Comparing and Optimizing an Identity-Based Post-Quantum Scheme

WILLIAM BRANTE & MAX MONTEBOVI
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Comparing and Optimizing an Identity-Based Post-Quantum Scheme

Department of Electrical and Information Technology, Lund University

Supervisors: Guo Qian (EIT), Nilson Anders (Bosch)
Examiner: Johansson Thomas

By
Brante William
Montebovi Max

2024

Security is a process, not a product.
– Bruce Schneier

Abstract

The objective of this thesis is to evaluate the performance of the IBE scheme over NTRU lattices presented by Ducas, Lyubashevsky, and Prest (2014) on different platforms. The motivation behind this study was to explore a post-quantum IBE scheme and determine its viability on different platforms and specifically on an ARM64 CPU. Our goal was to parallelize the bottlenecks identified during our initial tests using SIMD instructions and GPU programming. Additionally, we aimed to demonstrate that IBE schemes may be better suited for certain scenarios than traditional PKI. We used CUDA to execute parts of the code on GPUs in one case, SIMD instructions to increase computing performance in the other cases and comparing this to the use of the open source library NFLlib. A profiler was used to identify hotspots during the initial executions and optimization continued from there. We found that the user key generation with CUDA was slower than the standard implementation, but encryption and decryption both gained a boost, similar to the NFLlib results. Tests with AVX2 SIMD instructions showed similar performance to the standard implementation for user key generation, while the encryption and decryption achieved performance increases. The Neon SIMD instructions used on the ARM64 platform made user key generation, encryption and decryption faster than the standard implementation.

Contents

Acknowledgement **x**

1 Introduction **1**
- 1.1 Literature review . . . 1
- 1.2 Motivation . . . 1
- 1.3 Goal . . . 2
- 1.4 Contributions . . . 2
- 1.5 Overview . . . 2

2 Background **4**
- 2.1 Key exchange methods . . . 4
- 2.2 Public key encryption . . . 5
  - 2.2.1 RSA . . . 7
- 2.3 Identity-based encryption . . . 8
  - 2.3.1 Key revocation . . . 10
- 2.4 Post-quantum cryptology & quantum computers . . . 10
  - 2.4.1 How quantum computers breaks cryptography . . . 11
  - 2.4.2 Hybrid schemes . . . 13
- 2.5 Lattices . . . 13
- 2.6 NTRU . . . 14
  - 2.6.1 Key creation . . . 15
  - 2.6.2 Encryption and decryption . . . 16
  - 2.6.3 Signing . . . 16
  - 2.6.4 NTRU lattice . . . 17
- 2.7 Learning with errors . . . 18
- 2.8 IBE NTRU . . . 18
  - 2.8.1 IBE key generation . . . 19
  - 2.8.2 Encryption and decryption . . . 20
  - 2.8.3 IBE threat model . . . 20
- 2.9 Parallelization technologies . . . 23
  - 2.9.1 CUDA . . . 23
  - 2.9.2 SIMD . . . 23

3 Methodology **26**

3.1 Client server setup for ID-based NTRU in C++ 26
3.2 Hardware 28
3.3 Software 28

4 Implementation and Pseudocode **30**
4.1 CUDA implementation 30
4.2 SIMD implementaions 34
4.3 NFL implementation 41

5 Results **44**
5.1 Master key creation 44
5.2 User key generation, encryption and decryption 45
5.2.1 x86 45
5.2.2 ARM 49
5.3 Distributions 52
5.3.1 x86 52
5.3.2 ARM64 56

6 Discussion **59**
6.1 Measurements 59
6.2 Comparison with PKI setup 61
6.2.1 Architectural differences 61
6.2.2 Identification 63
6.2.3 Key generation and management 64
6.2.4 Differences in maintaining CIA 65
6.3 IBE in practice 65
6.3.1 Internal vehicle communication 65
6.3.2 Email system 66
6.3.3 Updating and distributing keys 67

7 Summary and outlook **68**
7.1 Results 68
7.2 Future work 69
7.3 Outlook 69

Bibliography **74**

Appendix A **75**

List of Figures

2.1 Digital signature scheme. 6
2.2 Model of a public key cryptosystem where Alice sends an encrypted message to Bob. 7
2.3 Model of an identity-based signature scheme. Alice signs the message with her ID combined with a signature key so Bob can verify the sender of the message. 9
2.4 Model of an identity-based cryptosystem showing the steps needed for Alice to send an encrypted message to Bob. A signature is also necessary for Bob to be able to confirm the source of the encrypted message. 10
2.5 A 2-dimensional lattice (dots) with one good vector basis (black) and one bad vector basis (grey). 14
2.6 Threat model for an IBE system. It shows two users requesting to join but only Alice has the right. The model also shows that both server and users need a secure storage to store the keys in. 21
2.7 ALU using SIMD. 25

3.1 UML diagram of the Client and Server classes. 27

5.1 Median master key generation time plotted for all platforms. 45
5.2 Median user key generation time on the x86-1 and x86-2 platforms for the standard, CUDA and SIMD (AVX2) implementations. 48
5.3 Median encryption time on the x86-1 and x86-2 platforms for the standard, CUDA, NFLlib and SIMD (AVX2) implementations. 49
5.4 Median decryption time on the x86-1 and x86-2 platforms for the standard, CUDA, NFLlib and SIMD (AVX2) implementations. 49
5.5 Median user key generation time using the standard and SIMD (Neon) implementations on the ARM64 platform. 51
5.6 Median encryption time using the standard and SIMD (Neon) implementations on the ARM64 platform. 51
5.7 Median decryption time using the standard and SIMD (Neon) implementations on the ARM64 platform. 52

5.8 Master secret key and user key generations. 53
5.9 User key generations. 54
5.10 Encryptions on platform x86-2 using Standard and AVX2 methods on two different charts. 54
5.11 Encryption times on platform x86-2 using GPU and NFL methods on two different charts. 55
5.12 Decryption times on platform x86-2 using Standard and AVX2 methods on two different charts. 55
5.13 Decryption times on platform x86-2 using GPU and NFL methods on two different charts. 56
5.14 Distribution of 10 Master Key generations on platform ARM64 for N=1024. 57
5.15 User key generations. 57
5.16 Distribution on 1000 encryptions on platform ARM64 using Standard and Neon methods on two different charts. 58
5.17 Distribution on 1000 decryptions on platform ARM64 using Standard and Neon methods on two different charts. 58

6.1 Certificate request and management for a user in PKI. The user sends a certificate signing request to the certificate authority and gets a certificate in return. 62
6.2 Identity-based encryption scheme setup. The model shows a user requesting the Key Generation Center / Trusted Authority to join. The figure also shows what both entities have and the steps being taken during a request to join. 63

7.1 Gperf chart of the original implementation, key generation part. 75
7.2 Gperf chart of the original implementation, encryption and decrytion part. 76

List of Tables

3.1 The different platforms that were used to perform measurements on. . . . 28

5.1 Times in s for Master Key generation on the different platforms. 45
5.2 Times in ms for different implementations on x86-1. 46
5.3 Speedup relative to the standard run with the same N (standard time/new time) on x86-1. 47
5.4 Times in ms for different implementations on x86-2. 47
5.5 Speedup relative to the standard run with the same N (standard time/new time) on x86-2. 48
5.6 Times in ms for different implementations run on ARM64. 50
5.7 Speedup relative to the standard run with the same N (standard time/new time) on ARM64. 50

List of acronyms

ALU	Arithmetic Logic Unit
AVX	Advanced Vector Extensions
CA	Certificate Authority
CIA	Confidentiality, Integrity and Availability
CPU	Central Processing Unit
CRT	Chinese Remainder Theorem
CSR	Certificate Signing Request
DH	Diffie-Hellman
FFT	Fast Fourier Transform
GPU	Graphics Processing Unit
IBE	Identity-Based Encryption
ID	Identity
IoT	Internet of Things
LWE	Learning With Errors
MITM	Man-In-The-Middle
MPK	Master Public Key
NIST	National Institute of Standards and Technology
MSK	Master Secret Key
NFLlib	NTT-based Fast Lattice library
NP	Non-deterministic Polynomal time
NTRU	N-th degree Truncated polynomial Ring Units
NTT	Number Theoretic Transform
PKI	Public Key Infrastructure
RA	Registration Authority
SIMD	Single Instruction, Multiple Data
SNDL	Store Now, Decrypt Later
SVP	Shortest Vector Problem
TA	Trusted Authority
TPM	Trusted Platform Module

Popular Science Summary

With the accelerated quantum computer development, we have come to a point where current public key infrastructure is under threat and threatens the safety of individuals, corporations and institutions. Thankfully other methods classified as post-quantum secure algorithms are being developed and tested. One such is an identity-based encryption scheme relying on the Learning With Errors (LWE) and NTRUSign algorithm which we implemented and tested on different devices.

The security of the digital world we use today relies on public key infrastructure (PKI), cryptography which uses one public key, one private key and a proof of ownership of the public key. Most widely adopted versions use RSA or a version of the Diffie-Hellman key exchange. The problem is that both RSA and Diffie-Hellman can be broken using a decent quantum computer. For this reason other algorithms need to be considered, two of which are NTRU and LWE. In identity-based encryption schemes you replace the public key with a known ID, this removes the need for a ”proof of ownership” which could decrease the complexity of the overall system.

We took an existing implementation in C++ of a specific identity-based encryption scheme, modified and optimized it using CUDA, SIMD and the C++ library NFLlib. We implemented our code on different devices and found that even weaker hardware could deliver acceptable performance. It was possible to speed up encryption/decryption using all optimization techniques however using SIMD proved the best with speedups as high as 13 times faster. Generating private keys for users only improved when using SIMD on an ARM64 processor. We also compared the scheme to standard PKI and found that it could be beneficial to use an identity-based scheme in some instances. Some processing time and storage needed for authentication in ordinary PKI can be avoided using identity-based encryption, however distribution of keys remains a problem. Revoking an ID is also a problem and it would therefore be recommended to use time constrained IDs over static IDs.

Acknowledgments

We would like to thank our supervisor, Anders Nilson at Bosch R&D Lund who provided us with guidance and support throughout this entire master thesis. We would also like to thank Qian Guo, our supervisor at LTH who also gave us valuable feedback when writing this report.

Chapter 1

Introduction

1.1 Literature review

The idea of an identity-based cryptosystem was first introduced by Adi Shamir in 1985 and has come a long way since. A contribution to the field was the work by Ducas and Lyubashevsky and Prest in 2014, who presented an efficient IBE scheme over NTRU lattices. Their scheme showed that key generation, encryption, and decryption could be done with reasonable parameters on a laptop in an acceptable time. The work and results by Ducas et al. were promising but further exploration was needed to test the scheme on more platforms and its potential for further increased speeds. Compared to other IBE schemes this appeared more promising and overall complete while also being argumentatively post-quantum secure.

1.2 Motivation

The evolution of quantum computers has accelerated in recent years and doesn't appear to be slowing down. There exists quantum algorithms that could break the cryptography currently used today and it is important to find replacements for this reason. The most commonly used asymmetric cryptographic schemes also rely on public key infrastructure (PKI) which uses certificates to validate the authenticity of public keys. PKI therefore requires some managing overhead that can be avoided by using identity-based encryption.

1.3 Goal

With lots of research pouring into quantum computers we wanted to look at some promising post-quantum secure schemes as well as combining these with identity-based encryption. The ID-based NTRU scheme we chose came with the post-quantum security of NTRU-lattices and practicality of identity-based encryption. Another interesting part is to compare identity-based encryption with the traditional, widely used public key infrastructure schemes to find areas were one is more suitable than the other.

We also wanted to test this scheme in different scenarios to see how this type of scheme holds up for different practical use cases. The goal was to set up a client-server system where the server acted as a trusted authority, responsible for key generation and the clients were entities with an identity and had the ability to both encrypt and decrypt messages sent amongst each other. We would also measure the time for different procedures in the scheme and implement it in various ways to try and make it faster.

To test and time the identity-based scheme, we used different techniques to try to speed up encryption, decryption and key generation. These techniques were CUDA programming for GPU acceleration and SIMD instructions to fully utilise registers and the ALU. These techniques were compared to both the standard implementation and an implementation using the NFLlib library, mentioned by Ducas et al. We also found it important to test the identity-based encryption scheme on an ARM64 and use ARM Neon SIMD architecture in the implementation to see the adaptability of the IBE scheme.

1.4 Contributions

We did multiple implementation variants of Ducas et al.'s identity-based encryption scheme and tested them on different hardware. We tested GPU and SIMD optimization and most importantly ran tests on a comparatively low powered ARM64 processor with Neon SIMD optimization. Our results showed that it is possible to run an IBE scheme on an ARM64 and also the fact that a speedup with Neon SIMD instructions is possible and worth implementing fully on lattice based encryption.

1.5 Overview

Firstly, the background in this thesis covers the fundamentals of cryptography, subjects like key exchange, PKI, and the main topic: identity-based encryption. The background

also covers post-quantum cryptography, quantum computers and how hybrid schemes are an option for the near future. The identity-based scheme used for our tests is described as well. How the encryption, decryption and key generation works are also covered besides the underlying math of the scheme. Lastly the methods for parallelization are described with the GPU programming language CUDA first and then what SIMD is and how it works.

The methodology explains how the client/server setup works for the ID-based NTRU scheme and how it was implemented. Gperf was used to find bottlenecks in the code and as a indicator to where optimisation was needed and those charts can be found in Appendix A.

Three different hardware platforms were used for the testing and those were ARM64, x86-1 and x86-2. The ARM64 platform was performance wise the weakest of the three, x86-1 was a desktop computer and x86-2 was a virtual machine running on Google Cloud Computing. The different implementations are the standard (C++), CUDA, Intel Intrinsics SIMD and Neon SIMD, as well as a fifth implementation using a C++ library optimized for lattice based operations called NFLlib.

The pseudocode and explanation of the implementations are covered in chapter four. From these different implementations and tests on different hardware, times for encryption, decryption and different key generations was obtained and are shown in various tables in the result section. To make it easier to compare, the speedup was also calculated as well as charts showing the distributions of our results. Lastly this thesis discusses the obtained results, advantages with IBE, differences from PKI and scenarios where IBE could be worth considering.

Chapter 2

Background

2.1 Key exchange methods

To be able to use symmetric encryption the communicating parties must share a secret key. One option the two parties have is to exchange the key in person, to make sure both have a copy of the key and no one else. The key exchange could also be done out-of-band via a courier, a letter or through some other trusted method.

Now only of historical interest, the American computer scientist and mathematician Ralph Merkle had an idea for a key exchange method that could be done over an insecure channel. His idea was that for two parties; Alice and Bob, to agree on a key, Bob would send multiple puzzles to Alice were each puzzle could be solved after some computing and the solution would give Alice a unique secret key. Each puzzle has an unique identifier so Alice can send the identifier in clear text back to Bob along with the message encrypted with the key. Bob can look up the key for that identifier and decrypt the message. If a third party; Eve, eavesdrops the channel she would not get hold of the key and her best chance to get the key is to solve all the puzzles Bob sends to Alice. With many puzzles this will be too computationally expensive for Eve to successfully decrypt the message [44]. This key would then be used for both encrypting and decrypting the messages, i.e. a symmetric key. The key exchange he came up with made it possible for two communicating parties to agree on a key without too much effort and an eavesdropping third party would have to commit a lot more effort in order to get the key [14, 25].

In 1976 Whitfield Diffie and Martin Hellman suggested a new public key distribution scheme that is based on the the mathematically hard discrete logarithm problem. For

Alice and Bob to perform the DH (Diffie-Hellman) key exchange they use two public parameters chosen by a trusted third party. These parameters are an integer g and a prime p. Alice picks a secret integer a and sends the value $A \equiv g^a \pmod p$ and Bob sends the value $B \equiv g^b \pmod p$ where b is a secret integer chosen by Bob. They then use these values to compute the shared secret by raising the published value to their respective secret integer modulus p and get the following:

$$A^b \equiv B^a \equiv g^{ab} \pmod p$$

A third party listening to the channel will only get the values A and B and even with the public parameters p and g, the shared secret $g^{ab} \pmod p$ cannot be computed without the secret values a or b [37].

If the third party listening to the channel is active then the classical Diffie-Hellman protocol can be exposed to a Man In The Middle (MITM) attack. If Eve wants to perform a MITM attack on Alice and Bob's secret sharing, Eve sits in the middle and acts like she is the intended target. Alice and Bob are unaware of Eve's presence and think they communicate directly with each other. When Alice asks for Bob's key, Eve will relay this query to Bob. When Bob sends his key Eve will save that and give Alice another key. Eve can then decrypt and read all Alice's messages and then encrypt them and send them to Bob without Alice and Bob knowing. Preventing a man in the middle attack on a Diffie-Hellman key exchange protocol can be done by using authentication, which we will cover later, or another way is to not send the keys in plain but rather somewhat scrambled [19].

2.2 Public key encryption

In a public key cryptosystem there exists two non-symmetric keys, one public and one private key. This system can be used both for encryption and one-way authentication. If Alice wants to send Bob a message she can encrypt the message with Bob's public key and then Bob can decrypt it with his private key. If Alice would like to sign this message so Bob can verify it was she who sent it, Alice can put her signature and encrypt it with her private key and Bob can decrypt the signature with Alice's public key. Thus verifying that Alice is the sender. Another useful part of public key encryption is that the number of keys needed for encrypted communication is drastically smaller than that of symmetric key encryption. Take the example of N participants: if each participant would want to be able to communicate with each other using symmetric keys,

there would need to be one key for each communication channel, resulting in $\frac{(N-1)N}{2}$ keys. But using asymmetric keys, only $2N$ distinct keys are required; one public and one private key for each participant [14].

A public key infrastructure includes a CA (Certificate Authority) and an RA (Registration Authority) to create, handle and distribute digital certificates. A digital certificate, also known as an identity certificate, is used to prove that a public key is valid and it contains the public key, identity of its owner and a digital signature of someone that has verified the content. The tasks of the RA and CA can often be done by the same entity but specifically the RA makes sure that the user that receives the certificate is legitimate while the CA generates and verifies certificates. A PKI usually also have a certificate storage to store certificates and that storage could also contain a revocation list of certificates that are no longer valid [27].

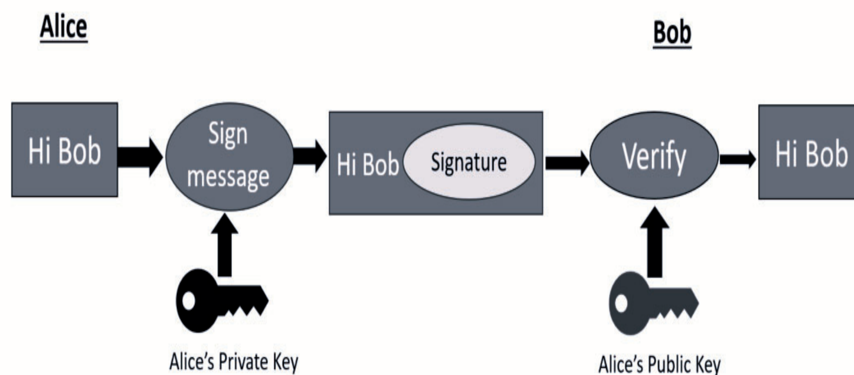


Figure 2.1: Digital signature scheme.

If someone wants to send an encrypted message and also prove that the message actually is from them some form of information must be added to the message. Digital signatures solve this problem and they act as the equivalence of signing a non-digital paper with ink. A digital signature scheme needs a couple of parameters added to the PKI. Firstly, it needs another pair of private and public keys. Then it needs a signing algorithm and an algorithm for verification. The scheme works by using the private key together with the signing algorithm and the message, or a hash of the message to create

a signature. Since only the sender has access to the private signature key, the signature will ensure the integrity of the message. When the receiver gets the signed message, the signature must be verified. This is done with the verification algorithm that takes the message, the signature and the public verification key. These steps are shown in figure 2.1 where Alice sends a message to Bob. If the signature matches with the sender's private signature key and the message, the verification will pass. If it does not match, the receiver cannot know who sent it [37]. In figure 2.1 it is not sufficient for Alice to only encrypt "Alice" or some other static message with her private key and use that as the signature because an eavesdropper can take that signature, attach it to their messages, and pretend the messages come from Alice. Instead her signature should be unique in each signed message which can be done by using information sent in the message.

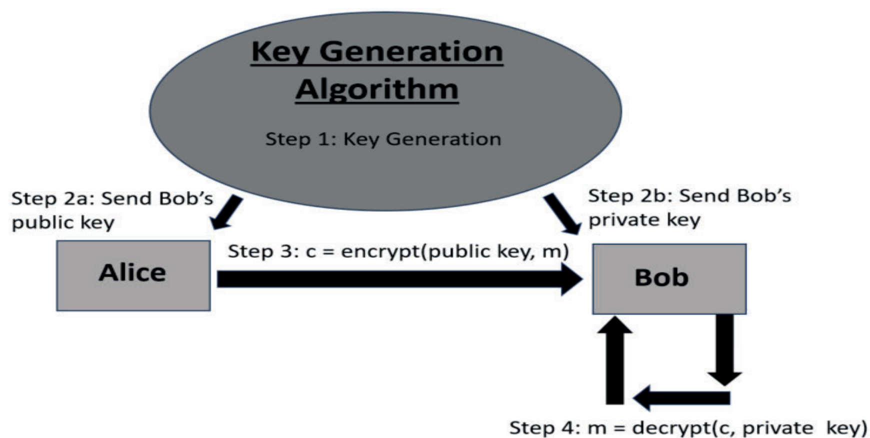


Figure 2.2: Model of a public key cryptosystem where Alice sends an encrypted message to Bob.

2.2.1 RSA

In 1978 Rivest, Shamir and Adleman provided an implementation of the public key cryptosystem Diffie and Hellman proposed in 1976. Their implementation is called RSA after its creators, a so called "one-way" function because it is easy to compute in one direction and difficult in the opposite direction [31].

If Alice would like to send Bob a message using RSA she must use Bob's public key for encryption and Bob uses his private key for decryption. It begins with Bob choosing two secret primes p and q, the exponent e and its inverse $d \pmod{(p-1)(q-1)}$. Bob's public values become $N = pq$ and e. Alice can now take her message m and with Bob's public keys compute the ciphertext $c \equiv m^e \pmod{N}$ and send c to Bob. To decipher the message Bob computes:

$$c^d \equiv m^{ed} \equiv m^{1+k(p-1)(q-1)} \equiv m^1 m^{(p-1)(q-1)k} \pmod{N}$$

And utilizing a property of Euler's phi function we finally see that this equals the message m [31, 37]:

$$m^1 m^{(p-1)(q-1)k} \equiv m^1 1^k \equiv m \pmod{N}$$

2.3 Identity-based encryption

In 1984 Shamir introduced a new cryptographic scheme that enables two persons to both communicate securely and to verify the other person's signature without storing a list of public keys or exchanging keys. The scheme has a key generation center that gives every new person in the network information on how to sign, verify, encrypt and decrypt messages irrespective of the identity of the other party. This scheme is similar to a public key cryptosystem but the main difference is that instead of generating a pair of public/secret keys the user can have an email address or a phone number as the public key, as long as it is unique to that user. This means that a certificate manager would no longer be needed since the public key is directly tied to some form of identity of the receiver. To send a message from Alice to Bob, Alice can encrypt it by using Bob's public key, based on for instance his email address, that Alice already knows. Bob can then decrypt the message with the private key he got when he joined the network [35,38].

The first thing that must happen is that a key generation center must generate a master secret key and a master public key. The key generation center shown in fig 2.4 not only sends its users their secret keys but also a master public key needed for encryption and this key is the same key for all users. Every user's public key is a combination of the master public key and the user's ID and therefore the master public key must be generated initially. The master secret key must also be generated in the beginning because that key is used to generate the secret decryption keys for the users. The next step comes when users want to join the network, then the generator must create personal secret keys, using the master secret key, for each user [11].

Each user in the network has the master public key, their own secret key and they also know the identities of the others in the network. To send an encrypted message to another user the only things needed are the receiver's ID, the master public key and the message, this is shown as point four in figure 2.4. Compared with public key infrastructure were the sender must either ask or search for the public key of the receiver, here the sender uses the receiver's ID combined with the master public key. Points five and six in figure 2.4 illustrate the decryption of a ciphertext. The decryption uses the user's personal secret key to generate the original message [35].

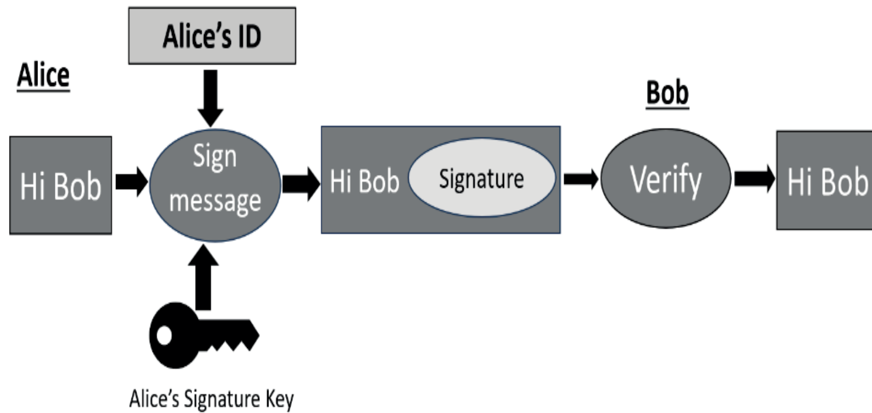


Figure 2.3: Model of an identity-based signature scheme. Alice signs the message with her ID combined with a signature key so Bob can verify the sender of the message.

To sign a message in an IBE scheme another key must be created by the key generation center, a signature creation key. This key is often bound to an identity and it is used for signatures [27]. Like signatures in a PKI, there must also exist some form of algorithm for verifying the signature and also an algorithm for signing. In figure 2.3 Alice sends a signed message to Bob. She signs the message with her signature creation key based on her ID. Bob receives the message and verifies it with an algorithm that takes Alice's ID and the message as input and checks if the signature is valid [35].

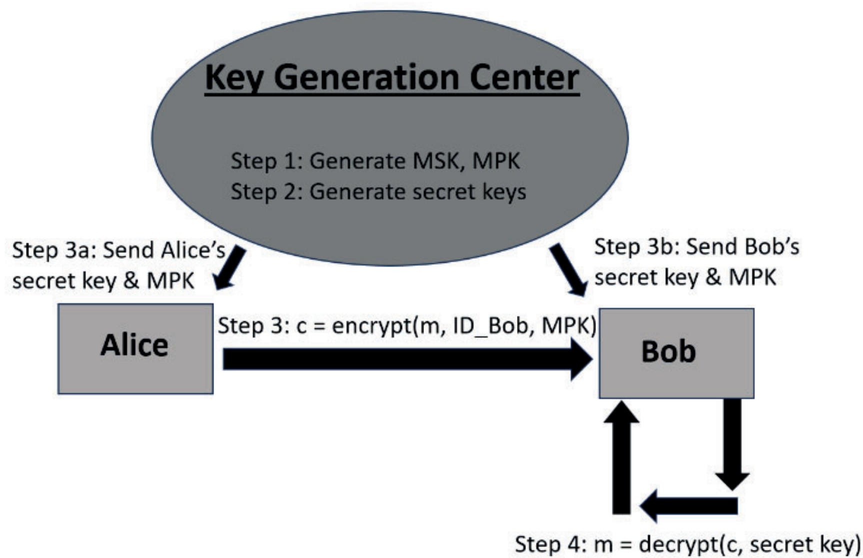


Figure 2.4: Model of an identity-based cryptosystem showing the steps needed for Alice to send an encrypted message to Bob. A signature is also necessary for Bob to be able to confirm the source of the encrypted message.

2.3.1 Key revocation

The key revocation is a bit different because Bob should not have to create a new email address every time his key expires. This can be solved by letting Alice use Bob's email address in combination with the current day, month or year to force Bob to obtain a new private key in order to have regular private key updates. The only thing Alice needs to keep track of with this system is the date. An interesting thing with this scheme is that Alice can send Bob messages into the future because Bob can only decrypt them once he gets the proper private key for that date. When Bob should no longer be able to decrypt his messages the company/network stops issuing new keys for Bob [8].

2.4 Post-quantum cryptology & quantum computers

RSA and the Diffie Hellman key exchange method are based on the mathematical problems of large integer factorization and discrete logarithms. The security in these crypto-

graphic methods depend on the fact that these mathematical problems are hard to solve. There exist algorithms that simplify the problems, such as Quadratic Sieve for integer factorization, but even these algorithms depend on a lot of trial and error in order to reach an answer, so much so that given a large enough solution space it is computationally infeasible with classical computers. There exists algorithms for quantum computers that can take huge shortcuts and make these mathematical problems much easier to solve. Therefore there is a need for cryptosystems that are secure not only today but in the future as well and that means that they need to be quantum secure [36, 37, 41].

2.4.1 How quantum computers breaks cryptography

To understand how quantum computers can break current cryptography we need to start off with taking a look into the difference between bits used in classical computers and bits used in quantum computers called qubits. An ordinary bit takes one of two states; inactive and active or more commonly recognised as 0 and 1. This means that N bits can represent one of 2^N states (00, 01, 10 or 11 for $N = 2$) and the same goes for calculations that use bits, one input state and one output state at a time. This is in line with our natural way of thinking of calculations, however qubits are different. A qubit can also be used to represent two states 0 and 1 but unlike ordinary bits it also exists in a quantum state where we view it as existing in the two states at the same time. An advantage of quantum computers is therefore that they do not run calculations for one input and one output at a time but calculations for all inputs and all outputs at the same time by fixing the input qubits and output qubits in a specific quantum entanglement. A quantum computer with N output qubits can thereby calculate all possible 2^N solutions simultaneously. This quantum solution state is however not very practical since the information about all except one random state disappears if we try to observe it directly [34]. It is however possible to extract the necessary information needed to solve the mathematical problems of RSA and the Diffie Hellman key exchange [36].

Now we are ready to break cryptography and we begin with breaking RSA using a classical computer and we start off with a factoring algorithm similar to the Quadratic Sieve algorithm.

Given an integer $N = pq$ where p and q are primes our initial goal is to find a pair (g, r) such that the following is true for $g, r \in \mathbb{Z}_N$:

$$g^r \equiv 1 \pmod{N}$$

We do this by fixing g to a random integer > 1 and iteratively increase r. Once we find a pair (g, r) that fulfills the condition we say that the period given g is r since:

$$g^{i+kr} \equiv z_i \pmod{N} \tag{2.1}$$

for all k and some fixed $g, r, i, z_i \in \mathbb{Z}_N$.

If r is odd we need to pick a new g and find a new pair (g, r) where r is even. But when r is even we have:

$$g^r = 1 + tN \Leftrightarrow$$

$$g^r - 1 = tN \Leftrightarrow$$

$$(g^{\frac{r}{2}} + 1)(g^{\frac{r}{2}} - 1) = t_1 t_2 t_3 ... pq$$

where t is an integer with prime factors $t_1, t_2, t_3....$ From this we can see that p and q must be prime factors of either $p' = (g^{\frac{r}{2}} + 1)$, $q' = (g^{\frac{r}{2}} - 1)$ or both and to check we compute $gcd(p', N)$ which equals either 1, N or p. If we get p we can easily calculate $\frac{N}{p} = q$, otherwise we try the same with q' and if that fails we start over with a new pair of (g, r), until finally we end up with the secret primes p and q [36].

This algorithm is the basis of what is known as Shor's algorithm, a promising algorithm that can run on quantum computers and break both RSA and the Diffie Hellman key exchange. The next part of the algorithm is the quantum algorithm and is more complicated than the first part but the general idea is the following:

Given $N = pq$, use n qubits to represent the input quantum state X, then pick an integer g and fix the output quantum state to be the remainder Y such that $g^X = tN + Y$ for some integer t. Now X and Y exist in quantum entanglement, X will represent all possible exponents x and Y will represent all associated remainders y, we also see that the pair (x, y) is equivalent to $(i + kr, z_i)$ in equation 2.1. From this we can conclude that given g, the input state X is periodic with period r and so is Y where the number of possible states in Y is equivalent to r. With the help of this observation, more math and the Quantum Fourier Transform we can retrieve $\frac{1}{r}$, directly get p', q' and continue from there to get p and q [36].

Now the only reason this algorithm has not broken our current cryptosystems yet is because of inaccuracies in today's quantum computers. X and Y are hard to represent perfectly and they introduces a lot of errors to the result. To get rid of these errors either higher quality qubits or a larger amount of qubits is required and the gap between the

available and required qubits shrink every year. It might only be a matter of time until quantum computers are powerful enough to break standard cryptography.

2.4.2 Hybrid schemes

With the threat of quantum computers breaking the cryptography used today the idea of combining today's cryptography with post-quantum cryptography has emerged. This idea upholds regular security while also mitigating the risk of quantum attacks [7]. Since classic cryptography has been around for a long time there exists secure schemes but with quantum cryptography it is harder to say because those schemes have not been around as long and have not been tested and analyzed as thoroughly. This is a reason why the Federal Office for Information Security in Germany currently recommends not fully switching to new quantum secure schemes but using post-quantum cryptography in combination with established algorithms instead [2, 30].

Another reason hybrid schemes could be worth implementing is because of the SNDL (Store Now Decrypt Later) attacks. The attack idea is to capture valuable information that is encrypted with today's cryptographic methods and decrypt them once powerful enough quantum computers are available. SNDL is an attack that can be initiated immediately, requiring only storage to initiate. For most data this is not a major problem but some data could be sensitive even if it is decrypted far into the future, for example state secrets and medical records. This attack is therefore another reason to transition into post-quantum cryptographic schemes and not hold off until the last minute [17].

2.5 Lattices

A lattice, in general, is a subgroup of the Euclidean space R^m generated by a matrix $\mathbf{A} = (a_1, a_2, ..., a_n) \in R^{m \times n}$. The dimension of the lattice is defined by n and any lattice with a dimension ≥ 2 has infinitely many bases. Figure 2.5 shows a two dimensional lattice grid with two different bases drawn as arrows. Different linear combinations of the vectors a_1 and a_2 would generate all lattice points shown, as would b_1 and b_2 or any other two different vectors from the grid [43].

A difficult and important mathematical lattice problem is the SVP (Shortest Vector Problem). The problem definition is: given a lattice basis **A**, find the shortest non-zero vector or in other words find the intersection point closest to the origin in the grid [26, 43]. SVP is NP-hard under certain conditions and in general it is said to be an extremely

hard problem to solve for regular computers and also most importantly, there exist no quantum scheme that can be used either [4, 37].

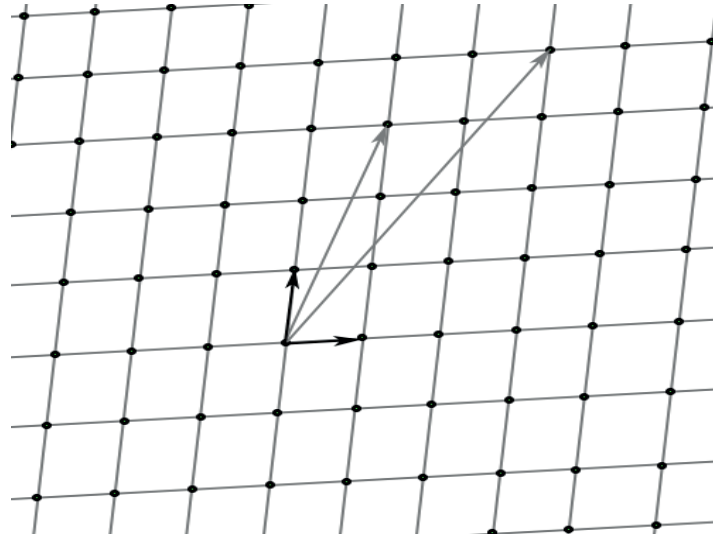


Figure 2.5: A 2-dimensional lattice (dots) with one good vector basis (black) and one bad vector basis (grey).

2.6 NTRU

In 1998 Hoffstein, Pipher and Silverman introduced NTRU as a new cryptosystem. The encryption is based on polynomial algebra and modular reduction while the decryption is based on a procedure for unmixing the steps done in the encryption. The security relies on the polynomial mixing combined with double modular reduction. NTRU is closely related to lattices since key recovery can be framed as a shortest vector problem for a certain lattice. The security of the scheme can therefore be considered as secure as it is hard to find very short vectors (SVP) [16, 37].

The NTRU cryptosystem has four public integer parameters (N, p, q, d) decided by a trusted third party and two private polynomials of degree $N - 1$: f and g. In order for the system to work the parameters need to be chosen according to some rules:

1. $gcd(N, q) = gcd(p, q) = 1$
2. $q > (6d + 1)p$

3. f needs to contain $(d+1)$ coefficients of value 1 and d coefficients of value -1 with the rest equal to 0.

4. g needs to contain both d coefficients of value 1 and d coefficients of value -1 with the rest equal to 0.

The system works in a polynomial ring $\mathcal{R} = \frac{\mathbb{Z}[x]}{x^N - 1}$, where an element $F \in \mathcal{R}$ is a polynomial in the form [20]:

$$F = \sum_{i=0}^{N-1} F_i x^i = \{a_0 + a_1 x + ... + a_{N-1} x^{N-1}\}$$

Multiplication in the ring takes the form of a cyclic convolution product and is represented as

$$\mathbf{a}(x) * \mathbf{b}(x) = \mathbf{c}(x) = \{c_0 + c_1 x + ... + c_{N-1} x^{N-1}\}$$

where

$$c_k = \sum_{i+j \equiv k (modN)} = a_i b_{k-i}$$

for $0 \leq i, j \leq N-1$ and $\mathbf{a}(x), \mathbf{b}(x), \mathbf{c}(x) \in \mathcal{R}$. To perform multiplication modulo p simply reduce the coefficient, $c_k \pmod p$ [16, 37].

2.6.1 Key creation

For Alice to create an NTRU key she must first generate the two polynomials f and g, then compute the inverses f_p^{-1} and f_q^{-1} defined as:

$$f * f_p^{-1} \equiv 1 \pmod p, \; f * f_q^{-1} \equiv 1 \pmod q$$

If the inverses do not exist a new f needs to be generated. Alice can then generate the public key:

$$h \equiv f_q^{-1} * g \pmod q$$

The recommended key sizes giving the highest security is 1595-bits for the private key and 4024-bits for the public key. For moderate security, that can be used for television or cellphone transmissions, the private key size is 340-bits and for the public key it is 642-bits [16].

2.6.2 Encryption and decryption

When Bob wants to send a message to Alice he chooses a polynomial ϕ on the same principle as g (rule 4) and encrypts his message $m \in \mathcal{R}_p$ by computing

$$c \equiv p\phi * h + m \pmod{q}.$$

Bob can now send the encrypted message c to Alice for her to decrypt. When Alice receives the message c from Bob she computes

$$a \equiv f * c \equiv f * p\phi * h + f * m \pmod{q} = p\phi * g + f * m \pmod{q}$$

and then center-lifts a so that the coefficients of the polynomial a ends up between $-q/2$ and $q/2$. This makes it such that

$$a = p\phi * g + f * m \in \mathcal{R}. \tag{2.2}$$

Finally to recover the message Alice uses the inverse f_p^{-1} to do one last calculation and recover m:

$$m \equiv f_p^{-1} * a \pmod{p}.$$

Notice how a in equation 2.2 is not only congruent to an expression that is similar to Bob's encryption process, but is actually equal to the expression. This is because of rule 2 in the rule list which bounds the constants in the polynomial a and is what enables the equality. This in turn makes it possible for Alice to extract the exact value of m [16,37].

2.6.3 Signing

The NTRU cryptographic system can also modified and be used as a signing algorithm. If we again take a look at the cipher text equation:

$$c \equiv p\phi * h + m \pmod{q}.$$

We can modify this a bit and use it as a signature on a document ρ. If we set $t = H(\rho||r)$, where H is a public hashing function converting a digital document into a polynomial in a ring, r is a random string making the polynomial valid and '||' is concatenation. We

can then re-write the equation as the following:

$$t \equiv s * h + v \pmod{q}.$$

The signature then becomes (s, r) since computing s is hard unless you have access to the secret NTRU polynomials f and g. The signature is also easily verifiable for anyone by doing the comparison:

$$||(s, s * h - H(\rho||r)|| < B$$

where B is some public security threshold ensuring that both s and v are small [15].

2.6.4 NTRU lattice

The NTRU key recovery can, for a special sort of lattice, be mapped to the shortest vector problem [37]. The public key

$$h(x) = h_0 + h_1 x + h_2 x^2 + ... + h_{N-1} x^{N-1}$$

has an NTRU 2N-dimensional lattice associated to it generated by the rows of the matrix

$$M_h = \begin{pmatrix} I & h0 & qI \end{pmatrix} = \left(\begin{array}{cccc|cccc} 1 & 0 & \dots & 0 & h_0 & h_1 & \dots & h_{N-1} \\ 0 & 1 & \dots & 0 & h_{N-1} & h_0 & \dots & h_{N-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 & h_1 & h_2 & \dots & h_0 \\ \hline 0 & 0 & \dots & 0 & q & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & 0 & q & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & q \end{array}\right)$$

where the upper right quarter consists of cyclical permutations of the coefficients from $h(x)$.

As mentioned earlier the security of NTRU depends on the SVP in the lattice generated by M_h. If Eve somehow can solve the SVP for the lattice within a small factor the vector might work as a decryption key [37]. The LLL (Lenstra, Lenstra and Lovász) algorithm is a lattice basis reduction algorithm that helps solve the SVP to some factor in polynomial time [22]. The LLL algorithm reduces the basis of a lattice and the shortest vector can sometimes be taken from said basis. If N is large (in high dimensions)

the LLL algorithm does not find small vectors in the NTRU lattice, so with well chosen parameters the NTRU cryptosystem stays secure [37]. Recent work in this area has shown that the LLL algorithm can be improved to become faster and also work in larger dimensions. In 2023 Ryan and Heninger introduced a new improved version of the LLL algorithm. It is more efficient than the previous best adaption of the LLL, however it does not affect the security of today's lattice based cryptosystems [32].

2.7 Learning with errors

Learning with errors is a problem that was introduced by Oded Regev which can also be reduced to a lattice problem. The learning with errors problem is a hard problem to solve and in short is the problem of solving s for $As = b - e$ given A and $b + e$ where $s, b, e \in \mathbb{Z}_q^n$ and $A \in \mathbb{Z}_q^{m \times n}$. The problem also states that the elements of A are uniformly random but the elements of e are small with a high probability, such as drawn from a Discrete Gaussian distribution [23, 24]. The problem can be used as an encryption scheme to encrypt a single bit x.

If $(A, b + e)$ is public then someone can select k row vectors $a_i \in A$ and the corresponding elements $b_{ei} \in (b+e)$ to send $[\sum_k a_i, x \cdot \frac{q}{2} + \sum_k b_{ei}] = [a', b'] \in \mathbb{Z}_q^2$. Anyone with the secret key s can then with a very high probability decrypt the bit accurately by checking whether $b' - a's$ is closer to $\frac{q}{2}$ $(x = 1)$ or 0 $(x = 0)$. This idea has then further been expanded to use elements of polynomial rings instead of $\mathbb{Z}_q$ and is called the Ring learning with errors key exchange mechanism. As the name implies, Ring learning with errors key exchange mechanism is a scheme made to transfer keys similarly to the Diffie-Hellman key exchange but this key exchange is, unlike Diffie-Hellman, fortunately quantum secure [10].

2.8 IBE NTRU

This identity-based scheme is based on NTRU and learning with errors and was created by Ducas, Lyubashevsky and Prest to create a practical lattice cryptosystem that has reasonable key sizes. Like NTRU this system works in the ring $\mathcal{R}_q = \mathbb{Z}[x]/(x^N - 1)$.The system also uses the GPV trapdoor sampling algorithm introduced by Gentry, Peikert and Vaikuntanathan in 2008. The GPV algorithm can generate short lattice vectors while not revealing any information about the trapdoor [11, 13].

2.8.1 IBE key generation

In IBE-schemes a master key is needed to generate secret keys for the users and this scheme is no different. The Master secret key is a matrix $\mathbf{B} \in Z^{2N \times 2N}$ where N is a chosen parameter that is a power of two. **B** is a short basis for the lattice Λ utilizing the GPV algorithm to generate short vectors without leaking information, here the basis **B** becomes the trap-door. **B** is generated by the two random NTRU polynomials f and g with a fixed square norm.

With $F, G \in \mathcal{R}_q$ satisfying

$$g * G - g * F = q \; and \; h = g * f^{-1} \bmod q$$

the two matrices **B** and **A** can be created. Where

$$\mathbf{B} = \begin{pmatrix} \mathcal{A}(g) & -\mathcal{A}(f) \\ \mathcal{A}(G) & -\mathcal{A}(F) \end{pmatrix} \text{ and } \mathbf{A} = \begin{pmatrix} -\mathcal{A} & I_N q I_N & 0_N \end{pmatrix}$$

generate the same lattice and the anticirculant matrix of f is

$$\mathcal{A}(f) = \begin{pmatrix} f_0 & f_1 & \dots & f_{N-1} \\ -f_{N-1} & f_0 & \dots & f_{N-2} \\ \vdots & \vdots & \ddots & \vdots \\ -f_1 & -f_2 & \dots & f_0 \end{pmatrix}$$

Since everyone has access to the public key h they can generate the matrix **A**. The problem is that even though it generates the same lattice as **B** it does not give a good basis for solving lattice problems because if h is uniformly distributed in $\mathcal{R}_q$, the matrix **A** has a large orthogonal defect [11].

Users in this scheme must have their own secret key to be able to decrypt messages sent to them. The secret key extraction first checks if the user's secret key is in the storage and if it is then it simply outputs the secret key. Otherwise the secret key is created by a combination of the hashed user ID (t) and the short elements (s_1, s_2) sampled from the short basis **B** such that $s_1 + s_2 * h = t$. This is done by comparing t to the lattice Λ and setting one of the closest lattice points to be $s_2 * h$ through Gaussian sampling. The secret key s_2 is then sent to the user and also stored [11].

2.8.2 Encryption and decryption

Ring LWE is used for the encryption scheme. The encryption of a message m to a user with a certain ID is done by first choosing three elements $\{r, e_1, e_2\} \in \{1, 0, -1\}^N$ and a $k \in \{0, 1\}^N$ that is uniformly distributed and serves as a key. Encryption also uses the two hash functions H and H'.The output of the encryption is the triple $(u, v, m \oplus H'(k))$ where ,

$$u = r * h + e_1 \textit{ and } v = r * t + e_2 + \lfloor q/2 \rfloor \cdot k \textit{ where } u, v \in \mathcal{R}_q \textit{ and } t = H(id)$$

and the least significant bits of v are dropped [11, 23].

For the decryption the receiver can calculate k by computing

$$k = \lfloor \frac{v - u * s_2}{q/2} \rceil$$

and then taking the third element from the encryption output $(m \oplus H'(k))$ and performing XOR with $H'(k)$ to get the message,

$$m = m \oplus H'(k) \oplus H'(k).$$

For a correct decryption the coefficients of $r * s_1 + e_2 - e_1 * s_2$ must have a magnitude less than $q/4$ [11, 23].

2.8.3 IBE threat model

CIA triad

The "CIA triad", where CIA stands for Confidentiality, Integrity and Availability, is a model often used when looking for vulnerabilities in a system. Confidentiality is about making sure the information is only accessible to authorized users. Integrity is about ensuring authenticity of the information and also protection against any form of modification to said information. Lastly Availability means that the information should always be accessible in a reliable and timely manner. These are the three cornerstones of information security [1].

To maintain confidentiality in an identity-based encryption scheme, the private keys

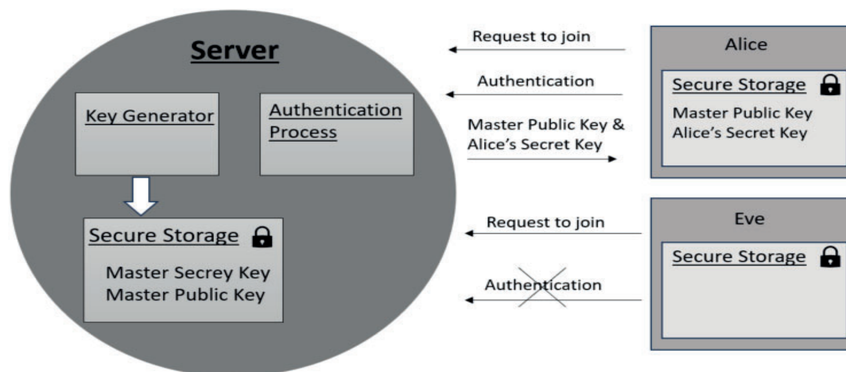


Figure 2.6: Threat model for an IBE system. It shows two users requesting to join but only Alice has the right. The model also shows that both server and users need a secure storage to store the keys in.

must be generated and managed securely because only the intended recipient should be able to decrypt the encrypted information. The master secret key must also be stored securely because with access to it, all encryption can be decrypted. Since only the generator has this key, and it does not need to be transferred, it is the keeping of the master secret key that must be secure, as can be seen in figure 2.6 [35].

There are more aspects to integrity, namely authenticity, freshness and non-repudiation. Authenticity means that the information is original and its source is validated. Non-repudiation proves the validity of the information by having indisputable proof of authenticity and validity, thus making sure the sender cannot deny their involvement. Non-repudiation guarantees authenticity but authenticity does not guarantee non-repudiation [33]. Freshness is a descriptor of how old the data is and in an encryption system this is relevant to prevent replay attacks, i.e. when the same message is sent multiple times. Time stamps are used to measure the freshness of data and they can be used in messages [6, 29]. Regarding the authenticity and non-repudiation of an IBE-scheme the sender must be able to prove its identity and a way to do this is with digital signatures. Then the receiver can prove that the alleged sender actually is the one who sent the message [18].

Lastly to consider is the availability of an IBE-scheme and this comes down to making sure all parties have the keys needed to perform encryption and decryption. A scenario

that could damage the availability of the encryption scheme is if a private key is leaked, then the system must have a strategy for key revocation. This strategy must cover who can revoke a key, how to inform all users of the revocation and lastly how to handle all messages sent encrypted with the leaked key [39].

Another important aspect is the transferring of the secret key and the master public key from the key generation center to the user. This must be done over a secure channel which means the user and key generation center must somehow set one up. If this channel is not secure then the confidentiality and the authenticity of the secret key is not protected [27].

Message recovery attack

A strategy for attacking the encryption scheme is to try to recover the errors e_1 and e_2, to recover the message. With the ciphertext (u,v) one can formulate the equation

$$(t * h^{-1}) * e_1 - e_2 = (t * h^{-1}) * u - v \text{ mod } q.$$

That equation can be converted to finding the vector $(e_1, e_2, 1)$ in a $2N + 1$ dimensional lattice. However, with proper parameters for the scheme this method is not practical [11].

Side channel attacks

Instead of trying to break the crypto one could resort to side channel attacks. Side channel attacks are based on indirect information from the crypto scheme. Examples of this kind of information are cache accesses, power consumption and how much time different computations take. Even though a crypto scheme is mathematically secure it could be insecure as an implementation. One way to counter side channel attacks is by masking and the foundation of masking is secret sharing which splits a secret value into multiple shares. A function is also needed to recover the secret from some or all the shares. Older versions of NTRU are vulnerable to side channel attacks and in article [42] the secret decryption key is recovered with high probability. There exists newer versions of NTRU that are fully masked and because of that secure against side channel attacks but these versions have a much higher computational time [21, 42].

2.9 Parallelization technologies

2.9.1 CUDA

CUDA is a GPU (Graphics Processing Unit) programming language created by Nvidia to make it easier for software developers to utilize the performance of GPUs in regular tasks. The CUDA programming language uses both the CPU (Central Processing Unit) and GPU for computational tasks in something known as heterogeneous computing where sequential tasks are handled by the CPU and tasks that can be performed in parallel is offloaded to the GPU. Therefore the performance boost gained from CUDA is dependent on the tasks themselves [9].

A CPU is designed to compute a few potentially hard tasks while the GPU is designed to run a lot of simple tasks. The CPU is suited to run operating systems and programs with a lot of different tasks that can be done in any order while the GPU can take care of tasks that can be divided into smaller subtasks that can be worked on individually. Initially, the GPU's main purpose was to speed up graphics rendering and geometry transformations but since it can process data simultaneously it is also used in areas like artificial intelligence, machine learning and other compute-intensive tasks. The reason a GPU can perform parallel processing is because it can run thousands of threads simultaneously [9, 28].

CUDA keeps track and controls all threads with something known as a grid. The grid contains blocks where each block contains warps that are a collection of 32 threads. Each block contain multiple warps resulting in hundreds or thousands of threads per block, depending on the used GPU architecture. The best situation for CUDA is in so called embarrassingly parallel problems. These problems should have almost no inter-block communication meaning that a block can perform computations without depending on the results or data from other blocks. If all the blocks can compute on their own the performance will be the highest because the blocks will not have to wait for each other to finish [9].

2.9.2 SIMD

SIMD (Single Instruction, Multiple Data) is a method to get increased performance when computing on CPUs. By using SIMD, parallel processing of data sets is possible even on single cores [40]. Usually a CPU runs its instructions on single words (64 bit data) or less but Single Instruction Multiple Data allows the CPU to run the same

instruction on entire data sets at the same time. When a SIMD instruction is fetched by the CPU, the ALU (Arithmetic Logic Unit) takes two vectors instead of two values as input. Then the ALU performs the fetched instruction and returns a new vector as can be seen in figure 2.7. The ALU both stores its output and fetches its input from registers, therefore with larger registers more data can be handled at the same time and using more registers reduces data movement to and from the cache memory which is another time consuming component [5, 12].

SIMD is not a single instruction set but consists of multiple different ones for different CPU architectures. SSE (Streaming SIMD Extensions), AVX (Advanced Vector Extensions) and their extended versions are common instruction sets for the x86 architecture used by both Intel and AMD. AVX2 for example uses 16 registers to enable single instructions on 256 bits of data, this means that eight 32 bit values or four 64 bit values can be processed at the same time. The further extension AVX-512 can handle 512 bits of data by doubling the number of available registers and in turn double the data sets that can be processed at once [5].

There are two distinct SIMD instruction sets for ARM processors called Neon and SVE. Neon is the standard and although having similar instruction support as those of the x86 architecture the maximum number of bits that can be operated on is only 128. The performance boost should therefore theoretically be half of AVX2 but since the two architectures are fundamentally different such a claim is hard to back up and would differ on a case to case basis. SVE supports operations on larger bit-ranges, namely multiples of 128 bits up to 2048 bits but is only supported by some ARM processors.

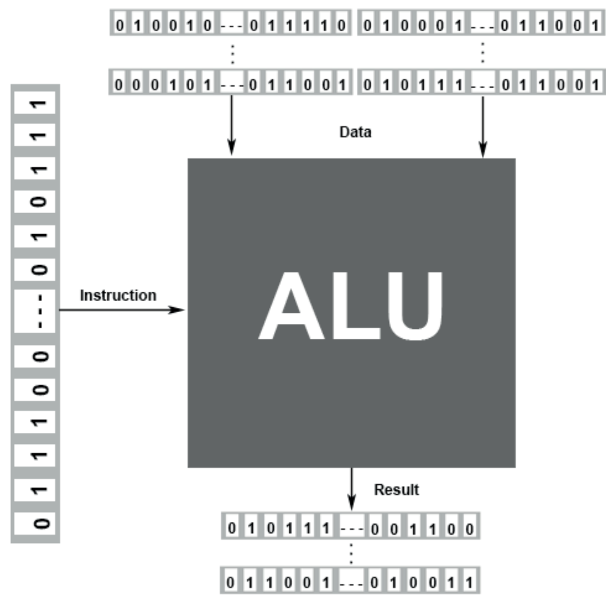


Figure 2.7: ALU using SIMD.

Chapter 3

Methodology

3.1 Client server setup for ID-based NTRU in C++

From [11] there is a repository link to Prest's GitHub, containing a proof of concept of an identity-based encryption scheme over NTRU lattices. Although written by the three researchers Ducas, Lyubashevsky and Prest we will for the sake of simplicity, in short refer to the code as Prest's code since it is hosted on his GitHub. The implementation was written in C++ and is measuring the time for creating keys and performing encryption and decryption, however Prest mentions that their code is meant as a proof of concept and should not be used in real world implementations.

Their code was created using the NTLlib library for dealing with polynomials which were used to represent keys and IDs. The algebraic functions and different Fourier transforms were added by Prest and could be found in the files FFT.cc and Algebra.cc located in the same repository. These functions were used in the file Scheme.cc which contains the functions needed for an encryption scheme, like the key generation functions, encryption and decryption. The file also contained a few functions used for benchmarking performance. The recommended compiler for the code was the g++ compiler together with the "-0fast" flag for optimization.

To get a more practical "test case" the changes made on the original code was to turn it from a test case environment into a client/server structure. Two classes were added for the new structures, a client class and a server class. The server class was implemented with the purpose of managing master keys and extracting keys for the clients based on their IDs. The client class was made to represent a user of the system, the client has access to a server and its own keys which grants it the ability to encrypt and

decrypt messages. We reused and refactored code from the Scheme.cc file in the new classes when creating keys, encrypting and decrypting messages. The classes can be seen in figure 3.1 and the type MSK_Data used by MSKD is a simple C++ structure containing the NTRU secret polynoms f and g as well as the values F and G where $f * G - g * F = q$, as explained in [15]. MSK_Data also contains the NTRU lattice created by the secret polynoms, their FFT-transform and the Grahm-Schmidt values used by the GPV during key extraction. The other data type MPK_Data is only a structure containing the NTRU public polynom h and its FFT-transform.

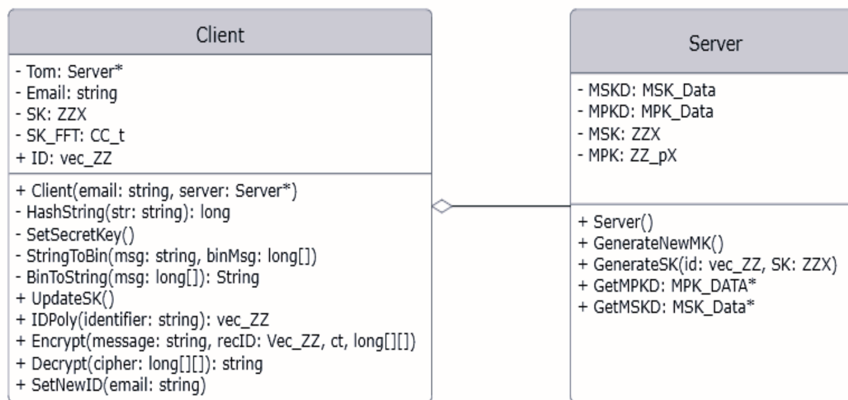


Figure 3.1: UML diagram of the Client and Server classes.

To find out which methods that were taking the most time and would require most of our focus, Google Performance Tools (gperftools) was used, the result can be found in Appendix A. It showed where in the code the computer spent most time and thus showing the bottlenecks in the code that could perhaps be parallelised or at least made more efficient. It became clear that a lot of time was spent computing the dot product when running the code. This was because the function was used heavily during key generation, both for the master key and during user key extraction. Since encryption and decryption are the most performed actions in the cryptosystem they are also the most time critical parts of the system and the speed of the encryption/decryption process reflect the speed of the over all system. Using gperftools to look at the encryption/decryption process we saw that the most concerned methods were the Fourier transform and the reverse Fourier transform used for polynomial convolution.

Encryption/decryption operation is used most often, then user key generation. Master key generation is not as common, and occurs least often. Generating master keys is only done when setting up a trusted server or in order to perform key revocation in the form of regular updates or key leakage. When a master key has been generated a new user key has to be extracted for all users in the IBE system, this could amount to hundreds or many thousands of users that need new keys. This was taken into consideration when updating the functions in Prest's code as well as the fact that the original code was written for 128 bit floating point accuracy while our optimization methods are restricted to support at most 64 bit floating point numbers. This loss in accuracy produces rounding errors which impacts the results of some of the operations in the code which we needed to be aware of when implementing our changes to the code. Taking these factor into consideration we decided to not update the code for master key generation since it is not used very often and errors produced in the master key could produce errors in the rest of the system.

3.2 Hardware

When gathering our results we wanted to test our implementation on diverse platforms to gain an understanding of what type of hardware is required to get good enough performance as well as how big of an impact hardware had on performance. We therefore decided to run our code on 3 different platforms with different hardware and limitations. The platforms we used and their hardware specifications can be summarized in table 3.1.

Name	RAM	CPU	GPU	SIMD
ARM64	8	2.4GHz ARM Cortex-A76	-	Neon (128 bit)
x86-1	16	4.4GHz Intel i7 4790k	NVIDIA RTX3080	AVX2 (256 bit)
x86-2	-	2.2GHz Intel Xeon	NVIDIA L-4	AVX2 (256 bit)

Table 3.1: The different platforms that were used to perform measurements on.

3.3 Software

Both the original and our implementation used a few non-standard C++ libraries to perform operations and they are worth mentioning here. The first two used in the original

implementation are NTL and GMP. NTL is an open source C++ library made for number theory and operates on on arbitrary length integers in values, vectors, matrices and polynomials. GMP on the other hand is an open source C++ library with support for arbitrary length integers, rational and floating point numbers. In Prest's revised versions there are mentions of the NFLlib library which is an extension of the open source XPIR library [3]. NFLlib is a library using Number Theoretic Transform (NTT) and the Chinese Remainder Theorem (CRT) to optimize operations in polynomial rings, which makes it ideal for use in lattice based cryptography. NFLlib is dependent on another library, the GMP extension MPFR, which is also required for NFLlib to run. Finally in our implementation we use SHA-512 hashing of string identities, which we gained access to by using the popular openssl library.

Chapter 4

Implementation and Pseudocode

In this chapter we show the implementations made by us in pseudo code. Most implementations are similar to Prest's original code but with a few variations while others differ more significantly. All implementations of encryption and decryption share the same set of variables and have the same purpose in each implementation. What the variables mean and represent has previously been presented in section 2.8.2. However just like Prest's original code we let k equal m and avoid the hash and XOR process for the encryption. We did this since it is separate from the rest of the process and can be done anywhere, whenever needed and is a fast operation even for large N. For a key and message length of $N = 2048$ the hash and XOR function only took a single microsecond using SHA-512 and concatenation which is the size of the impact on our results.

4.1 CUDA implementation

Writing code using CUDA requires writing separate code for the CPU and GPU. The code running on the CPU is ordinary C++ code known as host code and the code running on the GPU is known as kernels or device code and is also C++ code but with a few caveats. Kernels are run fully in parallel with multiple threads running the same block of code which means that if not careful, race conditions will occur. Because it is the same code running on all threads, a thread needs to be able to be uniquely distinguished and this is done through the dimension, block and index attributes available in code. From these unique attributes you can conclude how and which operations should be run by each thread. In our pseudo code these are represented by ”ID”.

The host code is responsible for memory management in the sense that it allocates mem-

ory on both the CPU and the GPU as well as transferring data between CPU memory and GPU memory. Host code also decides how threads should be ordered before triggering kernel calls. As mentioned previously, the optimum number of threads largely depend on the GPU architecture. Both the NVIDIA RTX 3080 and L-4 support 1024 threads per block so in our host code we only used multiple blocks when the number of parallel operations were greater than 1024.

When it comes to the dot-product implementation (Algorithm 1) it needs to be noted that it has the same drawback as the SIMD implementations: the maximum floating point precision is 64 bit which is lower than the 128 bit precision of the original. This could theoretically introduce rounding errors and affect the end result but this did not show in the way it is used within the code. We tried two different versions of DotProduct in CUDA, one where summation was done in host code and one where it was done in device code (Algorithm 1), surprisingly there was little to no difference in the two implementations.

The implementations for encryption and decryption in CUDA (Algorithms 2, 3, 4, 5) are quite different to the original implementations, this is because these do not use FFT for the convolution operation. Instead we use the more primitive but easier to turn parallel vector-matrix multiplication. In order to perform convolution in the cyclic group using the two polynomials $P_1 = a_0 + a_1x^1 + \cdots + a_{N-1}x^{N-1}$ and $P_2 = b_0 + b_1x^1 + \cdots + b_{N-1}x^{N-1}$ we let the P_1 be represented by the vector,

$$v = [a_0, a_1, ..., a_{N-1}]$$

and P_2 be transformed into the circular matrix,

$$B = \begin{pmatrix} b_0 & -b_{N-1} & \dots & -b_1 \\ b_1 & b_0 & \dots & -b_2 \\ \vdots & \vdots & & \vdots \\ b_{N-2} & b_{N-3} & \dots & -b_1 \\ b_{N-1} & b_{N-2} & \dots & b_0 \end{pmatrix}$$

the convolution $P_1 * P_2$ can then be performed through $vB \pmod q \in \frac{\mathbb{Z}[x]}{x^N-1}$. This is what we implemented for our GPU kernels instead of using FFT.

Algorithm 1 DotProductKernel(sum, x_1, x_2)

1: products$_{SHARED}$[ID] $\leftarrow$ x_1[ID] $\cdot$ x_2[ID]
2: **if** ID$_{THREADS}$ = 0 **then**
3: sum$_{LOCAL}$ $\leftarrow 0$
4: **for** $i \in \{0, 1, 2, ..., \frac{THREADS}{BLOCKS} - 1\}$ **do**
5: sum$_{LOCAL}$ $\leftarrow$ sum$_{LOCAL}$ + products$_{SHARED}[i]$
6: **end for**
7: sum $\leftarrow$ sum + sum$_{LOCAL}$
8: **end if**

Algorithm 2 EncryptionKernel(u, v, x, m, t, h, r)

1: $u_0 \leftarrow 0$
2: $v_0 \leftarrow 0$
3: $e_1 \leftarrow_{random} \{-1, 0, 1\}$
4: $e_2 \leftarrow_{random} \{-1, 0, 1\}$
5: **for** $i \in \{0, 1, 2, ..., N - 1\}$ **do**
6: **if** $i \leq$ ID **then**
7: $u_0 \leftarrow u_0 + h[ID - i] \cdot r[i] \mod q$
8: $v_0 \leftarrow v_0 + t[ID - i] \cdot r[i] \mod q$
9: **else**
10: $u_0 \leftarrow u_0 - h[ID - i] \cdot r[i] \mod q$
11: $v_0 \leftarrow v_0 - t[ID - i] \cdot r[i] \mod q$
12: **end if**
13: **end for**
14: $u[ID] \leftarrow u_0 + e_1 \mod q$
15: $v[ID] \leftarrow v_0 + e_2 + \frac{q}{2} m[ID] \mod q$

Algorithm 3 DecryptionKernel(m, u, v, s)

1: $\mathrm{m}_0 \leftarrow 0$
2: **for** $i \in \{0, 1, 2, ..., N-1\}$ **do**
3:     **if** $i \leq$ ID **then**
4:         $\mathrm{m}_0 \leftarrow \mathrm{m}_0 + \mathrm{u}[\mathrm{ID} - i] \cdot \mathrm{s}[i] \mod q$
5:     **else**
6:         $\mathrm{m}_0 \leftarrow \mathrm{m}_0 - \mathrm{u}[\mathrm{ID} - i] \cdot \mathrm{s}[i] \mod q$
7:     **end if**
8: **end for**
9: $\mathrm{m}_0 \leftarrow \mathrm{v}[\mathrm{ID}] - \mathrm{m}_0 \mod q$
10: **if** $\mathrm{m}_0 < \frac{q}{4}$ **or** $\frac{3q}{4} < \mathrm{m}_0$ **then**
11:     $\mathrm{m}[\mathrm{ID}] \leftarrow 0$
12: **else**
13:     $\mathrm{m}[\mathrm{ID}] \leftarrow 1$
14: **end if**

Algorithm 4 GPUEncryption(m, t, h)

1: u_{CUDA}, v_{CUDA}
2: $\mathrm{m}_{CUDA} \leftarrow \mathrm{m}$
3: $\mathrm{t}_{CUDA} \leftarrow \mathrm{t}$
4: $\mathrm{h}_{CUDA} \leftarrow \mathrm{h}$
5: $\mathrm{r}_{CUDA} \leftarrow_{random} \{-1, 0, 1\}^N$
6: EncryptionKernel(u_{CUDA}, v_{CUDA}, m_{CUDA}, t_{CUDA}, h_{CUDA}, r_{CUDA})
7: $\mathrm{u} \leftarrow \mathrm{u}_{CUDA}$
8: $\mathrm{v} \leftarrow \mathrm{v}_{CUDA}$
9: **return** [u,v]

Algorithm 5 GPUDecryption(u, v, s)

1: m_{CUDA}
2: $\mathrm{u}_{CUDA} \leftarrow \mathrm{u}$
3: $\mathrm{v}_{CUDA} \leftarrow \mathrm{v}$
4: $\mathrm{s}_{CUDA} \leftarrow \mathrm{s}$
5: DecryptionKernel(m_{CUDA}, u_{CUDA}, v_{CUDA}, s_{CUDA})
6: $\mathrm{m} \leftarrow \mathrm{m}_{CUDA}$
7: **return** m

4.2 SIMD implementaions

For the SIMD implementation the code for the Fourier transform (Algorithms 8, 10), reverse Fourier transform (Algorithms 9, 11), the dot product (Algorithms 6, 7) and the supporting code for these (Algorithms 12, 13, 14, 15) was changed. Although we had two different SIMD implementations, one for ARM and one for x86, the general ideas are the same and that is why the pseudo code is so similar. A C++ library containing the Neon extension was used for the ARM chip and a library containing the AVX2 intrinsics was used for the x86 chips. The 128 bit floating point precision used by the original code was not compatible with our SIMD instructions since the maximum support is 64 bit floating point. For this reason we had to downgrade the precision which enabled us to perform operations on two 64 bit elements using Neon or four 64 bit elements using AVX2 at the same time. This downgrade in precision had the same results as it had for our GPU implementation: no affect on the outcome in all tests.

DotProduct was implemented first and the basic idea was pretty straightforward; to reduce the number of multiplications and additions needed. This was done by moving the maximum number of elements into SIMD vectors and performing the same operation on multiple elements at the same time. Effectively speeding up the algorithm by a factor close to the vector sizes.

Implementing the Fast Fourier Transform algorithms with SIMD was less straightforward. To be able to implement the FFT algorithms using SIMD a complex data type compatible with SIMD was needed. This was done by storing the real and imaginary parts of complex numbers in different SIMD vectors and overriding each mathematical operation with their complex counterpart using SIMD instructions. The rest followed similarly to the idea of the DotProduct implementation: reduce the number of mathematical operations in each step and adapt the code to use SIMD instructions instead of ordinary C++ statements such as EveryOther in algorithms 8, 10 and ReverseEveryOther in algorithms 9, 11. ω was not converted into the complex SIMD type since doing so did not increase performance but reduce accuracy because of the many recursive multiplications and small differences between two ωs.

Algorithm 6 DotProduct(x_1, x_2)

1: $\text{sum}_{SIMD256} \leftarrow 0$
2: **for** $i \in \{0, 4, 8, 12, ...\}$ **do**
3: $\quad \text{reg}^1_{SIMD256} \leftarrow x_1[i, ..., i + 3]$
4: $\quad \text{reg}^2_{SIMD256} \leftarrow x_2[i, ..., i + 3]$
5: $\quad \text{sum}_{SIMD256} \leftarrow \text{sum}_{SIMD256} + (\text{reg}^1_{SIMD256} \cdot \text{reg}^2_{SIMD256})$
6: **end for**
7: $\text{sum}_{64} \leftarrow \text{Reduce}(\text{sum}_{SIMD256})$
8: **return** sum_{64}

Algorithm 7 DotProduct(x_1, x_2)

1: $\text{sum}_{SIMD128} \leftarrow 0$
2: **for** $i \in \{0, 2, 4, 6, ...\}$ **do**
3: $\quad \text{reg}^1_{SIMD128} \leftarrow x_1[i, i + 1]$
4: $\quad \text{reg}^2_{SIMD128} \leftarrow x_2[i, i + 1]$
5: $\quad \text{sum}_{SIMD128} \leftarrow \text{sum}_{SIMD128} + (\text{reg}^1_{SIMD128} \cdot \text{reg}^2_{SIMD128})$
6: **end for**
7: $\text{sum}_{64} \leftarrow \text{Reduce}(\text{sum}_{SIMD128})$
8: **return** sum_{64}

Algorithm 8 FFTStep($f_{SIMD256}$, n, ω)

1: **if** $n = 4$ **then**
2: $f_{4\times64} \leftarrow f_{SIMD256}[0]$
3: $fft_{4\times64}[0] \leftarrow f_{4\times64}[0] + f_{4\times64}[2]i + \frac{1}{\sqrt{2}}(f_{4\times64}[1] + f_{4\times64}[3]) \cdot (1, i)$
4: $fft_{4\times64}[1] \leftarrow f_{4\times64}[0] - f_{4\times64}[2]i + \frac{1}{\sqrt{2}}(f_{4\times64}[1] - f_{4\times64}[3]) \cdot (-1, i)$
5: $fft_{4\times64}[2] \leftarrow f_{4\times64}[0] + f_{4\times64}[2]i + \frac{1}{\sqrt{2}}(f_{4\times64}[1] + f_{4\times64}[3]) \cdot (-1, -i)$
6: $fft_{4\times64}[3] \leftarrow f_{4\times64}[0] - f_{4\times64}[2]i + \frac{1}{\sqrt{2}}(f_{4\times64}[1] - f_{4\times64}[3]) \cdot (1, -i)$
7: $fft_{SIMD256}[0] \leftarrow fft_{4\times64}$
8: **else**
9: $f_{even} \leftarrow$ EveryOther($f_{SIMD256}$, 0)
10: $f_{odd} \leftarrow$ EveryOther($f_{SIMD256}$, 1)
11: $fft_{even} \leftarrow$ FFTStep(f_{even}, $\frac{n}{2}$, ω^2)
12: $fft_{odd} \leftarrow$ FFTStep(f_{odd}, $\frac{n}{2}$, ω^2)
13: $\omega^k_{SIMD256} \leftarrow [\omega, \omega^3, \omega^5, \omega^7]$
14: **for** $k \in \{0, 1, 2, ..., \frac{n}{4} - 1\}$ **do**
15: $fft_{SIMD256}[k] \leftarrow fft_{even}[k \mod \frac{n}{8}] + \omega^k_{SIMD256} \cdot fft_{odd}[k \mod \frac{n}{8}]$
16: $\omega^k_{SIMD256} \leftarrow \omega^8 \cdot \omega^k_{SIMD256}$
17: **end for**
18: **end if**
19: **return** $fft_{SIMD256}$

Algorithm 9 ReverseFFTStep($\text{fft}_{SIMD256}$, n, ω)

1: **if** $n = 4$ **then**
2: $\quad \text{fft}_{4\times64} \leftarrow \text{fft}_{SIMD256}[0]$
3: $\quad \text{f}_{4\times64}[0] \leftarrow \text{fft}_{4\times64}[0] + \text{fft}_{4\times64}[2] + \text{fft}_{4\times64}[1] + \text{fft}_{4\times64}[3]$
4: $\quad \text{f}_{4\times64}[1] \leftarrow (\text{fft}_{4\times64}[0] - \text{fft}_{4\times64}[2]) \cdot (1, -i) + (\text{fft}_{4\times64}[1] - \text{fft}_{4\times64}[3]) \cdot (-1, -i)$
5: $\quad \text{f}_{4\times64}[2] \leftarrow \text{fft}_{4\times64}[0] + \text{fft}_{4\times64}[2] - (\text{fft}_{4\times64}[1] + \text{fft}_{4\times64}[3])$
6: $\quad \text{f}_{4\times64}[3] \leftarrow (\text{fft}_{4\times64}[0] - \text{fft}_{4\times64}[2]) \cdot (1, -i) - (\text{fft}_{4\times64}[1] - \text{fft}_{4\times64}[3]) \cdot (-1, -i)$
7: $\quad \text{f}_{SIMD256}[0] \leftarrow [\frac{1}{4}, \frac{1}{4\sqrt{2}}, -\frac{i}{4}, -\frac{i}{4\sqrt{2}}] \cdot \text{f}_{4\times64}$
8: **else**
9: $\quad \omega^k_{SIMD256} \leftarrow [\omega, \omega^3, \omega^5, \omega^7]$
10: $\quad$ **for** $k \in \{0, 1, 2, ..., \frac{n}{8} - 1\}$ **do**
11: $\quad\quad \text{fft}_{even}[k] \leftarrow \frac{1}{2}(\text{fft}_{SIMD256}[k] + \text{fft}_{SIMD256}[k + \frac{n}{8}])$
12: $\quad\quad \text{fft}_{odd}[k] \leftarrow \frac{1}{2}\omega^k_{SIMD256} \cdot (\text{fft}_{SIMD256}[k] - \text{fft}_{SIMD256}[k + \frac{n}{8}])$
13: $\quad\quad \omega^k_{SIMD256} \leftarrow \omega^8 \cdot \omega^k_{SIMD256}$
14: $\quad$ **end for**
15: $\quad \text{f}_{even} \leftarrow$ ReverseFFTStep(fft_{even}, $\frac{n}{2}$, ω^2)
16: $\quad \text{f}_{odd} \leftarrow$ ReverseFFTStep(fft_{odd}, $\frac{n}{2}$, ω^2)
17: $\quad \text{f}_{SIMD256} \leftarrow$ ReverseEveryOther(f_{even}, f_{odd})
18: **end if**
19: **return** $\text{f}_{SIMD256}$

Algorithm 10 FFTStep($\text{f}_{SIMD128}, n, \omega$)

1: **if** $n = 2$ **then**
2: $\quad \text{f}_{2\times64} \leftarrow \text{f}_{SIMD128}[0]$
3: $\quad \text{fft}_{2\times64}[0] \leftarrow \text{f}_{2\times64}[0] + \text{f}_{2\times64}[1]i$
4: $\quad \text{fft}_{2\times64}[1] \leftarrow \text{f}_{2\times64}[0] - \text{f}_{4\times64}[2]i$
5: $\quad \text{fft}_{SIMD128}[0] \leftarrow \text{fft}_{2\times64}$
6: **else**
7: $\quad \text{f}_{even} \leftarrow$ EveryOther($\text{f}_{SIMD128}, 0$)
8: $\quad \text{f}_{odd} \leftarrow$ EveryOther($\text{f}_{SIMD128}, 1$)
9: $\quad \text{fft}_{even} \leftarrow$ FFTStep($\text{f}_{even}, \frac{n}{2}, \omega^2$)
10: $\quad \text{fft}_{odd} \leftarrow$ FFTStep($\text{f}_{odd}, \frac{n}{2}, \omega^2$)
11: $\quad \omega^k_{SIMD128} \leftarrow [\omega, \omega^3]$
12: $\quad$ **for** $k \in \{0, 1, 2, ..., \frac{n}{2} - 1\}$ **do**
13: $\quad\quad \text{fft}_{SIMD128}[k] \leftarrow \text{fft}_{even}[k \mod \frac{n}{4}] + \omega^k_{SIMD128} \cdot \text{fft}_{odd}[k \mod \frac{n}{4}]$
14: $\quad\quad \omega^k_{SIMD128} \leftarrow \omega^4 \cdot \omega^k_{SIMD128}$
15: $\quad$ **end for**
16: **end if**
17: **return** $\text{fft}_{SIMD128}$

Algorithm 11 ReverseFFTStep($\text{fft}_{SIMD128}, n, \omega$)

1: **if** $n = 2$ **then**
2: $\quad \text{fft}_{2\times64} \leftarrow \text{fft}_{SIMD128}[0]$
3: $\quad \text{f}_{2\times64}[0] \leftarrow \text{fft}_{2\times64}[0] + \text{fft}_{2\times64}[1]$
4: $\quad \text{f}_{2\times64}[1] \leftarrow (\text{fft}_{2\times64}[0] - \text{fft}_{2\times64}[1]) \cdot (0, -i)$
5: $\quad \text{f}_{SIMD128}[0] \leftarrow \frac{1}{2}\text{f}_{2\times64}$
6: **else**
7: $\quad \omega^k_{SIMD128} \leftarrow [\omega, \omega^3]$
8: $\quad$ **for** $k \in \{0, 1, 2, ..., \frac{n}{4} - 1\}$ **do**
9: $\quad\quad \text{fft}_{even}[k] \leftarrow \frac{1}{2}(\text{fft}_{SIMD128}[k] + \text{fft}_{SIMD128}[k + \frac{n}{4}])$
10: $\quad\quad \text{fft}_{odd}[k] \leftarrow \frac{1}{2}\omega^k_{SIMD128} \cdot (\text{fft}_{SIMD128}[k] - \text{fft}_{SIMD128}[k + \frac{n}{4}])$
11: $\quad\quad \omega^k_{SIMD128} \leftarrow \omega^4 \cdot \omega^k_{SIMD128}$
12: $\quad$ **end for**
13: $\quad \text{f}_{even} \leftarrow$ ReverseFFTStep($\text{fft}_{even}, \frac{n}{2}, \omega^2$)
14: $\quad \text{f}_{odd} \leftarrow$ ReverseFFTStep($\text{fft}_{odd}, \frac{n}{2}, \omega^2$)
15: $\quad \text{f}_{SIMD128} \leftarrow$ ReverseEveryOther($\text{f}_{even}, \text{f}_{odd}$)
16: **end if**
17: **return** $\text{f}_{SIMD128}$

Algorithm 12 XFFT(f_X)

1: $\text{f}_{SIMD} \leftarrow$ Convert(f_X)
2: $\text{fft}_{SIMD} \leftarrow$ FFTStep(f_{SIMD}, N, $e^{\frac{i\pi}{N}}$)
3: $\text{fft}_{imag} \leftarrow$ Convert(fft_{SIMD})
4: **return** fft_{imag}

Algorithm 13 XReverseFFT(fft_{imag})

1: $\text{fft}_{SIMD} \leftarrow$ Convert(fft_{imag})
2: $\text{f}_{SIMD} \leftarrow$ ReverseFFTStep(fft_{SIMD}, N, $e^{-\frac{i\pi}{N}}$)
3: $\text{f}_X \leftarrow$ Convert(f_{SIMD})
4: **return** f_X

Algorithm 14 SIMDEncryption(m, t, h)

1: u_0, v_0
2: r $\leftarrow_{random} \{-1, 0, 1\}^N$
3: $\mathrm{fft}_r \leftarrow$ XFFT(r)
4: $\mathrm{fft}_t \leftarrow$ XFFT(t)
5: $\mathrm{fft}_h \leftarrow$ XFFT(h)
6: **for** $i \in \{0, 1, 2, ..., N-1\}$ **do**
7: $\quad \mathrm{u}_0[i] \leftarrow \mathrm{fft}_r[i] \cdot \mathrm{fft}_h[i]$
8: $\quad \mathrm{v}_0[i] \leftarrow \mathrm{fft}_r[i] \cdot \mathrm{fft}_t[i]$
9: **end for**
10: u $\leftarrow$ XReverseFFT(u_0)
11: v $\leftarrow$ XReverseFFT(v_0)
12: **for** $i \in \{0, 1, 2, ..., N-1\}$ **do**
13: $\quad \mathrm{e}_1 \leftarrow_{random} \{-1, 0, 1\}$
14: $\quad \mathrm{e}_2 \leftarrow_{random} \{-1, 0, 1\}$
15: $\quad \mathrm{u}[i] \leftarrow \mathrm{u}[i] + \mathrm{e}_1 \mod q$
16: $\quad \mathrm{v}[i] \leftarrow \mathrm{v}[i] + \mathrm{e}_2 + \frac{q}{2}\mathrm{m}[i] \mod q$
17: **end for**
18: **return** [u,v]

Algorithm 15 SIMDDecryption(u, v, s)

1: m_0
2: $\mathrm{fft}_u \leftarrow$ XFFT(u)
3: $\mathrm{fft}_s \leftarrow$ XFFT(s)
4: **for** $i \in \{0, 1, 2, ..., N-1\}$ **do**
5: $\quad \mathrm{m}_0[i] \leftarrow \mathrm{fft}_u[i] \cdot \mathrm{fft}_s[i]$
6: **end for**
7: $\mathrm{m} \leftarrow$ XReverseFFT(m_0)
8: **for** $i \in \{0, 1, 2, ..., N-1\}$ **do**
9: $\quad \mathrm{x} \leftarrow \mathrm{v}[i] - \mathrm{m}[i] \mod q$
10: $\quad$ **if** $\mathrm{x} < \frac{q}{4}$ **or** $\frac{3q}{4} < \mathrm{x}$ **then**
11: $\quad\quad \mathrm{m}[i] \leftarrow 0$
12: $\quad$ **else**
13: $\quad\quad \mathrm{m}[i] \leftarrow 1$
14: $\quad$ **end if**
15: **end for**
16: **return** m

4.3 NFL implementation

Although not mentioned in the original paper by Ducas et al. [11], in later revised versions and presentations the NFLlib library is mentioned and referred to as the preferred encryption method. However no available code implementing it existed and the documentation for the library is limited. Looking at code of the files included in the library we were able to find available functions and all pre-computed values used in the library. From this it was possible to conclude what functions produced our targeted operation: convolution on polynomials in a cyclic group.

The problem is that the main modulus $q_1 q_2 q_3 ... = Q$ used in CRT is based on the pre-computed values and is therefore static which is not ideal for us. So in order to get results in our cyclic group we had to set Q to be larger than a certain threshold based on our q and N. Calculating this threshold is done by looking at the polynomials used in encryption/decryption, these polynomials have max degree $N-1$ and maximum constant value $q-1$. Now looking at the discrete convolution function,

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m]$$

and fitting this to our encryption/decryption polynomials we get:

$$\max((f * g)[n]) = \sum_{m=0}^{N} (q-1)(q-1) = N(q^2 - 2q + 1) \leq Nq^2 + 1$$

This is the maximum value gained before modular reduction that needs to be supported. For $N = 2048 = 2^{11}$ and $q \approx 2^{27}$ we get the threshold $2^{65} < Q$. We could now perform accurate convolution but we still needed to manage large values which are handled as negative values by the NFLlib library. Therefore an extra check was needed after convolution which reinterpreted negative (large) values by converting $x = -a + Q$ to $y = -a + q$. We now had a working convolution in our intended cyclic group and the rest of the encryption/decryption code could be implemented as normal (Algorithms 16, 17).

About 30% of the NFLlib library is written in assembly without support for ARM, for this reason it was not possible to run the NFL tests directly on the ARM64 hardware.

Algorithm 16 NFLEncryption(m, t, h)

1: $\mathrm{r}_{GMP} \leftarrow_{random} \{-1, 0, 1\}^N$
2: $\mathrm{t}_{GMP} \leftarrow \mathrm{t}$
3: $\mathrm{h}_{GMP} \leftarrow \mathrm{h}$
4: $\mathrm{r}_{NFL} \leftarrow \mathrm{r}_{GMP}$
5: $\mathrm{t}_{NFL} \leftarrow \mathrm{t}_{GMP}$
6: $\mathrm{h}_{NFL} \leftarrow \mathrm{h}_{GMP}$
7: $\mathrm{u}_{NFL} \leftarrow \mathrm{r}_{NFL} * \mathrm{h}_{NFL}$
8: $\mathrm{v}_{NFL} \leftarrow \mathrm{r}_{NFL} * \mathrm{t}_{NFL}$
9: $\mathrm{u}_{GMP} \leftarrow \mathrm{u}_{NFL}$
10: $\mathrm{v}_{GMP} \leftarrow \mathrm{v}_{NFL}$
11: **for** $i \in \{0, 1, 2, ..., N-1\}$ **do**
12: $\quad \mathrm{e}_1 \leftarrow_{random} \{-1, 0, 1\}$
13: $\quad \mathrm{e}_2 \leftarrow_{random} \{-1, 0, 1\}$
14: $\quad \mathrm{u}_{GMP}[i] \leftarrow \mathrm{u}_{GMP}[i] \mod q$
15: $\quad \mathrm{v}_{GMP}[i] \leftarrow \mathrm{v}_{GMP}[i] \mod q$
16: $\quad \mathrm{u}[i] \leftarrow \mathrm{u}_{GMP}[i] + \mathrm{e}_1 \mod q$
17: $\quad \mathrm{v}[i] \leftarrow \mathrm{v}_{GMP}[i] + \mathrm{e}_2 + \frac{q}{2}\mathrm{m}[i] \mod q$
18: **end for**
19: **return** [u,v]

Algorithm 17 NFLDecryption(u, v, s)

```
1: m
2: u_GMP ← u
3: v_GMP ← t
4: s_GMP ← h
5: u_NFL ← u_GMP
6: s_NFL ← s_GMP
7: m_NFL ← u_NFL * s_NFL
8: m_GMP ← m_NFL
9: for i ∈ {0, 1, 2, ..., N − 1} do
10:     m_GMP[i] ← m_GMP[i]  mod q
11:     x← v_GMP[i] − m_GMP[i]  mod q
12:     if x < q/4 or 3q/4 < x then
13:         m[i] ← 0
14:     else
15:         m[i] ← 1
16:     end if
17: end for
18: return m
```

Chapter 5

Results

After each method had been implemented and thoroughly tested the code was run on each platform and benchmarks were performed for different values of N. The values 512, 1024 and 2048 were selected as appropriate values as this spans the security range from acceptable to very high and a random prime on the order of 2^{27} was used as q (as was the case for the original report by Ducas et al. [11]). In this chapter we present the results from these benchmarks in appropriate tables and graphs.

5.1 Master key creation

Table 5.1 shows the time it took to generate the master secret key for various sizes of N on different hardware. When N is increased so is the time it takes to create the key. The same standard code was used on all three platforms for this measurement and as can be seen by the results the ARM64 processor was slowest while the x86 platforms performed quite similar to each other. The table is plotted in figure 5.1 which shows the non-linearity of the procedure.

Platform	N	Time
ARM64	512	4.36
ARM64	1024	22.85
ARM64	2048	152.23
x86-1	512	1.79
x86-1	1024	8.91
x86-1	2048	46.57
x86-2	512	2.44
x86-2	1024	11.89
x86-2	2048	58.78

Table 5.1: Times in s for Master Key generation on the different platforms.

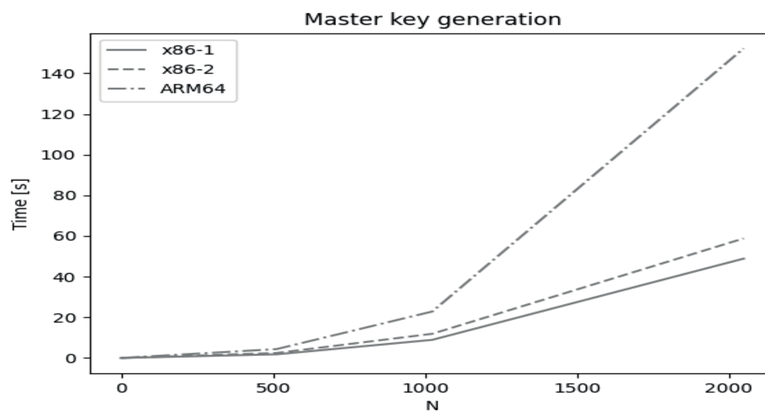


Figure 5.1: Median master key generation time plotted for all platforms.

5.2 User key generation, encryption and decryption

5.2.1 x86

Table 5.2 contains the times it took on the desktop computer (x86-1) for generating user keys, encrypt and decrypt. The standard implementation is faster than all optimized implementations when it comes to generating user keys as seen in figure 5.2a. The standard and the AVX2 implementation performs similarly but the CUDA implementation is far worse. However the encryption and decryption is faster on the optimized versions and in table 5.3 we can see how many times faster the different algorithms

were compared to the standard implementation. On all optimization variants the speed up increased when N increased for decryption and encryption, this can be seen in figure 5.3a and 5.4a.

The results from the virtual machine (x86-2) are shown in table 5.4 and the speedups are shown in table 5.5. Like the desktop computer the standard implementation had the fastest user key generation but the SIMD optimization is very close because the speedup is close to one, this can also be seen in 5.2b where Standard and SIMD are plotted very close together. Encryption and decryption are faster on all three of the optimizations. The largest speedup is the same for the virtual machine and the desktop computer and it is the SIMD optimization for N = 2048. Looking at figure 5.3b and 5.4b we can see that the speedup increases as N increases.

Type	**N**	**SK Gen**	**Encryption**	**Decryption**
Standard	512	7.29	0.57	0.27
Standard	1024	24.90	2.23	1.06
Standard	2048	93.10	8.22	3.99
SIMD	512	7.34	0.14	0.04
SIMD	1024	26.12	0.29	0.09
SIMD	2048	106.47	0.62	0.19
GPU	512	497.54	0.92	0.49
GPU	1024	1035.29	1.18	0.57
GPU	2048	2015.83	1.92	0.88
NFL	512	-	0.40	0.19
NFL	1024	-	0.80	0.40
NFL	2048	-	1.65	0.84

Table 5.2: Times in ms for different implementations on x86-1.

Type	N	SK Gen	Encryption	Decryption
Standard	512	1	1	1
Standard	1024	1	1	1
Standard	2048	1	1	1
SIMD	512	0.99	4.15	6.61
SIMD	1024	0.95	7.70	11.90
SIMD	2048	0.87	13.24	21.00
GPU	512	0.02	0.63	0.55
GPU	1024	0.02	1.90	1.87
GPU	2048	0.05	4.28	4.51
NFL	512	-	1.44	1.43
NFL	1024	-	2.81	2.68
NFL	2048	-	5.00	4.74

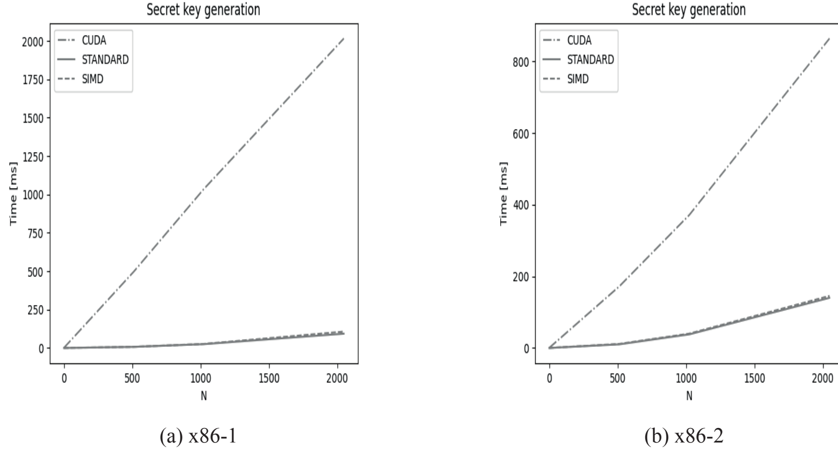
Table 5.3: Speedup relative to the standard run with the same N (standard time/new time) on x86-1.

Type	N	SK Gen	Encryption	Decryption
Standard	512	10.14	0.67	0.28
Standard	1024	38.04	2.93	1.20
Standard	2048	140.02	12.18	5.85
SIMD	512	11.22	0.22	0.06
SIMD	1024	39.96	0.45	0.13
SIMD	2048	145.31	0.94	0.28
GPU	512	171.70	0.49	0.24
GPU	1024	371.34	1.02	0.45
GPU	2048	864.02	1.86	0.75
NFL	512	-	0.57	0.28
NFL	1024	-	1.16	0.57
NFL	2048	-	2.42	1.19

Table 5.4: Times in ms for different implementations on x86-2.

Type	N	SK Gen	Encryption	Decryption
Standard	512	1	1	1
Standard	1024	1	1	1
Standard	2048	1	1	1
SIMD	512	0.90	3.09	4.73
SIMD	1024	0.95	6.51	9.28
SIMD	2048	0.96	12.95	21.27
GPU	512	0.06	1.37	1.20
GPU	1024	0.10	2.88	2.68
GPU	2048	0.16	6.57	7.79
NFL	512	-	1.17	1.022
NFL	1024	-	2.53	2.09
NFL	2048	-	5.04	4.93

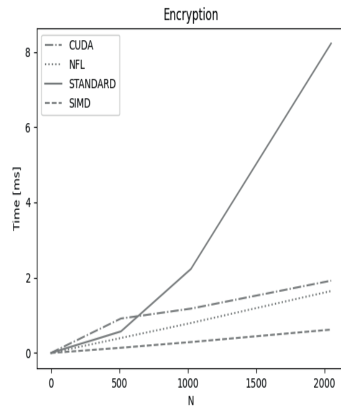
Table 5.5: Speedup relative to the standard run with the same N (standard time/new time) on x86-2.



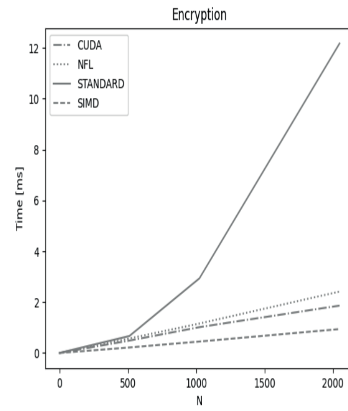
(a) x86-1

(b) x86-2

Figure 5.2: Median user key generation time on the x86-1 and x86-2 platforms for the standard, CUDA and SIMD (AVX2) implementations.

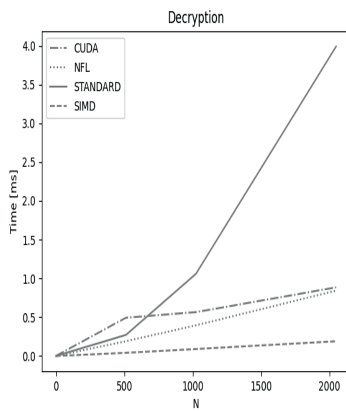


(a) x86-1

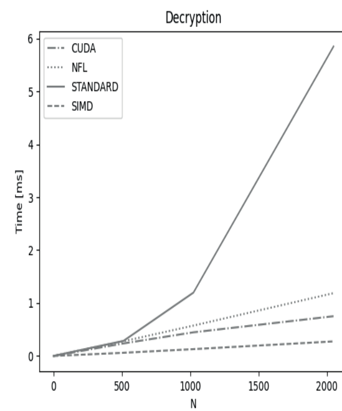


(b) x86-2

Figure 5.3: Median encryption time on the x86-1 and x86-2 platforms for the standard, CUDA, NFLlib and SIMD (AVX2) implementations.



(a) x86-1



(b) x86-2

Figure 5.4: Median decryption time on the x86-1 and x86-2 platforms for the standard, CUDA, NFLlib and SIMD (AVX2) implementations.

5.2.2 ARM

On the ARM64, the only available optimization was using Neon SIMD instructions and our results show that user key generation, encryption and decryption all became faster,

as shown in tables 5.6 and 5.7. The speedup is also dependent on N which can be seen in figures 5.5, 5.6 and 5.7. The speedup with these SIMD instructions are almost the same regardless of the value of N for all three algorithms, with the user key generation speedup at 1.5, the encryption speedup at 2.1 and the decryption speedup at 2.2.

Type	N	SK Gen	Encryption	Decryption
Standard	512	149.24	7.51	3.79
Standard	1024	559.20	17.44	8.82
Standard	2048	2134.56	42.40	21.26
SIMD	512	97.46	3.59	1.74
SIMD	1024	365.58	8.19	3.95
SIMD	2048	1363.90	17.71	8.58

Table 5.6: Times in ms for different implementations run on ARM64.

Type	N	SK Gen	Encryption	Decryption
Standard	512	1	1	1
Standard	1024	1	1	1
Standard	2048	1	1	1
SIMD	512	1.53	2.09	2.18
SIMD	1024	1.53	2.13	2.24
SIMD	2048	1.57	2.40	2.48

Table 5.7: Speedup relative to the standard run with the same N (standard time/new time) on ARM64.

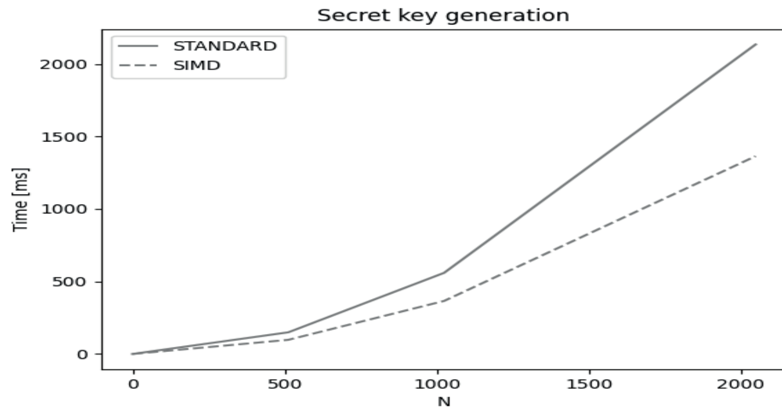


Figure 5.5: Median user key generation time using the standard and SIMD (Neon) implementations on the ARM64 platform.

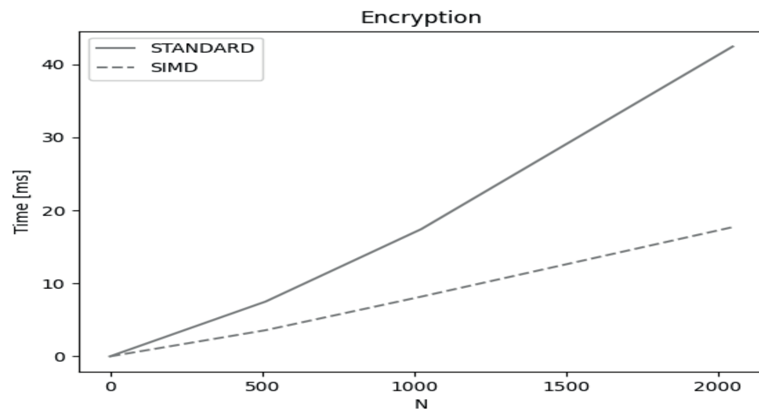


Figure 5.6: Median encryption time using the standard and SIMD (Neon) implementations on the ARM64 platform.

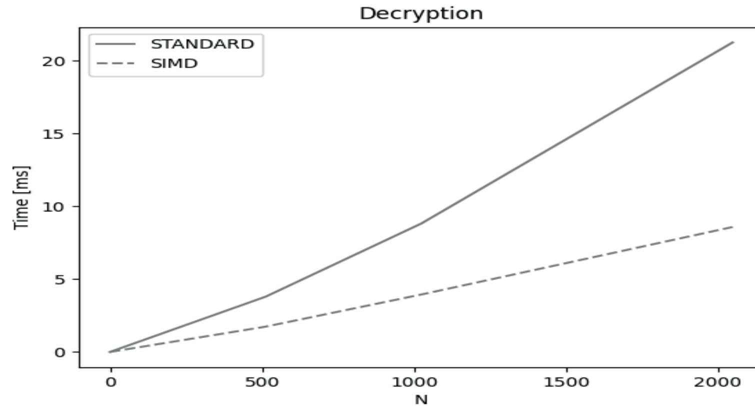


Figure 5.7: Median decryption time using the standard and SIMD (Neon) implementations on the ARM64 platform.

5.3 Distributions

Our previous results use the median value obtained from the benchmarks but in order to gain a deeper understanding of how the implementations perform we also present the distribution of these values. Since the implementations on the two x86 platforms are the same, only distribution figures from the x86-2 runs are presented here and for similar reasons only runs for N=1024 are filtered.

Looking at all encryption and decryption graphs one important note needs to be considered. All implementations had a number of outliers outside of the graph area, some even as high as 10 times the value of their median. Including these in the plot made the graphs unreadable and for this reason the graphs were trimmed around the median results.

5.3.1 x86

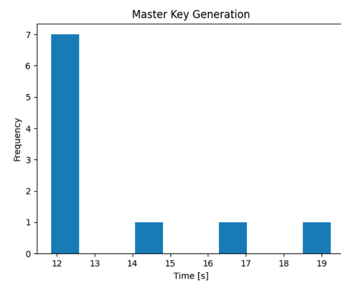
Figure 5.8a shows the distribution of the times it took to generate the master secret key on the x86-2 platform for N=1024. Most of the generations took about the same time as the median, 11.9 seconds, but it could be argued that the sample size is too small for proper interpretation.

Looking instead at user key generation in figures 5.8b, 5.9a, and 5.9b they all have

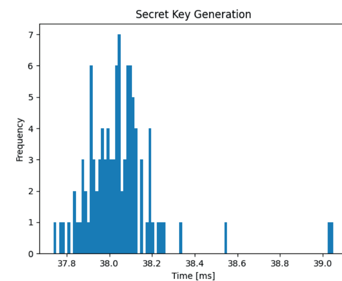
the look of an almost normal distribution. For N=1024 with the standard method on the x86-2 platform most of the generations were in a 0.4 ms interval between 37.8 ms and 38.2 ms. The AVX2 SIMD optimizations also had most of its generations within the median ± 0.2 ms while the GPU optimization had a larger span of about 4 ms with more outliers, still on the x86-2 with N=1024.

The encryption times on the x86-2 with different methods are shown in figures 5.10a, 5.10b, 5.11a and 5.11b. The three first mentioned distributions are similar and with an early peak, indicating that most results are close to optimal and slowly decreases after the peak. Distribution graph 5.11b also has an early peak but is more skewed towards the center giving more of a normal distribution. All four graphs show a clear peak and narrow range which is the ideal look for a well performing encryption implementation.

Decryption times for the x86-2 platform are shown in 5.12a, 5.12b, 5.13a and 5.13b. Decryption is slightly less computational expensive than encryption and all decryptions showed similar distribution to their encryption counterpart with the same or narrower spread. The most noteworthy distribution is that of figure 5.12b which has an extreme peak at the very beginning indicating very good performance.

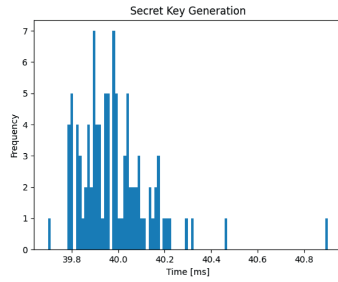


(a) Distribution of 10 Master Key generations on platform x86-2 for N=1024.

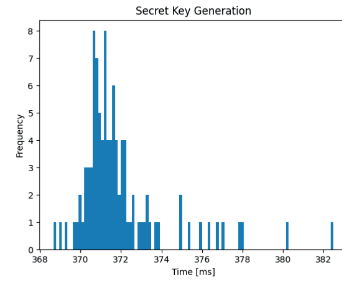


(b) Distribution of 100 user key generations on platform x86-2 using Standard method for N=1024.

Figure 5.8: Master secret key and user key generations.

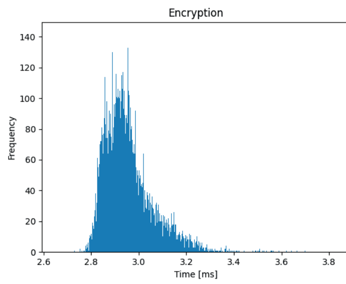


(a) Distribution of 100 user key generations on platform x86-2 using AVX2 method for N=1024.

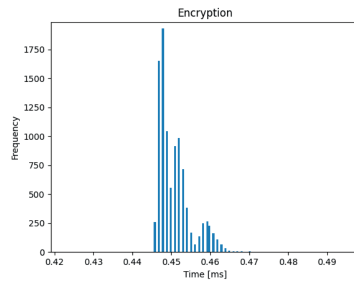


(b) Distribution of 100 user key generations on platform x86-2 using GPU method for N=1024.

Figure 5.9: User key generations.

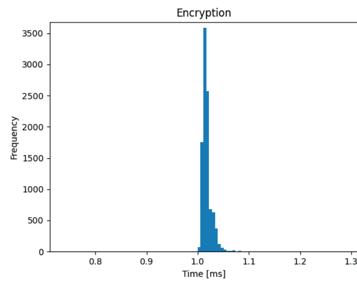


(a) Distribution of 10000 encryptions on platform x86-2 using Standard method for N=1024.

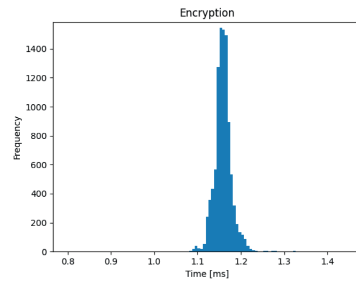


(b) Distribution of 10000 encryptions on platform x86-2 using AVX2 method for N=1024.

Figure 5.10: Encryptions on platform x86-2 using Standard and AVX2 methods on two different charts.

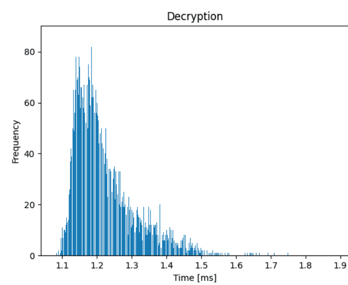


(a) Distribution of 10000 encryptions on platform x86-2 using GPU method for N=1024.

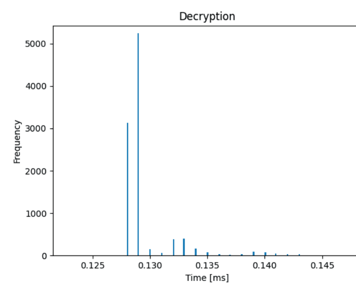


(b) Distribution of 10000 encryptions on platform x86-2 using NFL method for N=1024.

Figure 5.11: Encryption times on platform x86-2 using GPU and NFL methods on two different charts.

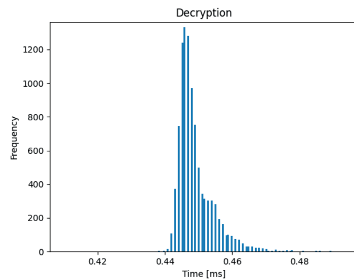


(a) Distribution of 10000 decryptions on platform x86-2 using Standard method for N=1024.

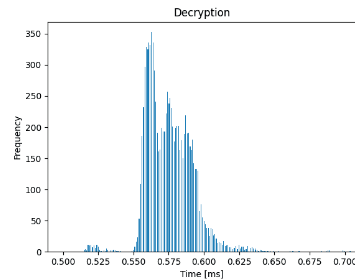


(b) Distribution of 10000 decryptions on platform x86-2 using AVX2 method for N=1024.

Figure 5.12: Decryption times on platform x86-2 using Standard and AVX2 methods on two different charts.



(a) Distribution of 10000 decryptions on platform x86-2 using GPU method for N=1024.



(b) Distribution of 10000 decryptions on platform x86-2 using NFL method for N=1024.

Figure 5.13: Decryption times on platform x86-2 using GPU and NFL methods on two different charts.

5.3.2 ARM64

Taking a look at master secret key generation on ARM64 shown in figure 5.14 we see that most of the generations were also close to the median 22.8 seconds but almost as many were closer to 27 seconds. A better representative value would therefor have been somewhere in between, around 25 seconds. Again the sample size could be argued to be too small.

In figures 5.15a and 5.15b the distribution of user key generations is shown and like on x86-2 they too look normal distributed except for a few outliers. When it comes to the range of the distributions, most of the generation times lies within 3 ms intervals and even wider if single values are included.

The distributions in 5.16a and 5.16b are encryption times from the ARM64 with the standard and Neon SIMD methods. Most of the times for the standard and SIMD methods lies within a 0.15 ms interval but there appears to be a slightly larger spread for the Neon method even though the rate between fast results and most common results appear to be similar.

The distribution of decryption for the ARM64 can be seen in 5.17a and 5.17b. For the Neon method most decryptions where in the extremely narrow range of between 3.94 ms and 3.95 ms. Meanwhile the standard decryption method on the ARM64 had most of the decryptions in the range between 8.8 ms and 8.7 ms, a slightly larger in-

terval. As for the similarities between encryption and decryption shapes, the standard methods both have close to a normal distribution and so does Neon decryption while Neon encryption stands out on its own with the least normal looking distribution.

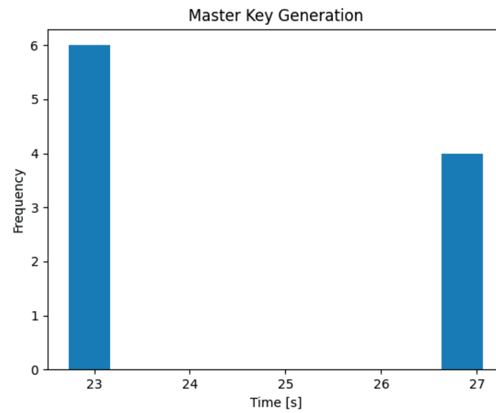
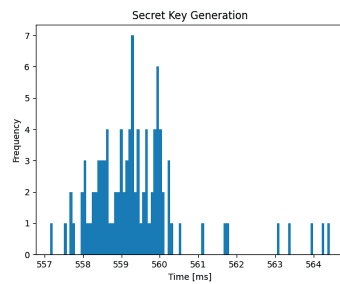
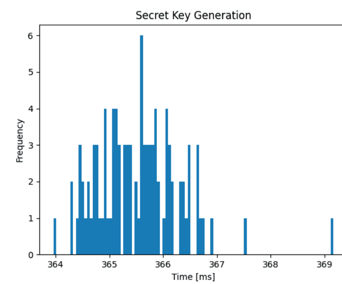


Figure 5.14: Distribution of 10 Master Key generations on platform ARM64 for N=1024.

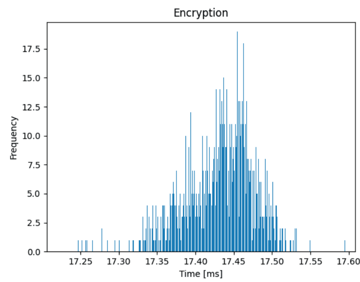


(a) Distribution of 100 user key generations on platform ARM64 using Standard method for N=1024.

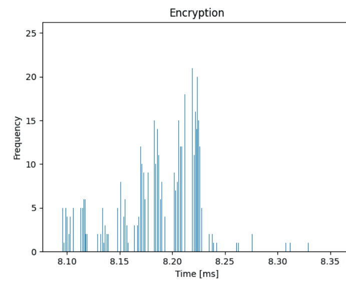


(b) Distribution of 100 user key generations on platform ARM64 using Neon method for N=1024.

Figure 5.15: User key generations.

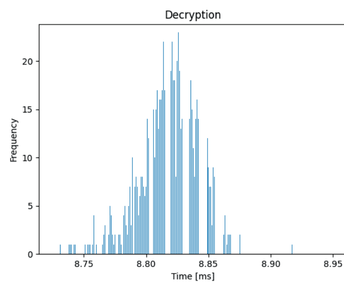


(a) Distribution of 1000 encryptions on platform ARM64 using Standard method for N=1024.

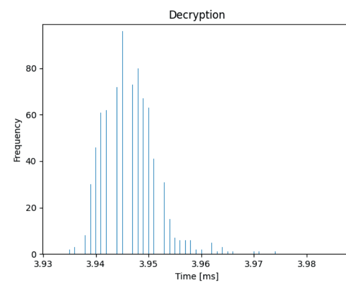


(b) Distribution of 1000 encryptions on platform ARM64 using Neon method for N=1024.

Figure 5.16: Distribution on 1000 encryptions on platform ARM64 using Standard and Neon methods on two different charts.



(a) Distribution of 1000 decryptions on platform ARM64 using Standard method for N=1024.



(b) Distribution of 1000 decryptions on platform ARM64 using Neon method for N=1024.

Figure 5.17: Distribution on 1000 decryptions on platform ARM64 using Standard and Neon methods on two different charts.

Chapter 6

Discussion

6.1 Measurements

Results of the previous chapter showed that using CUDA to try and optimize the implementation was not as good as we had initially hoped. Again looking at user key generation in graphs 5.2a and 5.2b we see how badly the GPU performs on a task that is only semi-parallel such as dot-product. We suspect that the low efficiency from the CUDA dot-product implementation stems from all the data copying between CPU and GPU that is required before and after calling a kernel. The arguments x1 and x2 in algorithm 1 has the length 2N and needs to be filled and fetched between each kernel call and although the product part of dot-product can be performed in parallel, the summation needs to be done sequentially which is not optimal for CUDA code. An indication that this is the case is the linearity in graph 5.2a and 5.2b.

As for the encryption and decryption using CUDA the performance boost is better and even outperforms the NFL implementation on the x86-2 tests. With more computations performed in parallel during both encryption and decryption it was expected that the CUDA implementation would perform well, making it possible to convert the $O(N\log(N))$ runtime of using FFT for convolution into $O(N)$ by using vector-matrix multiplication in parallel instead. This small difference did not appear to make a big enough difference to outperform FFT using AVX2. The drawbacks are most likely the same as for dot-product, copying between memory in algorithms 4 and 5 takes time. Since the encryption algorithm is less sequentially dependent, it could perhaps be possible to change more of Prest's original code to run more on the GPU side, avoid some data copying and gain slightly better results.

Prest's performance boosts of using the NFLlib library for encryption and decryption was stated to be up to 10x. Our results did not achieve this and it is unclear how it could have been achieved as our maximum boost was closer to 5x. However during implementation, tests were performed using the pre-computed CRT value of $q_1 q_2 q_3 \cdots = Q$ and not the intended q. Writing the encryption algorithm this way allowed us to ignore the conversions to mod q discussed in section 4.3 and the difference between standard and NFL became a factor close to 10. The problem is that the value q becomes static doing it this way and in order to change it, some or all pre-computed values in the NFL library needs to be recomputed and set but finding how and which was not possible for us to find even when looking at the available code for the library. Doing it this way also removes the customizability implementing in software provides. Even if we did not gain the 10x performance boost we had hoped for, NFL did perform as well and even better than our CUDA implementation which requires dedicated GPU hardware.

Another surprising result was the out-performance of the AVX2 dot-product implementation by the Neon implementation. The performance of standard user key generation and AVX2 generation were about the same or worse while Neon user key generation had a boost of around 50%. The expected outcome was that the boost from AVX2 was supposed to be about twice the boost of Neon since AVX2 supports a SIMD vector size twice the size of that of the Neon vector size. Comparing the AVX2 encryption/decryption results to the Neon encryption/decryption results are more in line with our expectations. By comparing the results from tables 5.3, 5.5 and 5.7 we see that there is no single factor that can be used to show the difference, this is because the difference is non-linear and most likely dependent on N and the FFT runtime $O(N \log(N))$ becoming $\frac{N}{2}\log(\frac{N}{2})$ for Neon and $\frac{N}{4}\log(\frac{N}{4})$ for AVX2.

Also visible in graphs 5.3a, 5.3b and 5.6 is how dependent the speed-ups are on N. In short, larger sizes of N benefits more by the optimizations so increasing N even further to 4096, 8192, 16384 and so on would benefit more and more. The problem is that N > 2048 is not necessary for security reasons today and the loss in overall performance makes it less desirable even if overall security increases with larger N. Another thing previously not discussed is how the overall data rate is also dependent on N. On ARM64 with N=512 the data rate using standard encryption is

$$\frac{1}{7.51 \cdot 10^{-3}} \cdot 512 = 133 \cdot 512 \approx 68\text{Kbps}$$

and

$$\frac{1}{42.4 \cdot 10^{-3}} \cdot 2048 = 23.6 \cdot 2048 \approx 48\text{Kbps}$$

for N=2048. For Neon encryption these rates become

$$\frac{1}{3.59 \cdot 10^{-3}} \cdot 512 = 279 \cdot 512 \approx 143\text{Kbps}$$

and

$$\frac{1}{17.71 \cdot 10^{-3}} \cdot 2048 = 56 \cdot 2048 \approx 116\text{Kbps}$$

respectively. This shows the relative boost in data rates when N increases. In the standard implementation we have a 29% loss in data rate using N=2048 compared to using N=512, this same number is only 19% using Neon and although the difference is small it is still worth noting.

If we compare the results to the most widely used PKI algorithm used today; RSA, our measured speeds are relatively good. RSA also drastically decreases performance as key sizes increase, however for RSA performance limitations occurs during the decryption phase. Taking a recommended key size of 4096 bits for RSA, the decryption time takes hundreds to thousands of milliseconds depending on the hardware. Comparing this to table 5.6 we see that the longer of our two phases, encryption, only takes 29 to 35 milliseconds for the same amount of data on low performance hardware.

Still the overall data rates on ARM64, even with Neon optimization, 143 Kbps is modest for modern applications. For these cases it would be recommended to use this IBE scheme as the initial procedure of a key exchange before switching over to a faster, symmetric, post-quantum secure, algorithm such as AES. This is usually already how it is done with PKI and RSA today. Comparing AES to the IBE scheme is hard because encrypting 256 bits (largest key size of AES) would take the same amount of time as the times presented in figure 5.6, since encryption works on entire messages and 256 bits is smaller than a single message in all cases.

6.2 Comparison with PKI setup

6.2.1 Architectural differences

A public key infrastructure often contains a CA, an RA and a certificate storage to handle the certificates needed whilst an identity-based setup handles most of this with the

TA (trusted authority). The TA is accountable for creating and handing out all keys and this can take place immediately when a network is created. This could decrease the load on the client side because the clients only need to know the master public key, its user key and the IDs of the other clients. Compared with a PKI where the clients must not only have its decryption key but also a CA trust store. The trust store is necessary to be able to check certificates to find out if they should be trusted, meaning it is signed by a trusted CA, and has not been revoked or expired.

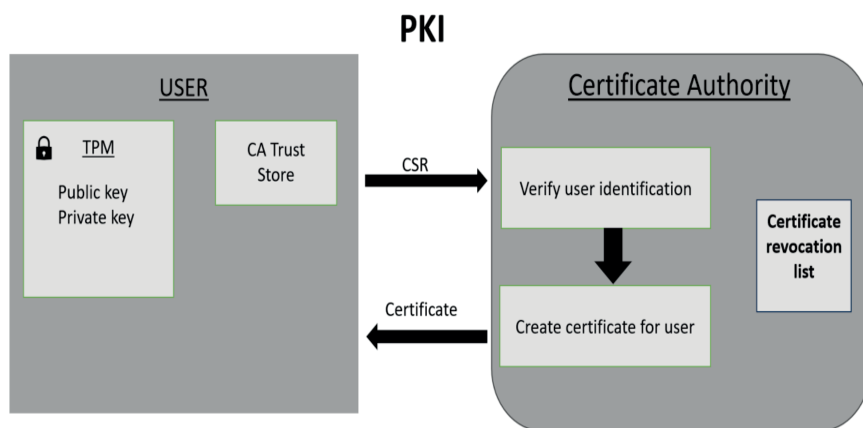


Figure 6.1: Certificate request and management for a user in PKI. The user sends a certificate signing request to the certificate authority and gets a certificate in return.

An identity-based system needs a trusted third party that generates the keys needed. In a PKI the users could actually create their own keys but they would still need certificates and for that a third party is needed, it is often a CA. This means both systems need some sort of third party but the key generation can be done differently.

Since an identity-based system does not need a certificate storage it can be easier to scale up. To add new users the TA simply generates a user key, gives it to the user and then the TA could easily broadcast the ID of the new user in plain-text to the other users since the ID is not a secret. An identity-based system with many users also avoids the overhead for managing and storing all the certificates needed, compared to a PKI where the certificate storage must be able to store all certificates and keep a revocation list updated. As can be seen in figure 6.2 the TA in the IBE scheme does not need to store

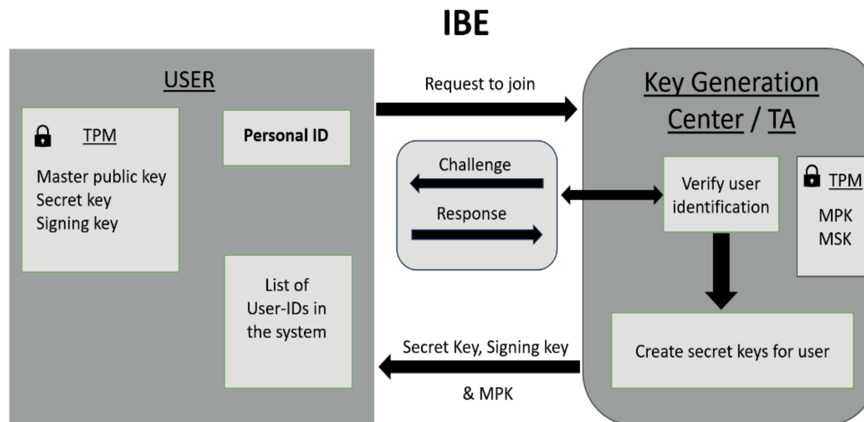


Figure 6.2: Identity-based encryption scheme setup. The model shows a user requesting the Key Generation Center / Trusted Authority to join. The figure also shows what both entities have and the steps being taken during a request to join.

much more than the keys.

6.2.2 Identification

In a public key infrastructure the CA generates certificates and makes sure a certificate is bound to the correct key. The certificates contain a digital signature of a third party that has verified the content. If one, verifying the certificate chain, trusts this third party that has signed it, the public key is clear to use. Meanwhile in an identity-based setup the key acts as an identifier because it is created with publicly identifiable information of that user [27]. Instead of searching and checking certificates for a public key that is valid, the users in an identity-based system encrypts messages with a master public key combined with the recipient's ID.

Certificates are used to create trust between users and in systems it is often desired to have mutual trust, meaning that the sender and receiver can trust each other. In figure 6.1 it is shown how a user in a PKI sends a CSR (Certificate Signing Request) to the CA and how the CA verifies the user and later creates and sends a certificate back. A CSR is how a user applies for a digital certificate from a certificate authority in a public key infrastructure.

6.2.3 Key generation and management

A client can choose to generate their own key pair and let the CA verify it or to let the CA generate the key pair from scratch in a public key cryptosystem. There are scenarios where it could be better to create the keys at the client side and some scenarios where there is an advantage to create the keys at the CA. The ability to recover the encrypted information is easier if a CA generates the keys because the CA can store a backup. If the keys are generated by the client it becomes clear that there only exists one such key pair and this would support non-repudiation and authenticity.

In an identity-based cryptosystem it is a trusted authority that acts as a key generation center that creates all the keys. Then the users create the public keys by combining a master public key with the ID of the recipient and in some cases some other information, for instance the date. In a PKI one public and one private key is generated for each user while in an identity-based system everyone gets the same master public key and one unique secret decryption key, so there are fewer unique keys in an identity-based cryptosystem. Both in IBE and PKI it is reasonable to use a TPM (Trusted Platform Module) to safely keep the keys. The TPM is a standard for crypto processors to handle keys and it provides confidentiality and integrity by not letting anyone unauthorized access the keys.

Traditional public key infrastructure keeps a certificate revocation list in the certificate storage to keep up with expired and revoked keys. In some systems a certificate revocation list could be hard to manage, one solution for this is to implement a short lifespan for the certificates, meaning that new keys should be created frequently. In an identity-based solution this would be more efficient because there are no certificates to create and revoke but also one could use the date as an input with the ID and the master public key. This would force all users to fetch a new user key from the generator every new time period, by doing this the system could have regular key updates. There are multiple reasons to use more input than just identity for encryption, beyond regular key updates. If a key should be revoked one cannot simply revoke a person's identity, therefore it is a good idea to use something more than just the ID if the ID is very troublesome to change (phone number or email address).

One problem with updating keys is that the new keys must be sent using a secure channel. For instance if the old keys are leaked the old channel would be unavailable. Another reason the previous channel should not be used is if one key is broken, all keys

after will also be available. So distributing new keys in a secure way is a problem for both PKI and identity-based systems.

6.2.4 Differences in maintaining CIA

Since public key infrastructure has been around for a long time there are existing techniques for how the CIA triad is fulfilled, for instance using TPMs and certificates. However, there are things that need to be addressed in an identity-based encryption scheme to maintain the CIA triad. Users in an IBE scheme can keep the keys secure in a TPM like they are in a public key infrastructure. This gives confidentiality and integrity to the keys because only the TA and the users have access to the keys. An important difference between IBE and PKI is that the TA in an IBE scheme has access to all keys, because it has the master secret key, and can therefore decrypt everything sent. A PKI can on the other hand choose to let the users create the keys and then only let the CA sign the certificates so no one but the user has access to the keys. This is still not a problem for the IBE scheme because the security at this TA should be a higher than the security at the users and therefore it is not easier to access the keys from the TA, the consequences on the other hand will be a lot greater if the TA is compromised.

Both a PKI and an IBE scheme can use signatures on their messages, it is done differently but the outcome is the same. The differences are that a public key infrastructure needs a new key pair for each user and the required certificate for the public key. In an identity-based encryption scheme the users only need one more key for signing and the TA must be able to generate this signature creation key that is also identification based. Signatures gives the receiver the possibility to verify the authenticity of the message and also to prove that the message came from a known sender.

6.3 IBE in practice

PKI is currently the most used scheme but there are some areas where identity-based encryption has an edge.

6.3.1 Internal vehicle communication

Inside a vehicle, multiple entities want to communicate with each other encrypted. A case like this is suitable for identity-based encryption because the entities know each other and therefore can use IDs instead of keeping track of certificates.

The modern cars of today consist of multiple computers or more specifically Electronic Control Units (ECU) handling different tasks. To handle internal communication between the ECUs in the vehicle, identity-based encryption could be an option. There is usually one ECU stronger than the others and that one can act as the TA. Generating keys will take some time but as our results show it is possible on an ARM64. This would also enable the possibility to have regular key updates because the TA is in the system itself. Key updates with traditional PKI would mean that each ECU has to create a new key pair and send a certificate signing request to a certificate authority. This would also occur in the production line to get the first key pair and the accompanying certificate. Changing PKI to IBE would reduce the amount of certificates each ECU would have to handle and a CA and CSRs would not be necessary.

6.3.2 Email system

An email system using an identity-based encryption and signing scheme would resemble an ordinary mail system. The sender only needs the ID of the recipient to send an encrypted message. The recipient can verify the signature of the sender with a verification algorithm that takes the sender's ID as input and also decrypts the message with the user key.

Companies could easily implement such a solution for their employees to handle internal communication and in this setup the company itself works as the TA. When a new employee joins the company they get two secret keys based on an id, this ID could be some combination of name, work title or phone number. The two keys are needed for encryption and signing. They also need the encryption, decryption, signing and verification algorithms but those are handled by the email system. To send a message in this identity-based system the sender signs the message with its own identity and encrypts it with the receiver's identity and then the message is ready to be sent. The receiver's job is also easy because the decryption is done with its own identity and to verify the signature the sender's identity is used. In a company everyone knows the identity of the others so there is no extra time spent searching for public keys.

If the company wants to add frequent key updates a solution could be to include the date in the ID used for encryption and signing. This would force the users to fetch the new keys every time they get updated. Updating the keys every week would mean that if someone got hold of the master secret key they could only decrypt the messages from

that week and if the security requirements are higher the key updated can be more frequent. The email system must also be able to handle if an employee quits or someone should not be a part of the system anymore and therefore the TA/company must be able to revoke keys. If the keys are updated frequently in the system the TA simply stops issuing new keys to the employee that should have his keys revoked. If the system does not have frequent key updates then it would be suitable to do a key update and give new keys to all employees except the one who should not have new keys.

6.3.3 Updating and distributing keys

When the TA generates a new master secret key it must also generate new private secret keys and a master public key and somehow distribute these to the users in the encryption scheme. The difficulty of this problem depends on the system the encryption scheme is used in. For an email system inside a company the distribution could be solved easily because they share the same network, some secure file transfer protocol would be enough. If the TA and the users do not share the same network this problem becomes trickier. Some sort of key exchange is necessary to be able to update keys regularly and keep the keys secure.

Chapter 7

Summary and outlook

7.1 Results

As has previously been discussed, an area that could benefit from IBE is IoT devices such as those in modern vehicles. It was for this reason important to see how an IBE implementation would perform on such hardware. Seeing as our tests and optimization showed promising results on an ARM processor, this should be seen as an optimistic outcome.

Our results also showed that using SIMD for optimising the scheme proved easy to implement and surprisingly beneficial. This observation should not limit itself to this specific scheme but rather all polynomial based encryption schemes as operations on polynomials can usually be represented as list-wise operations. For this reason we recommend that SIMD should be considered for all implementations of lattice based encryption schemes. Fully parallel LWE with dedicated hardware is an option to increase speeds. However, using CUDA to use both the CPU and GPU is not recommended since expensive, high end GPUs did not outperform the simpler and built-in SIMD hardware.

Our implementations is on par or even outperforms the commonly used RSA algorithm when it comes to the sum of the encryption and decryption process but has much larger key sizes (multiple kilobytes). RSA decryption is much slower than encryption which negatively affects servers in a client-server setup, as it is usually the servers task to decrypt and clients task to encrypt. This makes servers susceptible to CPU burn attacks while encryption and decryption is more similar on this NTRU IBE which makes it less vulnerable to such attacks.

7.2 Future work

It would be interesting to convert and test even more of the original code into SIMD. Having gone through the entire code there is not much to gain by further trying to optimize encryption or decryption with SIMD, but both master secret key generation and user key generation could be optimized further by updating more than DotProduct. Using AVX-512 for x86 architecture and SVE for ARM architecture could also be worth investigating further, as well as updating the NFLlib to use Neon ARM instructions to see its performance on slower hardware.

7.3 Outlook

Quantum computers are not here yet and it is not certain there ever will exist working quantum computers capable of breaking today's cryptographic schemes. However, if quantum computers ever become an option it would be wise to be prepared and have post-quantum encryption algorithms ready and preferably in use. Because of the Store Now, Decrypt Later attacks and the fact that memory is cheap nowadays implementing hybrid schemes is something worth looking into. Hybrid schemes are also recommended by the Bundesamt für Sicherheit in der Informationstechnik (BSI). The National Institute of Standards and Technology (NIST) is also on the way towards quantum secure cryptography with standardized algorithms that can withstand attacks from Shor's algorithm implemented on quantum computers.

Bibliography

[1] Executive summary — NIST SP 1800-25 documentation. https://www.nccoe.nist.gov/publication/1800-25/VolA/index.html. Accessed: 2024-4-18.

[2] Quantum-safe cryptography ‑fundamentals, current developments and recommendations. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Brochure/quantum-safe-cryptography.pdf?__blob=publicationFile&v=6. Accessed: 2024-4-25.

[3] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. Xpir: Private information retrieval for everyone. Cryptology ePrint Archive, Paper 2014/1025, 2014. https://eprint.iacr.org/2014/1025.

[4] Martin R Albrecht, Miloš Prokop, Yixin Shen, and Petros Wallden. Variational quantum solutions to the shortest vector problem. *Quantum*, 7:933, 2023.

[5] Hossein Amiri and Asadollah Shahbahrami. Simd programming using intel vector extensions. *Journal of Parallel and Distributed Computing*, 135:83–100, 2020.

[6] Ahmed M Bedewy, Yin Sun, and Ness B Shroff. Optimizing data freshness, throughput, and delay in multi-server information-update systems. In *2016 IEEE International Symposium on Information Theory (ISIT)*, pages 2569–2573. IEEE, 2016.

[7] Nina Bindel, Jacqueline Brendel, Marc Fischlin, Brian Goncalves, and Douglas Stebila. Hybrid key encapsulation mechanisms and authenticated key exchange. In *Post-Quantum Cryptography: 10th International Conference, PQCrypto 2019, Chongqing, China, May 8–10, 2019 Revised Selected Papers 10*, pages 206–226. Springer, 2019.

[8] Dan Boneh and Matt Franklin. Identity-based encryption from the weil pairing. In *Annual international cryptology conference*, pages 213–229. Springer, 2001.

[9] Shane Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[10] Jintai Ding, Xiang Xie, and Xiaodong Lin. A simple provably secure key exchange scheme based on the learning with errors problem. Cryptology ePrint Archive, Paper 2012/688, 2012. `https://eprint.iacr.org/2012/688`.

[11] Léo Ducas, Vadim Lyubashevsky, and Thomas Prest. Efficient identity-based encryption over ntru lattices. In *Advances in Cryptology–ASIACRYPT 2014: 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, ROC, December 7-11, 2014, Proceedings, Part II 20*, pages 22–41. Springer, 2014.

[12] Pierre Fortin, Ambroise Fleury, François Lemaire, and Michael Monagan. High-performance simd modular arithmetic for polynomial evaluation. *Concurrency and Computation: Practice and Experience*, 33(16):e6270, 2021.

[13] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 197–206, 2008.

[14] Martin Hellman and Whitfield Diffie. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[15] Jeffrey Hoffstein, Nick Howgrave-Graham, Jill Pipher, Joseph H. Silverman, and William Whyte. Ntrusign: Digital signatures using the ntru lattice. In Marc Joye, editor, *Topics in Cryptology — CT-RSA 2003*, pages 122–140, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[16] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. Ntru: A ring-based public key cryptosystem. In *International algorithmic number theory symposium*, pages 267–288. Springer, 1998.

[17] David Joseph, Rafael Misoczki, Marc Manzano, Joe Tricot, Fernando Dominguez Pinuaga, Olivier Lacombe, Stefan Leichenauer, Jack Hidary, Phil Venables, and Royal Hansen. Transitioning organizations to post-quantum cryptography. *Nature*, 605(7909):237–243, 2022.

[18] Ravneet Kaur and Amandeep Kaur. Digital signature. In *2012 International Conference on Computing Sciences*, pages 295–301. IEEE, 2012.

[19] Aqeel Sahi Khader and David Lai. Preventing man-in-the-middle attack in diffie-hellman key exchange protocol. In *2015 22nd international conference on telecommunications (ICT)*, pages 204–208. IEEE, 2015.

[20] Dinesh Khattar and Neha Agrawal. *Rings*, pages 185–190. Springer International Publishing, Cham, 2023.

[21] Markus Krausz, Georg Land, Jan Richter-Brockmann, and Tim Güneysu. Efficiently masking polynomial inversion at arbitrary order. In *International Conference on Post-Quantum Cryptography*, pages 309–326. Springer, 2022.

[22] Arjen K Lenstra, Hendrik Willem Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische annalen*, 261(ARTICLE):515–534, 1982.

[23] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *Journal of the ACM (JACM)*, 60(6):1–35, 2013.

[24] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for ring-lwe cryptography. In *Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings 32*, pages 35–54. Springer, 2013.

[25] R Merkle. Secure communications over an insecure channel. submitted to. *Comm. ACM*.

[26] Daniele Micciancio. The shortest vector in a lattice is hard to approximate to within some constant. *SIAM journal on Computing*, 30(6):2008–2035, 2001.

[27] Kenneth G Paterson and Geraint Price. A comparison between traditional public key infrastructures and identity-based cryptography. *information security Technical Report*, 8(3):57–72, 2003.

[28] Jon Peddie. *The History of the GPU-Steps to Invention*. Springer Nature, 2023.

[29] Verónika Peralta. Data freshness and data accuracy: A state of the art. *Instituto de Computacion, Facultad de Ingenieria, Universidad de la Republica2006*, 2006.

[30] Sara Ricci, Patrik Dobias, Lukas Malina, Jan Hajny, and Petr Jedlicka. Hybrid keys in practice: Combining classical, quantum and post-quantum cryptography. *IEEE Access*, 2024.

[31] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[32] Keegan Ryan and Nadia Heninger. Fast practical lattice reduction through iterated compression. *Cryptology ePrint Archive*, 2023.

[33] Spyridon Samonas and David Coss. The cia strikes back: Redefining confidentiality, integrity and availability in security. *Journal of Information System Security*, 10(3), 2014.

[34] Benjamin Schumacher. Quantum coding. *Phys. Rev. A*, 51:2738–2747, Apr 1995.

[35] Adi Shamir. Identity-based cryptosystems and signature schemes. In *Advances in Cryptology: Proceedings of CRYPTO 84 4*, pages 47–53. Springer, 1985.

[36] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[37] Joseph H Silverman, Jill Pipher, and Jeffrey Hoffstein. *An introduction to mathematical cryptography*. Springer, 2 edition, 2008.

[38] Kuldeep Singh, Manohar Sai Burra, and Soumyadev Maity. Identity based encryption and broadcast using hybrid cryptographic techniques. In *2022 International Conference on Recent Trends in Microelectronics, Automation, Computing and Communications Systems (ICMACC)*, pages 1–6. IEEE, 2022.

[39] Tage Stabell-Kulø and Simone Lupetti. Public-key cryptography and availability. In *Computer Safety, Reliability, and Security: 24th International Conference, SAFECOMP 2005, Fredrikstad, Norway, September 28-30, 2005. Proceedings 24*, pages 222–232. Springer, 2005.

[40] Zewen Sun, Zhifang Li, and Chuliang Weng. Co-utilizing simd and scalar to accelerate the data analytics workloads. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 637–649. IEEE, 2023.

[41] Atharva Takalkar and Bahubali Shiragapur. Quantum cryptography: Mathematical modelling and security analysis. In *2023 3rd Asian Conference on Innovation in Technology (ASIANCON)*, pages 01–07. IEEE, 2023.

[42] Tristen Teague, Mayeesha Mahzabin, Alexander Nelson, David Andrews, and Miaoqing Huang. Towards cloud-based infrastructure for post-quantum cryptography side-channel attack analysis. In *2023 IEEE Design Methodologies Conference (DMC)*, pages 1–6. IEEE, 2023.

[43] Xiaoyun Wang, Guangwu Xu, and Yang Yu. Lattice-based cryptography: A survey. *Chinese Annals of Mathematics, Series B*, 44(6):945–960, 2023.

[44] Wikipedia contributors. Merkle's puzzles — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Merkle%27s_Puzzles&oldid=1208543827`, 2024. [Online; accessed 17-April-2024].

Appendix A

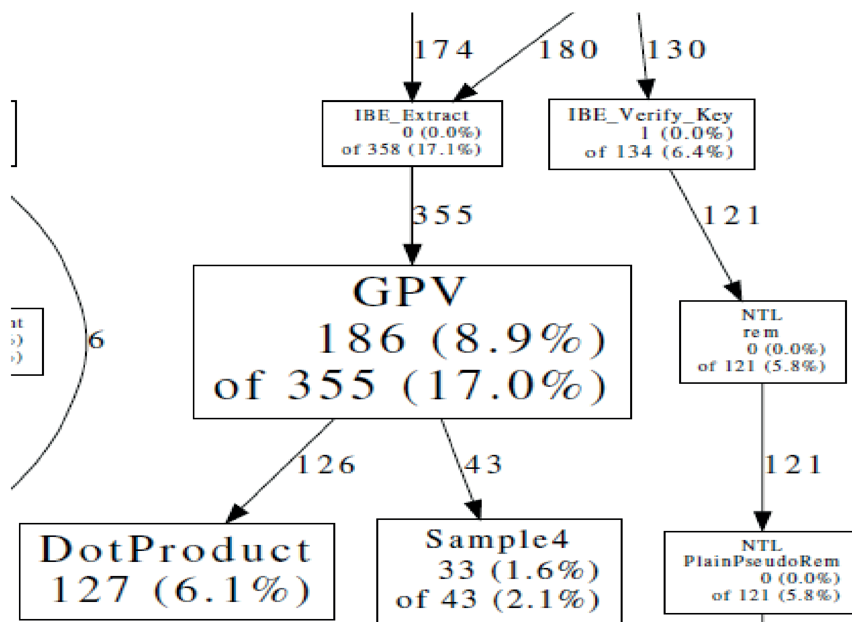


Figure 7.1: Gperf chart of the original implementation, key generation part.

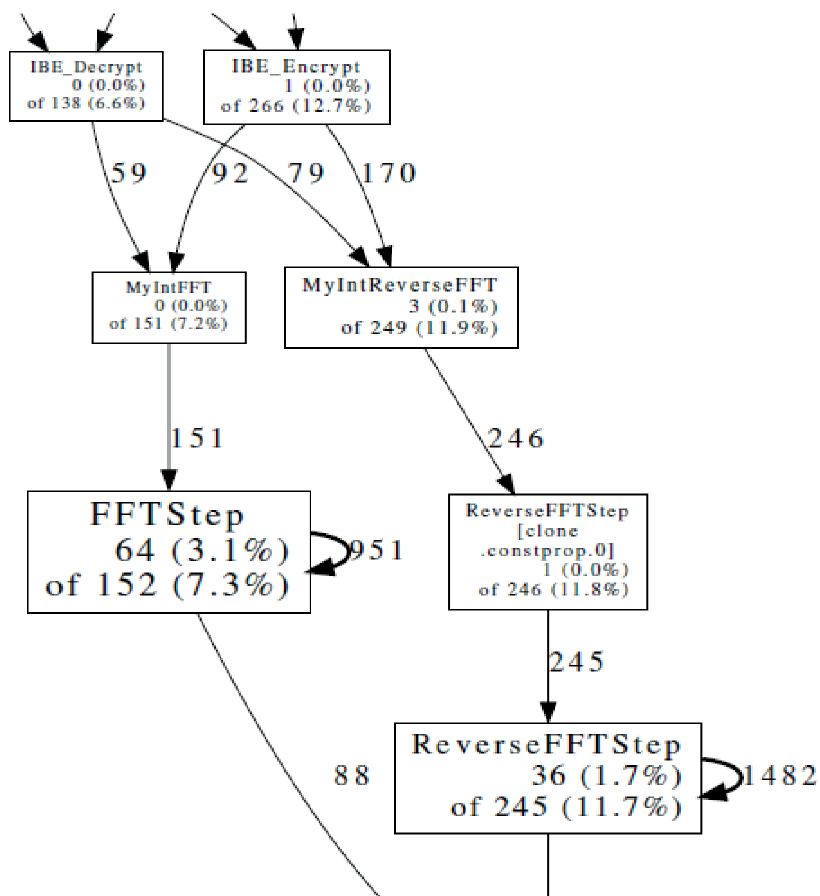


Figure 7.2: Gperf chart of the original implementation, encryption and decrytion part.

Printed by Tryckeriet i E-huset, Lund 2024

LUND UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2024-1012
http://www.eit.lth.se