

Fastest Homomorphic Encryption in the West Made Faster

Jonas Bertels

Thesis submitted for the degree of
Master of Science in
Electrical Engineering, option
Electronics and Chip Design

Thesis supervisor:
Prof. dr. ir. Ingrid Verbauwhede

Mentors:
Ir. Michiel Van Beirendonck
Ir. Furkan Turan
Ir. Jose Maria Bermudo Mera
Ir. Angshuman Karmakar

© Copyright KU Leuven

Without written permission of the thesis supervisor and the author it is forbidden to reproduce or adapt in any form or by any means any part of this publication. Requests for obtaining the right to reproduce or utilize parts of this publication should be addressed to Departement Elektrotechniek, Kasteelpark Arenberg 10 postbus 2440, B-3001 Heverlee, +32-16-321130 or by email info@esat.kuleuven.be.

A written permission of the thesis supervisor is also required to use the methods, products, schematics and programmes described in this work for industrial or commercial use, and for submitting this publication in scientific contests.

Preface

Thanks and acknowledgement

Jonas Bertels

Contents

Preface	i
Abstract	iii
List of Figures and Tables	iv
List of Abbreviations and Symbols	vi
1 Introduction to homomorphic encryption	1
1.1 An Introduction To Encryption	1
1.2 An Introduction To Learning With Errors	2
1.3 Homomorphic Encryption	3
1.4 Encryption Libraries And Homomorphic Encryption Schemes	4
1.5 FHEW (Fastest Homomorphic Encryption In The West)	5
1.6 Hardware Acceleration	6
2 Preliminaries	9
2.1 Definitions	9
2.2 Ring Learning With Errors	9
2.3 FHEW (Fastest Homomorphic Encryption In The West)	11
2.4 Bootstrapping	13
2.5 Notes On Previous Optimisations	17
3 The Number Theoretic Transform	19
3.1 Introduction To Number Theoretic Transform	19
3.2 Hardware Implementations Of NTT	22
4 Hardware Implementation	27
4.1 An Overview Of The Hardware Implementation	27
4.2 Mert's NTT implementation	28
4.3 Leveraging The NTT For FHEW	30
4.4 Memory Interface	33
5 Results & Conclusion	35
A FHEW Algorithm flowchart	39
Bibliography	41

Abstract

The `abstract` environment contains a more extensive overview of the work. But it should be limited to one page.

List of Figures and Tables

List of Figures

1.1	One Time Pad [25]	2
1.2	The jewel box [2]	4
2.1	Binary Message representation of $m * \frac{q}{2} + e$ in $\text{LWE}_s^2(m, \frac{q}{4})$ (taken from [9])	10
2.2	Messages in $\text{LWE}_s^4(m, \frac{q}{8})$ [9]	11
2.3	Messages in $\text{LWE}_s^4(m, \frac{q}{16})$ with smaller error [9]	11
2.4	Result of the addition [9]	12
2.5	Rounding the sum [9]	12
2.6	Creating an AND gate [9]	12
2.7	The final NAND gate [9]	12
2.8	Our LWE ciphertext is encrypted under an RGSW ciphertext so that it can be decrypted using LWE secret keys, which themselves have been encrypted under RGSW	13
3.1	Cooley Tukey FFT/NTT butterfly [4]	20
3.2	Gentleman Sande FFT/NTT butterfly [4]	20
4.1	Diagram of the RGSW accumulator implementation	28
4.2	Mert's datapath [18]	29
4.3	Mert's implementation of GS NTT and INTT for a ring size N of 8	30
4.4	Bitreversal in the GS INTT	31
4.5	Secret Key accumulation integrated in CT	34

List of Tables

2.1	Creating an AND gate from the sum of 2 bits	11
2.2	Initial polynomial for v=1 and n=8	15
2.3	Initial polynomial for v=1 and n=8	16
2.4	Results after subtraction by $a_i * s_i = 2$	16
2.5	Parameters for STD128	17

3.1	Hardware implementations of NTT (green is open source, yellow is open source (to be released) and red is closed source)	23
3.2	Paper and code Source for the HW implementations	24
3.3	Resources for the U280	25
3.4	FOM's for the different implementations when considering the U280 as target (lower is better)	26

List of Abbreviations and Symbols

Abbreviations

CC	Clock Cycle
FHEW	Fastest Homomorphic Encryption in the West
NTT	Number Theoretic Transform
PSNR	Peak Signal-to-Noise ratio

Symbols

42	“The Answer to the Ultimate Question of Life, the Universe, and Everything” according to [1]
c	Speed of light
E	Energy
m	Mass
π	The number pi

Chapter 1

Introduction to homomorphic encryption

1.1 An Introduction To Encryption

Let's say it's 1943 and you are the head of a resistance movement that helps downed pilots escape. To get the pilots out of Europe, you need to be able to communicate with Great Britain to be able to arrange a pick-up in Spain. The only way to do this quickly and reliably is via radio. However, you are met with a serious problem: anyone can listen in to your broadcast. Therefore you need to send your message in a code. The British Special Operations Executive provided booklets called "one-time pads". These contained long rows of letters, as seen in Figure 1.1 and were used to turn the message, which we call "plain-text" into a jumble of seemingly meaningless letters. This process of turning plain-text messages into messages that cannot be understood by eavesdroppers is called encryption. Only the receiver in Britain, with the only copy of the one-time pad in the world, would be able to turn the encrypted message back into the original message.

What has been described is called symmetric key encryption, because both the sender and receiver are using the same key. The use of it in WW2 posed certain problems: records were kept of all intercepted messages by the occupying forces, even when they were not understood. This way, when a radio operator and his one-time pad were captured, earlier messages could still be deciphered. Secondly, some way had to be found to provide radio operators with the one-time pad booklet, and a new booklet had to be created for every radio operator.

How to provide a user with a key without accidentally giving an eavesdropper the opportunity to acquire this key has always been a problem. To solve this, public-key cryptography was created. In public-key cryptography, there is both a secret and a public key. The receiver (we'll call her Alice) makes her public key available to anyone who wants it, including potential eavesdroppers. However, the public key can only be used to *encrypt* the data. Once a potential sender (we'll call him Bob)

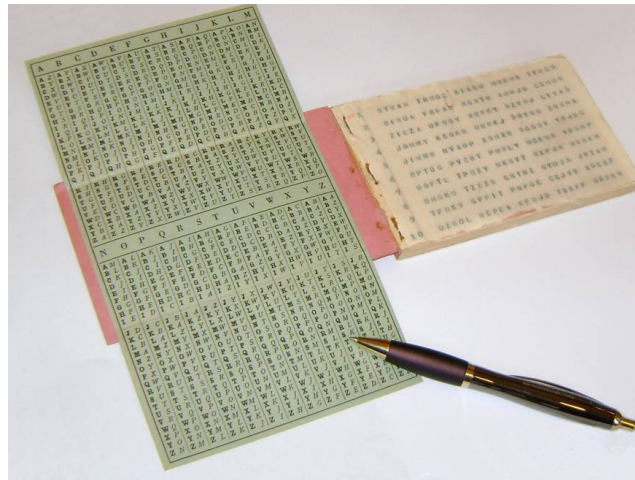


FIGURE 1.1: One Time Pad [25]

has encrypted his message using the public key, he cannot reverse this process.

It is as if Alice had sent every person in the world a safe with a lock but without a key. If someone like Bob wants to send Alice a message, they simply take the safe, place their message inside of it, and lock the safe. Bob can't unlock the safe anymore, but he can send the safe to Alice. Alice has a key that fits for all the safes she sent out, so she can unlock the safe and retrieve Bob's message, without an eavesdropper (who we'll call Eve from now on) being able to read the message, even if Eve has full access to the safe which contains Bob's message.

1.2 An Introduction To Learning With Errors

Because there are many uses for cryptography, from securing messages to securing bank accounts, there are many cryptographic schemes, i.e., ways of encrypting and decrypting information. As previously mentioned, public/secret key schemes need to have a public-key (i.e., a safe in which a sender can store information) and a secret key (i.e., a key to that safe that the receiver can use). If you are the receiver and you've chosen a secret key, you should be able to create a public key from this secret key. However, the opposite should obviously be impossible. After all, the public key is being given away to anyone and should not give (much) information about the secret key. Given enough time, you might be able to reverse engineer the public key into a secret key, so we call a scheme "secure" (i.e. good enough) if it takes so long and requires so many resources that it might as well be impossible. Impossible meaning that the required time to break the secret key is on the order of a small country dedicating all its computers for millennia for high enough security needs.

These schemes have been in use for the past 50 years. However, quantum computers, which may become a commercial reality in the future, could speed up the process of breaking these schemes massively. This means that there is a pressing need for new schemes that are based on public keys that cannot be reverse-engineered into their secret keys.

Learning With Errors is a basis for many of these new schemes. In addition to being resistant to quantum attacks, LWE has another interesting feature: it can be used for Homomorphic Encryption.

1.3 Homomorphic Encryption

Homomorphic encryption means encryption on which functions can still be executed. If you encrypt the value 3 and you encrypt the value 5, you can send both these encrypted values to a server and ask the server to add them together, then return the result back to you. Normally, the decrypted result would be some useless random value. Under a homomorphic encryption scheme, however, the result after decryption would actually be 8. Fast-working homomorphic encryption would have a wide-ranging impact on the financial sector, the web services sector and many other sectors in which a company in need of computation cannot trust the providers of computation to access their data.

There are two types of homomorphic encryption: Leveled Homomorphic Encryption and Fully Homomorphic Encryption. Leveled Homomorphic Encryption allows for a limited number of computations, Fully Homomorphic Encryption allows for an infinite number of computations. In 2009 Gentry [14] showed that Fully Homomorphic encryption, i.e. doing an arbitrary amount of operations on encrypted data, is possible.

Gentry explained this using an analogy: imagine a jeweler, Alice, who wants to let her workers work on precious gems without them being able to steal the gems [2]. She creates a box that allows the workers to manipulate the jewels. This box represents Leveled Homomorphic Encryption, because while the workers can do some operations, eventually the gloves stop working (so that the workers can no longer continue the work, although the jewels are still safe). Thus, the workers have to bring the box back to Alice so that she can unlock the box and retrieve the finished jewels. If the workers haven't finished their work yet, she must take the half-finished jewels from the box with the "worn" gloves and place them in a box with new gloves so that the workers can continue their work.

The "box" in this analogy is of course our encryption scheme "Learning With Errors". "Alice" is the user while the workers represent a server that can do operations (such as addition, multiplication or others) on the encrypted data without being able



FIGURE 1.2: The jewel box [2]

to access it. The fact that gloves wear out in our analogy is representative of the fact that the homomorphic encryption schemes use some form of numerical noise to mask the message. As we do addition, or multiplication, the amount of noise increases. Multiplication increases the noise at a much greater rate than addition. Due to this noise increase, decryption of the data (whose last step is usually a sort of rounding operation to remove noise) will at some point no longer return the correct answer. At that point, our “worker” (i.e. the server which is doing the operations) has to stop or the output he gives will be wrong.

The idea behind "bootstrapping" is to turn Leveled Homomorphic Encryption into Fully Homomorphic Encryption. This can be done by performing the decryption, i.e., the unlocking of the box, while everything is encrypted under another encryption. In our analogy, this would be like locking our box in another box that already has a key for our box inside of it. (Since we are using public/secret key encryption, we can place things into a locked box with the public key without having to unlock said box. This is a feature of public-key encryption).

We call this unlocking while keeping the box locked under a different box "bootstrapping". We call it "bootstrapping" because it allows us to keep working on the encrypted message. Our workers can open the inner box using the secret key which is taped to the inside of the outer box and continue working on the jewels (see Figure 1.2). For an arbitrary amount of boxes, we can thus compute an arbitrary amount of functions on data that stay encrypted.

1.4 Encryption Libraries And Homomorphic Encryption Schemes

Several software libraries have been created to provide software developers with the tools required to use schemes in their messenger apps, database programs, financial

application software, ...

We can split the homomorphic schemes that are currently the focus of most research into 3 generations. The first generation consists of older schemes that are so inefficient they are no longer being used. The second generation consists of schemes like the Fan-Vercauteren [10] scheme, the Brakerski-Gentry-Vaikuntanathan scheme and the Gentry-Sahai-Waters scheme, which are primarily used for Leveled Homomorphic Encryption. The second-generation schemes allow for multiplication and addition but have a very compute-intensive bootstrapping process. The third generation schemes such as FHEW and THFE only allow for 1 gate (such as a AND/OR gate), and bootstrap immediately, but the bootstrapping happens relatively fast ($\approx 100\text{ms}$)[19]. In this master thesis, I have accelerated one of the third generation schemes, namely Fastest Homomorphic Encryption in the West (FHEW) but it is still worth to first consider the Fan-Vercautren scheme.

1.4.1 Fan-Vercauteren: Usually Leveled Homomorphic Encryption

FV has been the subject of hardware acceleration by multiple groups [20, 31, 28]. It is also implemented in the homomorphic encryption libraries SEAL [6] and PALISADE [5] among others.

The problem with the second generation schemes as previously mentioned is that bootstrapping takes so long the schemes usually forgo the bootstrapping for practical reasons. This means that any user of the libraries has a limited number of multiplications to work with (as we have seen, the limited noise growth of additions is usually negligible). This imposes constraints on the programmer.

In summary then: second-generation schemes execute fairly quickly, and as such schemes like FV are good for a certain class of programs with a limited amount of operations but a large amount of data. However, they only allow for a limited number of computations, and algorithms that are already out there and require a lot of instructions will thus not easily translate into programs that can homomorphically execute data. This can also be seen in a hardware acceleration for the Fan-Vercauteren scheme done by COSIC in 2019 [28]. This implementation could do FV very quickly, at 400 homomorphic multiplications per second, but is limited to a multiplicative depth of 4. Although this is enough for a lot of applications, it can limit its flexibility in a general role.

1.5 FHEW (Fastest Homomorphic Encryption In The West)

FHEW (the name is a reference to the Fastest Fourier Transform in the West library [11]) is part of the so-called third-generation schemes[7, 3]. It attempts to tackle

the long length of the bootstrapping process by only executing one NAND gate (other simple gate functions are also possible) and then immediately bootstrapping. This makes it a true fully homomorphic scheme, as the bootstrapping can be done within a reasonable amount of time, namely 137 milliseconds for NAND + bootstrap for 128-bit security on an intel i7 [19]. In other words, while only one NAND gate can be done at a time (later papers show parallelisation is possible), there is no limit on the depth of our circuit, and as all functions can be written as a combination of logic gates, there is no limit on the functions that can be executed.

Because of its flexibility, a FHEW implementation that operated fast enough would have a large advantage in usability over a Leveled Homomorphic Encryption scheme. Programmers using FHEW do not need to worry about the noise growth of the scheme they are using, or when they should return their data to the client. The entire FHEW scheme provides the programmer with the basic binary gates, and as such, and the programmer only has to worry about building his program from these binary gates. The programmer does not have to worry about the internal noise growth when using a Fully Homomorphic Scheme. It is for this reason that this scheme was chosen to be accelerated.

1.6 Hardware Acceleration

We have previously mentioned the term *Hardware Acceleration* when talking about an implementation of Fan-Vercauteren. Hardware acceleration means building a specialized electronic circuit to make a certain function run faster (in less time) or using less energy. Electronics built for one function only are faster than electronics built to execute software. The reason for this is simple: Specialized circuits can dedicate all their resources to executing very specific operations and as such the surfeit of resources can be dedicated to doing as many operations in parallel as possible. This increasing parallelism means increasing speed, as long as the algorithm itself can be implemented in parallel. In this thesis, we use a FPGA, which is a form of *programmable* electronics, where different electrical components are wired together according to a netlist that can be programmed from a computer. The other option for hardware acceleration would be to create an ASIC, which is an Application Specific Integrated Circuit. An ASIC can provide much larger speed gains than an FPGA, but it is significantly more expensive to create, as new hardware has to be physically created for it, and it is also much less flexible. Once created, in most cases it cannot be reprogrammed.

The main advantage of FPGA's is that they can provide a speedup of a factor 10 to a factor 100 for a scheme like FHEW.[28] This is a smaller speed-up than an ASIC implementing the same algorithm would provide, as an ASIC does not dedicate resources to reprogrammability, but it is still a great improvement. Nonetheless, FPGA's are still specialized electronics. Once an implementation is shown to work well and is accepted to be worth it, an ASIC based on the FPGA implementation

might one day end up in a consumer device, such as laptop, smartphone, or in the case of homomorphic encryption, a server, but this is unlikely to occur soon, and requires significant investment.

In other words, Hardware Acceleration requires a new chip to become part of the server hardware when we use an ASIC, or for cloud servers to already contain an FPGA. Additionally, the encryption schemes which are hardware accelerated are still in flux, and there is no guarantee that a much faster scheme isn't just over the horizon, rendering the investments in a specially built ASIC pointless. Software is much easier to deploy and requires no expensive investment. However, recently web hosting companies such as Amazon Web Services have started making servers with programmable FPGA's available [30], which makes deploying hardware almost as easy as deploying software. Given that the computational bottlenecks for Homomorphic Encryption schemes mainly lie at the server-side, hardware acceleration becomes a perfect solution to the speed problems faced by the Homomorphic Encryption schemes.

Chapter 2

Preliminaries

2.1 Definitions

Lattice-based cryptography uses polynomial rings for various optimizations. The polynomial rings we use are defined as $R = \mathbb{Z}[x]/f(x)$ with $f(x) = x^d + 1$ and $d = 2^m$ with $m \in \mathbb{N}$. We also define $R_q = R/qR = \mathbb{Z}_q[x]/f(x)$, where again $f(x) = x^d + 1$ and $d = 2^m$ but \mathbb{Z}_q defines the set of integers between $(-q/2, q/2]$. [10]

For our error terms, we consider the discrete Gaussian sampling function $D_{\mathbb{Z}, \sigma}$. To get our error term, we can simply sample our coefficients from this Discrete Gaussian function, so that the error distribution $\chi = D_{\mathbb{Z}, \sigma}^d$.

2.2 Ring Learning With Errors

Many homomorphic encryption schemes [10] are based on the Ring Learning With Errors Problem, which states the following: For many vectors $(a_i, b_i) \in (R_q, R_q)$, $b_i = a_i * s + e_i$ with e_i being a randomly sampled error term and s being our secret key and a_i being a randomly sampled but known vector, it is not computationally feasible to find s . In other words, (a_i, b_i) is a good public key since we can't reverse engineer the secret from the public key. If we consider the problem from the case of the server: we cannot reverse engineer (a_i, b_i) into s , even if we have a very large quantity of a_i 's and b_i 's.

First, we define the LWE encryption of a message $\tilde{m} \in R_q$ with a secret key s as: [19]

$$\text{LWE}_s^t(\tilde{m}, E) = (a, as + e + \tilde{m}) \quad (2.1)$$

with $a \leftarrow R_q$ and randomly sampled and $e \leftarrow \chi_\sigma^d < E$ with χ_σ being a discrete Gaussian distribution with σ as parameter which is sampled d times (once for every coefficient) to form χ_σ^d . The secret key s can be sampled from R_q or from a smaller distribution. Using smaller distributions improves the performance, but reduces the security. For the purpose of this work we focus on Gaussian (i.e. the most general)

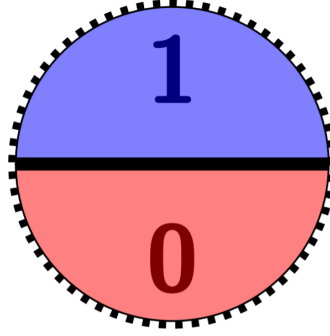


FIGURE 2.1: Binary Message representation of $m * \frac{q}{2} + e$ in $\text{LWE}_s^2(m, \frac{q}{4})$ (taken from [9])

secret keys.

Our actual message m is a bit, not an integer. We have $\tilde{m} = \frac{q}{t} * m$ with q as always our modulus ($=512$ see parameter set STD128 in Table 2.5 [19]) and t being dependent on which LWE scheme we use (we therefore write it as LWE_s^t).

Figure 2.1 shows a representation of such a message. From the representation, it is immediately obvious that the error should not exceed $\frac{q}{4} = 128$.

We can now define our decryption function as:

$$\text{LWE}_s^{-1}(a, b) = b - a * s = \tilde{m} + e \quad (2.2)$$

We can then recover m from \tilde{m} by multiplying with $\frac{t}{q}$ and rounding the error away.

2.2.1 The Decryption Function

As previously mentioned, our message is a single bit. If it is 0, we could represent it as some error, if it is 1, we could represent it as $\frac{q}{2} (= 256)$ with some error added to it, as shown in Figure 2.1. For LWE_s^t with $t = 2$, our maximum error is equal to $\pm \frac{q}{2t} = 128$. We can now present LWE_s^4 . This representation allows for 4 possible messages, with the message being multiplied by $\frac{q}{t} = 128$, which is $\frac{q}{4}$ (Figure 2.2). Our maximum error value, in this case, is $\pm \frac{q}{8} = 64$. This is written as $\text{LWE}_s^4(\tilde{m}, q/8)$. **Nothing stops us from reducing the possible error further though**, as can be seen on the right of Figure 2.3, so that there are values that are never reached by a given message and error (white space in Figure 2.3). This is written as $\text{LWE}_s^4(\tilde{m}, q/16)$.

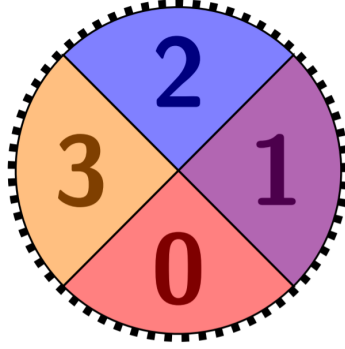


FIGURE 2.2: Messages in $\text{LWE}_s^4(m, \frac{q}{8})$ [9]

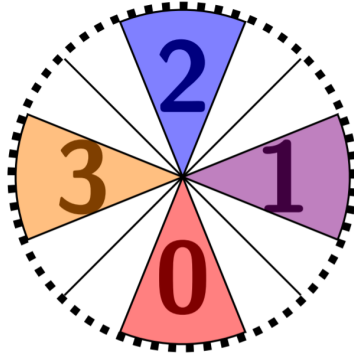


FIGURE 2.3: Messages in $\text{LWE}_s^4(m, \frac{q}{16})$ with smaller error [9]

Result of the Sum	0	1	2	3
Result of a AND gate	False (0)	False (0)	True (1)	N/A

TABLE 2.1: Creating an AND gate from the sum of 2 bits

2.3 FHEW (Fastest Homomorphic Encryption In The West)

We can use **these** messages for addition (Figure 2.4). After addition, the result now has more error than the beginning, because the noise adds up.

Our interest lies in making a functional NAND gate. For a NAND gate, consider Table 2.1. We first create an AND gate from the addition of 2 encrypted messages. If the resulting sum's error stays small enough to decrypt correctly (i.e., if the colours of Figure 2.4 do not shade into each other), we can map a result of 2 to a True outcome of the AND gate, and a result of 0 or 1 to a False outcome of the AND gate.

If we consider everything on the top left to be 1 and everything on the bottom

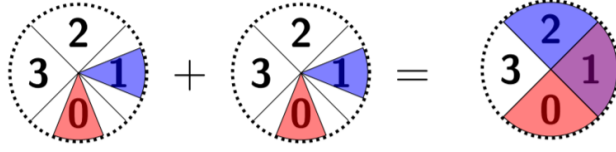


FIGURE 2.4: Result of the addition [9]

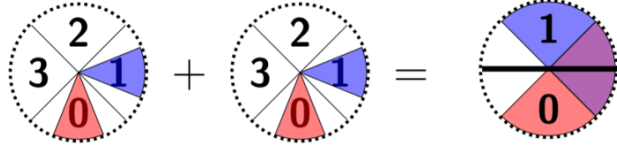


FIGURE 2.5: Rounding the sum [9]

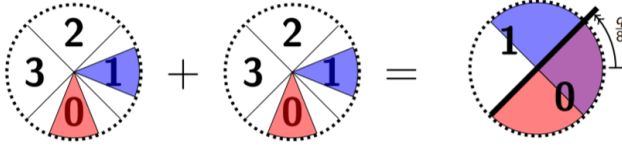


FIGURE 2.6: Creating an AND gate [9]

right to be 0, we have an AND gate (Figure 2.6 and Table 2.1). To make this an NAND, we rotate by $\frac{q}{2}$, in other words, we consider the bottom right to be 1 and the other half to be a 0, and this results in the results that we expect from a NAND gate for the given input values (Figure 2.7).

If we look at the result of our NAND gate, we immediately see a lot less white space than previously. As previously mentioned, the noise has increased. We originally had $2 \text{LWE}_s^4(\tilde{m}, q/16)$ inputs, and now we have one $\text{LWE}_s^4(\tilde{m}, q/8)$ output. Clearly, if we were to use this output as the input of our next binary gate, it is possible that we would no longer decrypt the correct result. As such we need to find some way of reducing the noise.

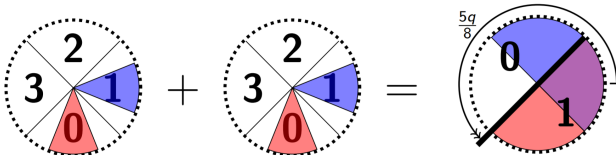


FIGURE 2.7: The final NAND gate [9]

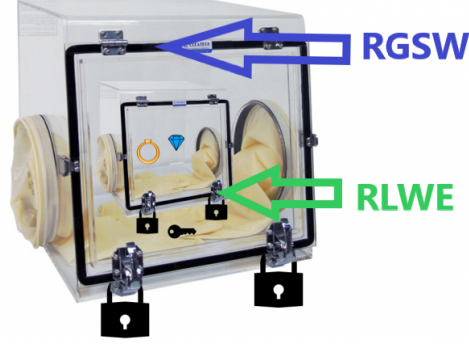


FIGURE 2.8: Our LWE ciphertext is encrypted under an RGSW ciphertext so that it can be decrypted using LWE secret keys, which themselves have been encrypted under RGSW

2.4 Bootstrapping

2.4.1 Introduction To Bootstrapping In FHEW

What we've discussed until now is the *underlying* encryption system. If we go back to our analogy from the introduction, our LWE scheme is the box that the workers use to assemble the jewels. As we have seen, the gloves on this box are one-use-only (we only do one NAND gate, or a similarly small binary gate operation to go from $\text{LWE}_s^4(\tilde{m}, q/16)$ to $\text{LWE}_s^4(\tilde{m}, q/8)$). The workers now need a larger box in which they can put this LWE box, so that they can unlock the LWE box and put the jewels in a new (small, LWE) box with new gloves (in other words, with reduced maximum error $q/8$). This (larger) box will be provided by RGSW, which we will define in the next paragraph. By encrypting our secret keys under RGSW, we can safely send them to the server without breaking security by giving the server the means to decipher the LWE ciphertext into plain text. Instead, we will decipher our $\text{RGSW}(\text{LWE}(m))$ into a $\text{RGSW}(m)$, just like the jewels never leave the security of the outer box once the inner box has been unlocked.

As mentioned before this process is called bootstrapping. Bootstrapping is done by executing decryption operations homomorphically, i.e. doing the decryption operations with an encrypted secret key. This reduces the noise back down to the original level, completing our algorithm.

2.4.2 Definitions For Bootstrapping

We define RLWE' [8] [19]:

$$\text{RLWE}'_s(m) = (\text{RLWE}_s(m), \text{RLWE}_s(B * m), \text{RLWE}_s(B^2 * m), \dots, \text{RLWE}_s(B^{k-1} * m)) \quad (2.3)$$

with $B^k = q$. (In our parameter set, $B = 128 = 2^7$). The reason for this definition is that for the outer box, we want a system that creates as little noise

as possible to ensure that the bootstrapping step doesn't add more noise than it removes. By breaking RGSW into bases, less noise is generated. Multiplications are defined as:

$$(\odot) : R \times \text{RLWE}' \rightarrow \text{RLWE} : d \in \mathbf{R}, \mathbf{c} \in \text{RLWE}' : d \odot (\mathbf{c}_0, \dots, \mathbf{c}_{k-1}) = \sum_i d_i \mathbf{c}_i \quad (2.4)$$

We can extend this multiplication definition to a multiplication that results in RLWE' :

$$(\odot') : R \times \text{RLWE}' \rightarrow \text{RLWE}' : d \odot \mathbf{C} = ((B^0 * d) \odot \mathbf{C}, (B^1 * d) \odot \mathbf{C}, \dots, (B^{k-1} * d) \odot \mathbf{C}) \quad (2.5)$$

And finally, using these definitions, we can define our RGSW scheme, which will allow us to multiply ciphertexts together. We need this ability if we want to be able to run our RLWE decryption while everything is encrypted under RGSW.

$$\text{RGSW}_s(m) = (\text{RLWE}'_s(-s * m), \text{RLWE}'_s(m)) \quad (2.6)$$

To be able to use our secret key s encrypted under RGSW, we need to be able to multiply an RGSW ciphertext $\text{RGSW}(m_1) = (\mathbf{c}, \mathbf{c}')$ with a RLWE' ciphertext $\text{RLWE}'_s(m_0, e_0) = (a, b)$ and get a result that when decrypted gives the product of the 2 messages [19]:

$$(a, b) \diamond (\mathbf{c}, \mathbf{c}') = \langle (a, b), (\mathbf{c}, \mathbf{c}') \rangle = a \odot \mathbf{c} + b \odot \mathbf{c}' \quad (2.7)$$

$$= a \odot \text{RLWE}'_s(-s * m_1) + b \odot \text{RLWE}'_s(m_1) \quad (2.8)$$

$$= \text{RLWE}'_s((b - a * s) * m_1) = \text{RLWE}'_s((m_0 + e_0) * m_1) \quad (2.9)$$

This $\text{RGSW} \times \text{RLWE}' \rightarrow \text{RLWE}$ can be turned into a $\text{RGSW} \times \text{RLWE}' \rightarrow \text{RLWE}'$, and then we have everything we need to do our bootstrapping.

2.4.3 Bootstrapping Algorithm

We now turn our attention to the bootstrapping process itself, first formally using the algorithm described in [19]. The bootstrapping consists of 3 steps: Initialization (see Algorithm 1), Accumulation (Algorithm 2) and Extraction (which simply consists of taking the first value of the second part of the RGSW result).

Initialisation

The way that we implement both our rounding function and NAND gate that we previously described is as a look-up table. In a look-up table, the result of an input i is stored at index i and executing the function thus simple requires looking at the value at index i . For example, we can turn a NAND gate into a look up table by first calculating the sum of the 2 inputs, a and b . For index $a+b$, we then have the Look-Up Table 2.2.

Algorithm 1: Initialization[19]**Data:** b such that $(\mathbf{a}, b) = c_0 + c_1$ with c_0 and c_1 encrypted input of NAND**Result:** $\mathbf{ACC}_f[b] = \text{RLWE} \left(\sum_{i=0}^{q/2-1} f(b-i) * X^i \right) \in R_q^2$ **begin** **for** $i = 0, 1, \dots, q/2 - 1$ **do** $m_i = f(b - i);$ Note that f is our look-up table **end** $\mathbf{m} = \sum_{i < q/2} m_i * X^i = (m_0, m_1, \dots, m_{q/2-1});$ Return $(0, \mathbf{m});$ **end****Algorithm 2:** A Single Accumulation Step[19]**Data:** \mathbf{ACC} , an RGSW ciphertext and $E(s) = \{\mathbf{Z}_{j,v} = \text{RGSW}(X^{v*B_r^s*s} | j < \log_{B_r}(q), v \in \mathbb{Z}_{B_r}) \in \text{RGSW}^{B_r * \log_{B_r}(q)}$ **Result:** \mathbf{ACC} , updated with $a_i * s_i$ **begin** **for** $j = 0, 1, \dots, \log_{B_r}(q) - 1$ **do** $c_j = \lfloor c / B_r^j \rfloor \bmod B;$ **if** $c_j > 0$ **then** $\mathbf{ACC} \leftarrow \mathbf{ACC} \diamond \mathbf{Z}_{j,c_j};$ **end** **end** Return $\mathbf{ACC};$ **end**

$a + b = 3$ (not possible)	$a + b = 2$	$a + b = 1$	$a + b = 0$
N/A	0	1	1

TABLE 2.2: Initial polynomial for $v=1$ and $n=8$

In FHEW, a polynomial is used as a look-up table. The value at index i in the table is given by the coefficient of X^i in the polynomial. To explain our scheme, we will use tables rather than polynomials, and reduce the length of the table/polynomial from $n = 512$ to $n = 8$.

Our look-up table is initialized to the formula seen in algorithm 1 (see Table 2.3). This LUT is an implementation of our binary gate (our NAND for instance) and our rounding function (which is the final step in a LWE decryption, after the subtraction of $a_i \cdot s_i$ for all i). The only operation that we will from now on perform on this look-up table is to rotate the indices. With rotation, we mean that we change the

2. PRELIMINARIES

Index of Coefficient	7	6	5	4	3	2	1	0
Value of Coefficient	0	0	0	0	$f(-2) = f(2)$	$f(-1) = f(3)$	$f(0)$	$f(1)$

TABLE 2.3: Initial polynomial for $v=1$ and $n=8$

Index of Coefficient	7	6	5	4	3	2	1	0
Value of Coefficient	0	0	$f(-2) = f(2)$	$f(-1) = f(3)$	$f(0)$	$f(1)$	0	0

TABLE 2.4: Results after subtraction by $a_i * s_i = 2$

look-up table so for index i

$$i \rightarrow i + r \pmod{\text{LUT size}} \quad (2.10)$$

Rotating a look-up table is the same as adding or subtracting from the input index of the look-up table. It does not affect the function that the look-up table represents, only the index.

When we initialize our look-up table, we give it an initial rotation, so that the result of $g(\text{LWE}_s^t(\tilde{m}, q))$, with $g(x)$ our combined NAND and rounding function, is found under the index 0. We can also see this LUT as a RLWE ciphertext, as described in algorithm 1. That the LUT is also a RLWE ciphertext is important in the accumulation step.

Accumulation

Another way to look at the LUT is as a noiseless RLWE encryption of a polynomial, with as coefficients the results of the rounding+NAND function. (Table 2.3). A rotation of the look-up table by a value is an addition of that value to the index. Addition on the LWE level is then performed using multiplication on the RGSW level. The reason for this is again our look-up table. When we rotate the look-up table leftward, from our LWE perspective we are subtracting $a_j * s_j$ from the index (which was originally v). From an RGSW or RLWE perspective, a rotation by a value $a_j \cdot s_j$ is a multiplication of the polynomial with a value $X^{a_j \cdot s_j}$.

Now that we know we can do additions on the $\text{LWE}_s^4(\tilde{m}, q/8)$, we know we can decrypt it. So, we simply perform a series of $\text{RGSW} \odot' \text{RLWE}' \rightarrow \text{RLWE}'$ multiplications to perform the decryption $b - \mathbf{a} * \mathbf{s}$.

Table 2.4 shows one rotation through a look-up table.

Extraction

After extracting the correct value from the polynomial (i.e. the coefficient of X^0), which is the constant term or the rightmost value in our table, a key-switching step occurs to change from the keys used in the Accumulator back to the keys of the

n	q	N	Q	B_g	B_r
512	512	1024	$134215681 < 2^{27}$	128	23

TABLE 2.5: Parameters for STD128

RLWE. This step does not concern this thesis as we are mainly interested in resolving the bottleneck operation of FHEW, which is in the accumulator loop.

The full bootstrapping algorithm can be found in flowchart form in Appendix A.

The parameters for which our hardware implementation was designed are given by Table 2.5.

2.5 Notes On Previous Optimisations

The version that is accelerated in this thesis is an optimized version([19]) of the FHEW scheme originally introduced in 2014 [8]. The only optimisation paper on this scheme yet was an effort to make FHEW run on multicore CPU and GPU's, where they achieved a speedup of about 2.2 using CUDA on the 2015 version of FHEW. [15]

Another optimisation paper introduced NuFHE, which uses another third-generation scheme called TFHE with GPU acceleration and succeeded in a 100 times speedup, bringing the cost of homomorphic encryption down to 0.13 ms/bit for binary gates (for FFT implementation). For implementations using the NTT, it is 0.35 ms/bit [21]. However, this scheme implemented full adders and took advantage of the fact that multiple bits can be packed into one cipher text, then divided the total bootstrapping speed by the number of bits in one adder. No attempt at packing is made in this thesis.

Chapter 3

The Number Theoretic Transform

3.1 Introduction To Number Theoretic Transform

The Number Theoretic Transform [4] [16] [24] is an algorithm for performing polynomial (ring) multiplication. Most variants of it are almost identical to the Fast Fourier Transform, meaning it executes quickly (in $O(\frac{n}{2} \log n)$ time). The idea of the algorithm is to transform a polynomial into a form that allows polynomial multiplication to be done as point-wise multiplication, which executes in linear time. Then the result is transformed back into polynomial form. Given that executing the addition of one $a_i * s_i$ secret key to our accumulator requires $d_r * 2 * d_g (= 16)$ NTT's, and we perform this step 512 times, it is obvious why high performance for this transformation is necessary.

3.1.1 Mathematics Behind NTT

The NTT is a Discrete Fourier Transform with the twiddle factor changed from a complex number so that $\omega^{\frac{N}{2}} = 1$ to a $\omega \in \mathbb{Z}_Q$ with again $\omega^N = 1$, where ω is a primitive root of unity ($\omega^{\frac{N}{2}} = -1$)

The discrete Fourier is given by the expression ([4], equation 3.1).

$$X_r = \sum_{l=0}^{N-1} x_l \omega^{rl} \quad (3.1)$$

As always, N is our ring size ($=1024$ for our parameters) and Q is the modulus of the coefficients ($=$ a 27-bit number)). The expression for the NTT is identical, but with $\omega \in \mathbb{Z}_q$. We also define our **standard** root of unity as $\omega_N = \omega$, because as previously mentioned, $\omega_N^N = 1$, and following this definition we define $\omega_{\frac{N}{2}} = \omega^2$ because $\omega_{\frac{N}{2}}^{\frac{N}{2}} = \omega^2 = 1$

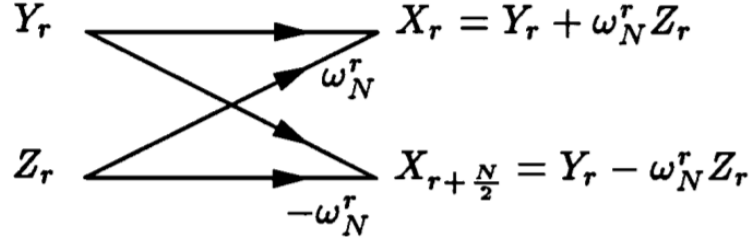


FIGURE 3.1: Cooley Tukey FFT/NTT butterfly [4]

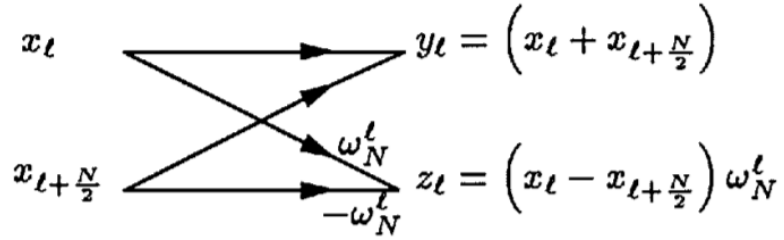


FIGURE 3.2: Gentleman Sande FFT/NTT butterfly [4]

There are now 2 methods that we consider for performing the radix-2 FFT/NTT: The Cooley-Tukey and the Gentleman Sande. The easiest way to show how these work is via 2 butterfly diagrams (Figure 3.1 and Figure 3.2).

The Cooley-Tukey Butterfly:

First we split 3.1 into an even and an odd parts:

$$X_r = \sum_{k=0}^{\frac{N}{2}-1} x_{2k} \omega_N^{2rk} + \omega_N^r \sum_{k=0}^{\frac{N}{2}-1} x_{2k+1} \omega_N^{2rk} \quad (3.2)$$

Let us define the following:

$$Y_r = \sum_{k=0}^{\frac{N}{2}-1} y_k \omega_N^{rk}, y_k = x_{2r} \quad (3.3)$$

$$Z_r = \sum_{k=0}^{\frac{N}{2}-1} z_k \omega_N^{rk}, z_k = x_{2r+1} \quad (3.4)$$

As such we can write the equation as:

$$X_r = Y_r + \omega_N^r * Z_r \quad (3.5)$$

Now we can consider the result of this equation for 2 cases: for the case that $r \in [0, \frac{N}{2} - 1]$ and $r \in [\frac{N}{2}, N - 1]$.

In the first case, the equation does not vary, in the second case, because $\omega_N^{r+N/2} = -\omega_N^r$ and because $\omega_N^{r+N/2} = \omega_N^r$ so that $Y_{r+N/2} = Y_r$ and $Z_{r+N/2} = Z_r$, we can write the following:

$$X_{r+N/2} = Y_{r+N/2} + \omega_N^{r+N/2} * Z_{r+N/2} = Y_r - \omega_N^r * Z_r \quad (3.6)$$

Note that we use ω^2 instead of ω for calculating these NTT's.

The Gentleman-Sande Butterfly:

The Gentleman-Sande works by splitting 3.1 into a first half and second half:

$$X_r = \sum_{k=0}^{\frac{N}{2}-1} x_k \omega_N^{rk} + \sum_{k=\frac{N}{2}}^{N-1} \omega_N^{rk} = \sum_{k=0}^{\frac{N}{2}-1} (x_k + x_{k+\frac{N}{2}} \omega_N^{r\frac{N}{2}}) \omega_N^{rk} \quad (3.7)$$

For even $r = 2 * l$, we have $\omega_N^{r\frac{N}{2}} = \omega_N^{2*l*\frac{N}{2}} = 1$, so this equation becomes:

$$X_{2l} = \sum_{k=0}^{\frac{N}{2}-1} (x_l + x_{l+\frac{N}{2}}) \omega_N^{2kl} \quad (3.8)$$

and for odd $r = 2 * l + 1$ we have $\omega_N^{r\frac{N}{2}} = \omega_N^{(2*l+1)*\frac{N}{2}} = -1$ so:

$$X_{2l+1} = \sum_{k=0}^{\frac{N}{2}-1} (x_l - x_{l+\frac{N}{2}}) \omega_N^{(2l+1)k} \quad (3.9)$$

3.1.2 The Polynomial Multiplication And Negatively Wrapped Convolution

To perform the polynomial multiplication, we must not just perform an NTT but also an INTT. Luckily, the INTT is defined as: [4]

$$X_r = N^{-1} \sum_{l=0}^{N-1} x_l \omega^{-rl} \quad (3.10)$$

Therefore, the INTT can be performed as an NTT by simply changing the ω to their inverse and performing multiplication with the modular inverse of n . ($N^{-1} * N = 1 \pmod{q}$).

Most implementations of NTT result in an output with bit-reversed indices [4]. This means that to find the correct value, it is necessary to take the index of that

value, write it in binary form for $\log_2 N$, and "flip" this index in such a way that for example 1010000000 become 0000000101. Then we use this flipped index to find the correct value. Indeed, in the PALISADE library's implementation, the NTT implementation takes a normally-order polynomial and returns a bit-reversed, transformed polynomial. The INTT does the opposite. As such, in the PALISADE library implementation, there is no need for extra steps to undo this bit-reversal.

Normally the following optimisation is applied to polynomial multiplication with NTT. In the standard case, we multiply together 2 polynomials of ring size $2N$ after an NTT with half of the inputs being zeros, and do a reduction step of the polynomial modulo $X^N + 1$ [24].

There is however an optimisation that can be applied to avoid the reduction step and do the multiplication with polynomials of size N instead. This optimisation is the *negatively wrapped convolution*, where we use a ψ so that $\psi^2 = \omega \pmod{q}$, and we calculate

$$\hat{a} = (a[0], a[1] * \psi, a[2] * \psi^2, \dots, a[n-1] * \psi^{n-1}) \quad (3.11)$$

and

$$\hat{b} = (b[0], b[1] * \psi, b[2] * \psi^2, \dots, b[n-1] * \psi^{n-1}) \quad (3.12)$$

We can then calculate

$$\hat{c} = \mathbf{INTT}(\mathbf{NTT}(\hat{a}) \circ \mathbf{NTT}(\hat{b})) \quad (3.13)$$

and it turns out that for

$$c = \mathbf{a} * \mathbf{b} \pmod{X^N + 1} \quad (3.14)$$

we have

$$c = (\hat{c}[0], \hat{c}[1] * \psi^{-1}, \hat{c}[2] * \psi^{-2}, \dots, \hat{c}[n-1] * \psi^{-(n-1)}) \quad (3.15)$$

This multiplication with ψ or ψ^{-1} can be integrated in our multiplication with the ω factors [24]. How this is done depends on how the algorithm is implemented, and as such, we first have to determine which implementation of NTT to choose from.


3.2 Hardware Implementations Of NTT

Because NTT is a common feature among many post-quantum cryptography implementations, there are many HW implementations for it available. **As such we created Table 3.1.**

3.2.1 Criteria For Good NTT hardware

Open source vs. closed source

Because the NTT is widely used in cryptographical applications, there are many papers discussing hardware designs of the NTT. As such there is no point in creating a new NTT hardware accelerator from scratch. Additionally, many hardware papers



Author	Year	FPGA	NTT Area	#CC NTT	Frequency	$\mathbf{c} = \mathbf{a} \odot \mathbf{b}$ Area	#CC $\mathbf{c} = \mathbf{a} \odot \mathbf{b}$	N	$\log_2 Q$
Oder	2017	Arktix-7	N/A	35840	125			1024	13.6
Roy	2018	UltraScale		87582	200	64K/25K/0.4K/0.2K	5349567	4096	180
Wang	2020	Arktix-7			126	1735/758/6/0	11,455	1024	28.35
Fritzmann	2020	Arktix-7	2908/170/9/0	18537	45.47		36,102	1024	13.6
Roy	2014	Virtex 6	1536/953/1/3	2304	278			512	13.6
Kuo	2017	Artix-7	2832/1381/8/10	2616	131			1024	13.6
Rashmi	2019	Artix-7			0.1	no area for NTT	33792	1024	13.6
Du	2016	Spartan-6			241	251slice/1/4.5	11826	1024	16
Hartshorn	2020	Virtex 7	9233/8585/128/8	1294	179		1936	1024	N/A
Mert	2019	Virtex 7			211.5	1208/556/14/14	7970	1024	32
Mert	2020	Virtex-7	575/0/3/11	5160	125			1024	14
Mert	2020	Virtex-7	2584/0/24/16	680	125			1024	14
Mert	2020	Virtex-7	17188/0/96/48	200	125			1024	14
Mert	2020	Virtex-7	38093/0/224/48	250	125			1024	29

TABLE 3.1: Hardware implementations of NTT (green is open source, yellow is open source (to be released) and red is closed source)

Name	Paper	Source
Oder	[23]	[22]
Roy, Furkan	[32]	[26]
Wang	[34]	[33]
Fritzmann	[13]	[12]
Roy, Sujoy	[29]	[27]
Mert	[18]	[17]

TABLE 3.2: Paper and code Source for the HW implementations

make their source code available. Table 3.1 mainly lists open-source implementations, but closed source implementations can also be interesting as they often describe the optimisations that make them successful.

3.2.2 Time to perform one NTT

The primary purpose of our hardware acceleration is to *accelerate*. Therefore it is vital that the FHEW is performed faster than the 137 ms currently achieved on an Intel-i7.

The design that stands out for this characteristic is Mert’s parametric NTT implementation, with its 250 clock cycle run-time at 125 MHz for the largest implementation parameters that the paper lists and for the design parameters set that we are targeting. If we used the next fastest implementation instead (Harthorn’s, which is closed source), it would take 1294 clock cycles at 179 MHz to perform one NTT. For 8096 NTT’s, this means:

$$T_{\text{run}} = \frac{1294 * 8096}{179 * 10^6} = 58.5 \text{ ms} \quad (3.16)$$

In other words, we would only be twice as fast simply running FHEW on (unoptimized) software. The reason for this slow execution speed of many NTT implementations is that area is often an important concern. Since the bootstrapping algorithm for FHEW runs on large servers, and any hardware acceleration is likely to be done on large data center accelerator cards, such as the U280, the primary concern here is execution time.

There is one Caveat: during FHEW, we are able to perform certain NTT’s (up to 2 INTT’s and 8 forward NTT’s) in parallel. Taking full advantage of this with Harthorn’s implementation would give an execution time of 14.8 ms, which is almost 10 times faster than the 137 ms time on an Intel i7 and might be a worthwhile speed-up. However, the design is still closed source and the next 2 fastest designs were made for $\log_2 q = 13.6$, with one not having publicly available source code and the other being designed for a ring size of 512 instead of 1024.

Resource Name	LUT	FF	DSP	BRAM
Amount	1304000	2607000	9024	1490

TABLE 3.3: Resources for the U280

3.2.3 Number of NTT's per data accelerator card

While the first concern of any hardware acceleration is to accelerate the algorithm in question, it is also important to consider how much area the implementation takes up, because, for small implementations, we can run many instances in parallel. Nonetheless, it is better to have a design that takes up twice as much area but runs twice as fast, than having two designs running in parallel. They both have the same throughput, but the former has a latency that is twice as good as the latter. Therefore, we should only go for a lower area implementation if the area scales down significantly faster than the NTT execution time.

To be able to judge the merits of the different implementations for their use in the Alveo U280, the following figure of merit was devised:

$$\frac{C * \max_i(\mu_i)}{f_c/(200\text{MHz})} \quad (3.17)$$

C is the number of clock cycles to perform 1 NTT. μ_i is the ratio of the resources one NTT implementation takes up (for example 48 BRAMS), to the total amount of BRAMS available on a U280 (1490 BRAM's). f_c is the clock frequency of the NTT implementation, which is normalized to 200 MHz in our figure of merit. The total amount of ~~BRAM~~ resources given by Table 3.3.

Our FOM for the different implementations is then given by Table 3.4. The figure of merit effectively measures how many cycles it would take on average to do 1 NTT if the hardware implementations scaled up perfectly, with the frequency being normalized to 200 MHz to give a fair comparison. A lower FOM is better.

We see that Roy's 2014 NTT implementation scores very well, but this is because it works for only half the ring size. T required area or required number of cycles will more than double with a doubling of the ring size because the NTT is a $O(\frac{n}{2} \log n)$ process. Mert's implementation doesn't just achieve fast execution, but is very area efficient as well.

Name	Year	$\log_2 Q$	N	Figure Of Merit
Wang	2020	28.35	1024	12.1
Fritzmann	2020	13.6	1024	181.8
Roy, Vercauteren	2014	13.6	512	3.3
Kuo	2017	13.6	1024	26.8
Hartshorn	2020	N/A	1024	20.5
Mert	2019	32	1024	35.4
Mert	2020	14	1024	61.0
Mert	2020	14	1024	11.7
Mert	2020	14	1024	10.3
Mert	2020	29	1024	12.9

TABLE 3.4: FOM's for the different implementations when considering the U280 as target (lower is better)

Chapter 4

Hardware Implementation

4.1 An Overview Of The Hardware Implementation

To understand the HW implementation of FHEW created for this thesis, it is vital to first understand the algorithm described in appendix A. We can split the flow of the algorithm into 2 parts: the outer RLWE loop, and the inner RGSW loop. The outer RLWE loop is fairly trivial to implement in HW, as it requires the calculation of one modular subtraction, for 1024 values. From both an area and performance perspective, it makes barely any difference whether one sends the $n(= 512)$ values of \mathbf{a} (each $\log_2(q)(= 9)$ bits) to the accelerator core, or the $n * d_r$ values of \mathbf{c} (each $\log_2(B_r) = \log_2(23) = 5$ bits). These values can be calculated in advance, and will only be used for determining which part of the secret key is used to rotate the RGSW Accumulator.

4.1.1 The inner control loop

The interesting part and most performance-sensitive part of the algorithm lies in the inner control loop. The inner loop can be broken up into 4 parts: Two INTT's, Signed Digit Decomposition, Eight ($= 2 * d_g$) NTT's and RGSW secret key multiplication. Counter-intuitively, the INTT is performed first. The reason for this is that we start with an accumulator on which the NTT has already been performed. We start this way because we wish to begin and end with ciphertext that allow for easy polynomial multiplication ("EVALUATION" state in the Palisade library). After performing this INTT to bring the accumulator back to the "normal" state("COEFFICIENT" state in the Palisade library), we perform the signed digit decompose. This is a simple algorithm that breaks the values of the Accumulator into values between $[-B_g/2, B_g/2)$. This creates $2 * d_g = 2 * \log_{B_g}(Q) = 2 * 4$ polynomials. As previously mentioned, we perform because doing the multiplication in this fashion decreases the noise growth. Thirdly, an NTT is performed to bring the polynomials back to the "EVALUATION" form. This allows us to in the last step multiply with the secret key, with a different secret key for both parts of the RGSW, resulting in $2 * 2 * d_g = (16)$ multiplications, and $2 * d_g$ additions for each new RGSW part, for a total of $2 * 2 * d_g = 16$ additions. (See Appendix A).

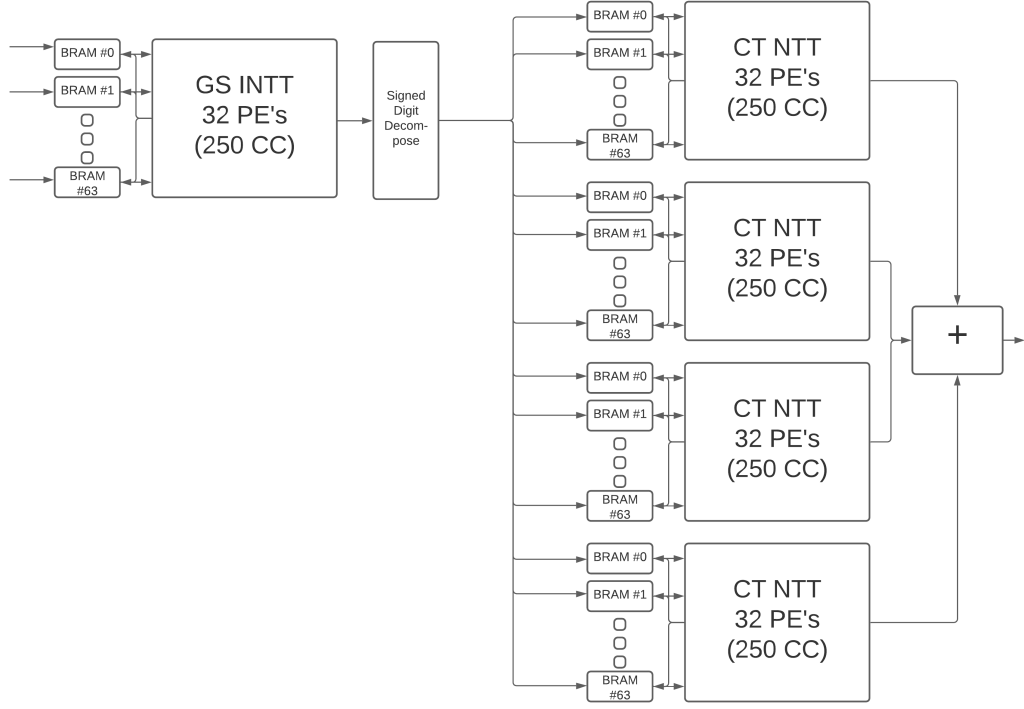


FIGURE 4.1: Diagram of the RGSW accumulator implementation

4.1.2 A Note On The Device Choice

The device targeted for this thesis was the ZCU102 prototype board, although an Alveo U280 data center accelerator card would be more suitable for applications using FHEW. The former was chosen because it simplified design and implementation. It turns out that the ZCU102 is powerful enough to allow for a significant, almost 40-fold speed-up, **but a fast DDR4 interface.**

4.2 Mert's NTT implementation

As we have seen, Mert's design was the only open-source HW implementation that worked fast enough to provide a functional NTT and INTT for the FHEW implementation. It can provide this because it is **so** parametric: if required, one could also create a very low-area implementation using the exact same implementation. However, this comes at a cost: the implementation is Gentleman Sande, and always takes normal-order inputs and outputs in bit-reversed order. (see **Chapter 2**). This means that applying the INTT on the result of an NTT operation will not give the original input to the NTT. First, one must bit-reverse the order of the outputs of the NTT, followed by a bit-reversal of the order of the outputs of the INTT. The negatively wrapped convolution described in **Chapter 2** is not supported either. Finally, while the NTT (or INTT) is executed in 250 clock cycles, no allowance is

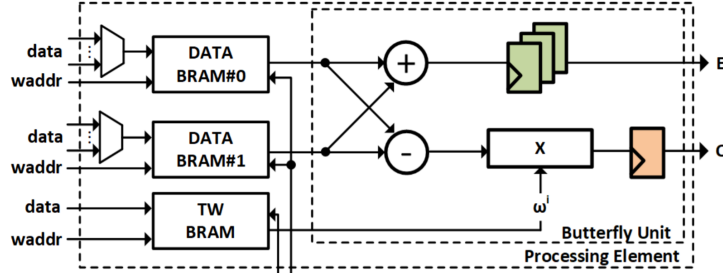


FIGURE 4.2: Mert's datapath [18]

made for outputting the values, which takes as many cycles as the ring size $N (= 1024)$ is large, and the output values are stored both at a different location within the same BRAM and in a different manner than the input values, meaning it is quite inconvenient to use the same design for both INTT and NTT.

4.2.1 Mert's design

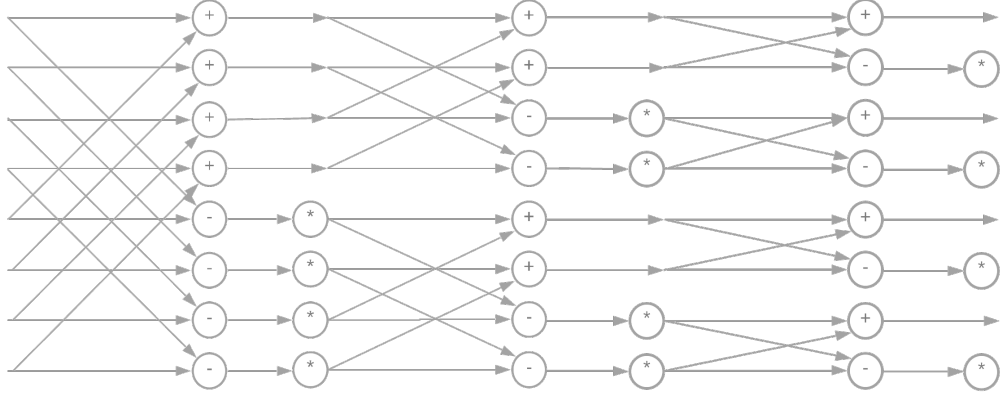
Before continuing to discuss this implementation, the basics of Mert's design must be considered. It allows for 3 parameters to be set: Ring Size, which for parameter set STD128 is 1024, Modulus Size, which is the size of Q and is 27 bit, and finally the number of processing elements. This number of processing elements determines the speed and (to a degree) the area of the design. For the purpose of this implementation, we chose a PE number of 32, but we kept everything parametric to the greatest extent possible. In hindsight, it would have been slightly more optimal to have chosen a PE number of 16 for the ZCU102 and a PE number of 64 or larger for the U280. The number of 32 was chosen because it was the largest parameter set mentioned in [18].

For each PE element, there are 2 BRAM's to hold the coefficients which will go through the GS datapath, and 1 BRAM to hold the twiddle factors (Figure 4.2). Under the original design, the twiddle factors have to be loaded in as well, but to reduce the complexity of interfacing, and to save some area, the BRAM's for the twiddle factors were changed to read initialize with the correct values in them.

As we have 64 BRAM's, there are logically $N/(2 * PE) = 16$ values in each BRAM. These values must be written back to new BRAM's after every stage, as a PE block can only use the 2 BRAM's adjacent to it to calculate the results for a given stage.

A stage is the execution of one butterfly on every pair of elements in the polynomial. In Figure 4.3, we can see an example of $\log_2(N) = 3$ stages with a ring size of $N = 8$. The twiddle factors which we use in the multiplier are given by:

$$\omega^{2^{(\text{stage}) * k}} \quad (4.1)$$

FIGURE 4.3: Mert's implementation of GS NTT and INTT for a ring size N of 8

with k being a counter modulo $2^{(2-\text{stage})}$.

4.3 Leveraging The NTT For FHEW

4.3.1 Modifying The GS INTT

As previously mentioned, Mert did not include support for *negatively wrapped convolution*. Pöppelmann [24] explains how the twiddle factors of an INTT can be modified to achieve this nonetheless. Because the twiddle factors are calculated in advance, which can be done in a scripting language such as python, the additional complexity of the twiddle factors has no impact on the hardware implementation.

It is worth noting that the GS INTT butterfly shown in Figure 1 of [24] is not the same GS INTT butterfly depicted in Figure 4.3. This is the result of [24] describing a GS INTT which takes a polynomial in bit-reversed ordering and returns it to normal ordering. Mert's GS INTT takes a polynomial with indices in normal order, and bit-reverses these indices. However, we can turn one into the other by bit-reversing the indices before and after running the INTT. This adds extra overhead, but because in Mert's design we have 32 different Processing Elements, with values spread out amongst them, a bit reversing step has to happen in any case in order to bring the values from the BRAM's of the NTT to the BRAM's of the INTT.

The advantage of this separate Index Reversal Block is that the twiddle factors calculated in [24] can be used in Mert without requiring changes to the addressing. The only significant modification (from an area perspective, not from a design complexity perspective) is the addition of the Index Reversal Block (Figure 4.4).

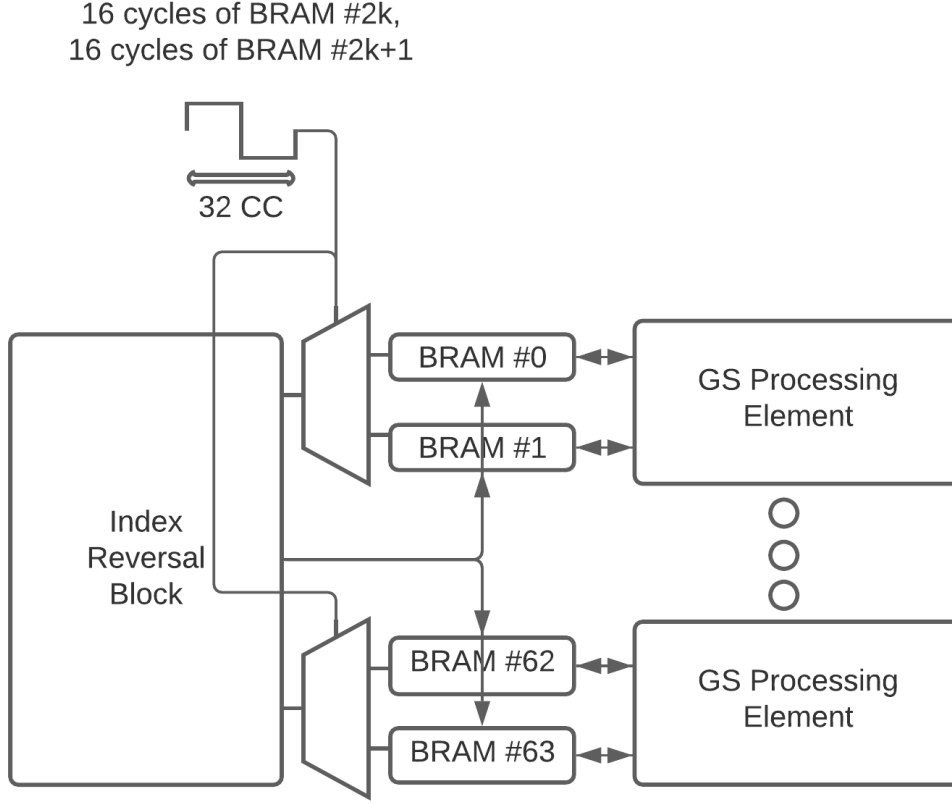


FIGURE 4.4: Bitreversal in the GS INTT

4.3.2 Overview Of Index Bit Reversal

To bit-reverse the indices, the greatest problem is that each BRAM can only read and write one value at a time. A very simple solution would be to read out the values one by one, but this would take $N = 1024$ clock cycles, making the bit reversal the bottleneck of the design. Clearly, as the maximum amount of data that can be read out in a clock cycle is two times the number of processing elements ($=64$), the minimum number of clock cycles for the index bit-reversal is $N/(2 * PE) = 16$. This would require 64, 64-element-wide multiplexers (with 1 element = 27 bits). Since the critical path should not be through a non-essential block like the index reversal block, and running the bit reversal in 32 cycles does not impose as significant delay as a reduction in clock speed, the final design uses 32 multiplexers, each 64 elements wide.

Now that the read limitation is solved, we face the write limitation: element 0 and element 2 are stored in BRAM 0 and BRAM 2 respectively. However, when they are bit reversed, they will become element 0 and element $256 = 0X\ 01000$

00000 = bitreverse(0x 00000 00000). However, both element 0 and element 256 will be stored in BRAM 0. As previously mentioned, we cannot write 2 elements to the same BRAM at the same time, as there is only 1 read port and 1 write port. Turning each BRAM into a set of $N/(2 * PE) = 16$ registers is not a solution as the BRAM's are also used in Mert to store temporary values. (Note that for the Ultrascale Architecture, the minimum number of elements for any BRAM with a data width of approximately 32 bits is 512 [36]). The other solution, first reading into a separate register, and then loading the values from the register into the BRAM, is inelegant, and requires a large number of registers.

The solution is to read out the BRAM's in such a way that no two values will write to the same BRAM in the same clock cycle. This solution requires little additional logic beyond the multiplexer but requires a complicated (but efficient) calculation of the read and write addresses. The first thing to consider is that in Mert's implementation, values are written differently to the input than they are found in the output. Consider "inputaddr" to be the (4-bit) address of an element in its BRAM, "inputBRAM" to be the (6-bit) index of the BRAM it is found in, and likewise for "outputaddr" and "outputBRAM".

To explain the scheme in this section, $y[x]$ is defined as a bit at position x (indexed from 0) of a value y . In this way, $y[x] = \lfloor \frac{y}{2^x} \rfloor \bmod 2$. Similarly, $y[b : a]$ is defined as a range of bits from position a to b , so that $y[b : a] = \lfloor \frac{y}{2^a} \rfloor \bmod 2^{b-a}$.

The element with index i is then found at :

$$\text{inputBRAM} = \{i[4 : 0], i[9]\} \quad (4.2)$$

$$\text{inputaddr} = i[8 : 5] \quad (4.3)$$

for values before an NTT or INTT commences and

$$\text{outputBRAM} = i[5 : 0] \quad (4.4)$$

$$\text{outputaddr} = i[9 : 6] \quad (4.5)$$

when the NTT or INTT has finished.

To solve the write limitation, we set:

$$\text{readaddr} = \text{BRAM}[5 : 2] + \text{cycle} \quad (4.6)$$

with cycle being the current clock cycle of the bit-reversal. In other words, it is a value which starts at 0 and goes to 31. By reading in this fashion, no two values will be read away to the same BRAM in the same clock cycle.

The bit-reversal block then guarantees through the synthesis of 64 multiplexers that each elements ends up in the correct BRAM in 32 cycles, and also provides the correct write address for each of the 32 elements as output. The bit-reversal step at the start and the bit-reversal step at the end, plus the integrated division by N which is required for the INTT add a total of 80 cycles to the INTT run time of 250 cycles.

4.3.3 Creating CT NTT From GS NTT

As can be seen in [24], the structure in Figure 4.3 is the same structure as for the CT NTT, both of which go from normal order to bit-reversed order, but with a different datapath. Changing the datapath in Figure 4.2 so that the multiplication comes before the addition and the subtraction converts the GS NTT to a CT NTT. Similar to the GS INTT, we can modify the weights to support the *negatively wrapped convolution*.

4.3.4 Non-NTT Components

The Signed Digit Decompose, as previously mentioned, can be implemented almost verbatim from the palisade library, and can be placed between the readout from the INTT BRAM's and the NTT BRAM's (we do not re-use the INTT BRAM because there is only one for every four NTT BRAM's required and it would complicate the design for only a small improvement in area).

The secret key multiplication requires either a separate block for multiplication and addition, or it can be integrated into CT NTT. A compromise between these 2 approaches was used, where the multiplication and first addition was performed in the CT NTT datapath, and the subsequent addition between the $d_g = 4$ CT NTT's is performed in a separate block. Integrating the addition into the CT NTT datapath would extend the execution time by 64 cycles, which is almost 10% of the total run time, while providing area savings equal to only 64 27 bit adders.

Because we run both parts of the RGSW accumulator in the same space, the BRAM's of the CT NTT are extended to fit in the results of the secret key multiplication with the first part until the second part is calculated and is summed together with the first part inside the CT NTT datapath (Figure 4.5).

4.4 Memory Interface

The current rate-limiting step of the design is the memory interface. In order to perform the multiplication with the secret key in time with the CT NTT, we must be able to fetch $N * d_r * 2 = 8096$ elements in the 640 cycles at 200 MHz it takes to complete half of one accumulator step. In other words, we would need a memory BW of

$$\frac{8096 * 200 * 20^6 * 27}{640} = 68.3\text{Gbps} \quad (4.7)$$

Storing the secret key on the ZCU102 FPGA would neither be possible, as it takes up a total of [19]:

$$4nNd_rB_rd_g\log_2(Q) = 10.4\text{Gigabits} \quad (4.8)$$

This exceeds the maximum of 32.1 Megabits that can be generated. [35]

what is written next depends on how the interfacing goes, so I'm holding off. I'm also still wondering how to split this with results and conclusion

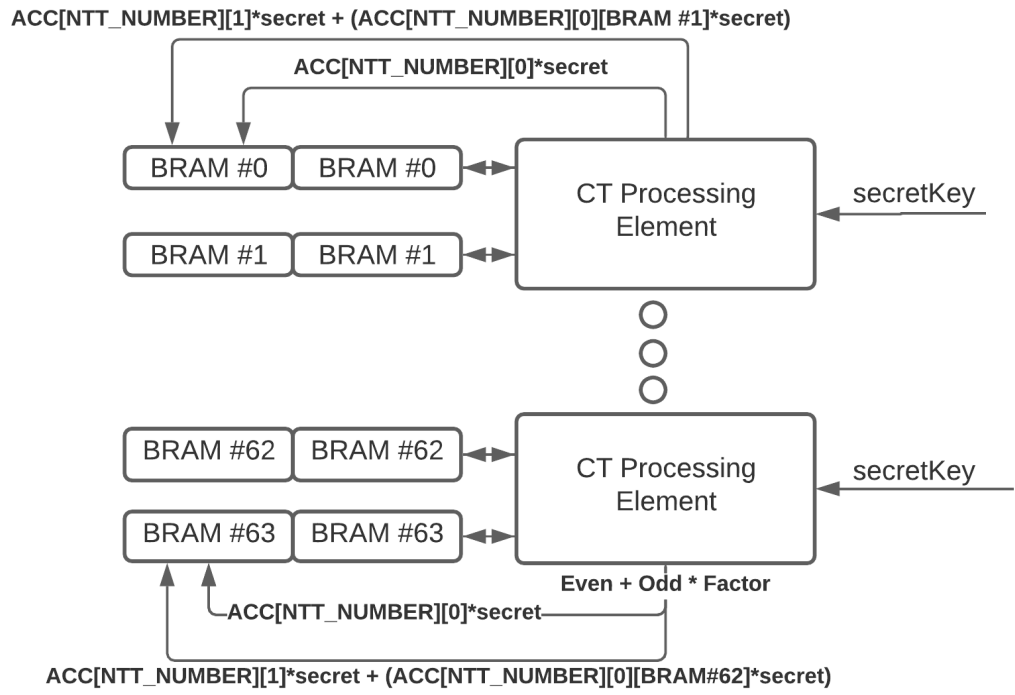


FIGURE 4.5: Secret Key accumulation integrated in CT

Chapter 5

Results & Conclusion

The final chapter contains the overall conclusion. It also contains suggestions for future work and industrial applications.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetur adipiscing elit. In

hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetur.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

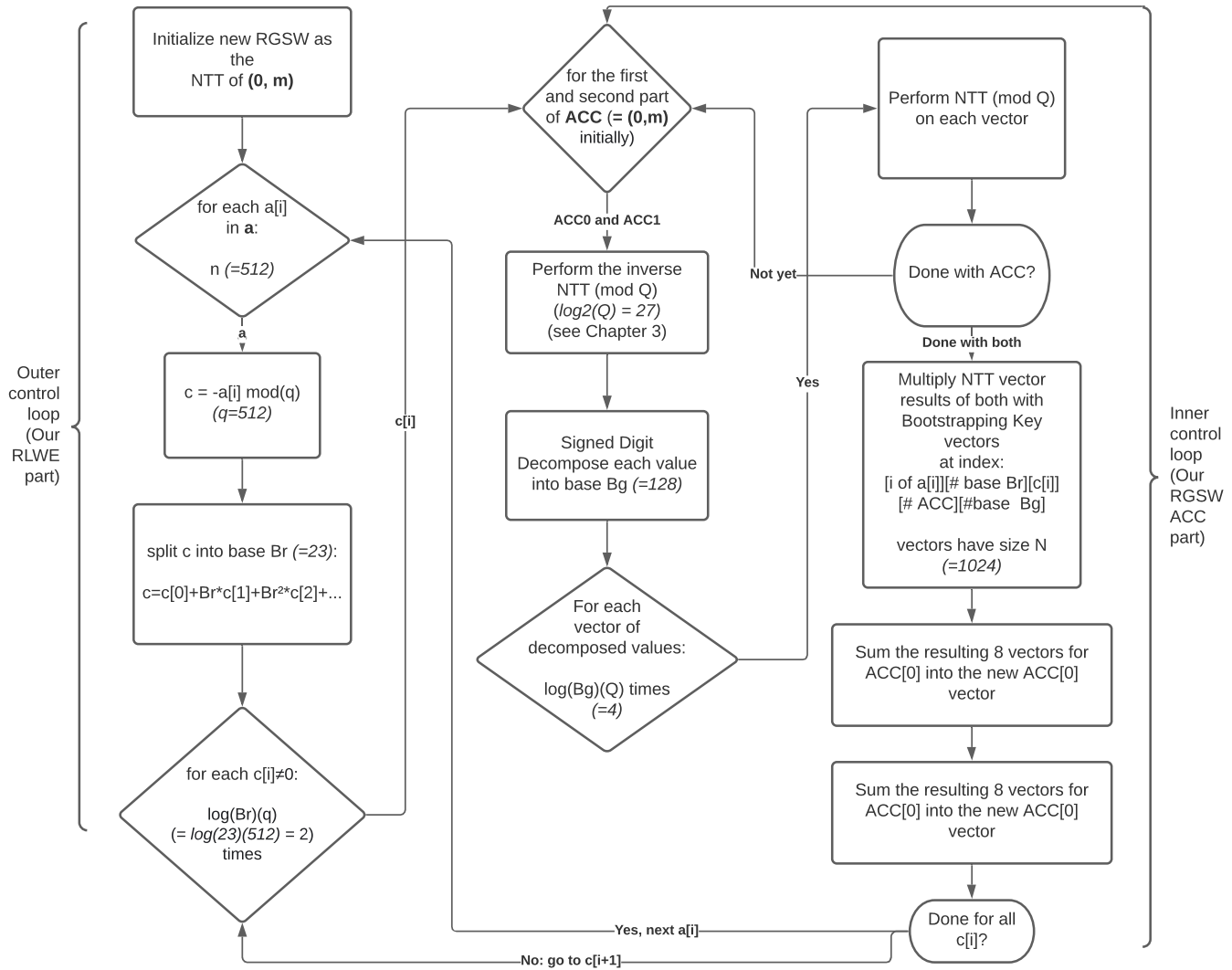
Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Appendices

Appendix A

FHEW Algorithm flowchart

This flowchart was created from the algorithm found in [19]. Note that the leftmost column of the flow chart represents the outer control loop, while the 2 rightmost columns represent the inner control loop. These control loops broadly correspond to the RLWE and the RGSW schemes respectively.



Bibliography

- [1] D. Adams. *The Hitchhiker's Guide to the Galaxy*. Del Rey (reprint), 1995. ISBN-13: 978-0345391803.
- [2] C. Bonte. Co6gc: Homomorphic encryption (part 1): Computing with secrets. URL: <https://www.esat.kuleuven.be/cosic/blog/co6gc-homomorphic-encryption-part-1-computing-with-secrets/>, last checked on 2020-03-25.
- [3] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Ttfe: Fast fully homomorphic encryption over the torus. Cryptology ePrint Archive, Report 2018/421, 2018. <https://eprint.iacr.org/2018/421>.
- [4] E. Chu, E. Chu, and A. George. *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*. Computational Mathematics Series. CRC-Press, 2000.
- [5] P. community. Palisade. <https://gitlab.com/palisade/palisade-release>, 2021.
- [6] Cryptography and M. Privacy Research Group. Microsoft seal. <https://github.com/microsoft/SEAL>, 2021.
- [7] L. Ducas and D. Micciancio. Fhew: Bootstrapping homomorphic encryption in less than a second. Cryptology ePrint Archive, Report 2014/816, 2014. <https://eprint.iacr.org/2014/816>.
- [8] L. Ducas and D. Micciancio. Fhew: Bootstrapping homomorphic encryption in less than a second. Cryptology ePrint Archive, Report 2014/816, 2014. <https://eprint.iacr.org/2014/816>.
- [9] L. Ducas and D. Micciancio. Fhew: Homomorphic encryption bootstrapping in less than a second1. University Lecture, 2015.
- [10] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.
- [11] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on “Program Generation, Optimization, and Platform Adaptation”.

- [12] Fritzmann. Risq-v. <https://gitlab.lrz.de/tueisec/post-quantum-crypto>, 2020.
- [13] T. Fritzmann, G. Sigl, and J. Sepúlveda. Risq-v: Tightly coupled risc-v accelerators for post-quantum cryptography. Cryptology ePrint Archive, Report 2020/446, 2020. <https://eprint.iacr.org/2020/446>.
- [14] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, page 169–178, New York, NY, USA, 2009. Association for Computing Machinery.
- [15] X. Lei, R. Guo, F. Zhang, L. Wang, R. Xu, and G. Qu. Accelerating homomorphic full adder based on fhw using multicore cpu and gpus. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 2508–2513, 2019.
- [16] P. Longa and M. Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. Cryptology ePrint Archive, Report 2016/504, 2016. <https://eprint.iacr.org/2016/504>.
- [17] Mert. Parametric ntt/intt hardware. <https://github.com/acmert/parametric-ntt>, 2020.
- [18] A. C. Mert, E. Karabulut, E. Ozturk, E. Savas, and A. Aysu. An extensive study of flexible design methods for the number theoretic transform. *IEEE Transactions on Computers*, pages 1–1, 2020.
- [19] D. Micciancio and Y. Polyakov. Bootstrapping in fhw-like cryptosystems. *IACR Cryptol. ePrint Arch.*, 2020:86, 2020.
- [20] V. Migliore, C. Seguin, M. M. Real, V. Lapotre, A. Tisserand, C. Fontaine, G. Gogniat, and R. Tessier. A high-speed accelerator for homomorphic encryption using the karatsuba algorithm. *ACM Trans. Embed. Comput. Syst.*, 16(5s), Sept. 2017.
- [21] NuCypher. Nufhe, a gpu-powered torus fhe implementation. URL: <https://nufhe.readthedocs.io/en/latest/>, last checked on 2020-03-26.
- [22] T. Oder and T. Güneysu. New hope. <https://www.seceng.ruhr-uni-bochum.de/research/publications/implementing-newhope-simple-key-exchange-low-cost/>, 2017.
- [23] T. Oder and T. Güneysu. Implementing the newhope-simple key exchange on low-cost fpgas. In T. Lange and O. Dunkelman, editors, *Progress in Cryptology – LATINCRYPT 2017*, pages 128–142, Cham, 2019. Springer International Publishing.

-
- [24] T. Pöppelmann, T. Oder, and T. Güneysu. High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers. Cryptology ePrint Archive, Report 2015/382, 2015. <https://eprint.iacr.org/2015/382>.
 - [25] D. Rijmenants. One-time pad. URL: <http://users.telenet.be/d.rijmenants/en/onetimepad.htm>, last checked on 2020-04-13.
 - [26] F. Roy. Heaws. <https://github.com/KULeuven-COSIC/HEAT>, 2020.
 - [27] S. Roy. Compact ring-lwe cryptoprocessor. <https://gitlab.lrz.de/tueisec/post-quantum-crypto>, 2014.
 - [28] S. Roy, F. Turan, K. Järvinen, F. Vercauteren, and I. Verbauwhede. Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data. In *2019 25TH IEEE INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE (HPCA)*, International Symposium on High-Performance Computer Architecture-Proceedings, pages 387–398, United States, 2019. IEEE. International symposium on high performance computer architecture, HPCA ; Conference date: 16-02-2019 Through 20-02-2019.
 - [29] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede. Compact ring-lwe cryptoprocessor. In L. Batina and M. Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, pages 371–391, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
 - [30] A. W. Services. User guide for linux instances: Fpga instances. URL: <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/accelerated-computing-instances.html#fpga-instances>, last checked on 2020-04-14.
 - [31] S. Sinha Roy, K. Järvinen, J. Vliegen, F. Vercauteren, and I. Verbauwhede. Hepcloud: An fpga-based multicore processor for fv somewhat homomorphic function evaluation. *IEEE Transactions on Computers*, 67(11):1637–1650, 2018.
 - [32] S. Sinha Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede. Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 387–398, 2019.
 - [33] Wang. qtesla. <https://caslab.csl.yale.edu/code/qtesla-hw-sw-platform/>, 2020.
 - [34] W. Wang, S. Tian, B. Jungk, N. Bindel, P. Longa, and J. Szefer. Parameterized hardware accelerators for lattice-based cryptography and their application to the hw/sw co-design of qtesla. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):269–306, Jun. 2020.
 - [35] Xilinx. *ZCU102 Evaluation Board*, 06 2019. PS-Side: DDR4 SODIMM Socket.

BIBLIOGRAPHY

- [36] Xilinx. *UltraScale Architecture Memory Resources*, 3 2021. Block Ram Summary.