x^2 h(x) + (x^3 + x + 1)$$

hence,

$$q(x) = x^2, \;\; r(x) = x^3 + x + 1$$

○

2.1.7 Number Theoretic Problems

Cryptography is based on problems which are computationally infeasible to solve. Following the most common examples of such problems:

Integer Factorization Problem

Definition 2.24. Given $n \in \mathbb{Z}$, find the primes p_i for which $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$ with $e_i \geq 1$

RSA Problem

Definition 2.25. Given $n = pq \in \mathbb{Z}$, where p and q are odd primes, $e \in \mathbb{Z}$ such that $gcd(e, (p-1)(q-1)) = 1$ and $c \in \mathbb{Z}$, find $m \in \mathbb{Z}$ such that $m^e \equiv c \pmod{n}$

2.1.7.1 Discrete Logarithm Problem

Definition 2.26. Let G be a finite cyclic group of order n. Let α be a generator of G and $\beta \in G$. The discrete logarithm of β of the base α, denoted $\log_\alpha \beta$, is $x \in \mathbb{Z}$, $0 \leq x \leq n-1$ such that $\beta = \alpha^x$

Example. $p = 97$, then $\mathbb{Z}_{97}$ is a cyclic group of order $n = 96$. A generator of $\mathbb{Z}_{97}$ is $\alpha = 5$. Since $5^{23} \equiv 35 \ (mod\, 97)$, $log_5 35 = 32$ in $\mathbb{Z}_{97}$ ○

Definition 2.27. Given a prime p, a generator α of $\mathbb{Z}_p$ and the element $\beta \in \mathbb{Z}_p$, find the integer x, $0 \leq x \leq p-2$, such that $\alpha^x \equiv \beta \ (mod\ p)$

2.1.8 Cryptographic Schemes

2.1.8.1 Diffie-Hellman Key Exchange

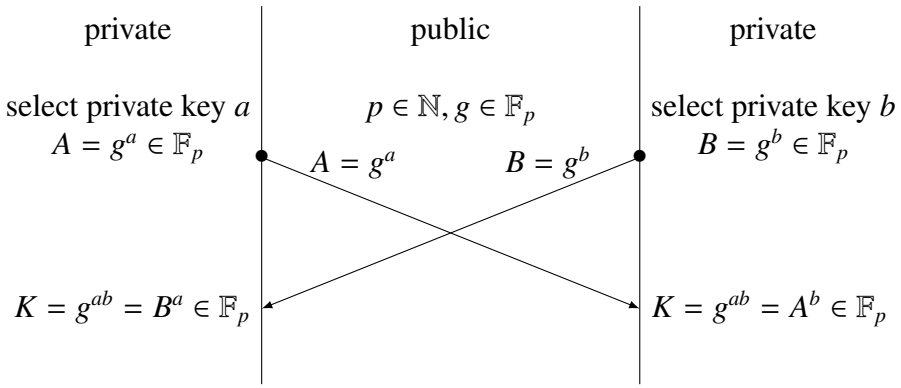


Figure 2.6: Diffie-Hellman key exchange

Diffie-Hellman key exchange is a typical public key exchange scheme, the scheme is illustrated in Fig. 2.6:

1. Alice and Bob agree on two prime numbers p and g where g is a primitive root of p which are shared publicly
2. Alice chooses a random number a as her private number
3. Bob chooses a random number b as his private number
4. Alice computes $A = g^a \pmod p$ and sends it to Bob
5. Bob computes $B = g^b \pmod p$ and sends it to Alice
6. Alice computes $K = B^a \pmod p = g^{b^a} \pmod p$
7. Bob computes $K = A^b \pmod p g^{a^b} \pmod p$
8. Whereas $g^{b^a} \pmod p = g^{a^b} \pmod p = g^{ab} \pmod p$

K is now the secure Key which Alice and Bob can use to encrypt and decrypt their messages. In order for an eavesdropper Eve to obtain K only knowing p, g, A, B, which describes the discrete logarithm problem in 2.27.

2.1.8.2 Elliptic-curve Cryptography

The computation with large integers as in the Diffie-Hellman key exchange scheme is computationally expensive. The same principle is used with elliptic curves, this scheme is faster and cryptographically more secure. Following Ti*k*Z images are inspired by [3] and the content is by [4].

An elliptic curve is defined with

$$y^2 = x^3 - ax + b \in \mathbb{R}, \tag{2.1}$$

must hold $4a^3 + 27b^2 \neq 0$ and $x^3 + ax = b$ contains no multiple roots.

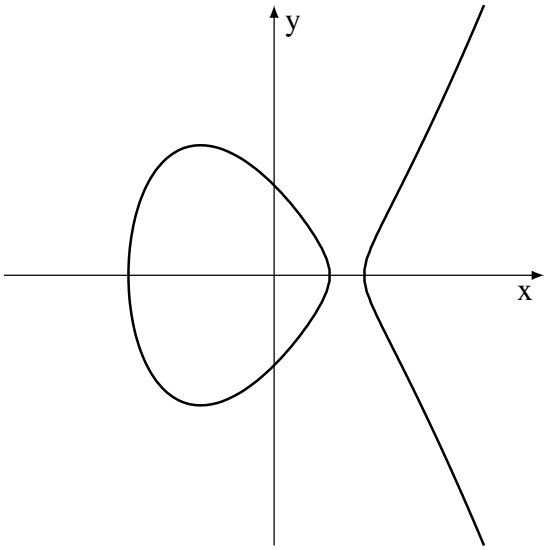


Figure 2.7: The elliptic curve defined by $y^2 = x^3 - 2x + 1 \in \mathbb{R}$

In order to form an abelian group (2.1.6), a operation is defined. In the case of elliptic curves the operation is called addition with the operator (+). Fig. 2.8 shows the geometrical representation of the addition as well as doubling one point. In order to be a complete group an identity element $\mathcal{O}$ is needed, where

- $P + \mathcal{O} = \mathcal{O} + P = P$
- $\mathcal{O} + \mathcal{O} = \mathcal{O}$
- $P + (-P) = \mathcal{O}$

The identity element is also called the point at infinity

The algebraic solution for the addition of the points (with coordinates x, y) $R = P + Q$ is

$$m = \begin{cases} \dfrac{y_P - y_Q}{x_P - x_Q} & \text{if } P \neq Q, P \neq -Q \\ \dfrac{3x_P^2 + a}{2y_P} & \text{if } P = Q \end{cases} \tag{2.2}$$

$$x_R = m^2 - x_P - x_Q \tag{2.3}$$

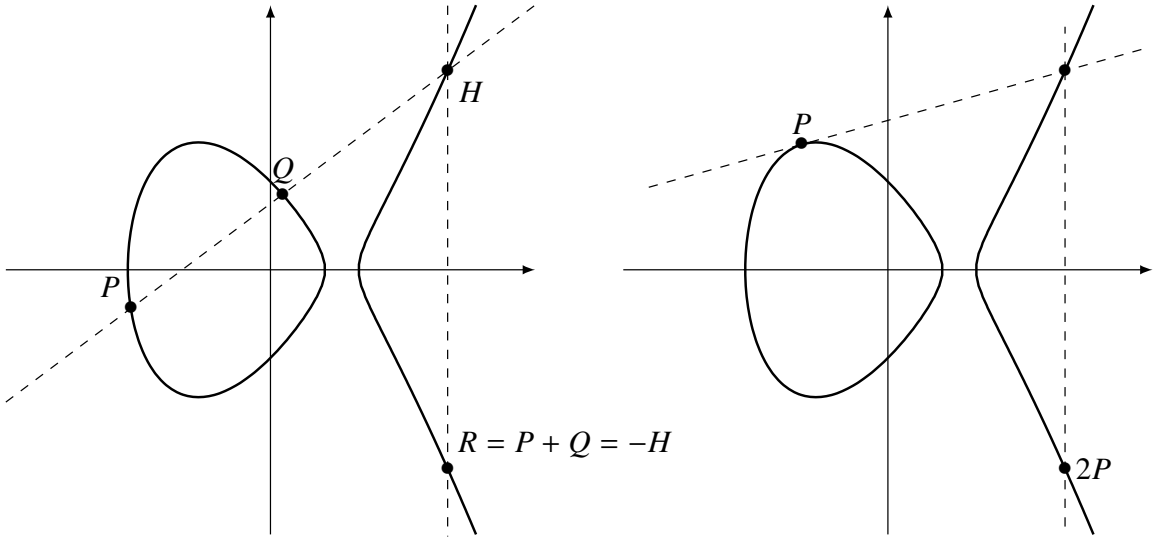


Figure 2.8: The elliptic curve defined by $y^2 = x^3 - 2x + 1 \in \mathbb{R}$, with the addition of $P + Q = R$ and $P + P = 2P$

$$y_R = m(x_P - x_R) - y_P \tag{2.4}$$

The order of a point is an important parameter of a specific point in an elliptic curve. It is defined by smallest positive number a point can be added by itself until $nP = O$. The idea is illustrated in Fig. 2.9.

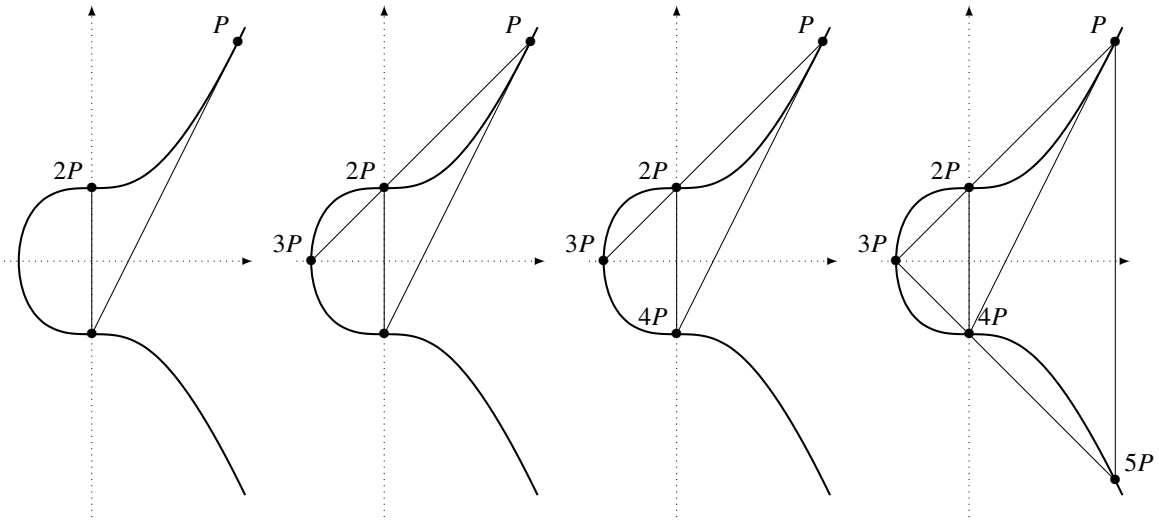


Figure 2.9: $y^2 = x^3 + 1$, P is chosen to be $(2, 3)$ the geometric construction of the points $2P, 3P, 4P, 5P$ are shown. When computing $6P$ one sees that the line between P and $5P$ is vertical and has no intersection, therefore $6P = O$ and the order of P is 6

Elliptic Curve over $\mathbb{F}_p$ Fig. 2.10 shows the elliptic curve defined by $y^2 = x^3 - 2x + 1$ over the finite field $\mathbb{Z}_{17}$. The specific points are computed as follows:

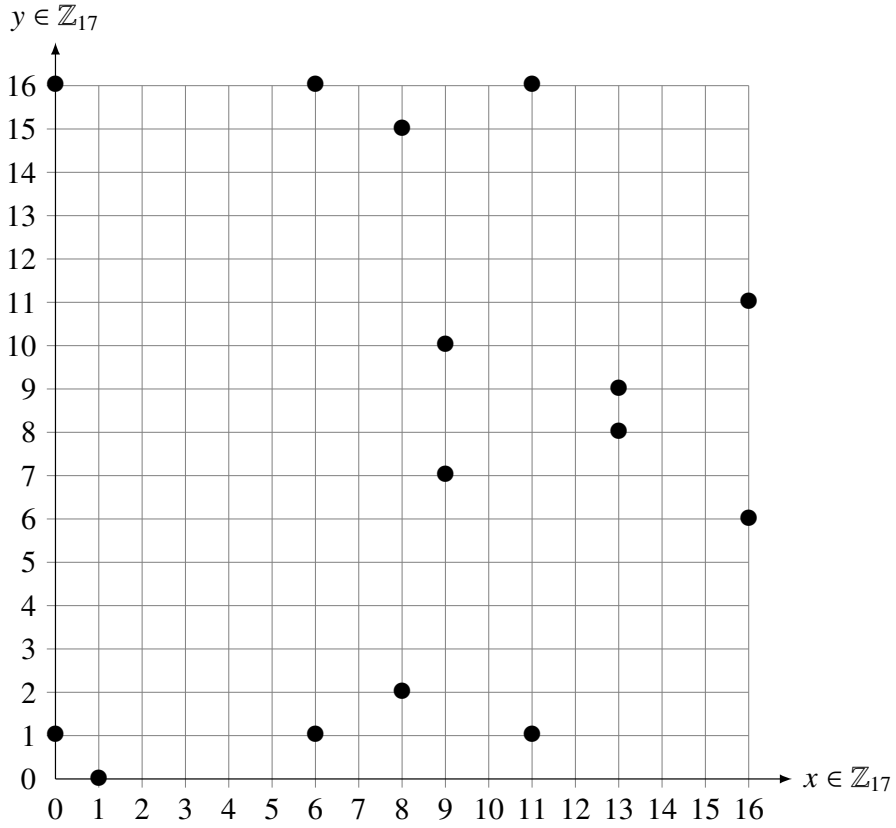


Figure 2.10: $y^2 = x^3 - 2x + 1$ over $\mathbb{Z}_{17}$

- $x = 0$ is a valuable point for $y = 1$ because:
- $1^2 = 0^3 - 2 \cdot 0 + 1 \equiv 1 \pmod{17}$
- $x = 3$ is not a valuable point for $y = 1$ because:
- $1^2 = 3^3 - 2 \cdot 3 + 1 = 27 - 6 + 1 = 22 \equiv 5 \pmod{17}$
- $x = 6$ is a valuable point for $y = 1$ because:
- $1^2 = 6^3 - 2 \cdot 6 + 1 = 216 - 12 + 1 = 205 \equiv 1 \pmod{17}$

Elliptic Curve Diffie-Hellman Key Exchange

$T = (p, a, b, G, n, h)$	set of parameters
$p \in \mathbb{P}$	defines field $\mathbb{F}_p$
$a, b \in \mathbb{F}_p$	defines elliptic curve
$G \in E(\mathbb{F}_p)$	base point
$n \in \mathbb{P}$	order of base point

1. Agree on $T = (p, a, b, G, n)$
2. Alice randomly chooses a secret integer d_A in the interval $[1, n - 1]$ and computes $Q_A = d_A G$

3. Bob randomly chooses a secret integer d_B in the interval $[1, n-1]$ and computes $Q_B = d_B G$
4. Alice and Bob exchange Q_A and Q_B
5. Alice computes $P_A = d_A Q_B$.
6. Bob computes $P_B = d_B QA$
7. If $P \neq (O)$ then Alice and Bob take the x-coordinate x_P as the shared key.

Where the discrete logarithm problem is to find the integer n that $nG = Q$.

2.1.8.3 Advanced Encryption Standard

Once a secure key with a given public key exchange algorithm is received, Alice can start to encrypt their messages. Advanced Encryption Standard (AES) is an encryption algorithm built as a symmetric block cipher [5].

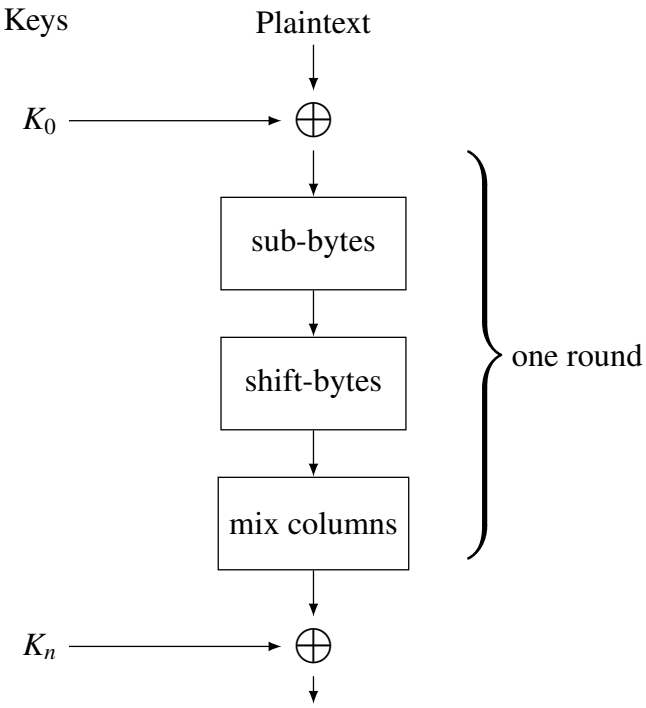


Figure 2.11: Order of operations for the AES with a different key K_n for each round

It is designed as a substitution-permutation (Fig. 2.11) network, where the bytes are arranged in a 4×4

$$\begin{array}{|cccc|} \hline b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \\ \hline \end{array} \tag{2.5}$$

array.

Add Round Key The byte array is combined with the key of the specific round using bitwise XOR.

Substitute Bytes Each byte is substituted and maps an 8-bit input, to an 8-bit output. This is implemented with a look-up-table and is therefore quite fast.

Shift Rows In the shift rows part, the first row is kept the same in the second row all bits are shifted by one position to the left, in the third row all bits are shifted by two positions to the left and so on.

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix} \longrightarrow \begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_5 & b_9 & b_{13} & b_1 \\ b_{10} & b_{14} & b_2 & b_6 \\ b_{15} & b_3 & b_7 & b_{11} \end{bmatrix} \tag{2.6}$$

Mix Columns Now after shifting all the rows, the columns are mixed. A column a_j is taken and multiplied by a small matrix to generate some randomness. The output column is b_j

$$\begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \begin{pmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{pmatrix} = \begin{pmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{pmatrix} \qquad 0 \leq j \leq 3$$

AES Key Schedule AES needs in total 11 sub keys one for the plaintext and then one for each of the ten rounds. If AES-128 is used, then the 128 bit key is split into four 32-bit words w_0, w_1, w_2, w_3. The computation of the key for the next round w_4, w_5, w_6, w_7 is shown in Fig. 2.12.

- The π-box stands for the rotation of a word $[w_0, w_1, w_2, w_3] \rightarrow [w_1, w_2, w_3, w_0]$
- The s-box stands for a substitution box as seen in the general AES algorithm
- r_n stands for the round constant which is computed individually every round.

2.2 Post Quantum Cryptography

PQC is a group of algorithms that are considered safe against quantum computing, or more precisely, no algorithm has yet been discovered that breaks the scheme.

The three difficult mathematical problems: the integer factorization problem, the discrete logarithm problem, or the discrete logarithm problem with elliptic curves, which are mainly used in current cryptographic algorithms can be solved with Shor's algorithm.

There are several theoretical foundations for PQC algorithms, but most of them that will be standardized are using lattice-based cryptography.

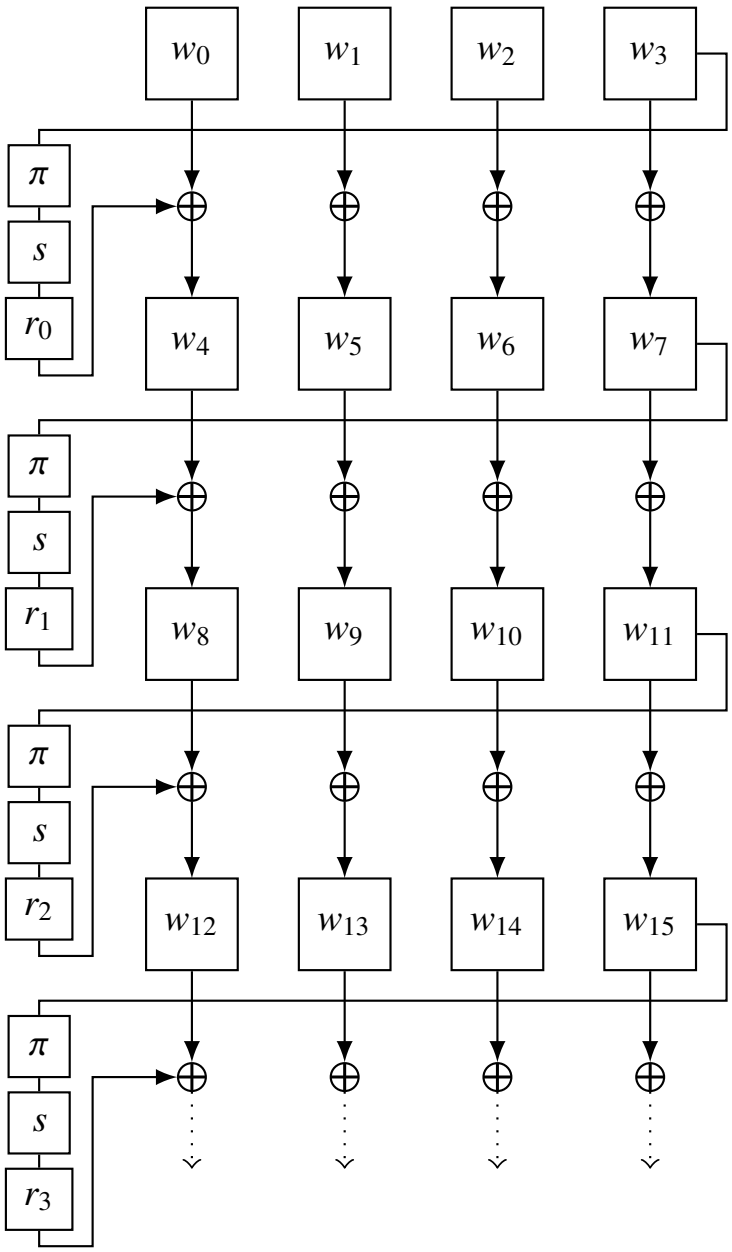


Figure 2.12: The key schedule for the advanced encryption standard is

2.2.1 Lattice Based Cryptography

A lot of following information is from Tanja Lange's lecture notes in *Selected Areas in Cryptology* [6]. A lattice L is a discrete additive subgroup of $\mathbb{R}_n$. Equivalently, a lattice can be defined as the set of all linear integer combinations of linearly independent vectors $\mathbf{b}_1, \ldots \mathbf{b}_d \in \mathbb{R}^n$,

$$L = \sum_{i=1}^{d} x_i \mathbf{b}_i \quad \in \mathbb{Z}^d \tag{2.7}$$

In Fig. 2.13 a two-dimensional lattice with two possible bases is shown.

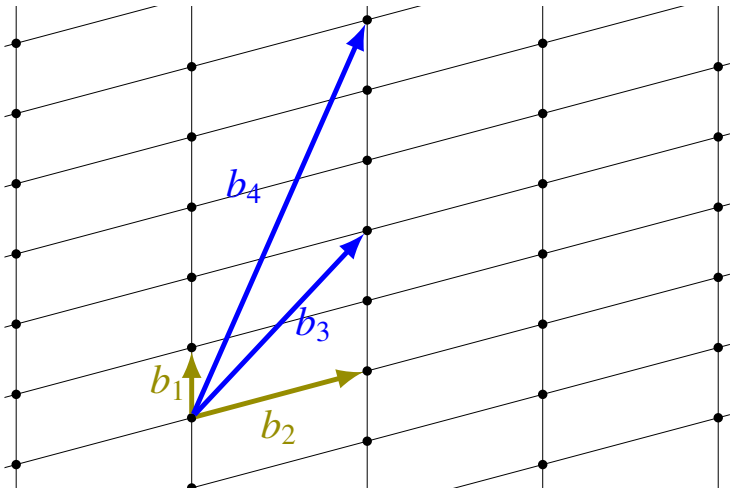


Figure 2.13: Two-dimensional lattice with two possible bases

Shortest Vector Problem With lattice based cryptography a short vector is of interest, the L_2-norm

$$\| (c_1, c_2, \ldots, c_n) \| = \sqrt{c_1^2 + c_2^2 + \cdots + c_n^2} \tag{2.8}$$

can be used to measure the length of a vector with coordinates c_i. The shortest vector problem is to find any vector with the shortest length. The problem is shown in Fig. 2.14 in two dimensions. The problem seems easy but image a lattice with numerous dimensions.

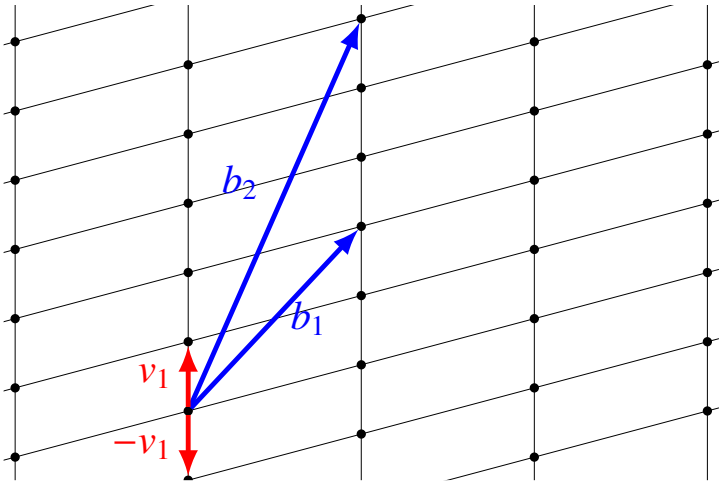


Figure 2.14: The shortest vector in this lattice field with this given basis is v_1, respectively v_2

Closest Vector Problem The closest vector problem, given a point t find the closest lattice in the lattice field. In Fig. 2.15 v would be the closest lattice.

Depending on the basis this can be a very hard task. In Fig. 2.16 a good basis with r_1 and r_2 is given. Drawing the closest lattice vectors (grid lines), the closest lattice point can be found easily with the intersection of the two lines.

In Fig. 2.17 a bad basis with r_1 and r_2 is given. Drawing the closet lattice vectors (grid lines), the closest lattice point can now not be found with the intersection of the two lines.

Babai's nearest plane algorithm solves the problem approximately [7].

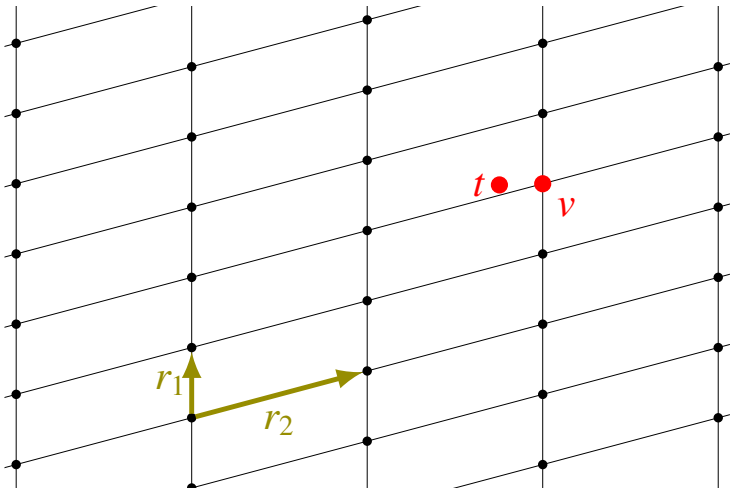


Figure 2.15: Illustrated closest vector problem, given a point t, find the closest grid in the grid array. In this case v would be the nearest grid

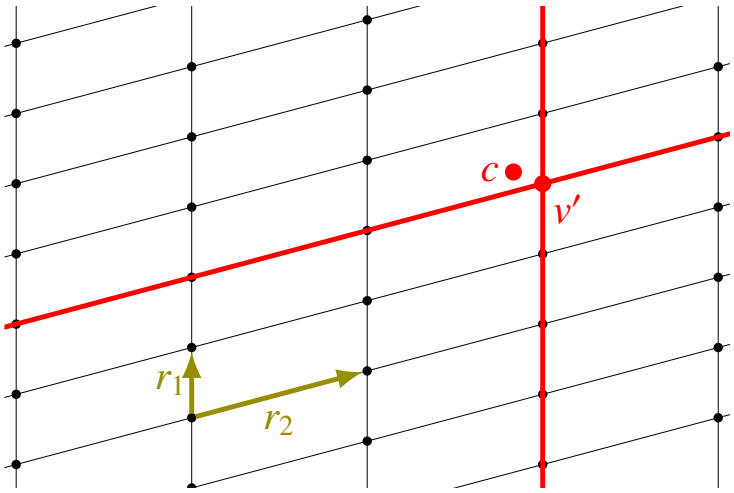


Figure 2.16: Finding the closest vector with a good basis

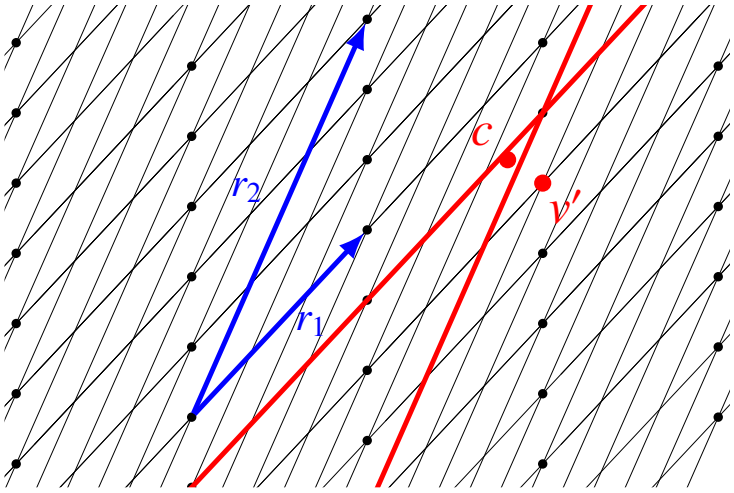


Figure 2.17: Finding the closest vector with a bad basis

GGH Cryptosystem The Goldreich-Goldwasser-Halevi (GGH) lattice-based cryptosystem is based on the fact that with a bad basis it is infeasible to solve the closest vector problem [8]. It is designed as follows:

- Private Key (good basis): R = $\begin{pmatrix}\mathbf{r}_1\\ \mathbf{r}_2\end{pmatrix}$
- Public Key (bad basis): B = $\begin{pmatrix}\mathbf{b}_1\\ \mathbf{b}_2\end{pmatrix}$
- Encrypt **m** and adding a small error **e** to it
  - $\mathbf{v} = \mathbf{m}B$
  - $\mathbf{c} = \mathbf{v} + \mathbf{e}$
- Decrypt **c**
  - $\mathbf{v}' = \lfloor \mathbf{c}R^{-1} \rceil R$
  - $\mathbf{m}' = \mathbf{v}'B^{-1}$

The fact that R is a good basis, the **v** vector can be retrieved with a suitable rounding function. With a bad basis that would not be possible.

2.2.1.1 NTRU

$$\sum_{i=0}^{n-1} a_i x^i + \sum_{i=0}^{n-1} b_i x^i = \sum_{i=0}^{n-1} (a_i + b_i) x^i$$

First proposed in [9] a NTRU cryptosystem has three integer parameters (N, p, q) and four sets $\mathcal{L}_f, \mathcal{L}_g, \mathcal{L}_\phi \mathcal{L}_m$ of polynomials with degree $N-1$ and with integer coefficients p, q that must not be primes but $\gcd(p, q) = 1$ and $q \ggg p$. It works in the ring $R = \mathbb{Z}[x]/(x^n - 1)$. An element $F \in R$ will be written as a polynomial or vector.

$$F = \sum_{i=0}^{N-1} F_i x^i = [F_0, F_1, \ldots, F_{N-1}]. \tag{2.9}$$

The $\circledast$ denotes the multiplication in R, it is the cyclic convolution,

$$F \circledast G = H \text{ with } H_k = \sum_{i=0}^{k} F_i G_{k-i} + \sum_{i=k+1}^{N-1} F_i G_{N+k-i} \tag{2.10}$$

Key Generation Alice picks two random polynomials $f, g \in \mathcal{L}_g$, f must satisfy the requirement that it has inverses modulo q and p. We will denote the inverses by F_q and F_p,

$$F_q \circledast f \equiv 1 \,(\mathrm{mod}\, q) \quad \text{and} \quad F_p \circledast f \equiv 1 \,(\mathrm{mod}\, p) \tag{2.11}$$

The public key h is computed

$$h \equiv F_q \circledast g \,(\mathrm{mod}\, q) \tag{2.12}$$

Encryption Bob selects a message m from the set of plaintexts $\mathcal{L}_m$ and randomly chooses a polynomial $\phi \in \mathcal{L}_\phi$ and computes the ciphertext c

$$c \equiv p\phi \circledast h = m \,(\mathrm{mod}\, q). \tag{2.13}$$

Decryption Alice decrypts c, With

$$a \equiv f \circledast c \,(\mathrm{mod}\, q) \tag{2.14}$$

$$m = F_p \circledast a \,(\mathrm{mod}\, p) \tag{2.15}$$

2.2.2 Falcon

Fast Fourier lattice-based compact signatures over NTRU or short Falcon is a lattice-based signature scheme. Following content is either based on or directly quoted (*italic*) from the Falcon specification [10].

The high-level design of Falcon is simple: we instantiate the theoretical framework described by Gentry, Peikert and Vaikuntanathan [11] for constructing hash-and-sign lattice-based signature schemes. This framework requires two ingredients:

- *A class of cryptographic lattices. We chose the class of NTRU lattices.*
- *A trapdoor sampler. We rely on a new technique which we call fast Fourier sampling.*

In a nutshell, the Falcon signature scheme may therefore be described as follows:

Falcon = GPV framework + NTRU lattices + Fast Fourier sampling

2.2.2.1 Genealogy of Falcon

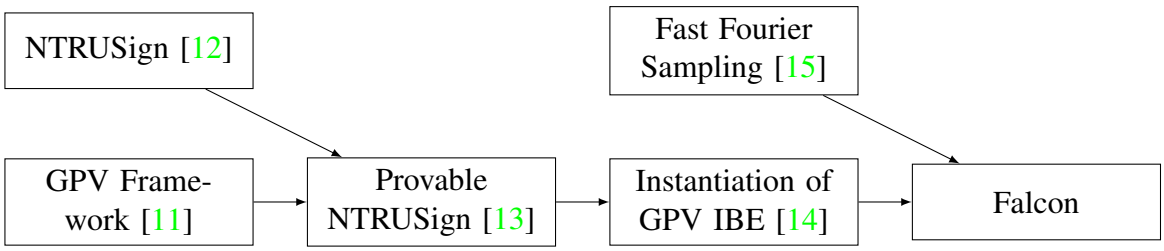


Figure 2.18: Genealogy of Falcon

Falcon is the product of many years of work, not only by the authors but also by others. This section explains how these works gradually led to Falcon as we know it.

The first work is the signature scheme NTRUSign [12] by Hoffstein et al., which was the first, along with GGH [8], to propose lattice-based signatures. The use of NTRU lattices by NTRUSign allows it to be very compact. However, both had a flaw in the deterministic signing procedure which led to devastating key-recovery attacks [16] [17].

At STOC 2008, Gentry, Peikert and Vaikuntanathan [11] proposed a method which not only corrected the flawed signing procedure but, even better, did it in a provably secure way. The

result was a generic framework (the GPV framework) for building secure hash-and-sign lattice-based signature schemes.

The next step towards Falcon was the work of Stehlé and Steinfeld [13], who combined the GPV framework with NTRU lattices. The result could be called a provably secure NTRUSign.

In a more practical work, Ducas et al. [14] proposed a practical instantiation and implementation of the IBE part of the GPV framework over NTRU lattices. This IBE can be converted in a straightforward manner into a signature scheme. However, doing this would have resulted in a signing time in $O(n^2)$.

To address the issue of a slow signing time, Ducas and Prest [15] proposed a new algorithm running in time $O(n \log n)$. However, how to practically instantiate this algorithm remained a open question. Falcon builds on these works to propose a practical lattice-based hash-and-sign scheme. Fig. 2.18 shows the genealogic tree of Falcon, the first of the many trees that this document contains.

2.2.2.2 The Gentry-Peikert-Vaikuntanathan Framework

In [10, p. 10]:

- Public Key: $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ ($m \geq n$) generating a q-ary lattice Λ
- Private Key: $\mathbf{B} \in \mathbb{Z}_q^{m \times m}$ generating $\Lambda_q^\perp$, where $\Lambda_q^\perp$ denotes the lattice orthogonal to $\Lambda \pmod q$: for any $\mathbf{x} \in \Lambda$ and $\mathbf{y} \in \Lambda_q^\perp$, we have $\langle \mathbf{x}, \mathbf{y} \rangle = 0 \pmod q$. Equivalently, the rows of $\mathbf{A}$ and $\mathbf{B}$ are pairwise orthogonal: $\mathbf{B} \times \mathbf{A}^t = \mathbf{0}$.
- Given a message m, a signature of m is a short value $\mathbf{s} \in \mathbb{Z}_q^m$ such that $\mathbf{sA} = H(m)$, where H: $\{0,1\}^* \to \mathbb{Z}_q^n$ is a hash function. Given $\mathbf{A}$, verifying that $\mathbf{s}$ is a valid signature is straightforward: it only requires to check that $\mathbf{s}$ is indeed short and verifies $\mathbf{sA}^t = H(m)$.
- Computing a valid signature is more delicate. First, a preimage $\mathbf{c}_0 \in \mathbb{Z}_q^m$ is computed, which verifies $\mathbf{c}_0\mathbf{A}^t = H(m)$. As $\mathbf{c}_0$ is not required to be short and $m \geq n$, this is simply done via standard linear algebra. $\mathbf{B}$ is then used in order to compute a vector $\mathbf{v} \in \Lambda_q^\perp$ close to $\mathbf{c}_0$. The difference $\mathbf{s} = \mathbf{c}_0 - \mathbf{v}$ is a valid signature: indeed, $\mathbf{sA}^t = \mathbf{c}_0\mathbf{A}^t - \mathbf{vA}^t = \mathbf{c} - 0 = H(m)$, and if $\mathbf{c}_0$ and $\mathbf{v}$ are close enough, then $\mathbf{s}$ is short.

2.2.2.3 NTRU Lattices

In [10, p. 12]:

Let $\phi = x^n + 1$ for $n = 2^k$ and $q \in \mathbb{N}^*$. NTRU consists of the polynomials $f, g, F, G \in \mathbb{Z}[x]/(\phi)$ for:

$$fG - gF = q \pmod \phi$$

Provided that f is invertible mod q, we can define the polynomial $h \leftarrow g \cdot f^{-1} \pmod q$, h will be the public key, whereas f, g, F, G are secret keys. The matrices

$$\left[\begin{array}{c|c} 1 & -h \\ \hline 0 & q \end{array}\right], \left[\begin{array}{c|c} f & g \\ \hline F & G \end{array}\right]$$

generate the same lattice. If f, g are generated with enough entropy then h will look pseudorandom. However, in practice, even when f, g are small, it remains hard to find small polynomials f', g' such that $h = g'(f')^{-1} \bmod q$. The hardness of this problem constitutes the NTRU assumption.

2.2.2.4 Instantiation with the GPV framework

The GPV framework is instantiated as follows:

- Public lattice basis,
$$\mathbf{A} = \left[\begin{array}{c|c} 1 & h^* \end{array}\right]$$
- The secret basis,
$$\mathbf{B} = \left[\begin{array}{c|c} g & -f \\ \hline G & -F \end{array}\right]$$

The signature of a message m consists of a salt r plus a pair of polynomials (s_1, s_2) such that $s_1 + s_2 h = H(r \,\|\, m)$. We note that since s_1 is completely determined by m, r and s_2, there is no need to send it: the signature can simply be (r, s_2).

2.2.2.5 Fast Fourier Sampling

The second choice when instantiating the GPV framework is the trapdoor sampler. A trapdoor sampler takes as input a matrix $\mathbf{A}$, a trapdoor T, a target $\mathbf{c}$ and outputs a short vector $\mathbf{s}$ such that $\mathbf{s}^t\mathbf{A} = \mathbf{c} \bmod q$. This is the same idea as in the GGH cryptosystem when we add a small error.

Ducas and Prest [15] proposed *fast Fourier nearest plane*, a variant of Babai's nearest plane algorithm for lattices over rings. It proceeds in a recursive way which is very similar to the fast Fourier transform, hence the name.

2.2.2.6 Overview of the Algorithm

Hashing The basis of every hash-and-sign signature scheme, is a proper hashing algorithm. Normally in order to sign a message or verify a signature one must hash the message. In the case of the Falcon algorithm, the message needs to be hashed into a polynomial in $\mathbb{Z}_q[x]/(\phi)$. SHAKE-256 is used as the hashing function.

- `SHAKE-256-Init()` denotes the initialization of a SHAKE-256 hashing context
- `SHAKE-256-Inject(ctx, str)` denotes the injection of the data `str` in the hashing context `ctx`
- `SHAKE-256-Extract(ctx, b)` denotes extraction from a hashing context `ctx` of `b` bits of pseudorandomness

Key Pair Generation involves choosing random f and g polynomials using an appropriate distribution that yields short, but not too short, vectors; then, the NTRU equation is solved to find matching F and G. The algorithm is shown in Algorithm 1.

Signature Generation consists of first hashing the message to sign, along with a random nonce (a pseudorandom string), into a polynomial c modulo ϕ, whose coefficients are uniformly mapped to integers in the 0 to $q-1$ range. Then, the signer uses his knowledge of the secret lattice basis (f, g, F, G) to produce a pair of short polynomials (s_1, s_2) such that $s_1 = c - s_2 h \bmod \phi \bmod q$ The algorithm is shown in Algorithm 2.

Signature Verification consists of recomputing s_1 from the hashed message c and the signature s_2, and then verifying that (s_1, s_2) is an appropriately short vector. Signature verification can be done entirely with integer computations modulo q. The algorithm is shown in Algorithm 3.

Private Keys are used to compute two additional elements, and may be recomputed dynamically, or stored along f, g and F: The FFT representations of f, g, F and G, ordered in the form of a matrix:

$$\hat{\mathbf{B}} = \left[\begin{array}{c|c} \text{FFT}(g) & -\text{FFT}(f) \\ \hline \text{FFT}(G) & -\text{FFT}(F) \end{array}\right],$$

FFT(a) being the fast Fourier transform of a in the underlying ring (here, $\mathbb{R}[x]/(\phi)$).

A binary tree called a Falcon Tree:

- A Falcon tree T of height 0 consists of a single node whose value is a real $\sigma > 0$
- A Falcon tree T of height κ verifies these properties:
  - The value of its root, noted T.value, is a polynomial $l \in \mathbb{Q}[x]/(x^n + 1)$ with $n = 2\kappa$.
  - Its left and right children, noted T.leftchild and T.rightchild, are Falcon trees of height $\kappa - 1$.

The structure is visualized in Fig. 2.19.

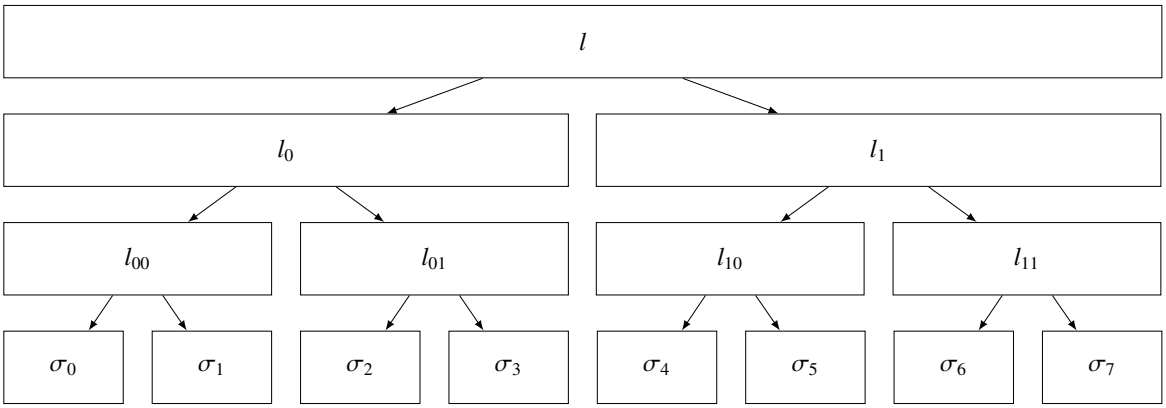


Figure 2.19: A Falcon tree of height 3

Algorithm 1 Keygen(ϕ, q)

Require: A monic polynomial $\phi \in \mathbb{Z}[x]$, a modulus q
Ensure: A secret key sk, a public key pk

$f, g, F, G \leftarrow \text{NTRUGen}(\phi, q)$
$\mathbf{B} \leftarrow \left[\begin{array}{c|c} g & -f \\ \hline G & -F \end{array}\right]$
$\hat{\mathbf{B}} \leftarrow \text{FFT}(\mathbf{B})$
$\mathbf{G} \leftarrow \hat{\mathbf{B}} \times \hat{\mathbf{B}}^*$
$\text{T} \leftarrow \text{ffLDL}^*(\mathbf{G})$
for each leaf of T **do**
    leaf.value $\leftarrow \sigma / \sqrt{leaf.value}$
end for
$\text{sk} \leftarrow (\hat{\mathbf{B}}, \text{T})$
$h \leftarrow g f^{-1} \bmod q$
$\text{pk} \leftarrow h$
return sk, pk

Algorithm 2 Sign(m, sk, $\lfloor \beta^2 \rfloor$)

Require: A message m, a secret key sk, a bound $\lfloor \beta^2 \rfloor$
Ensure: A signature sig of m

$\text{r} \leftarrow \{0,1\}^{320} uniformly$
$c \leftarrow \text{HashToPoint}(\text{r} \parallel \text{m}, q, n)$
$\mathbf{t} \leftarrow \left(-\frac{1}{q}\text{FFT}(C) \odot \text{FFT}(F), \frac{1}{q}\text{FFT}(C) \odot \text{FFT}(f)\right)$ ▷ $\mathbf{t} = (\text{FFT}(c), \text{FFT}(0)) \cdot \hat{\mathbf{B}}^{-1}$
while $s = \perp$ **do**
    while $\| s \|^2 > \lfloor \beta^2 \rfloor$ **do**
        $\mathbf{z} \leftarrow \text{ffSampling}_n(\mathbf{t}, \text{T})$
        $\mathbf{s} = (\mathbf{t} - \mathbf{z})\hat{\mathbf{B}}$
    end while
    $(s_1, s_2) \leftarrow \text{iFFT}(\text{s})$ ▷ $s_1 + s_2 h = c \bmod (\phi, q)$
    $\text{s} \leftarrow \text{Compress}(s_2, 8 \cdot \text{sbytelen} - 328)$
end while
return sig = (r, s)

Algorithm 3 Verify(m, sig, pk, $\lfloor\beta^2\rfloor$)

Require: A message m, a signature sig = $(\mathbf{r}, \mathbf{s})$, a public key pk = $h \in \mathbb{Z}_q[x]/(\phi)$, a bound $\lfloor\beta^2\rceil$
Ensure: Accept or reject

$c \leftarrow \text{HashToPoint}(\mathsf{r} \parallel \mathsf{m}, q, n)$
$s_2 \leftarrow \text{Decompress}(s, 8 \cdot \text{sbytelen} - 328)$
if $s_2 = \perp$ **then**
    reject ▷ Reject invalid encoding
end if
$s_1 \leftarrow c - s_2 h \bmod q$ ▷ s_1 should be normalized between $\lceil -\frac{q}{2} \rceil$ and $\lfloor \frac{q}{2} \rfloor$
if $\| (s_1, s_2) \|^2 \leq \lfloor\beta^2\rfloor$ **then**
    accept
else
    reject ▷ Reject signatures that are too long
end if

2.2.3 Hardware Acceleration of PQC Algorithms

The NIST began the process of standardizing post-quantum cryptography in 2017. After four rounds, the first algorithms were selected for standardization in July 2022. The citation shows the according hardware acceleration on FPGAs.

- Public-key Encryption and Key-establishment Algorithms
  - CRYSTALS-KYBER [18]
    - FPGA implementation: [19, 20]
- Digital Signature Algorithms
  - CRYSTALS-DILITHIUM [21]
    - FPGA implementation: [22–24]
  - Falcon [10]
    - FPGA implementation (Signature verification): [25, 26]
  - SPHINCS+ [27]
    - FPGA implementation: [28–30]

The hardware acceleration with FPGA's has been done for all algorithms except Falcon. For Falcon only the signature verification has been accelerated. Due to the use of floating point operation and recursive function calls the implementation of the key and signature generation is still missing.

Chapter 3

Methods

3.1 High Level Synthesis

As it is usual in such competitions for new algorithms, they are tested on a wide variety of aspects. An important step is the transition of the scientific examination in a high level programming language such as *Python*, or *C/C++* to the testing and benchmarking on different types of hardware.

For FPGAs, such a problem would traditionally have been solved with an HDL-based approach. Where the algorithms are specified through design steps like verification, logic synthesis, placement and routing, timing analysis, testing, and validation. This approach normally leads to very good results but the time related overhead is substantial.

The design approach with a HLS can be very time saving. In the scientific study of such algorithms, as a rule, some kind of implementation for a Central Processing Unit (CPU) or a microcontroller is built in *C/C++*.

The typical HLS flow is illustrated in Fig. 3.1. Where one can see that, the main task can be divided into two separate tasks. The basis is usually a reference implementation in a high level programming language mostly in *C/C++*. The first task is to adapt the code in such a way that the HLS can process the code and provide a synthesizable hardware description. This so called *HLS-ready-code* is neither optimized for throughput, latency nor hardware utilization.

There are several programming paradigms which work fine in *C/C++* and make a lot of sense to use, but HLS cannot process them. In the past, many of such code modifications were necessary. As an example [26], spoke about *"replace library functions, change complex hierarchy of structure, change variable length arrays and pointers to a fixed-dimension array, remove file operations, replace recursions, and modify complex pointer operations to make the C code synthesizable"*.

With the latest versions of HLS, these clear adaptation criteria are no longer given. For example, it was stated that the *C* library-function `memcpy` had to be rewritten. With the latest version of HLS this function did not cause any problems. Other programming styles are still causing HLS to fail:

- Recursive functions
- Dynamically allocated memory
- Pointer comparison to the `NULL`-pointer

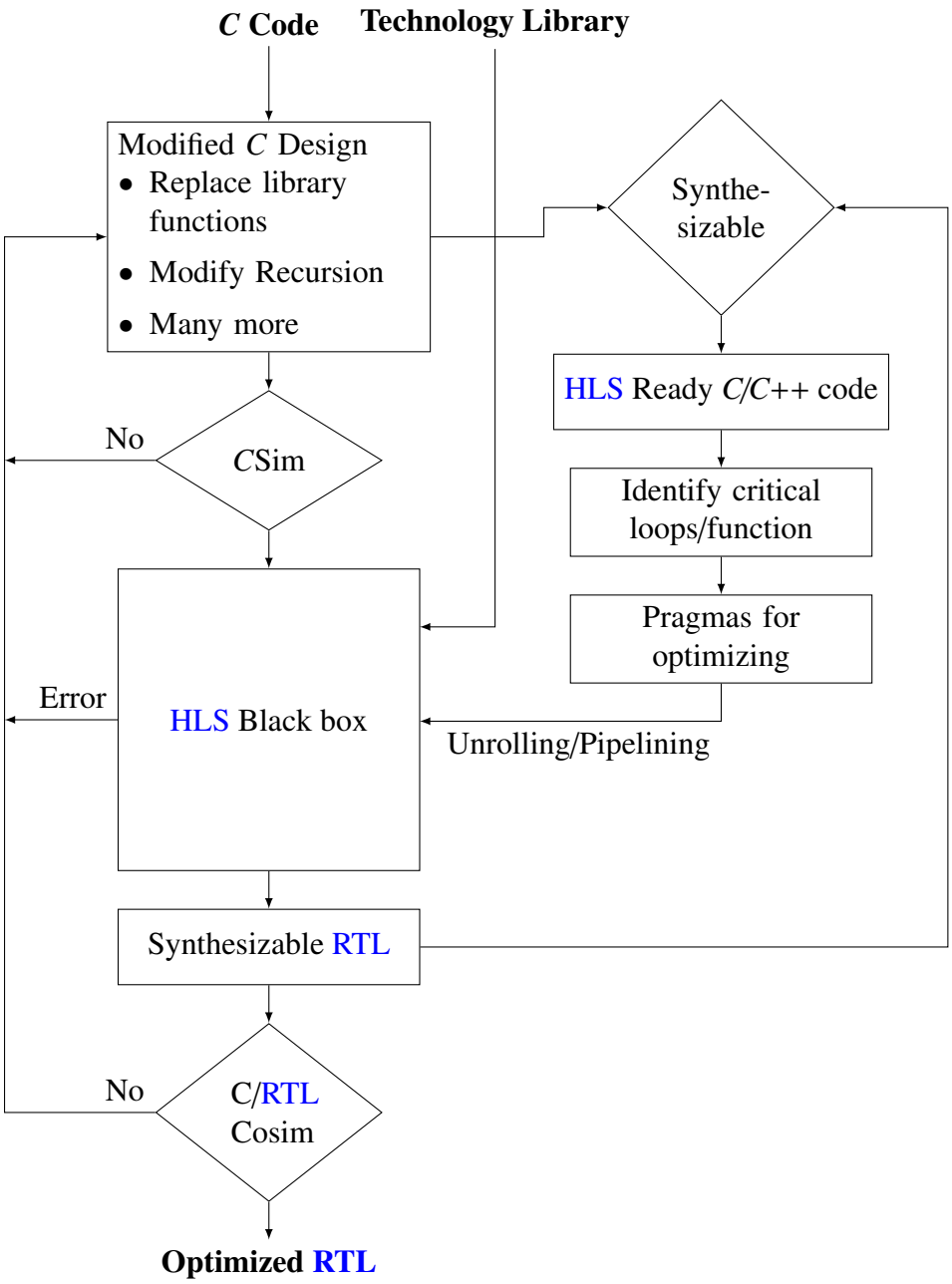


Figure 3.1: A typical HLS flow when taking a reference implementation in *C/C++* and trying to generate an optimized RTL, inspired by [26]

- `goto` statements
- Function pointers

3.1.1 Working Environment

When HLS is selected for a project, there is always the question of which provider to choose, whereby the FPGA to be used is decisive. In this project *Vitis HLS* from the FPGA provider *AMD-Xilinx* was used. The final version used is *Vitis 2022.2*, although version *Vitis 2021.2* was also used and no significant differences were found.

The traditional workflow in projects using HLS is based on the use of a Graphical User Interface (GUI), where a lot of explicit user intervention is required.

- Editing the code in *Vitis HLS*
- Running *C*-simulation in *Vitis HLS*
  - The *C* simulation is nothing more than compiling the code with a conventional compiler like *Clang* or *GCC* and executing the specified `main()` function. For some reason, this simulation in *Vitis HLS* is extremely slow. It can take up to minutes, while compiling over a normal termial takes no more than a few milliseconds, this depends on the program being executed. There is no known benefit from running the simulation in *Vitis HLS*.
- Running RTL Synthesis
- Exporting the generated RTL to a specific location
- Opening *Vivado* and importing the generated files
- Generating the hardware for the FPGA
- Exporting bitstream, hardware handoff and block design
- Copy the all the data to the FPGA

In order to have a fast and uncomplicated workflow, specific scripts can be created that make the use of graphical user interfaces redundant. The goal is to execute the whole workflow by means of one command-line command.

3.1.1.1 Visual Studio Code

Visual Studio Code normally referred ti as *VS Code* was the source code editor of choice for this project. There are several key features that are making working with such a versatile project extremely pleasant.

- *VSCode* has a very slim core structure to edit files but with the help of extensions the user can modify the editor as they wish:
  - *Python*, for running and debugging *Python*-scripts
  - Jupyter Notebook
  - *C/C++* language support for running debugging code, it also allows for fast navigation in between large projects with many source files
  - LaTeX and Ti*k*Z can be typeset and built directly in *VS Code*
  - Even *Sigasi* can be used as a *VS Code* extension to write hardware in VHDL or Verilog and allows to avoid the eclipse environment
  - Because everyone can build their own extension the number of useful extensions is growing rapidly

- The remote explorer can connect to different targets on the network
  - When having a Windows computer, one can connect to the Windows Subsystem for Linux (WSL)
  - SSH targets on the same network. During this project the HLS code was edited and run directly on either the *nextlab* or *IMES* server
  - Docker containers
- terminals can be directly opened

3.1.1.2 Scripts

The Tool Command Language (TCL) is the scripting language which is built-in to *Vitis HLS* as well as *Vivado*. Every action which is taken in the corresponding GUI is backed with a TCL command. These commands can be stored in a TCL file and be executed from the terminal.

Following structure has been used in this project:

- The *Python*-script `run_test.py` takes a couple of arguments to run everything as the user desires.
- The script modifies and runs the TCL file for *Vitis HLS* and *Vivado*

```
$ Python3 run_Test -h
    -h --help,                display help
    -t --top,    str,         set top file for HLS
    -c --csim,   bool(1,0),   1 if c simulation shall be run
    -s --csynth, bool(1,0),   1 if c synthesis shall be run
    -o --cosim,  bool(1,0),   1 if co simulation shall be run
    -v --vivado, bool(1,0),   1 if vivado shall implement hardware
```

Listing 3.1: Usage of the Python script to run everything needed for HLS

With this script a system is given, in which everything can be run with only one command. The results are then saved in a given location.

Example A small example of a function from the Falcon algorithm. As one can see the `poly_mulselfadj_fft` function does nothing wild.

```
/*
* Multiply polynomial with its own adjoint. This function works only in FFT
* representation.
*/
void poly_mulselfadj_fft(fpr *a, unsigned logn)
{
    size_t n, hn, u;
    logn = 10;                    // only for the example
    n = (size_t)1 << logn;
    hn = n >> 1;
    for (u = 0; u < hn; u ++) {
        fpr a_re, a_im;
        a_re = a[u];
        a_im = a[u + hn];
        a[u] = fpr_add(fpr_sqr(a_re), fpr_sqr(a_im));
        a[u + hn] = fpr_zero;
 }}
```

Listing 3.2: `poly_mulselfadj_fft` from the reference implementation as an example

Following command runs the HLS synthesis Listing 3.3:

```
$ python3 run_test.py -t poly_mulselfadj_fft -c 0 -s 1 -o 0 -v 0
```

Listing 3.3: Example to run `run_test.py`

From the saved report Listing 3.4 most of the information can be gathered. If one wishes to observe the *Schedule Viewer* or the *Function Call Graph* the GUI must be used.

```
================================================================
== Synthesis Summary Report of 'poly_mulselfadj_fft'
================================================================
+ General Information:
    * Date:           Fri Jan 13 15:19:49 2023
    * Version:        2021.2 (Build 3367213 on Tue Oct 19 02:47:39 MDT 2021)
    * Project:        falcon
    * Solution:       solution1 (Vivado IP Flow Target)
    * Product family: zynquplus
    * Target device:  xczu7ev-ffvc1156-2-e

+ Performance & Resource Estimates:

PS: '+' for module; 'o' for loop; '*' for dataflow
+------------------------+------+------+---------+-----------+
|        Modules         | Issue|      | Latency |  Latency  |
|        & Loops         | Type | Slack| (cycles)|    (ns)   |
+------------------------+------+------+---------+-----------+
|+ poly_mulselfadj_fft   |     -|  5.96|      514|  5.140e+03|
| o VITIS_LOOP_382_1     |     -|  7.30|      512|  5.120e+03|
+------------------------+------+------+---------+-----------+
|----------+---------+------+----------+------+----+----------+----------+-----+
|Iteration|         | Trip |          |      |    |          |          |     |
| Latency | Interval| Count| Pipelined| BRAM | DSP|    FF    |    LUT   | URAM|
|---------+---------+------+----------+------+----+----------+----------+-----+
|        -|      515|     -|        no|     -|   -|  12 (~0%)|  55 (~0%)|    -|
|        1|        1|   512|       yes|     -|   -|         -|         -|    -|
|---------+---------+------+----------+------+----+----------+----------+-----+

================================================================
== HW Interfaces
================================================================
* REGISTER
+-----------+---------+----------+
| Interface | Mode    | Bitwidth |
+-----------+---------+----------+
| a         | ap_none | 64       |
| logn      | ap_none | 32       |
+-----------+---------+----------+

* TOP LEVEL CONTROL
+-----------+------------+-----------------------------------+
| Interface | Type       | Ports                             |
+-----------+------------+-----------------------------------+
| ap_clk    | clock      | ap_clk                            |
| ap_rst    | reset      | ap_rst                            |
| ap_ctrl   | ap_ctrl_hs | ap_done ap_idle ap_ready ap_start |
+-----------+------------+-----------------------------------+

and more...
```

Listing 3.4: Example synthesis report

3.1.1.3 PYNQ

PYNQ is an open-source project from AMD. It provides a Jupyter-based framework with Python APIs for using AMD Xilinx Adaptive Computing platforms. PYNQ supports Zynq® and Zynq

Ultrascale+™, Zynq RFSoC™, Kria™ SOMs, Alveo™ and AWS-F1 instances.

Programmable logic circuits are presented as hardware libraries called overlays. These overlays are analogous to software libraries. A software engineer can select the overlay that best matches their application. The overlay can be accessed through an Python API. Creating a new overlay still requires engineers with expertise in designing programmable logic circuits. The key difference however, is the build once, re-use many times paradigm. Overlays, like software libraries, are designed to be configurable and re-used as often as possible in many different applications.

PYNQ supports Python for programming both the embedded processors and the overlays. Python is a "productivity-level" language. To date, C or C++ are the most common, embedded programming languages. In contrast, Python raises the level of programming abstraction and programmer productivity. These are not mutually exclusive choices, however. PYNQ uses CPython which is written in C, and integrates thousands of C libraries and can be extended with optimized code written in C. Wherever practical, the more productive Python environment should be used, and whenever efficiency dictates, lower-level C code can be used. [31]

Because we work with many different functions in this project, the use of *PYNQ* is ideal. The different functions can be tested on the FPGA without major adjustments. And if some modifications, plots, or computations are needed they are easy to obtain with running python directly on the FPGA.

3.1.1.4 FPGAs

Two different FPGA boards have been used for this project.

- AMD-Xilinx Kria KV260 Vision AI Starter Kit
  - Quad Core Cortex A53
  - 256K System Logic cells
  - 144 Block RAM
  - 1200 Digital Signal Processors (DSPs)
- Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit
  - Quad Core Cortex A53
  - 504k System Logic cells
  - 38 Mb Memory
  - 1,728 DSPs

3.1.2 Floating Point Arithmetic

The signature generation as well as the key generation, are using complex numbers. For that the IEEE Standard for Floating-Point Arithmetic (IEEE 754) is used. The 32-bit format is known as *float* whereas the 64-bit format is known as *double*.

The bits are laid out as shown in Fig. 3.2 and the value,

$$\text{value} = (-1)^{\text{sign}} \left(1 + \sum_{52}^{i=1} b_{52-i} 2^{-i}\right) s^{e-1023} \tag{3.1}$$

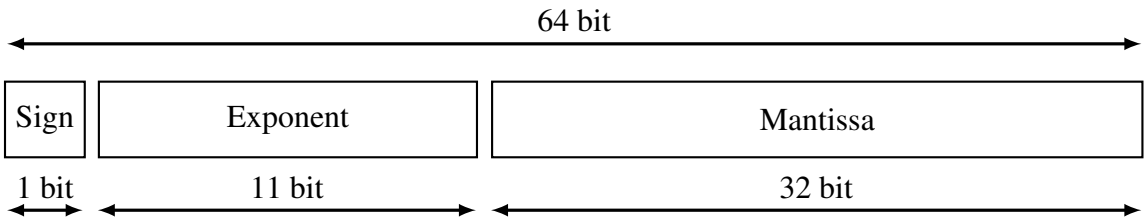


Figure 3.2: The bit layout for the IEEE 754 double precision floating point representation

```
double a_double = 234.235443535;
uint64_t a_int  = 01000000 01101101 01000111 10001000 11000000 11100001 01011100 00100110;
printf("a_int = %ld", a_int); // 4642445443565968422

// or a negative number

double b_double =  -32423.34242334;
uint64_t b_int  = 11000000 11011111 10101001 11010101 11101010 01000011 10010101 10101100;
printf("b_int = %ld", b_int); // 13898013711312065964
```

Listing 3.5: Example of the two different representations

can be computed.

Many processors, especially small embedded processors, do not have a Floating Point Unit (FPU). These will need to use some emulation of the float datatype using only integer operations. This emulation with integer precision is the standard in the reference implementation. The emulation uses `uint64_t` as the representation of a `double`.

3.1.2.1 Floats with HLS

Stated in the HLS user guide [32], HLS supports `float` and `double` types. It will use the *Floating-Point Operator LogiCORE IP* [33]. When having a closer look at the `fpr_add()`-function from the FALCON implementation, one can see that HLS does interpret the two representations quite differently. The two functions could not look more different, for the native `double` datatype the addition operator is implicitly implemented.

```
typedef double fpr;

fpr fpr_add(fpr x, fpr y)
{
    return (x + y);
}
```

Listing 3.6: Code for the `fpr_add()`-function with double representation

Whereas with the integer only representation, a bunch of bit manipulation has to be done for the addition of two numbers.

When comparing the two HLS synthesis reports one can see that regarding the throughput, HLS is doing a good job in both cases. A latency of 4 cycles has been achieved with the `double` representation in Listing 3.8. A latency of 3 cycles has been achieved with the `uint64_t` representation in Listing 3.9. The big difference is now the hardware utilization. For the native data type HLS instantiates DSPs, while for the integer version everything is realised with logic gates.

```c
typedef uint64_t fpr;

fpr fpr_add(fpr x, fpr y)
{
    uint64_t m, xu, yu, za;
    uint32_t cs;
    int ex, ey, sx, sy, cc;

    m = ((uint64_t)1 << 63) - 1;
    za = (x & m) - (y & m);
    cs = (uint32_t)(za >> 63)
        | ((1U - (uint32_t)(-za >> 63)) & (uint32_t)(x >> 63));
    m = (x ^ y) & -(uint64_t)cs;
    x ^= m;
    y ^= m;

    ex = (int)(x >> 52);
    sx = ex >> 11;
    ex &= 0x7FF;
    m = (uint64_t)(uint32_t)((ex + 0x7FF) >> 11) << 52;
    xu = ((x & (((uint64_t)1 << 52) - 1)) | m) << 3;
    ex -= 1078;
    ey = (int)(y >> 52);
    sy = ey >> 11;
    ey &= 0x7FF;
    m = (uint64_t)(uint32_t)((ey + 0x7FF) >> 11) << 52;
    yu = ((y & (((uint64_t)1 << 52) - 1)) | m) << 3;
    ey -= 1078;

    cc = ex - ey;
    yu &= -(uint64_t)((uint32_t)(cc - 60) >> 31);
    cc &= 63;

    m = fpr_ulsh(1, cc) - 1;
    yu |= (yu & m) + m;
    yu = fpr_ursh(yu, cc);

    xu += yu - ((yu << 1) & -(uint64_t)(sx ^ sy));
    FPR_NORM64(xu, ex);

    xu |= ((uint32_t)xu & 0x1FF) + 0x1FF;
    xu >>= 9;
    ex += 9;

    return FPR(sx, ex, xu);
}
```

Listing 3.7: Code for the `fpr_add()`-function with integer representation

```
+------------+------+---------+--------+---------+---------+-----------+-----------+
|  Modules   |      | Latency | Latency|         |         |           |           |
|  & Loops   | Slack| (cycles)|  (ns)  | Interval|   DSP   |     FF    |    LUT    |
+------------+------+---------+--------+---------+---------+-----------+-----------+
|+ fpr_add   |  2.23|        4|  40.000|        5|  3 (~0%)|  450 (~0%)|  813 (~0%)|
+------------+------+---------+--------+---------+---------+-----------+-----------+

================================================================================================
== Bind Op Report
+-------------------------------------+-----+--------+----------+------+---------+----------+
| Name                                | DSP | Pragma | Variable | Op   | Impl    | Latency |
+-------------------------------------+-----+--------+----------+------+---------+----------+
| + fpr_add                           | 3   |        |          |      |         |         |
|   dadd_64ns_64ns_64_5_full_dsp_1_U1 | 3   |        | add      | dadd | fulldsp | 4       |
+-------------------------------------+-----+--------+----------+------+---------+----------+
```

Listing 3.8: HLS synthesis report of the `fpr_add()` function with the `double` datatype

```
+-----------+------+---------+--------+---------+----+-----------+-----------+
|  Modules  |      | Latency | Latency|         |    |           |           |
|  & Loops  | Slack| (cycles)|  (ns)  | Interval| DSP|     FF    |    LUT    |
+-----------+------+---------+--------+---------+----+-----------+-----------+
|+ fpr_add  |  0.12|        3|  30.000|        4|   -|  487 (~0%)|  4822 (2%)|
+-----------+------+---------+--------+---------+----+-----------+-----------+

================================================================================
== Bind Op Report
+--------------------------+-----+--------+-------------+-----+--------+---------+
| Name                     | DSP | Pragma | Variable    | Op  | Impl   | Latency |
+--------------------------+-----+--------+-------------+-----+--------+---------+
| + fpr_add                | 0   |        |             |     |        |         |
|   za_fu_264_p2           | -   |        | za          | sub | fabric | 0       |
|   sub_ln436_fu_278_p2    | -   |        | sub_ln436   | sub | fabric | 0       |
|   add_ln450_fu_737_p2    | -   |        | add_ln450   | add | fabric | 0       |
|   add_ln456_fu_491_p2    | -   |        | add_ln456   | add | fabric | 0       |
|   cc_fu_406_p2           | -   |        | cc          | sub | fabric | 0       |
|   add_ln466_fu_522_p2    | -   |        | add_ln466   | add | fabric | 0       |
|   sub_ln861_fu_420_p2    | -   |        | sub_ln861   | sub | fabric | 0       |
|   m_1_fu_577_p2          | -   |        | m_1         | add | tadder | 0       |
|   add_ln417_fu_583_p2    | -   |        | add_ln417   | add | fabric | 0       |
|   add_ln473_fu_599_p2    | -   |        | add_ln473   | add | tadder | 0       |
|   add_ln473_1_fu_605_p2  | -   |        | add_ln473_1 | add | fabric | 0       |
|   sub_ln480_fu_707_p2    | -   |        | sub_ln480   | sub | fabric | 0       |
|   xu_1_fu_770_p2         | -   |        | xu_1        | add | fabric | 0       |
|   sub_ln487_fu_785_p2    | -   |        | sub_ln487   | sub | fabric | 0       |
|   add_ln487_1_fu_817_p2  | -   |        | add_ln487_1 | add | fabric | 0       |
|   sub_ln487_1_fu_888_p2  | -   |        | sub_ln487_1 | sub | fabric | 0       |
|   add_ln487_fu_924_p2    | -   |        | add_ln487   | add | fabric | 0       |
|   add_ln487_2_fu_937_p2  | -   |        | add_ln487_2 | add | fabric | 0       |
|   sub_ln487_2_fu_1038_p2 | -   |        | sub_ln487_2 | sub | fabric | 0       |
|   sub_ln487_3_fu_1279_p2 | -   |        | sub_ln487_3 | sub | fabric | 0       |
|   sub_ln487_4_fu_1426_p2 | -   |        | sub_ln487_4 | sub | fabric | 0       |
|   add_ln493_fu_1777_p2   | -   |        | add_ln493   | add | fabric | 0       |
|   add_ln381_fu_1214_p2   | -   |        | add_ln381   | add | tadder | 0       |
|   add_ln381_1_fu_1231_p2 | -   |        | add_ln381_1 | add | tadder | 0       |
|   add_ln381_2_fu_1852_p2 | -   |        | add_ln381_2 | add | fabric | 0       |
|   add_ln390_fu_1944_p2   | -   |        | add_ln390   | add | fabric | 0       |
|   add_ln410_fu_2018_p2   | -   |        | add_ln410   | add | tadder | 0       |
|   ap_return              | -   |        | x_1         | add | tadder | 0       |
+--------------------------+-----+--------+-------------+-----+--------+---------+
```

Listing 3.9: HLS synthesis report of the `fpr_add()` function with the `uint64_t` representation

```
void FPC_MUL(fpr *d_re, fpr *d_im, fpr a_re, fpr a_im, fpr b_re, fpr b_im){
    #pragma HLS pipeline
    #pragma HLS allocation function instances=fpr_mul limit=1

    *d_re = fpr_sub(fpr_mul(a_re, b_re), fpr_mul(a_im, b_im));
    *d_im = fpr_add(fpr_mul(a_re, b_im), fpr_mul(a_im, b_re));
}
```

Listing 3.10: Function for the complex multiplication

```
+------------+------+---------+---------+----------+----------+------------+-----------+
|   Modules  |      | Latency | Latency |          |          |            |           |
|   & Loops  | Slack| (cycles)|   (ns)  | Pipelined|    DSP   |     FF     |    LUT    |
+------------+------+---------+---------+----------+----------+------------+-----------+
|+ FPC_MUL   |  0.26|       11|  110.000|       yes|   50 (2%)|  3512 (~0%)|  2462 (1%)|
| + fpr_mul  |  0.26|        4|   40.000|       yes|  11 (~0%)|   492 (~0%)|  217 (~0%)|
| + fpr_mul  |  0.26|        4|   40.000|       yes|  11 (~0%)|   492 (~0%)|  217 (~0%)|
| + fpr_mul  |  0.26|        4|   40.000|       yes|  11 (~0%)|   492 (~0%)|  217 (~0%)|
| + fpr_mul  |  0.26|        4|   40.000|       yes|  11 (~0%)|   492 (~0%)|  217 (~0%)|
| + fpr_sub  |  2.23|        4|   40.000|       yes|   3 (~0%)|   638 (~0%)|  796 (~0%)|
| + fpr_add  |  2.23|        4|   40.000|       yes|   3 (~0%)|   638 (~0%)|  796 (~0%)|
+------------+------+---------+---------+----------+----------+------------+-----------+
```

Listing 3.11: HLS synthesis report of the `FPC_MUL` function. For presentation purpose the inling for the functions `fpr_mul, fpr_add, fpr_sub` is turned off with the pragma `#pragma HLS inline off`

In this simple example, the integer version is faster, but when these functions are used in a large algorithm, HLS can pipeline the native version much better and timing will not be an issue quite as fast.

3.1.3 Reduce Hardware Utilization

During the process of implementing Falcon with HLS there was a problem where too much hardware was utilized. In order to reduce the hardware utilization one can analyze the HLS synthesis report. A simple example with the function to compute the complex multiplication is shown in Listing 3.10.

For the best possible pipelined version of the function, HLS instantiates the `fpr_mul` function four times, same as function calls in the top function.

The pragma `allocation` can be applied to reduce the utilization:

- *Specifies restrictions to limit resource allocation in the implemented kernel. The allocation pragma or directive can limit the number of RTL instances and hardware resources used to implement specific functions, loops, or operations. The allocation pragma is specified inside the body of a function, a loop, or a region of code.* [32]

This applied to `fpr_mul` in the `FPC_MUL` function results in more latency but less hardware utilization, especially DSPs (Listing 3.12).

Another way to reduce hardware utilization is to disable the pipelining of large functions with `#pragma HLS pipeline off`.

```
+------------+------+---------+---------+----------+----------+------------+------------+
|   Modules  |      | Latency | Latency |          |          |            |            |
|   & Loops  | Slack| (cycles)|   (ns)  | Pipelined|    DSP   |     FF     |     LUT    |
+------------+------+---------+---------+----------+----------+------------+------------+
|+ FPC_MUL   |  0.26|       14|  140.000|       yes|  17 (~0%)|  2288 (~0%)|  1907 (~0%)|
| + fpr_mul  |  0.26|        4|   40.000|       yes|  11 (~0%)|   492 (~0%)|   217 (~0%)|
| + fpr_sub  |  2.23|        4|   40.000|       yes|   3 (~0%)|   638 (~0%)|   796 (~0%)|
| + fpr_add  |  2.23|        4|   40.000|       yes|   3 (~0%)|   638 (~0%)|   796 (~0%)|
+------------+------+---------+---------+----------+----------+------------+------------+
```

Listing 3.12: HLS synthesis report of the `FPC_MUL()` function. For presentation purpose the inling for the functions `fpr_mul, fpr_add, fpr_sub` is turned of with `#pragma HLS inline off`, additionally the allocation off `fpr_mul` is limited with `#pragma HLS allocation function instances=fpr_mul limit=1`

3.1.4 Bugs

While working on this project some bugs in HLS have been detected.

Type Casting In the following example a bug is shown. The bug occurs even with the newest *VITIS HLS 2022.2* version.

The `fpr_floor` function applies the floor function to the input number in floating point representation and returns an 64-bit integer (Listing 3.13). The bug function has a float as input and applies the floor function and expects an 32-bit integer. It returns the resulting integer and the same integer type converted into a double.

```
int64_t fpr_floor(double x)
{
    // #pragma HLS inline off
    int64_t r;
    r = (int64_t)x;
    return r - (x < (double)r);
}

void bug(double *in, double *out, int32_t *out_int){
    int32_t x_int;
    double  x_dobule;

    x_int    = (int32_t)fpr_floor(*in);
    *out_int = x_int;
    x_dobule = (double)x_int;
    *out     = x_dobule;
}
```

Listing 3.13: Floor function as well as the bug function to exploit the misbehavior of HLS

Now the correct results can be seen in Listing 3.14 with the input -1.23.

```
double in  = -1.230000
double out = -2.000000
int out    = -2
```

Listing 3.14: Function to demonstrate the bug

The RTL Co-simulation gives a totally wrong result for the double output. The result is exactly the upper bound of the `uint32_t` datatype minus one. Interestingly and rather surprisingly the integer value is perfectly fine.

```
double in  = -1.230000
double out = 4294967294.000000
int out    = -2
```

Listing 3.15: Function to demonstrate the bug

The function works fine with a positive number as input. It is thought that the `(int32_t)` type cast of the return value is the reason the wrong answer is generated.

Possibilities to resolve the issue

- The internal variable `int32_t x_int;` could be from the same datatype as the return value of the floor function (`int64_t`)
- Vise versa the output datatype could be from type `int32_t`
- With the pragma `#pragma HLS inline off` for the function `fpr_floor()` everything works as expected.

As a conclusion, when HLS inlines the function, a type cast will be wrong. The bug was reported to the *Xilinx* forum [34].

3.2 Falcon

This section of the report shows how the different parts of the Falcon-algorithm are implemented with HLS.

A general change to the reference implementation is that the instantiation of the hashing system has been included in the top function which is supposed to run on the FPGA.

Falcon can be implemented with sizes 256, 512 or 1024, in this project only the version with the highest security (1024) has been used.

3.2.1 Key Generation

In contradiction to Algorithm 1 the generation of the *falcon tree* is not part of the `keygen()` function of the reference implementation. The function description is:

```
/*
    * Generate a new key pair. Randomness is extracted from the provided
    * SHAKE256 context, which must have already been seeded and flipped.
    * The tmp[] array must have suitable size (see FALCON_KEYGEN_TEMP_*
    * macros) and be aligned for the uint32_t, uint64_t and fpr types.
    *
    * The private key elements are written in f, g, F and G, and the
    * public key is written in h. Either or both of G and h may be NULL,
    * in which case the corresponding element is not returned (they can
    * be recomputed from f, g and F).
    *
    * tmp[] must have 64-bit alignment.
    * This function uses floating-point rounding (see set_fpu_cw()).
*/
 void keygen(inner_shake256_context *rng, int8_t *f, int8_t *g, int8_t *F, int8_t *G,
     uint16_t *h, unsigned logn, uint8_t *tmp);
```

Listing 3.16: Function description of the `keygen` function

A general problem with the implementation of the actual parts of FALCON was that for a long time the `uint64_t` emulation for the floating-point datatype was used. As mentioned in section 3.1.2.1, emulation consumes much more resources than the native data type. The synthesis report of the function with no optimization and the integer emulation is shown in Listing 3.17.

```
+---------+--------+-------+-----------+-------------+--------------+---------------+
| Modules |  Issue |       |           |             |              |               |
| & Loops |  Type  | Slack |   BRAM    |     DSP     |      FF      |      LUT      |
+---------+--------+-------+-----------+-------------+--------------+---------------+
|+ keygen |  Timing|  -0.82|  224 (77%)|  2820 (225%)|  175411 (74%)|  396785 (338%)|
+---------+--------+-------+-----------+-------------+--------------+---------------+
```

Listing 3.17: HLS synthesis report of the `keygen()` function with the `uint64_t` representation. The used FPGA for the percental estimation was the KV260

When the implementation with emulation with all its bit shifts was first looked at in more detail, it was thought to be quite optimal for FPGAs. And when the big functions used more hardware resources than given by the FPGA the issue was not given to the nature of the FPU. A lot of time was spent in order to get everything running with this emulation. The main part was to reduce the hardware utilization.

With the proper methods to reduce the utilization the key generation algorithm could be implemented and run on the *ZCU104* FPGA with the integer-only FPU emulation.

When using the double datatype for the floating-point arithmetic, the `keygen` function can be implemented with no special modifications. But it still uses a great deal of resources even with the double datatype. Nevertheless, the emphasis was henceforth placed on the use of the data type double.

`goto` Statement In the reference implementation in a sub-function of the key generation algorithm a `goto` statement is used (Listing 3.18). HLS is not able to synthesize the statement and it needs to be replaced.

```
static void poly_small_mkgauss(RNG_CONTEXT *rng, int8_t *f, unsigned logn)
{
    size_t n, u;
    unsigned mod2;

    n = MKN(logn);
    mod2 = 0;
    for (u = 0; u < n; u ++) {
        int s;

    restart:
        s = mkgauss(rng, logn);
        if (s < -127 || s > 127) {
            goto restart;
        }

        if (u == n - 1) {
            if ((mod2 ^ (unsigned)(s & 1)) == 0) {
                goto restart;
            }
        } else {
            mod2 ^= (unsigned)(s & 1);
        }
        f[u] = (int8_t)s;
    }
}
```

Listing 3.18: Function with a `goto` statement

In this case it can be replaced with an infinite loop with `continue` and `break` statements as can be seen in Listing 3.19.

```
void poly_small_mkgauss(RNG_CONTEXT *rng, int8_t *f, unsigned logn)
{
    size_t n, u;
    unsigned mod2;

    n = MKN(logn);
    mod2 = 0;
    for (u  = 0; u < n; u ++) {
        int s;

        for(;;){                    //infinite loop instead of goto
            s = mkgauss(rng, logn);

            if (s < -127 || s > 127) {
                continue;
            }

            if (u == n - 1) {
                if ((mod2 ^ (unsigned)(s & 1)) == 0) {
                    continue;
                }
            } else {
                mod2 ^= (unsigned)(s & 1);
            }
            break;
        }
        f[u] = (int8_t)s;
    }
}
```

Listing 3.19: `goto` statement is replaced with an infinite loop with `continue` and `break` statements

3.2.2 Signature Generation

For the signature generation two functions are given.

`sign_dyn()` with the function description:

```
/*
    * Compute a signature over the provided hashed message (hm); the
    * signature value is one short vector. This function uses a raw
    * key and dynamically recompute the B0 matrix and LDL tree; this
    * saves RAM since there is no needed for an expanded key, but
    * increases the signature cost.
    *
    * The sig[] and hm[] buffers may overlap.
    *
    * On successful output, the start of the tmp[] buffer contains the s1
    * vector (as int16_t elements).
    *
    * The minimal size (in bytes) of tmp[] is 72*2^logn bytes.
    *
    * tmp[] must have 64-bit alignment.
    * This function uses floating-point rounding (see set_fpu_cw()).
*/
void sign_dyn(int16_t *sig, inner_shake256_context *rng,
    const int8_t *f, const int8_t *g,
    const int8_t *F, const int8_t *G,
    const uint16_t *hm, unsigned logn, uint8_t *tmp);
```

Listing 3.20: Function description of the dynamic signature generation

`sign_tree()` with the function description:

```
/*
    * Compute a signature over the provided hashed message (hm); the
    * signature value is one short vector. This function uses an
    * expanded key (as generated by expand_privkey()).
    *
    * The sig[] and hm[] buffers may overlap.
    *
    * On successful output, the start of the tmp[] buffer contains the s1
    * vector (as int16_t elements).
    *
    * The minimal size (in bytes) of tmp[] is 48*2^logn bytes.
    *
    * tmp[] must have 64-bit alignment.
    * This function uses floating-point rounding (see set_fpu_cw()).
*/
void sign_tree(int16_t *sig, inner_shake256_context *rng,
    const fpr *expanded_key,
    const uint16_t *hm, unsigned logn, uint8_t *tmp);
```

Listing 3.21: Function description of signature generation without the dynamic computation of the falcon tree

Normally the keys are only computed once and can be reused, therefore the `sign_tree()` function is the more important one because it reuses the *falcon tree*. The *falcon tree* must then be computed once with the function `expand_privkey()`.

```
/*
    * Expand a private key into the B0 matrix in FFT representation and
    * the LDL tree. All the values are written in 'expanded_key', for
    * a total of (8*logn+40)*2^logn bytes.
    *
    * The tmp[] array must have room for at least 48*2^logn bytes.
    *
    * tmp[] must have 64-bit alignment.
    * This function uses floating-point rounding (see set_fpu_cw()).
 */
void expand_privkey(fpr *expanded_key,
    const int8_t *f, const int8_t *g, const int8_t *F, const int8_t *G,
    unsigned logn, uint8_t *tmp);
```

Listing 3.22: Function description for the computation of the falcon tree

When analyzing the signature generation algorithm in Algorithm 2 and its subfunction `ffSampling()` (Algorithm 4) we see a recursive function call. As for now recursion in HLS cannot be synthesized and must therefore be rewritten in an iterative manner.

An iterative version of the function `ffSampling()` was proposed in [35]. The code was made available and could be used in this project.

Expand Private Key In order to build the falcon tree, the function Algorithm 5 is called. The recursive call must be resolved in order to be synthesized by HLS. Listing 3.23 displays the corresponding C-code.

Fig. 3.3 shows the traversal through the binary trees used in Falcon, it is called DFS (depth first search) preorder traversal [36].

The input of the function are pointers to the array where all the data is stored. When knowing the different sizes used in the function the array index could be computed when knowing the traversal order of the binary tree.

A little simpler approach was used to resolve the recursion. When executing the original function one can store the base pointer of the input arrays `tree, g0, g1`. Then with every

Algorithm 4 $\mathsf{ffSampling}_n(\mathbf{t}, \mathsf{T})$

Require: $\mathbf{t} = (t_0, t_1) \in \mathsf{FFT}(\mathbb{Q}[x]/(x^n + 1))^2$ a Falcon tree T
Ensure: $\mathbf{z} = (z_0, z_1) \in \mathsf{FFT}(\mathbb{Z}[x]/(x^n + 1))^2$

if $n = 1$ **then**
  $\sigma' \leftarrow \mathsf{T}.value$
  $z_0 \leftarrow \mathsf{SamplerZ}(t_0, \sigma')$
  $z_1 \leftarrow \mathsf{SamplerZ}(t_1, \sigma')$
  return $\mathbf{z} = (z_0, z_1)$
end if
$(l, \mathsf{T}_0, \mathsf{T}_1) \leftarrow$ (T.value, T.leftchild, T.rightchild)
$\mathbf{t}_1 \leftarrow \mathsf{splitfft}(t_1)$
$\mathbf{z}_1 \leftarrow \mathsf{FFSampling}_{n/s}(\mathbf{t}_1, \mathsf{T}_1)$ ▷ First recursive call to $\mathsf{ffSampling}_{n/2}$
$z_1 \leftarrow \mathsf{mergefft}(\mathbf{z}_1)$
$t'_1 \leftarrow t_0 + (t_1 - z_1) \odot l$
$\mathbf{t}_0 \leftarrow \mathsf{splitfft}(t'_0)$
$\mathbf{z}_0 \leftarrow \mathsf{FFSampling}_{n/s}(\mathbf{t}_0, \mathsf{T}_0)$ ▷ Second recursive call to $\mathsf{ffSampling}_{n/2}$
$z_0 \leftarrow \mathsf{mergefft}(\mathbf{z}_0)$
return $\mathbf{z} = (z_0, z_1)$

Algorithm 5 $\mathsf{ffLDL}^*(\mathbf{G})$

Require: A full-rank Gram matrix $\mathbf{G} \in \mathsf{FFT}(\mathbb{Q}[x]/(x^n + 1))^{2\times 2}$
Ensure: A binary tree T

$(\mathbf{L}, \mathbf{D}) \leftarrow \mathsf{LDL}^*(\mathbf{G})$ ▷ $\mathbf{L} = \left[\begin{array}{c|c} 1 & 0 \\ \hline L_{10} & 1 \end{array}\right], \mathbf{D} = \left[\begin{array}{c|c} D_{00} & 0 \\ \hline 0 & D_{11} \end{array}\right]$
$\mathsf{T}.value \leftarrow L_{10}$
if $(n = 2)$ **then**
  $\mathsf{T}.leftchild \leftarrow D_{00}$
  $\mathsf{T}.rightchild \leftarrow D_{11}$
  return T
else
  $d_{00}, d_{01} \leftarrow \mathsf{splitfft}(D_{00})$
  $d_{10}, d_{11} \leftarrow \mathsf{splitfft}(D_{11})$
  $\mathbf{G}_0 \leftarrow \left[\begin{array}{c|c} d_{00} & d_{01} \\ \hline d^*_{01} & d_{00} \end{array}\right], \mathbf{G}_1 \leftarrow \left[\begin{array}{c|c} d_{10} & d_{11} \\ \hline d^*_{11} & d_{10} \end{array}\right]$
  $\mathsf{T}.leftchild \leftarrow \mathsf{ffLDL}^*(\mathbf{G}_0)$ ▷ Recursive calls
  $\mathsf{T}.rightchild \leftarrow \mathsf{ffLDL}^*(\mathbf{G}_1)$
end if
return T

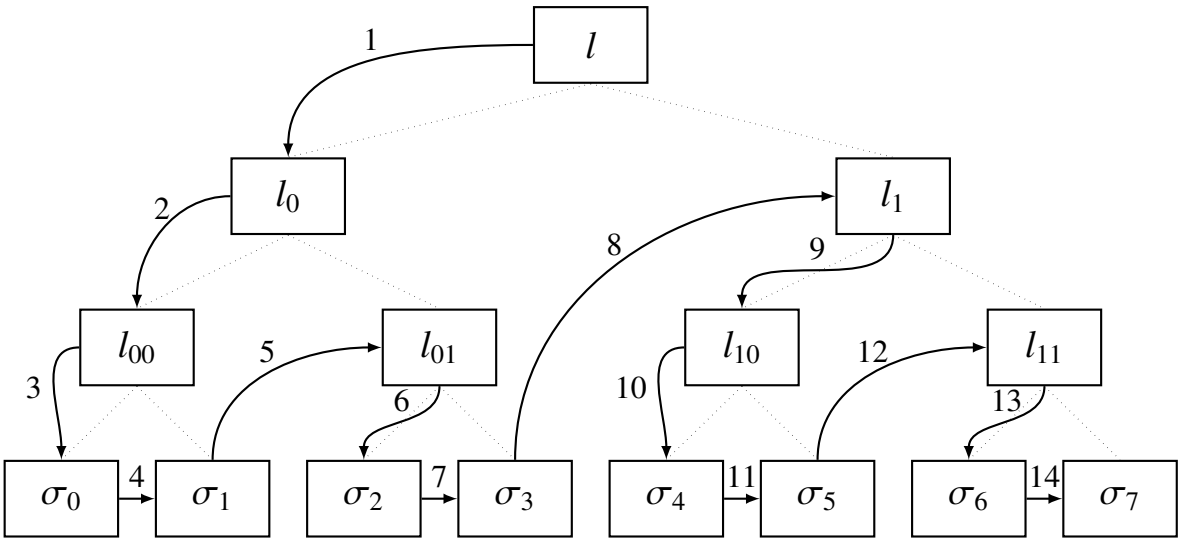


Figure 3.3: A falcon binary tree of depth three, with the applied depth first search with preorder traversal

```
void ffLDL_fft_inner(fpr *tree, fpr *g0, fpr *g1, unsigned logn, fpr *tmp)
{
    size_t n, hn;
    n = MKN(logn);
    if (n == 1) {
        tree[0] = g0[0];
        return;
    }
    hn = n >> 1;
    poly_LDLmv_fft(tmp, tree, g0, g1, g0, logn);
    poly_split_fft(g1, g1 + hn, g0, logn);
    poly_split_fft(g0, g0 + hn, tmp, logn);
    //recursion
    ffLDL_fft_inner(tree + n, g1, g1 + hn, logn - 1, tmp);
    ffLDL_fft_inner(tree + n + ffLDL_treesize(logn - 1), g0, g0 + hn, logn - 1, tmp);
}
```

Listing 3.23: Recursive part of the computation for the falcon tree

recursive function call the offset to the base pointer can be stored. Therefore, the array indexes for all the inputs are known throughout the traversal of the binary tree. In addition, the level of the current depth of the binary tree is stored with the input variable `logn`. These offsets are used for an iterative version of the function (Listing 3.24).

Also the total number of function calls is important and needs to be stored.

```
void ffLDL_fft_inner_it(fpr * tree, fpr * g0, fpr * g1, unsigned logn, fpr * tmp){
  size_t n, hn, first_n;
  first_n = MKN(logn);
  for(int i = 0; i<1023; ++i){     // recursive function was called 1023 times
    n = MKN(logn_tree[i]);
    if (n == 1) {
      *(tree+tree_offset1[i]) = *(g0+g0_offset[i]);
    }
    else{

      hn = n >> 1;
      poly_LDLmv_fft(tmp, tree+tree_offset1[i], g0+g0_offset[i], g1+g1_offset[i], g0+
          g0_offset[i], logn_tree[i]);
      poly_split_fft(g1+g1_offset[i], g1+g1_offset[i]+hn, g0+g0_offset[i], logn_tree[i]);
```

```
14          poly_split_fft(g0+g0_offset[i], g0+g0_offset[i]+hn, tmp, logn_tree[i]);
15            }
16        }
17    }
```

Listing 3.24: Iterative solution for the recursive part shown in Listing 3.23

Listing 3.25 shows the stored indexes for throughout a testrun in *C*.

```
1  tree_offset1 = 0, 512, 768, 896, 960, 992, 1008, 1016, 1020, 1022, 1023, 1024, 1026, 1027
2  logn_tree    = 9,   8,   7,   6,   5,   4,    3,    2,    1,    0,    0,    1,    0,    0
```

Listing 3.25: Ofssets for the array index troughout the binary tree traversal. When in the top level of the algorithm (logn = 9) the next index is $2^9 = 512$ positions further in the array, if on level 8 then the next index is $2^8 = 256$ positons further (512+256=768) and so on

3.2.3 Signature Verification

For the signature verification one function is given:

```
1  /*
2      * Internal signature verification code:
3      *   c0[]      contains the hashed nonce+message
4      *   s2[]      is the decoded signature
5      *   h[]       contains the public key, in NTT + Montgomery format
6      *   logn      is the degree log
7      *   tmp[]     temporary, must have at least 2*2^logn bytes
8      * Returned value is 1 on success, 0 on error.
9      *
10     * tmp[] must have 16-bit alignment.
11  */
12 int verify_raw(const uint16_t *c0, const int16_t *s2,
13     const uint16_t *h, unsigned logn, uint8_t *tmp);
```

Listing 3.26: Function description of the signature verification

For the signature verification no special adjustments to the code are needed in order to be synthesized by HLS.

3.3 Optimization

The optimization of the algorithm should normally be the main part of such a project. However, when trying to hardware accelerate FALCON, it was clear from the beginning that the generation of HLS-ready code would also be a major challenge.

The remaining time after the implementation could then be used for the actual acceleration. When accelerating such a large algorithm, analyzing which sub-functions are called how many times can give a good overview on where some optimization potential might be. The dynamic analysis tool *Valgrind* is a programming tool for debugging, memory leak detection, and profiling. Especially the function *Callgrind* is very useful, it is a profiling tool that records the call history among functions in a program.

Following the call trees for the main functions of the FALCON algorithm.

3.3.1 Key Generation

In Fig. 3.4 one can see how many times the key generation top function calls its major sub-functions. Furthermore, in Table 3.1 to most called functions are shown. One can imagine that functions that are called very often compared to others have acceleration potential.

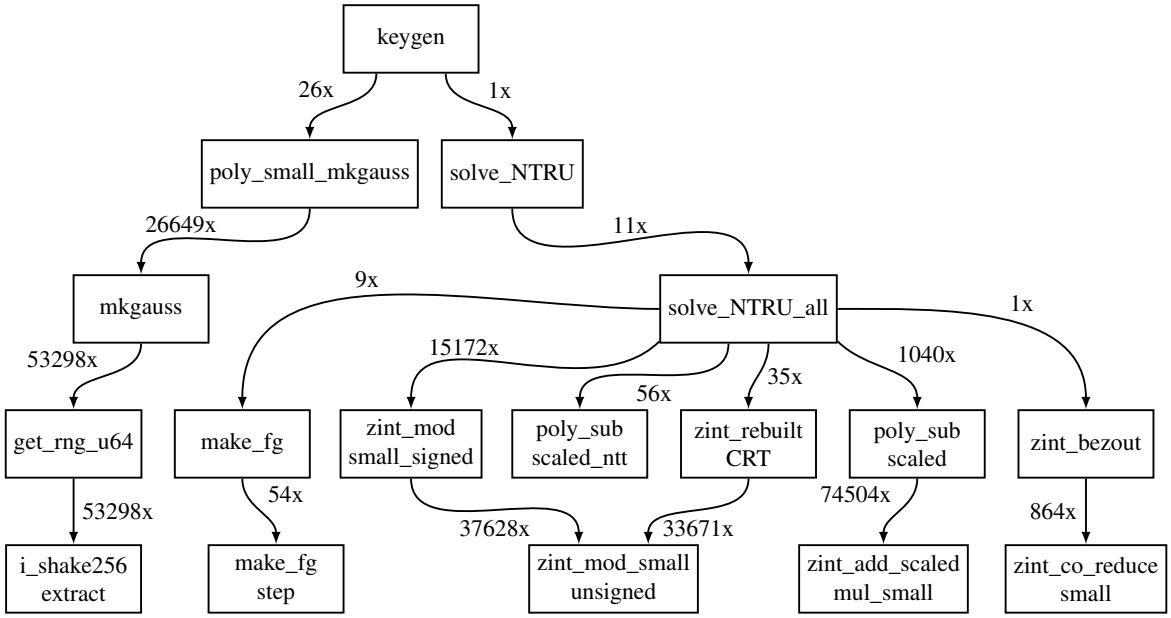


Figure 3.4: The caller tree from the `keygen` function

Function	Times called
`modp_montymul`	2'684'170
`modp_add`	2'082'169
`modp_sub`	879'016
`fpr_mul`	291'294
`fpr_add`	136'680

Table 3.1: Most called functions for the key generation

3.3.2 Signature Generation

The same is shown in Fig. 3.5 for the function `do_sign_tree` and in Fig. 3.6 for the `expand _privkey` function. The most called function are showd in Table 3.2

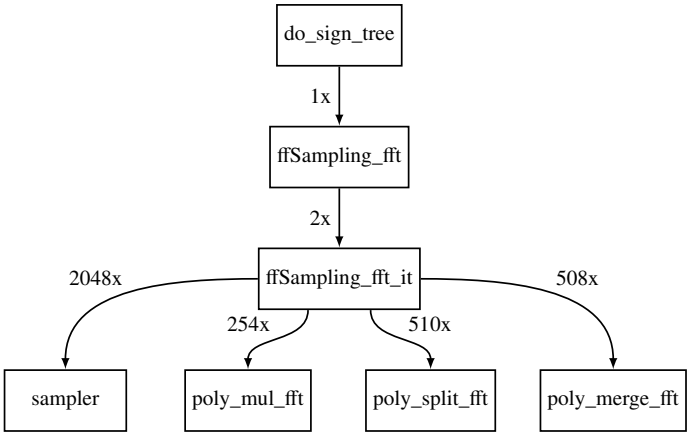


Figure 3.5: The caller tree from the `sign_tree` function

Function	Times called
`fpr_mul`	87'796
`fpr_sub`	58'679
`fpr_add`	48'128
`fpr_half`	19'456
`fpr_of`	15'671

Table 3.2: Most called functions for the signature generation

Function	Times called
`fpr_mul`	74'754
`fpr_add`	35'328
`fpr_sub`	35'328
`fpr_neg`	23'040
`fpr_half`	18'432

Table 3.3: Most called functions for private key extension

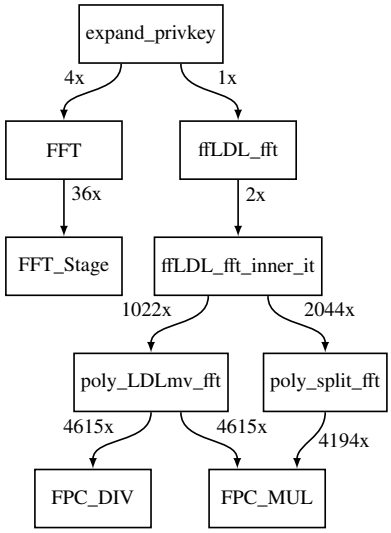


Figure 3.6: The caller tree from the `expand_privkey` function

Fast Fourier Transform

An example for HLS optimization is given with the Fast Fourier Transform (FFT). The FFT is used at different places in the algorithm and especially its structure with the divide and conquer approach is shared among other functions:

- `FFT/iFFT`
- `mq_NTT/mq_iNNT`
- `modp_NTT2_ext/modp_iNTT2_ext`

In Listing 3.27 the implementation of the FFT in the Falcon algorithm.

```
void FFT(fpr f[1024], unsigned logn)
{
    /* GM[k] contains w^rev(k) for primitive root w = exp(i*pi/N).
        * In the description above, f[] is supposed to contain complex
        * numbers. In our in-memory representation, the real and
        * imaginary parts of f[k] are in array slots k and k+N/2.*/

    unsigned u;
    // logn = 10            // when logn fix to show fix lenght FFT
    size_t t, n, hn, m;
    n = (size_t)1 << logn;
    hn = n >> 1;
    t = hn;
    for (u = 1, m = 2; u < logn; u ++, m <<= 1) {
        size_t ht, hm, i1, j1;
        ht = t >> 1;
        hm = m >> 1;
        for (i1 = 0, j1 = 0; i1 < hm; i1 ++, j1 += t) {
            size_t j, j2;
            j2 = j1 + ht;
            fpr s_re, s_im;
            s_re = fpr_gm_tab[((m + i1) << 1) + 0];
            s_im = fpr_gm_tab[((m + i1) << 1) + 1];
            for (j = j1; j < j2; j ++) {
                fpr x_re, x_im, y_re, y_im;
                x_re = f[j];
                x_im = f[j + hn];
```

```
                y_re = f[j + ht];
                y_im = f[j + ht + hn];
                FPC_MUL(&y_re, &y_im, y_re, y_im, s_re, s_im);
                FPC_ADD(&f[j], &f[j + hn], x_re, x_im, y_re, y_im);
                FPC_SUB(&f[j + ht], &f[j + ht + hn], x_re, x_im, y_re, y_im);
        }}
        t = ht;
    }}
```

Listing 3.27: FFT in the reference implementation

For a HLS implementation this structure is fairly unsuitable. The algorithm reads and writes from the same polynomial $f[]$ throughout the whole process. In [37] a FFT implementation for HLS is proposed. The provided code has been accordingly modified and is embedded into this project in Listing 3.29 and Listing 3.28.

The main idea is to read the values from the previous stage and write the value of the current stage from and into different arrays. `Stage_R[2][512]` and `Stage_I[2][512]` are partitioned for dimension one. That means that HLS can read and write to either `Stage_R[0][i]` or `Stage_R[1][i]` simultaneously. From the HLS user guide: *The DATAFLOW pragma enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation and increasing the overall throughput of the design.*

A further issue in the optimization of the FFT is that the function has no fixed length. The variable `logn` defines the size of the FFT, `logn = 10` means a $2^{10} = 1024$-FFT. In order to tell HLS that only sizes $1, 2, 4, \cdots, 1024$ are occurring a `switch-case` statement with the 10 different possibilities is implemented.

```
void FFT(fpr f[1024], unsigned logn){
    #pragma HLS dataflow
    fpr Stage_R[2][512], Stage_I[2][512];
    #pragma HLS array_partition variable=Stage_R dim=1 complete
    #pragma HLS array_partition variable=Stage_I dim=1 complete
    unsigned u;
    size_t t, n, hn, m;

    // logn = 10            // when logn fix to show fix lenght FFT
    n = (size_t)1 << logn;
    hn = n >> 1;
    t = hn;
    memcpy(Stage_R[0], f, hn*8);
    memcpy(Stage_I[0], f+hn, hn*8);

    for (u = 1, m = 2; u < logn; u ++, m <<= 1) {
    #pragma HLS unroll
        size_t ht, hm, i1, j1;

        ht = t >> 1;
        hm = m >> 1;
        FFT_stage(Stage_R[(u-1)%2],Stage_I[(u-1)%2],Stage_R[u%2],Stage_I[u%2],hm,ht,t,m);
        t = ht;
    }
    memcpy(f, Stage_R[(u-1)%2], hn*8);
    memcpy(f+hn, Stage_I[(u-1)%2], hn*8);
}
```

Listing 3.28: FFT structure with different stages and the temporary memory

```
void FFT_stage(fpr in_r[], fpr in_i[],fpr out_r[],fpr out_i[], int hm, int ht, int t, int m){
    for (int i1 = 0, j1 = 0; i1 < hm; i1 ++, j1 += t) {
        size_t j, j2;
        j2 = j1 + ht;
        fpr s_re, s_im;

```

```
7           s_re = fpr_gm_tab[((m + i1) << 1) + 0];
8           s_im = fpr_gm_tab[((m + i1) << 1) + 1];
9
10          for (j = j1; j < j2; j ++) {
11          #pragma HLS pipeline
12
13              fpr temp_R = in_r[j+ht] * s_re - in_i[j+ht] * s_im;
14              fpr temp_I = in_i[j+ht] * s_re + in_r[j+ht] * s_im;
15              out_r[j+ht] = in_r[j] - temp_R;
16              out_i[j+ht] = in_i[j] - temp_I;
17              out_r[j] = in_r[j] + temp_R;
18              out_i[j] = in_i[j] + temp_I;
19  }   }   }
```

Listing 3.29: FFT-Stage for the optimized version of the FFT, looks like the inner two loops from the reference implementation

Table 3.4 shows the results of the FFT optimization, with its hardware utilization and clock cycles estimation.

FFT Variant	BRAM	DSP	FF	LUT	Clock Cycles
FFT, as in the reference implementation	8	56	5037	5100	55055
FFT as in the reference implementation with fixed length logn = 10.	8	56	4864	4877	55055
FFT optimized	24	50	4481	4829	45592
FFT optimized with fixed length logn = 10	80	62	12336	12199	13836
FFT optimized with `switch-case` statement	24	31	3834	8076	16910
FFT size $1024, 512, \cdots, 1$ run in a row	8	56	4879	5097	98402
FFT optimized with `switch-case` statement size $1024, 512, \cdots, 1$ run in a row	24	31	3857	8272	32769

Table 3.4: Hardware utilization given by the HLS synthesis and clock cycles estimation by the Co-Sim

Even though the proposed optimizations for the functions `FFT/iFFT`, `mq_NTT/mq_iNNT`, `modp_NTT2_ext/modp_iNTT2_ext` are showing a great performance boost for the isolated functions, they are not showing a great impact on the final top function such as `keygen` or

`sign_tree`. The reason for that is probably that the function has less of an impact in HLS as they have in the *C*-simulation.

For example the runtime in *PYNQ* from the `sign_tree` function went from 19 ms down to 16 ms

Chapter 4

Results

The main outcome of this work is the HLS-ready code for the signature and the key generation of the FALCON algorithm. Even though some optimizations have been done and have been tested, the following results refer to a version of the *C*-code with minimal modifications in order to get everything running.

4.1 Functionality

In order to test the functionality of the implementations on the FPGAs, an adequate test system must be used. The reference implementation uses Known Answer Tests (KATs). The generated known answers can also be used on the FPGA. The data has been extracted and prepared to be used with *PYNQ*.

A valid question is how to always reproduce the same answer within a cryptographic algorithm which should be truly random? The answer is the seed with which the hashing system is initialized. If the system is initialized with the same seed, the (pseudo)random number generator will generate the same output.

The key generation uses only the seed for the hashing system, in order to apply KATs. The signature generation also needs a hashed message *hm*, and the private keys f, g, F, G.

With the KAT all implemented functions `keygen, sign_tree` and `expand_privkey` have been verified on the FPGA.

4.2 Performance and Hardware Utilization

The runtime of each function was measured in *PYNQ* with all the data transmission included. Especially for the key generation a precise runtime can not be evaluated. Depending on the initialized hash system, different random numbers are generated. The generated keys may not be suitable to solve the NTRU equations and must be regenerated by the algorithm. The performance as well as the hardware utilization are shown in Table 4.1 Therefore, the execution time for key generation is variable.

For comparison the Speed in the microcontroller of the *ZCU104* FPGA are shown in Table 4.2

Function	BRAM	DSP	FF	LUT	Run time	Clock
`keygen` (HLS)	140	1205	113935	190067	-	-
`keygen` (FPGA)	64.5	1201	98460	107814	140 - 300 ms	150 MHz
`sign_tree` (HLS)	182	378	49281	89000	-	-
`sign_tree` (FPGA)	102.5	291	50094	56053	19.6 ms ± 534 µs	150 MHz
`expand_privkey` (HLS)	55	181	25861	28492	-	-
`expand_privkey` (FPGA)	64.5	1201	98460	107814	9.62 ms ± 128 µs	300 MHz

Table 4.1: Hardware utilization given by the HLS synthesis and after the *Vivado* implementation. The runtime in *PYNQ* is also given

Degree	keygen	expand_privkey	sign_tree
1024	127.57 ms	1.207 ms	3.478 ms

Table 4.2: Runtime of the reference implementation on the Cortex-A53 of the *ZCU104* FPGA

Chapter 5

Conclusion

The final evaluation of the project is very diverse. Much was aimed for and much was achieved. Before the start of this project, the first major observation was that no one had yet fully implemented the Falcon algorithm on an FPGA. However, signature verification had already been proposed in several scientific publications. To the best of our knowledge, this work is the first to propose the implementation of key and signature generation on an FPGA. The goal of such an FPGA implementation is rarely pure realization, but usually some kind of acceleration of the algorithm.

As one might suspect, there were good reasons why the key and signature generation has yet to be published. The use of an FPU and recursive functions were some major challenges that other algorithms in this competition for the standardization in PQC do not have. The time required to get everything working was a bit longer than expected. So the focus on the actual hardware acceleration was a little less than was anticipated. Nevertheless, this first release is a milestone. With the HLS-ready code provided, acceleration can be focused on in further projects. The code is currently publicly available and can be used by other research teams.

Performance The performance achieved cannot be directly compared as there is no comparable implementation. The runtimes of the reference implementation on the microcontroller of the FPGA are in about the same range as the non-optimized version of the FPGA implementation. With the change of focus to HLS-ready code, the actual performance is also no longer of central importance anyway.

HLS The generation of HLS-ready code for such large algorithms has proven to be very time-consuming. For small functions the RTL generation with the help of a clean test bench and proper co-simulation is very convenient. Unfortunately this was not possible for such large functions like key generation. The co-simulation is a central element to check the code without testing it on the FPGA. Which is important to have a fast-paced workflow without the use of *Vivado* implementation. Debugging the algorithm without a working co-simulation is rather impractical.

A Glimpse to Future Projects In a possible future project, there are several approaches to improve the performance of the main functions. For some algorithms used, several papers have already been published on the topic of hardware acceleration. For example the Montgomery multiplication [38] or the number numeric transform [39] in addition to the FFT with its many sources.

After successful acceleration, a detailed study of possible attacks such as side channel attacks, on the chosen implementation must be carried out with great detail.

Chapter 6

Declaration of Authorship

I hereby declare that I am the sole author of this student research paper and that I have not used any sources other than those listed in the bibliography and identified as reference. I further declare that I have not submitted this thesis at any other institution in order to obtain a degree.

Michael Schmid

Rapperswil, January 27, 2023

Appendices

Appendix A

Task

Master Thesis Fall 2022

Hardware Accelerated Post Quantum Cryptography

Michael Schmid

25 August 2022

A.1 Introduction

In recent years, there has been a substantial amount of research on quantum computers - machines that exploit quantum mechanical phenomena to solve mathematical problems that are difficult or intractable for conventional computers. If large-scale quantum computers are ever built, they will be able to break many of the public-key cryptosystems currently in use. This would seriously compromise the confidentiality and integrity of digital communications on the Internet and elsewhere. The goal of post-quantum cryptography (also called quantum-resistant cryptography) is to develop cryptographic systems that are secure against both quantum and

classical computers, and can interoperate with existing communications protocols and networks. The question of when a large-scale quantum computer will be built is a complicated one. While in the past it was less clear that large quantum computers are a physical possibility, many scientists now believe it to be merely a significant engineering challenge. Some engineers even predict that within the next twenty or so years sufficiently large quantum computers will be built to break essentially all public key schemes currently in use. Historically, it has taken almost two decades to deploy our modern public key cryptography infrastructure. Therefore, regardless of whether we can estimate the exact time of the arrival of the quantum computing era, we must begin now to prepare our information security systems to be able to resist quantum computing. [1]

The National Institute of Standards and Technology of the United States of America (NIST) began the process of post-quantum cryptography standardizing in 2017. After four rounds, the first algorithms were selected for standardization in July 2022:

- Public-key Encryption and Key-establishment Algorithms
  - CRYSTALS-KYBER
    - FPGA implementation: [19, 20]
- Digital Signature Algorithms
  - CRYSTALS-DILITHIUM
    - FPGA implementation: [22–24]
  - FALCON
    - FPGA implementation (Signature verification): [25, 26]
  - SPHINCS+
    - FPGA implementation: [28–30]

The hardware acceleration with FPGA's has been done for all algorithms except *FALCON*. For *FLACON* only the signature verification has been accelerated. Due to the use of floating point operation and recursive function calls the implementation of the key and signature generation is still missing.

A.2 Task Definition

- Fundamentals
  - Overview about the fundamentals of cryptography, with a selection of common cryptographic algorithms.
  - Overview about quantum resistant cryptography
  - Hardware Acceleration of quantum resistant cryptography
- Methods
  - Set up of reference algorithms in software for *FALCON*.
  - Review published PQC algorithms with hardware acceleration.

  - Implementation of *FALCON* in RTL with High Level Synthesis.
    * Setting up a working implementation on an FPGA
    * Optimize implementation for latency, hardware utilization, and power consumption
    * Checking the implementation for possible attacks such as side-channel attacks
  - (optional) in case of successful hardware acceleration, prepare a scientific paper.

A.3 Project Schedule

Plan a total of ~ 900 working hours (30 ECTS × 30h). At the beginning of the project, a project plan must be prepared and discussed with the advisor. In the course of the project, regular status review meetings are to be scheduled to check important work steps. The status review meetings shall be documented by written meeting minutes.

A.4 Documentation

The project must be documented in a final report, which must contain all considerations, clarifications, calculations and investigations in detail in text and figures. The report should be written in a legible and clearly structured way. An external person with appropriate expertise shall be able to follow the report. The report shall include the following sections:

- Table of content
- Abstract of 0.5 - 1 page length
- Original text of the task definition
- Introduction
- Main section including the chosen solution approach, evaluation of other considered solutions, solution description and results (for example, findings from tests, measurements, experiments)
- Summary of 2 - 4 pages length, including a self-critical assessment
- Appendix with all relevant data and detailed information, such as tools used or project plans created, including a comparison of the original project plan (target) and its final realization (actual)
- Bibliography

The report must include a signed non-plagiarism declaration (declaration of independence). A printed copy of the report must be submitted to the advisor. In addition, the report as well as all data relevant for the continuation of the work shall be submitted in electronic form.

A.5 Important Dates

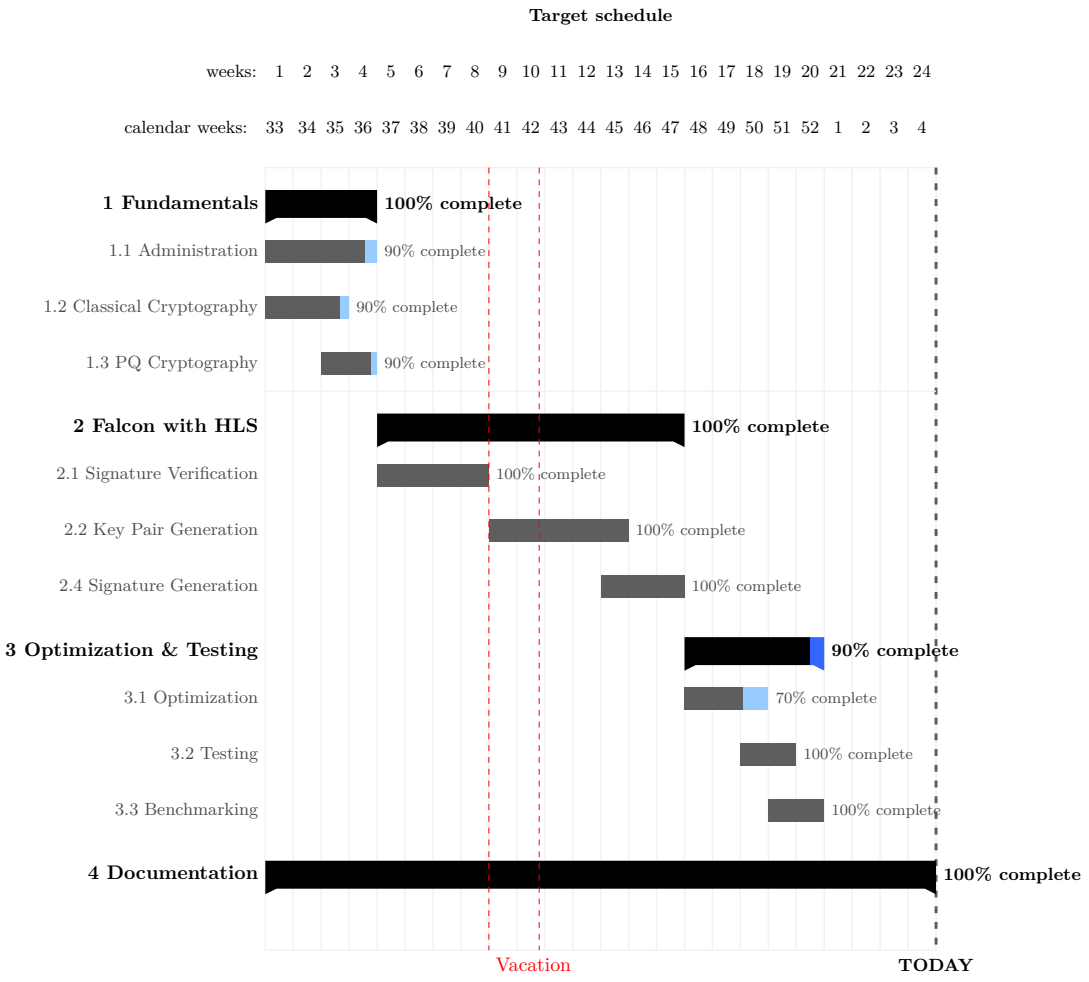
Project start	Monday, August 15, 2022
Fall break	10/10/2022 - 10/20/2022
Submission of report:	Friday, January 27, 2023
Final presentation/demo:	date by arrangement

A.6 Administrative Information

Supervisor:	Prof. Dr. Paul Zbinden
Advisor:	Prof. Dr. Tao Wei
Weekly meetings:	Thursday 4-5 pm EDT
Workplace/lab:	Tao's lab, URI

Appendix B

Schedule



Bibliography

[1] "NIST Post-Quantum Cryptography." https://csrc.nist.gov/Projects/post-quantum-cryptography, August 2022.

[2] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 2001.

[3] J. Jean, "Ti*k*Z for Cryptographers." https://www.iacr.org/authors/tikz/, 2016.

[4] Prof. Dr. Markus Hufschmid, "Lecture notes in Cryptography and Coding Theory," Autumn 2020.

[5] "Advanced Encryption Standard." https://en.wikipedia.org/wiki/Advanced_Encryption_Standard, January 2023.

[6] Tanja Lange, "Lecture notes in Selected Areas in Cryptology - Spring 2021 ." https://hyperelliptic.org/tanja/teaching/pqcrypto21/, Spring 2021.

[7] L. Babai, "On lovász' lattice reduction and the nearest lattice point problem - combinatorica."

[8] O. Goldreich, S. Goldwasser, and S. Halevi, "Public-key cryptosystems from lattice reduction problems," in *Advances in Cryptology — CRYPTO '97* (B. S. Kaliski, ed.), (Berlin, Heidelberg), pp. 112–131, Springer Berlin Heidelberg, 1997.

[9] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A ring-based public key cryptosystem," in *Algorithmic Number Theory* (J. P. Buhler, ed.), (Berlin, Heidelberg), pp. 267–288, Springer Berlin Heidelberg, 1998.

[10] "Falcon - Fast-Fourier Lattice-based Compact Signatures over NTRU ." https://falcon-sign.info/, January 2023.

[11] C. Gentry, C. Peikert, and V. Vaikuntanathan, "Trapdoors for hard lattices and new cryptographic constructions." Cryptology ePrint Archive, Paper 2007/432, 2007. https://eprint.iacr.org/2007/432.

[12] J. Hoffstein, N. Howgrave-Graham, J. Pipher, J. H. Silverman, and W. Whyte, "NTRUSign: Digital Signatures Using the NTRU Lattice," in *Topics in Cryptology — CT-RSA 2003* (M. Joye, ed.), (Berlin, Heidelberg), pp. 122–140, Springer Berlin Heidelberg, 2003.

[13] D. Stehlé and R. Steinfeld, "Making ntru as secure as worst-case problems over ideal lattices," in *Advances in Cryptology – EUROCRYPT 2011* (K. G. Paterson, ed.), (Berlin, Heidelberg), pp. 27–47, Springer Berlin Heidelberg, 2011.

[14] L. Ducas, V. Lyubashevsky, and T. Prest, "Efficient identity-based encryption over ntru lattices," in *Advances in Cryptology – ASIACRYPT 2014* (P. Sarkar and T. Iwata, eds.), (Berlin, Heidelberg), pp. 22–41, Springer Berlin Heidelberg, 2014.

[15] L. Ducas and T. Prest, "Fast fourier orthogonalization," in *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '16, (New York, NY, USA), p. 191–198, Association for Computing Machinery, 2016.

[16] P. Q. Nguyen and O. Regev, "Learning a parallelepiped: Cryptanalysis of ggh and ntru signatures," in *Advances in Cryptology - EUROCRYPT 2006* (S. Vaudenay, ed.), (Berlin, Heidelberg), pp. 271–288, Springer Berlin Heidelberg, 2006.

[17] L. Ducas and P. Q. Nguyen, "Learning a zonotope and more: Cryptanalysis of ntrusign countermeasures," in *Advances in Cryptology – ASIACRYPT 2012* (X. Wang and K. Sako, eds.), (Berlin, Heidelberg), pp. 433–450, Springer Berlin Heidelberg, 2012.

[18] "CRYSTALS-KYBER." https://pq-crystals.org/kyber/index.shtml, January 2023.

[19] Y. Xing and S. Li, "A compact hardware implementation of cca-secure key exchange mechanism crystals-kyber on fpga," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, p. 328–356, Feb. 2021.

[20] Z. Chen, Y. Ma, T. Chen, J. Lin, and J. Jing, "High-performance area-efficient polynomial ring processor for crystals-kyber on fpgas," *Integration*, vol. 78, pp. 25–35, 2021.

[21] "CRYSTALS-DILITHIUM." https://pq-crystals.org/dilithium/index.shtml, January 2023.

[22] S. Ricci, L. Malina, P. Jedlicka, D. Smékal, J. Hajny, P. Cibik, P. Dzurenda, and P. Dobias, "Implementing crystals-dilithium signature scheme on fpgas," in *The 16th International Conference on Availability, Reliability and Security*, ARES 2021, (New York, NY, USA), Association for Computing Machinery, 2021.

[23] L. Beckwith, D. T. Nguyen, and K. Gaj, "High-performance hardware implementation of crystals-dilithium," in *2021 International Conference on Field-Programmable Technology (ICFPT)*, pp. 1–10, 2021.

[24] D. Soni, K. Basu, M. Nabeel, N. Aaraj, M. Manzano, and R. Karri, *CRYSTALS-Dilithium*, pp. 13–30. Cham: Springer International Publishing, 2021.

[25] L. Beckwith, D. T. Nguyen, and K. Gaj, "High-performance hardware implementation of lattice-based digital signatures." Cryptology ePrint Archive, Paper 2022/217, 2022. https://eprint.iacr.org/2022/217.

[26] D. Soni, K. Basu, M. Nabeel, N. Aaraj, M. Manzano, and R. Karri, *FALCON*, pp. 31–41. Cham: Springer International Publishing, 2021.

[27] "SPHINCS+ Stateless hash-based signatures." https://sphincs.org/, January 2023.

[28] D. Amiet, L. Leuenberger, A. Curiger, and P. Zbinden, "Fpga-based sphincs+ implementations: Mind the glitch," in *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pp. 229–237, 2020.

[29] Q. Berthet, A. Upegui, L. Gantel, A. Duc, and G. Traverso, "An area-efficient sphincs+ post-quantum signature coprocessor," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 180–187, 2021.

[30] D. Soni, K. Basu, M. Nabeel, N. Aaraj, M. Manzano, and R. Karri, *SPHINCS+*, pp. 141–162. Cham: Springer International Publishing, 2021.

[31] "PYNQ: Python productivity for Adaptive Computing platforms." https://pynq.readthedocs.io/en/latest/index.html, January 2023.

[32] "Vitis High-Level Synthesis User Guide." https://docs.xilinx.com/r/en-US/ug1399-vitis-hls, January 2023.

[33] "Floating-Point Operator v7.1 LogiCORE IP Product Guide." https://docs.xilinx.com/v/u/en-US/pg060-floating-point, January 2023.

[34] "Bug Report Xilinx Forum." https://support.xilinx.com/s/question/0D54U00006FWwWpSAL/hls-bug-wrong-type-cast-when-inlining?language=en_US, January 2023.

[35] Senn, Andreas, and Wendler, Jan, "FALCON auf FPGA." Bachelor Thesis 2022.

[36] "4 Types of Tree Traversal Algorithms." https://towardsdatascience.com/4-types-of-tree-traversal-algorithms-d56328450846, January 2023.

[37] R. Kastner, J. Matai, and S. Neuendorffer, "Parallel Programming for FPGAs," *ArXiv e-prints*, May 2018.

[38] R. Elkhatib, R. Azarderakhsh, and M. Mozaffari-Kermani, "Highly optimized montgomery multiplier for sike primes on fpga," in *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*, pp. 64–71, 2020.

[39] A. C. Mert, E. Karabulut, E. Öztürk, E. Savaş, and A. Aysu, "An extensive study of flexible design methods for the number theoretic transform," *IEEE Transactions on Computers*, vol. 71, no. 11, pp. 2829–2843, 2022.

List of Figures

List of Tables

Listings