KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

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
Dr. ir. Angshuman Karmakar

Academic year 2021 – 2022

Preface

I'd like to extend many thanks to Michiel van Beirendonck for the patient explanations, insights and corrections he made over the course of this year. His help accelerated my insight into the algorithm I set out to accelerate. Secondly, many thanks to Furkan Turan for explaining the many ways to interface, for his help with the hardware aspects of the thesis and for his general guidance on working with the relevant hardware. Many thanks go to Angshuman Karmakar and Jose Maria Bermudo Mera for providing feedback on this thesis, and to Prof. dr. ir. Ingrid Verbauwhede for providing me with this interesting topic.

Jonas Bertels

Contents

Abstract

Fully Homomorphic Encryption is encryption that allows operations to be done on encrypted data indefinitely. It is often considered the holy grail of cryptography for its wide-ranging applications. Both Fully Homomorphic and Somewhat Homomorphic schemes exist and are already in somewhat limited use, but the slow computation time, especially of Fully Homomorphic Encryption schemes, limits their adoption. More specifically, Fully Homomorphic Encryption schemes are often based on schemes such as Learning With Errors, where messages are masked by adding numerical noise. This noise grows as operations are performed, and bootstrapping removes it, but usually at a steep computational cost.
In this thesis, we accelerate FHEW bootstrapping in hardware. By modifying a fast hardware NTT implementation [23], we were able to increase the execution speed of FHEW bootstrapping by a factor of 7.5, provided that a fast interface to the FHEW bootstrapping key is available.

List of Abbreviations

Abbreviations

ASIC	Application Specific Integrated Circuit
BRAM	Block RAM
CC	Clock Cycle
CT	Cooley-Tukey
FFT	Fast Fourier Transform
FHEW	Fastest Homomorphic Encryption in the West
FPGA	Field Programmable Gate Array
FV	Fan-Vercauteren
GS	Gentleman-Sande
INTT	Inverse NTT
LWE	Learning With Errors
NTT	Number Theoretic Transform
(R)GSW	(Ring) Gentry-Sahai-Waters
TFHE	Fully Homomorphic Encryption over the Torus

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 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 booklet, which is called a "key" in cryptography. 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

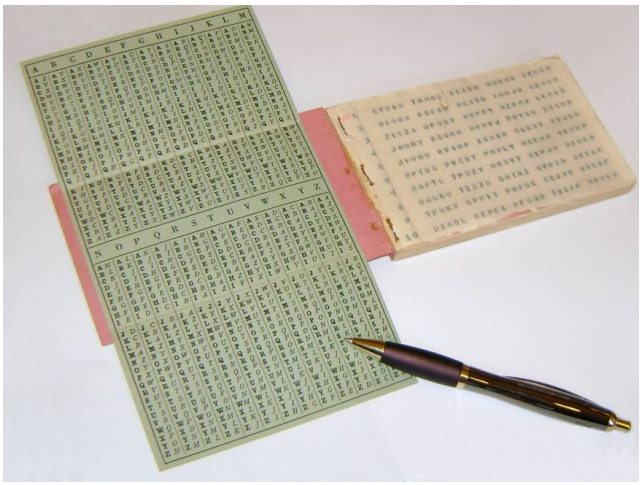


Figure 1.1: One Time Pad [32]

can only be used to *encrypt* the data. Once a potential sender (we'll call him Bob) 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 in 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 where we can still do operations on encrypted data. 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. 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 [1]. 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

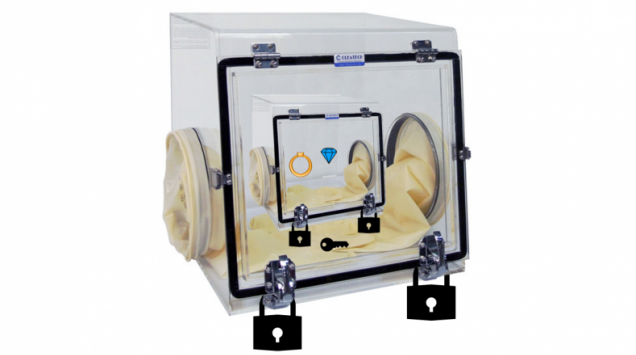


Figure 1.2: The jewel box [1]

(such as addition, multiplication or others) on the encrypted data without being able 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 three 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 (FV) [10] scheme, the Brakerski-Gentry-Vaikuntanathan [2] scheme and the Gentry-Sahai-Waters [15] 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 [7] and THFE [3] only allow for 1 gate (such as a AND/OR gate), and bootstrap immediately, but the bootstrapping happens relatively fast (≈100ms)[24]. In this master thesis, one of these third-generation schemes is accelerated, 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: Leveled Homomorphic Encryption

FV has been the subject of hardware acceleration by multiple groups [25, 38, 35]. 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 [35]. 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(R) Core(TM) i7-9700 CPU [24]. 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, the programmer only has to worry about building their 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 an 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 higher 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 FPGAs is that they can provide a speed-up of a factor

10 to a factor 100 for a scheme like FHEW.[35] 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, FPGAs 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's implementation might one day end up in a consumer device, such as a laptop, a 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 FPGAs available [37], 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

We define $\mathbb{Z}_q$ as the ring of integers modulo q, and $\mathbb{Z}_q^n$ as n elements x so that $x \in \mathbb{Z}_q$. Lattice-based cryptography uses polynomial rings for various optimizations. The polynomials rings we use are defined as $R = \mathbb{Z}[x]/f(x)$. In practice, a popular choice for $f(x)$ is the $2d$-th cyclotomic polynomial, 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 $d = 2^m$.

We consider the discrete Gaussian sampling function $D_{\mathbb{Z},\sigma}$. From this Discrete Gaussian function, we can sample values $\chi = D_{\mathbb{Z},\sigma}^d$.

We define $\odot$ as the element-wise multiplication between two different vectors or polynomials.

2.2 Learning With Errors

In this section, we consider LWE, which is the basis for many homomorphic encryption schemes [31].

2.2.1 Definition of LWE

Three parameters are used : the dimension n, the message modulus $t \geq 2$ and the ciphertext modulus q. We also define a randomized rounding function $\chi : \mathbb{R} \to \mathbb{Z}$ with error with e being a randomly sampled error term from χ [8]. The secret key $\mathbf{s}$ is a vector and the secret key space is $\mathbb{Z}_q^n$, and the secret key may be chosen uniformly from this space.

We now define the LWE distribution (a, b) by defining $(a, b) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$, with a being a randomly sampled but known vector, and:

$$b = a \odot s + e \tag{2.1}$$

There are primarily two problems based on LWE [29]: search-LWE and decision-LWE. From k independent samples $(a_i, b_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$, it is not computationally feasible to find s. This is the search-LWE problem, which is thus hard. If we consider the problem from the case of the server: we cannot reverse engineer (a, b) into s, even if we have a very large quantity of a's and b's. The decision-LWE problem attempts to distinguish between a uniformly sampled vector and a valid LWE-distribution (a, b) given k independent samples $(a_i, b_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ which can be either. This problem is also hard. A symmetric encryption scheme can then be based on these problems.

2.2.2 LWE Symmetric Encryption Scheme

Our actual message m is a bit, not an integer. To encrypt a message m from the plaintext space $\mathbb{Z}_t$, we first encode m into the ciphertext space $\mathbb{Z}_q$ as $\tilde{m} = \frac{q}{t} * m$ with q as always our modulus (=512, see parameter set STD128 in Table 2.5 [24]) and t being dependent on which LWE scheme we use (we therefore use the notation LWE_s^t).

Using these definitions for the message, we define the LWE encryption of a message bit m, so that $\frac{q*m}{t} = \tilde{m} \in \mathbb{Z}_q^n$ with a secret key s and a parameter t as: [24]

$$\text{LWE}_s^t(\tilde{m}, E) = \text{LWE}_s^t(\frac{q * m}{t}, E) = (a, a \odot s + e + \tilde{m}) = (a, b + \tilde{m}) \tag{2.2}$$

with $a \leftarrow \mathbb{Z}_q^n$ randomly sampled and $e \leftarrow \chi < E$ with χ as previously defined, a discrete Gaussian distribution with σ as parameter. The secret key s can be sampled from $\mathbb{Z}_q^n$ or from a smaller distribution. For the purpose of this work, we focus on Gaussian (i.e., the most general) secret keys.

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:

$$\text{LWE}_s^{-1}(a, b + \tilde{m}) = b + \tilde{m} - a * s = \tilde{m} + e \tag{2.3}$$

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

2.2.3 RLWE

We also define RLWE, for a message $\tilde{m} \in R_q$ under a secret key $s \in R$ as [24]:

$$\text{RLWE}_s(\tilde{m}) = (a, as + e + \tilde{m}) \tag{2.4}$$

with $a \leftarrow R_q$ (chosen uniformly at random) and $e \leftarrow \chi$.

We define RLWE because it is more efficient [21]. Lyubashevsky, Peikert and Regev [21] list two reasons. First of all, each product in RLWE contains d "simultaneously pseudorandom values" over $\mathbb{Z}_q$, as opposed to the one scalar given

in LWE. However, because the product is the result of polynomial multiplication, the computational cost is only on the order of $O(n \log(n))$. Secondly, a sample $(a, as + e + \tilde{m}) \in R_q \times R_q$ from RLWE can replace d samples of $(a, b) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$.

2.2.4 Homomorphic Addition and Multiplication

We briefly consider a simplified Fan-Vercauteren scheme as an example of a homomorphic scheme which supports addition and multiplication and is created using RLWE [10].

We again form the $\text{RLWE}_s(\tilde{m})$ ciphertext as:

$$ct = \text{RLWE}_s(\tilde{m}) = (a, b = a \cdot s + e + \tilde{m}) \tag{2.5}$$

with

$$a \leftarrow R_q, s \leftarrow R_q \text{ and } e \leftarrow \chi \tag{2.6}$$

We can add two ciphertexts encrypted under the same secret key:

$$\begin{aligned} ct' = HomAdd(ct_0, ct_1) &= (ct_0[0] + ct_1[0], ct_0[1] + ct_1[1]) \\ &= (a_0 + a_1, (a_0 + a_1) \cdot s + (e_0 + e_1) + (\tilde{m}_0 + \tilde{m}_1)) \end{aligned} \tag{2.7}$$

Such that $\text{RLWE}_s^{-1}(ct') = (\tilde{m}_0 + \tilde{m}_1) + (e_0 + e_1)$.

When we multiply two ciphertexts, we take their tensor product.

$$ct' = HomMul(ct_0, ct_1) = (ct_0[0] \cdot ct_1[0], ct_0[0] \cdot ct_1[1] + ct_0[1] \cdot ct_1[0], ct_0[1] \cdot ct_1[1]) \tag{2.8}$$

It can then be shown [10] that ct' is a ciphertext encrypting the product of $\tilde{m}_0$ and $\tilde{m}_1$. Note that because $\tilde{m} = \frac{q}{t} * m$, our product $\tilde{m}_0 \cdot \tilde{m}_1 = q^2/t^2 \cdot m_0 m_1$. Each multiplication will thus increase the size of the message we are encrypting. We would prefer to encrypt $q/t \cdot m_0 m_1$. Secondly, our ciphertext ct' has three elements as opposed to the original two elements of ct_0 and ct_1. To solve these issues, second-generation schemes like FV have a *rescaling* and *relinearisation* option, respectively. FHEW takes a different approach, which we will focus on in this thesis.

2.2.5 RLWE' and RGSW

We now introduce two more definitions based on RLWE, which we need when discussing FHEW. RLWE' is defined as [8] [24]:

$$\text{RLWE}'_s(m) = (\text{RLWE}_s(m), \text{RLWE}_s(B_g * m), \text{RLWE}_s(B_g^2 * m), ..., \text{RLWE}_s(B_g^{k-1} * m)) \tag{2.9}$$

with $B_g^k = q$. (In our parameter set, $B_g = 128 = 2^7$). The reason for this definition is that we want a system that creates as little noise as possible when operations are

completed. By breaking RGSW into bases, less noise is generated ([24], see section 5.1). Multiplications with a $d \in R_q$ so that $d = \sum_i B_g^i d_i$ are defined as:

$$(\odot) : R \times \text{RLWE}' \to \text{RLWE} : d \in R, \mathbf{c} \in \text{RLWE}' : d \odot (\mathbf{c}_0, ..., \mathbf{c}_{k-1}) = \sum_i d_i \mathbf{c}_i \quad (2.10)$$

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

$$(\odot') : R \times \text{RLWE}' \to \text{RLWE}' : d \odot \mathbf{C} = ((B_g^0 * d) \odot \mathbf{C}, (B_g^1 * d) \odot \mathbf{C}, ..., (B_g^{k-1} * d) \odot \mathbf{C}) \quad (2.11)$$

And finally, using these definitions, the RGSW scheme can also be defined, which will allow multiplication of ciphertexts. We need this ability if we want to run the RLWE decryption while everything is encrypted under RGSW (this will be explained in-depth in the bootstrapping section).

$$\text{RGSW}_s(m) = (\text{RLWE}'_s(-s * m), \text{RLWE}'_s(m)) \quad (2.12)$$

To be able to use our secret key s encrypted under RGSW, we can multiply an RGSW ciphertext $\text{RGSW}(m_1) = (\mathbf{c}, \mathbf{c'})$ with an 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 [24]:

$$(a, b) \diamond (\mathbf{c}, \mathbf{c'}) = \langle (a, b), (\mathbf{c}, \mathbf{c'}) \rangle = a \odot \mathbf{c} + b \odot \mathbf{c}' \quad (2.13)$$

$$= a \odot \text{RLWE'}_s(-s * m_1) + b \odot \text{RLWE'}_s(m_1) \quad (2.14)$$

$$= \text{RLWE}_s((b - a * s) * m_1) = \text{RLWE}_s((m_0 + e_0) * m_1) \quad (2.15)$$

This $\text{RGSW} \times \text{RLWE}' \to \text{RLWE}$ can be turned into a $\text{RGSW} \times \text{RLWE}' \to \text{RLWE}'$. In Chapter 4 in Algorithm 3, we will see how this multiplication is implemented. We will also see how the $\text{RGSW}_s(m) = (\text{RLWE}'_s(-s * m), \text{RLWE}'_s(m))$ is created through Algorithm 4. Briefly, each coefficient of the polynomial is broken down through a signed digit decomposition. Then a multiplication with an RLWE' vector occurs. As mentioned at the start of the section, using RLWE' instead of RLWE for the multiplication reduces the generated noise. Once the multiplication is completed, the results are summed.

2.3 FHEW (Fastest Homomorphic Encryption in the West)

We now consider, FHEW, the algorithm for which we design a hardware accelerator. FHEW uses LWE_s^2 (see equation 2.2), where our original 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 (see equation 2.2). 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

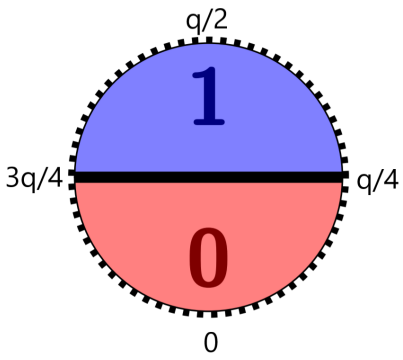


FIGURE 2.1: Binary Message representation of $m * \frac{q}{2} + e$ in $\text{LWE}_s^2(m, \frac{q}{4})$ (taken from[9])

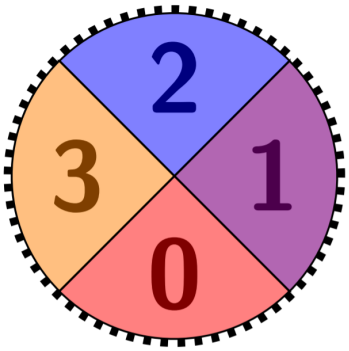


FIGURE 2.2: Messages in $\text{LWE}_s^4(m, \frac{q}{8})$ [9]

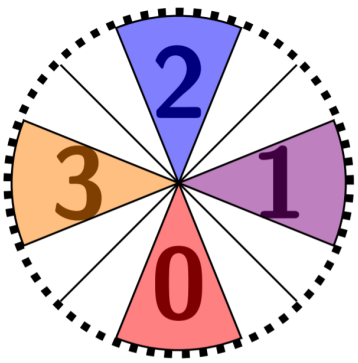


FIGURE 2.3: Messages in $\text{LWE}_s^4(m, \frac{q}{16})$ with smaller error [9]

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

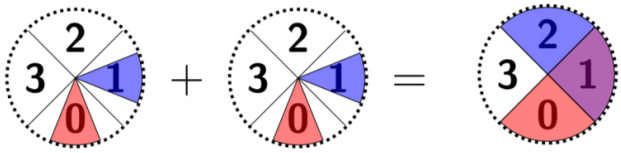


Figure 2.4: Result of the addition [9]

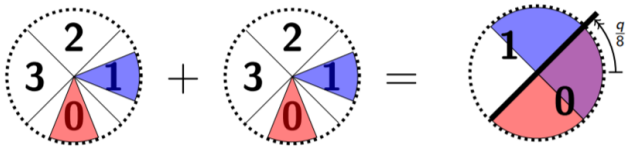


Figure 2.5: Creating an AND gate [9]

$\pm\frac{q}{8}(= 64)$. This is written as $\text{LWE}_s^4(\tilde{m}, q/8)$. The error can be reduced further, as can been 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 $\text{LWE}_s^4(\tilde{m}, q/16)$.

We can use $\text{LWE}_s^4(\tilde{m}, q/16)$ messages for addition (Figure 2.4) (again, see equation 2.2). After addition, the result now has more error than the beginning, because the noise adds up.

Our interest lies in making a functional binary gate, such as a NAND gate. The NAND gate is a universal gate, from which arbitrary circuits can be created. 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 right to be 0, we have an AND gate (Figure 2.5 and Table 2.1). To make this a 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 2.6).

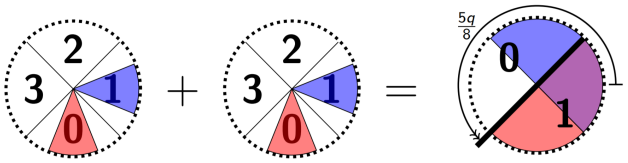


FIGURE 2.6: The final NAND gate [9]

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 two $\text{LWE}_s^4(\tilde{m}, q/16)$ inputs, and now we have one $\text{LWE}_s^2(\tilde{m}, q/8)$ 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.

2.4 Bootstrapping

As mentioned in the introduction, the FHEW scheme is motivated by the need for fast bootstrapping [8]. This fast bootstrapping is necessary for "practical" Fully Homomorphic Encryption.

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, the 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 $\text{LWE}_s^2(\tilde{m}, q/8)$). 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 $q/16$). This (larger) box will be provided by RGSW. 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.

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 Bootstrapping Algorithm

We now turn our attention to the bootstrapping process itself, first formally using the algorithm described in [24]. The bootstrapping consists of 3 steps: Initialization (see Algorithm 1), Accumulation (Algorithm 2) and Extraction (which simply consists of

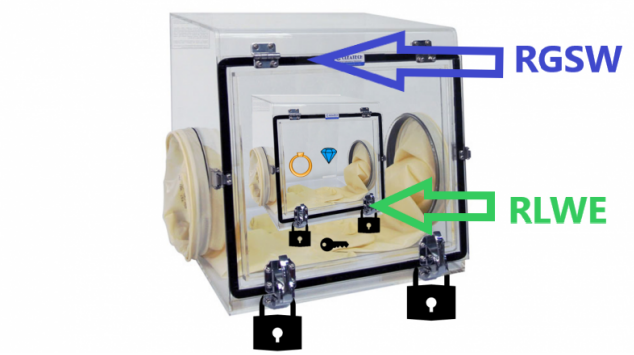


Figure 2.7: The LWE ciphertext is encrypted under an RGSW ciphertext so that it can be decrypted using LWE secret keys, which themselves have been encrypted under RGSW

taking the first value of the second part of the RGSW result).

Algorithm 1: Initialization[24]

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, ...q/2 − 1 do
        m_i = f(b − i);
        Note that f is our lookup table
    end
    m = Σ_{i<q/2} m_i * X^i = (m_0, m_1, ..., m_{q/2−1});
    Return (0, m);
end
```

Initialisation

We implement the rounding function and NAND gate described in the previous section as a lookup table. In a lookup table, the result of an input i is stored at index i and executing the function thus simply requires looking at the value at index i. For example, we can turn a NAND gate into a lookup table by first calculating the sum of the 2 inputs, a and b. For index a+b, we then have the results in Table 2.2.

In FHEW, a polynomial is used as a lookup 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 $n = 8$.

Algorithm 2: A Single Accumulation Step[24]

Data: **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: **ACC**, updated with $a_i * s_i$

```
begin
    for j = 0, 1, ... log_{B_r}(q) − 1 do
        c_j = ⌊c/B_r^j⌋ mod B;
        if c_j > 0 then
            ACC ← ACC ⋄ Z_{j,c_j};
        end
    end
    Return 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

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

Our lookup 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 lookup table is to rotate the indices. With rotation, we mean that we change the lookup table so for index i

$$i \rightarrow i + r \pmod{\text{LUT size}} \tag{2.16}$$

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

When we initialize our lookup 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 an RLWE ciphertext, as described in Algorithm 1. That the LUT is also an RLWE ciphertext is important in the accumulation step.

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 $a_i * s_i = 2$

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

Table 2.5: Parameters for STD128

Accumulation

Another way to look at the LUT is as a noiseless RLWE encryption of a polynomial. In this case the coefficients are the results of the rounding+NAND function (Table 2.3). A rotation of the lookup 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 lookup table. When we rotate the lookup table leftward, from the 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 RGSW $\odot'$ RLWE$'$ $\rightarrow$ RLWE$'$ multiplications to perform the decryption $b - \mathbf{a} * \mathbf{s}$. Table 2.4 shows one rotation through a lookup 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 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 in Table 2.5.

2.5 Notes on Previous Optimisations

The version that is accelerated in this thesis is an optimized variant ([24]) 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 speed-up of about 2.2 using CUDA on the 2015 version of FHEW [18].

Another optimisation paper introduced NuFHE [26], which uses another third-generation scheme called TFHE with GPU acceleration and succeeded in executing about a 100 times faster than the original scheme, 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. 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.

2.6 Conclusion

By understanding the working of the FHEW scheme, it is possible to understand the bottleneck operations, and from this gain understanding of how to best accelerate the scheme. The accumulation, which consists of n RGSW $\times$ RLWE$'$ $\rightarrow$ RLWE multiplications, is the bottleneck of the FHEW algorithm. Therefore, it is best to focus on the acceleration of this part. More specifically, an efficient multiplication module will be key to accelerating FHEW. An important algorithm for efficient polynomial multiplication, the Number Theoretic Transform, will be explained in the next chapter.

Chapter 3

The Number Theoretic Transform

3.1 Introduction to Number Theoretic Transform

The Number Theoretic Transform [4] [20] [30] is an algorithm for performing polynomial (ring) multiplication. Multiplying two polynomials is effectively a convolution between two coefficient vectors. Consider that we take a polynomial a and a polynomial b with coefficients a_0, a_1, ..., a_N, b_0, b_1, ..., b_N. We can define these as a vector a and b so that $a[i] = a_i$ and $b[i] = b_i$. We define a vector c so that:

$$c[i] = \sum_{k=0}^{N-1} a_k * b_{i-k} \qquad \text{where } b_l = 0 \text{ for all } 0 \leq l < N \tag{3.1}$$

If we now define a polynomial c so that $c_i = c[i]$, we see that this polynomial is the product of polynomial multiplication between a and b. We also see that the process for calculating the full multiplication will take N^2 multiplications.
To reduce the cost of polynomial multiplication, we can use the Discrete Fourier Transform, which has the following property:

$$\mathrm{FT}(c) = \mathrm{FT}(a) \odot \mathrm{FT}(b) \tag{3.2}$$

where $\odot$ is the pointwise multiplication.

Most variants of NTT are almost identical to the Fast Fourier Transform, which is a form of the Fourier Transform that executes quickly (in $\mathrm{O}(\frac{N}{2} \log N)$ time). The idea of using the algorithm is thus 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 $n = 512$ times, it is obvious why high performance for the NTT implementation is necessary.

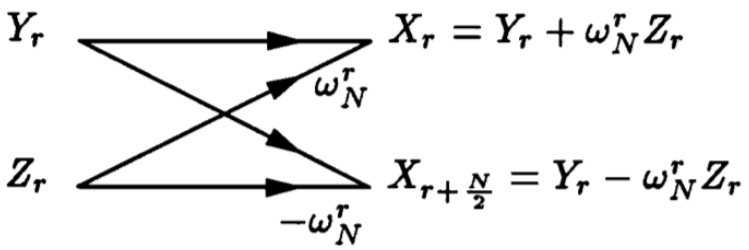


Figure 3.1: Cooley Tukey FFT/NTT butterfly [4]

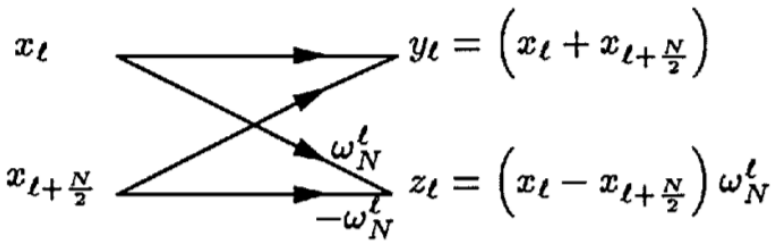


Figure 3.2: Gentleman Sande FFT/NTT butterfly [4]

3.1.1 Mathematics Behind NTT

We can see the NTT as a Discrete Fourier Transform with the twiddle factor changed from a complex number so that $\omega^N = 1$ to a $\omega \in \mathbb{Z}_Q$ with again $\omega^n = 1$, where ω is a primitive root of unity ($\omega^{N/2} = -1$)
The discrete Fourier is given by the expression ([4], equation 3.1).

$$X_r = \sum_{l=0}^{N-1} x_l \omega^{rl} \tag{3.3}$$

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 N-th root of unity as $\omega_N = \omega$, because as previously mentioned, $\omega_N^N = 1$.

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

The Cooley-Tukey Butterfly

First we split equation 3.3 into an even and an odd part:

$$X_r = \sum_{k=0}^{\frac{N}{2}-1} x_{2k}\omega_N^{2rk} + \omega_N^r \sum_{k=0}^{\frac{N}{2}-1} x_{2k+1}\omega_N^{2rk} \tag{3.4}$$

Let us define the following:

$$Y_r = \sum_{k=0}^{\frac{N}{2}-1} y_k \omega_{\frac{N}{2}}^{rk} \ , \ y_k = x_{2k} \tag{3.5}$$

$$Z_r = \sum_{k=0}^{\frac{N}{2}-1} z_k \omega_{\frac{N}{2}}^{rk} \ , \ z_k = x_{2k+1} \tag{3.6}$$

As such we can write the equation as:

$$X_r = Y_r + \omega_N^r * Z_r \tag{3.7}$$

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 $\omega_{\frac{N}{2}}^{r+N/2} = \omega_{\frac{N}{2}}^r$ so that $Y_{r+N/2} = Y_r$ and $Z_{r+N/2} = Z_r$,we can write the following:

$$X_{r+N/2} = Y_{r+N/2} + \omega_N^{r+N/2} * Z_{r+N/2} = Y_r - \omega_N^r * Z_r \tag{3.8}$$

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

The Gentleman-Sande Butterfly

The Gentleman-Sande works by splitting 3.3 into a first half and second half:

$$X_r = \sum_{k=0}^{\frac{N}{2}-1} x_k \omega_N^{rk} + \sum_{k=\frac{N}{2}}^{N-1} \omega_N^{rk} = \sum_{k=0}^{\frac{N}{2}-1} (x_k + x_{k+\frac{N}{2}} \omega_N^{r\frac{N}{2}}) \omega_N^{rk} \tag{3.9}$$

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:

$$X_{2l} = \sum_{k=0}^{\frac{N}{2}-1} (x_l + x_{l+\frac{N}{2}}) \omega_N^{2kl} \tag{3.10}$$

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:

$$X_{2l+1} = \sum_{k=0}^{\frac{N}{2}-1} (x_l - x_{l+\frac{N}{2}}) \omega_N^{(2l+1)k} \tag{3.11}$$

3.1.2 The Polynomial Multiplication

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

$$X_r = N^{-1} \sum_{l=0}^{N-1} x_l \omega^{-rl} \tag{3.12}$$

Therefore, the INTT can be performed as an NTT by simply changing the ω's 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.

3.1.3 Negative Wrapped Convolution

In a standard polynomial multiplication of two polynomials with ring size N using NTT, we use two polynomials of ring size $2N$ with half of the inputs being zeros. This way we can do our coefficient-wise multiplication after our NTT, which is followed by a reduction step of the polynomial modulo $X^N + 1$ [30].
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 *negative wrapped convolution* [30], where we use a ψ so that $\psi^2 = \omega \pmod{q}$, and we calculate

$$\hat{a} = (a[0], a[1] * \psi, a[2] * \psi^2, ..., a[n-1] * \psi^{n-1}) \tag{3.13}$$

and

$$\hat{b} = (b[0], b[1] * \psi, b[2] * \psi^2, ..., b[n-1] * \psi^{n-1}) \tag{3.14}$$

We can then calculate

$$\hat{c} = \mathbf{INTT}(\mathbf{NTT}(\hat{a}) \circ \mathbf{NTT}(\hat{b})) \tag{3.15}$$

and it turns out that for

$$c = \mathbf{a} * \mathbf{b} \pmod{X^N + 1} \tag{3.16}$$

we have

$$c = (\hat{c}[0], \hat{c}[1] * \psi^{-1}, \hat{c}[2] * \psi^{-2}, ..., \hat{c}[n-1] * \psi^{-(n-1)}) \tag{3.17}$$

This multiplication with ψ or ψ^{-1} can be integrated in our multiplication with the ω factors [30]. 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 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 make their source code available. Table 3.1 lists open-source implementations (in green), but closed source implementations (in red) can also be interesting as they often describe the optimisations that make them successful.

The table was compiled to find the NTT implementation that would best fill the needs of a FHEW hardware accelerator. For each design, it lists the platform on which it was created (we are designing for the UltraScale family, but most should be interchangeable). It also lists either the area the NTT implementation took up or the area the full polynomial multiplication implementation took up. Likewise, the number of clock cycles required to complete either, and the clock frequency are listed. Finally, we list the parameters for which the NTT was designed. The design parameters of our implementation are $N = 1024$ and $Q = 27$. NTT implementations that don't exactly match our modulus size aren't automatically excluded, as the modulus size is usually parametric, but using a design with a different ring size would be more difficult.

3.2.1 NTT Execution Time

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 et al.'s [23] 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 only one NTT core is used, this means our run time would be (for the 10240 NTT's that must be executed):

$$\mathbf{T}_{\text{run}} = \frac{250 * 10240}{125 * 10^6} = 20.5\mathbf{ms} \tag{3.18}$$

This run-time is in the order to be fast enough to justify hardware acceleration. Moreover, the design is highly parametric, so the run-time can be decreased further at the cost of increasing area.

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 et al.	2017	Arktix-7	N/A	35840	125			1024	13.6
Roy et al.	2018	UltraScale		87582	200	64K/25K/0.4K/0.2K	5349567	4096	180
Wang et al.	2020	Arktix-7			126	1735/758/6/0	11,455	1024	28.35
Fritzmann et al.	2020	Arktix-7	2908/170/9/0	18537	45.47		36,102	1024	13.6
Roy et al.	2014	Virtex 6	1536/953/1/3	2304	278			512	13.6
Kuo et al.	2017	Artix-7	2832/1381/8/10	2616	131			1024	13.6
Rashmi et al.	2019	Artix-7			0.1	no area for NTT	33792	1024	13.6
Du et al.	2016	Spartan-6			241	251slice/1/4.5	11826	1024	16
Hartshorn et al.	2020	Virtex 7	9233/8585/128/8	1294	179		1936	1024	N/A
Mert et al.	2019	Virtex 7			211.5	1208/556/14/14	7970	1024	32
Mert et al.	2020	Virtex-7	575/0/3/11	5160	125			1024	14
Mert et al.	2020	Virtex-7	2584/0/24/16	680	125			1024	14
Mert et al.	2020	Virtex-7	17188/0/96/48	200	125			1024	14
Mert et al.	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 et al.	[28]	[27]
Roy et al. (2018)	[39]	[33]
Wang et al.	[41]	[40]
Fritzmann et al.	[13]	[12]
Roy et al. (2014)	[36]	[34]
Mert et al.	[23]	[22]

TABLE 3.2: Paper and code Source for the HW implementations

If we used the next fastest implementation instead (Liu et al's [19], which is closed source), it would take 1294 clock cycles at 179 MHz to perform one NTT. For 10240 NTT's, this means:

$$\mathbf{T}_{\text{run}} = \frac{1294 * 10240}{179 * 10^6} = 74 \textbf{ ms} \tag{3.19}$$

If Liu et al's implementation were to be used, the final execution time would be less than half of 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 Alveo U280 FPGA, the primary concern when accelerating FHEW is the execution time, not the area taken up by the design.

In our previous calculation, we have assumed all NTT's are done in series. During FHEW, we are able to perform certain NTT's (up to 2 INTT's and $2 * d_r (= 8)$ forward NTT's) in parallel. Taking full advantage of this with Mert's implementation would give an execution time of 8.2 ms, which is almost 20 times faster than the 137 ms time on an Intel i7. Moreover, Mert's design is open source, and the execution time and area listed in Table 3.1 are given for a modulus size close to the one we are targeting.

3.2.2 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

Resource Name	LUT	FF	DSP	BRAM
Amount	1304000	2607000	9024	1490

Table 3.3: Resources for the U280

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)

Alveo U280, the following figure of merit was devised:

$$\frac{C * \max_i(\mu_i)}{f_c/(200\text{MHz})} \tag{3.20}$$

Here, 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 resources given in Table 3.3.

Our FOM for the different implementations is then given in 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 et al's 2014 NTT implementation scores very well, but this is because it works for only half the ring size. The 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 et al's implementation doesn't just achieve fast execution but is very area-efficient as well.

3.3 Conclusion

As the efficiency of the multiplication is key to the efficiency of the overall hardware implementation, a good understanding of the Number Theoretic Transform and how

it can be used in fast polynomial multiplication is important. Using the information, an NTT hardware implementation that would succeed in accelerating the FHEW accumulation step was chosen, and used in our hardware implementation.

Chapter 4

Hardware Implementation

The current FHEW (software) implementation provided in the PALISADE library can execute one binary gate in approximately 140 ms [24]. For many applications, 7 binary gates per second is too slow to be practical, especially when an entire CPU must be dedicated to one binary gate. In comparison, ENIAC, a very early digital computer built in 1945, could do 5000 additions or subtractions per second [16]. This motivates the hardware acceleration done in this thesis.

4.1 Hardware Acceleration

Hardware Acceleration is the creation or modification of a special-purpose circuit for the purpose of reducing the execution time of a given algorithm. The special-purpose circuit used in this thesis is an FPGA. We target Xilinx's Zynq Ultrascale+ FPGA found on Xilinx's ZCU102 Development Board. The algorithm we specifically accelerate in hardware is the accumulation step given by Algorithm 2. The rest of the FHEW scheme is then performed in "software". Just as the exact "hardware" targeted differs on the device, so does the "software". On the ZCU102, the "software" runs on the Arm Cortex-A53.

4.1.1 Hardware Acceleration 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. The ZCU102 is powerful enough to allow for a significant, almost 8-fold, speed-up of the accumulation algorithm. However, this requires a fast DDR4 (memory) interface to provide the bootstrapping key (which is the RGSW encryption of the secret key) in time with the hardware implementation. The latter was not implemented in this thesis as the focus lay on creating the general processing logic rather than a memory interface for a specific device.

4.1.2 Hardware and Software

As mentioned in Chapter 1, the bootstrapping step takes up 137 ms on an "Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz and 64 GB of RAM, running Ubuntu 18.04 LTS" [24]. The second most computationally intensive step (i.e., the step which takes the second most execution time for our purposes) is the key-switching step, which takes slightly less than 1 ms on the same processor. For this reason, we focus on the hardware acceleration of the bootstrapping only. In fact, we don't accelerate the full bootstrapping process, but only the accumulation step, as the initialisation step and extraction step take up a negligible amount of execution time [24].

To understand our hardware implementation, it is easier to consider the PALISADE source code [5], which we have written in pseudo-code in Algorithm 3, rather than Algorithm 2 [24]. Algorithm 2 is the more abstract accumulation algorithm found in Micciancio et al.'s paper comparing FHEW and TFHE [24].

4.2 An Overview of the Hardware Implementation

We now consider Algorithm 3, which is also described in flow chart form in Appendix A. We can split the flow of the algorithm into 2 parts: the outer RLWE loop (lines 1-5 in Algorithm 3), and the inner RGSW loop (lines 6-22).

4.2.1 The Outer Control Loop

The outer LWE loop is fairly trivial to implement in hardware, and one modular subtraction (see line 4 in Algorithm 3) is computed in software 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 c_j (line 4) (each $\log_2(B_r) = \log_2(23) = 5$ bits). These values are thus calculated in advance, and will only be used for determining which part of the bootstrapping key is used to rotate the RGSW Accumulator.

4.2.2 The Inner Control Loop

The most performance-sensitive part of the algorithm lies in the inner control loop (lines 6-22). The inner loop can be broken up into 4 parts: two INTT's (line 7), Signed Digit Decomposition (line 8), eight $(= 2 * d_g)$ NTT's (line 10) and RGSW bootstrapping key multiplication (line 17).

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 allows for easy polynomial multiplication ("EVALUATION" state in the PALISADE library). After performing this INTT to bring the accumulator back to the "normal" state("COEFFICIENT"

Algorithm 3: Accumulation described in Algorithm 2 from an implementation perspective
Note: This describes the full algorithm, and is equal to Algorithm 2 being executed n (= 512) times

Data: **ACC**, made up of 2 vectors size N, **secretKey**, an array of $d_r B_r * 2 * (2d_g)$ values, $\mathbf{a} = \{a_0, a_1, ..., a_{n-1}\}$
Result: **ACC**, updated with $a_i * s_i$ for $i = 0...n-1$

```
1  begin
2      for i = 0, 1, ...n − 1 do
3          for j = 0, 1, ... log_{B_r}(q) − 1 do
4              c_j = ⌊a_i / B_r^j⌋ mod B;
5              if c_j > 0 then
6                  for k = 0, 1 do
7                      CoefACC[k] = INTT(ACC[k]);
8                      {dcmp[k][d_g],.., dcmp[k][1], dcmp[k][0]} =
                        SignedDigitDecompose(CoefACC[k]);
9                      for l = 0, 1, ...d_g do
10                         evalACC[k][l] = NTT(dcmp[k][l]);
11                     end
12                 end
13                 for k = 0, 1 do
14                     ACC[k] = 0;
15                     for l = 0, 1, ...d_g − 1 do
16                         for m = 0, 1 do
17                             secretProduct = evalACC[m][l] *
                                secretKey[k][l][m];
18                             ACC[k] = ACC[k] + secretProduct;
19                         end
20                     end
21                 end
22             end
23         end
24     end
25     Return ACC;
26 end
```

state in the PALISADE library), we perform the signed digit decompose (see Algorithm 4). 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. Doing our bootstrapping key multiplication in this fashion decreases the noise growth [24], as shown in Chapter 2. 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 bootstrapping key, 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 and lines 16-19 in Algorithm 3).

Algorithm 4: Signed Digit Decompose

Data: **CoefACC**, a vector of size N
Result: **dcmp**[d_g]**, ..., dcmp[1], dcmp[0]**, which is the signed decomposition of each value in **CoefACC**

```
1  begin
2      for i = 0, 1, ...N − 1 do
3          if CoefACC[i] ≥ Q/2 then
4              d = CoefACC[i] − Q ;
5          end
6          else
7              d = CoefACC[i];
8          end
9          for j = 0, 1, ...d_r do
10             r = d (mod B)_g;
11             if r ≥ B_g/2 then
12                 r = r − B_g ;
13             end
14             d = d − r;
15             d = ⌊d/B_r⌋;
16             if r ≥ 0 then
17                 dcmp[j] = r ;
18             end
19             else
20                 dcmp[j] = Q − r ;
21             end
22         end
23     end
24     Return ACC;
25 end
```

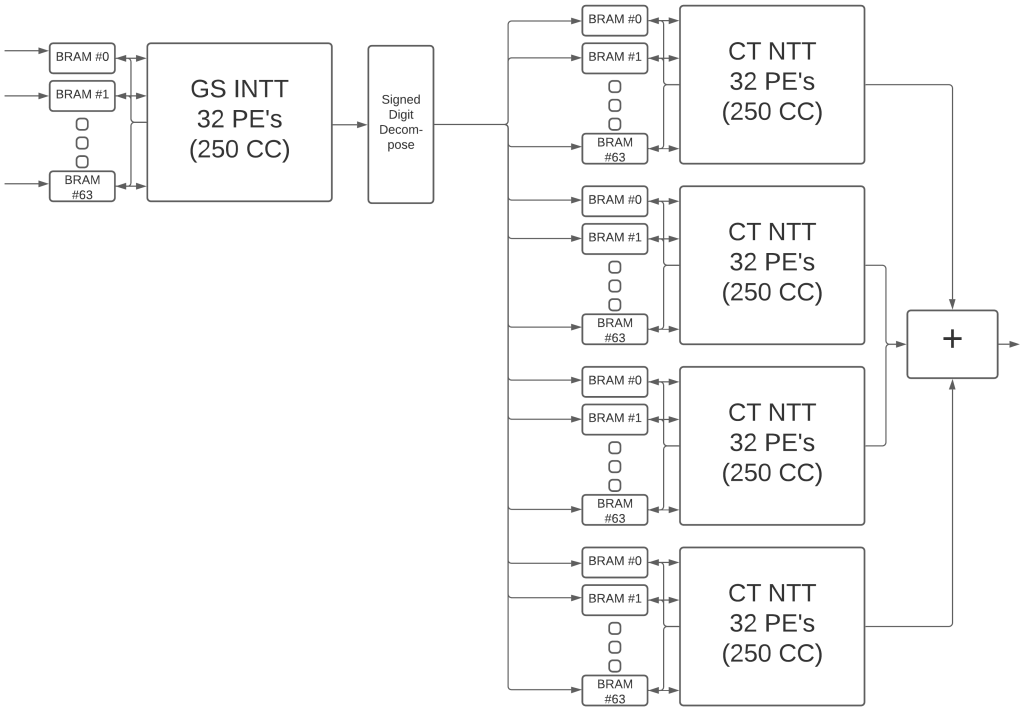


FIGURE 4.1: Diagram of the RGSW accumulator implementation

4.3 Mert et al.'s NTT Implementation

4.3.1 Implementation Advantages and Disadvantages

To the best of our knowledge, Mert et al.'s design is the only open-source hardware implementation with high enough performance for the FHEW implementation (see Chapter 3). It can provide this because the design is customizable with a parameter set. In other words, if required, one could also create a very low-area implementation using the exact same implementation. However, this comes at a flexibility cost: only the Gentleman Sande version of the NTT algorithm is implemented [11], with outputs using bit-reversed indices compared to the inputs (see Chapter 3). This means that the resulting coefficients of the NTT should be re-organized in memory before they can be used in the following INTT operation. Note that the reorganization in memory (=an index bit-reversal step), is costly in software, and is also difficult to achieve efficiently in hardware, especially when the coefficients are contained in multiple, separate, BRAMs with single read/write ports for each BRAM.

A second disadvantage of Mert et al's design is that there is no support for the negatively wrapped convolution described in Chapter 3. Thirdly, there is only one Q (= 27)-bit input port to the NTT module and Q (= 27)-bit output port. So, while the NTT (or INTT) is executed in 250 clock cycles, it takes N(= 1024) cycles to

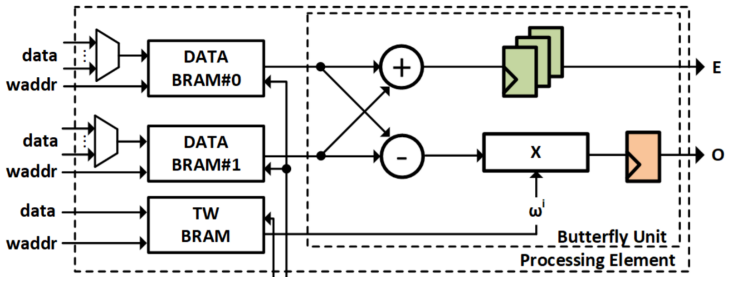


Figure 4.2: Mert et al.'s datapath [23]

load values into the BRAMs and $N(= 1024)$ cycles to output. Using Mert et al's design without modification would result in a 2250 CC execution time, where most of the time is spent on data transfers rather than calculations.

Finally, under the original design, the values would have to be output after executing every NTT or INTT, then placed back into the design, each step taking N clock cycles. The reason for this is that the output coefficients and input coefficients are stored in a different manner in the BRAMs. How the storage of inputs and outputs differs is explained in detail later.

4.3.2 Implementation Details

Before continuing to discuss this implementation, the basics of Mert et al.'s design must be considered. It allows for 3 parameters to be set:

- Ring Size $N = 1024$
- Modulus Size $Q < 2^{27}$
- Processing Elements $PE = 32$

The ring size and modulus size are found in the parameter set STD128 (see Table 2.5). The 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 (with a PE number of 32, the design cannot fully fit on the ZCU102) and a PE number of 64 or higher for the U280. The number of 32 was chosen because it was the largest parameter set mentioned in [23].

For each PE element, there are 2 BRAMs to store the coefficients which will go through the GS datapath, and 1 BRAM to hold the twiddle factors (Figure 4.2).

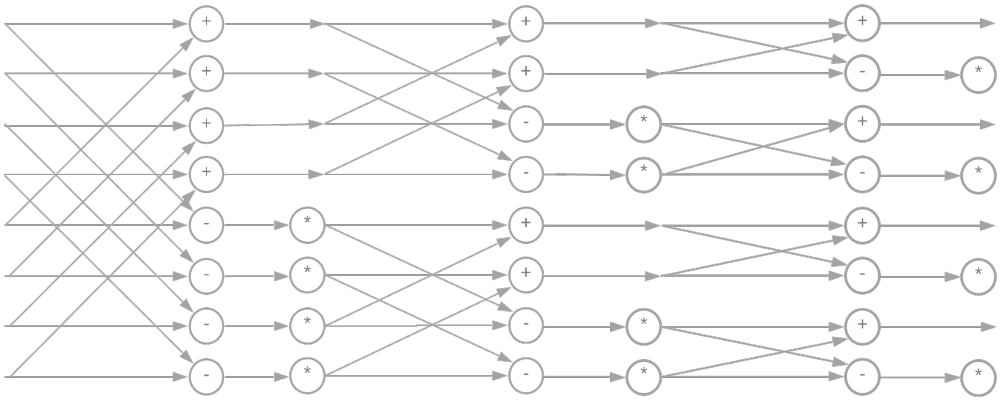


FIGURE 4.3: Mert et al.'s implementation of GS NTT and INTT for a ring size N of 8

Under the original design, the twiddle factors have to be loaded in at runtime as well, but to reduce the complexity of interfacing, and to save some area, the BRAMs for the twiddle factors were implemented as ROMs, the content of which is initialized at design time.

As we have 64 BRAMs for storing the coefficients for each NTT accelerator, there are logically $N/(2 * PE) = 16$ values in each (coefficient) BRAM at the start of the execution. The Gentleman-Sande NTT algorithm has multiple stages ($\log_2 N = 10$)[30]. After each stage, the values must be written back to new BRAMs, as a PE block can only use the 2 BRAMs 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 are used in the original multiplier are given by:

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

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

4.4 Leveraging the NTT for FHEW

Now that we have seen what Mert et al.'s design provides, we will discuss how we modified this design to support FHEW. These modifications can be split into three logical parts: the modification of the GS INTT to support *negatively wrapped convolution* and index bit-reversal, the modification of Mert et al.'s GS design into a CT design, and the integration of the bootstrapping key multiplication into this CT

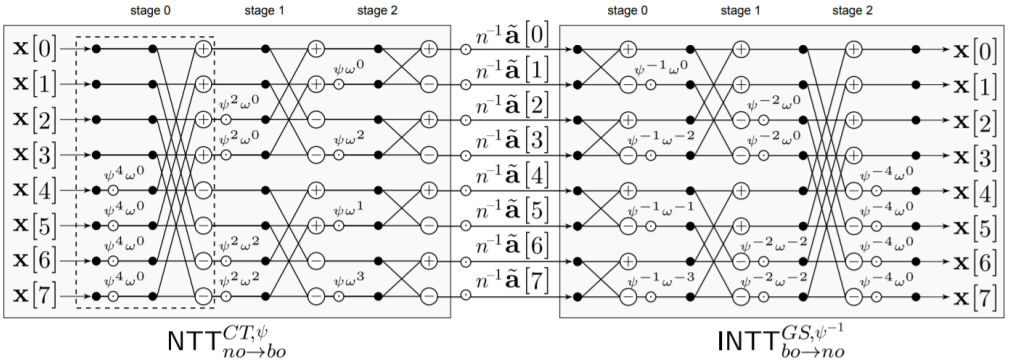


Figure 4.4: Poppelmann et al.'s representation of the GS and CT NTT [30]

design.

Note that these modifications are done for two reasons: Firstly, Mert et Al.'s design does not support the execution of more than one NTT or INTT by itself. Secondly, naive modifications to extend this support would be very inefficient, as noted in the previous section.

4.4.1 Modifying the GS INTT

As previously mentioned, Mert et al. did not include support for *negatively wrapped convolution*. Pöppelmann et al. [30] explain how the twiddle factors of an existing (normal) INTT algorithm (in software) can be modified to achieve this nonetheless. Pöppelman et al.'s method to modify the twiddle factors is applied to the twiddle factors used in Mert et al's design to create our design. Because the twiddle factors are calculated in advance, which is done by a Python script, 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 4.4 is not the same GS INTT butterfly depicted in Figure 4.3. This is the result of [30] describing a GS INTT which takes a polynomial in bit-reversed ordering and returns it to normal ordering. Mert et al.'s GS INTT takes a polynomial with indices in normal order, and bit-reverses these indices. We turn one butterfly into the other by bit-reversing the indices before and after running the INTT. Unfortunately, this adds extra overhead. However, in Mert et al.'s design, we have 32 different Processing Elements, each with 2 BRAMs, with values of the polynomial of size N (= 1024) spread out amongst them. Therefore a bit reversing step has to happen either way in order to bring the values from the BRAMs of the NTT to the BRAMs of the INTT. As such, the area added by a separate Index Reversal Block is not wasted.

The advantage of this separate Index Reversal Block is that the twiddle factors calculated in [30] can be used in our design 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 4.5).

The twiddle factor calculation for the INTT of this thesis is given by Algorithm 5. This operation is done in advance in a scripting language, as the twiddle factors remain constant for a given parameter set. Note that the algorithm aims at optimizing space usage by never storing the same twiddle factor twice. This optimisation was also present in Mert et al.'s original implementation. For the parameters which our implementation targets, this memory optimization does not reduce the size of the memory used. The reason for this is that the minimum size of a BRAM in our device is 18 Kbit [43]. The twiddle factors take up 36*$\log_2(Q)(= 36 * 27)$ bits in their optimised form, while they would take up $\log_2 N * N/(2 * PE) * \log_2(Q) = 160 * 27$ bits in their non-optimized form. In other words, the twiddle factors (for our parameter set) do not fully fill up the minimum size BRAM. Therefore, there is no point optimizing them to take up even less space. The same number of BRAM's will still be required for our design, regardless of whether we optimize the space taken up by the twiddle factors.

Therefore, we did not optimize the twiddle factor calculation of the CT NTT, as doing so would require rewriting not just the full addressing of the NTT but also the connections between the different PE's.

Algorithm 5: Twiddle factor calculation for the GS INTT

Data: **R** $(= 2^{33})$, a factor of 2 provided by Mert for the internal Montgomery Multiplication
Result: **TwiddleFactor** containing ω for each BRAM and ψ so that $\psi^2 = \omega$

begin
  for $BRAM = 0, 1, ...2 * PE - 1$ **do**
    for $j = 0, 1, ... \log_2(N) - 1$ **do**
      for $k = 0, ..., \min(1, \lfloor N/PE \rfloor >> j) - 1$ **do**
        exponent $= (2 * PE * 2^j * 2k + (2 * \text{BRAM} + 1) * 2^j) \pmod N$;
        TwiddleFactor[BRAM][j][k] $= \psi^{-\text{exponent}} * \mathbf{R} \pmod Q$;
      end
    end
  end
  Return **TwiddleFactor**;
end

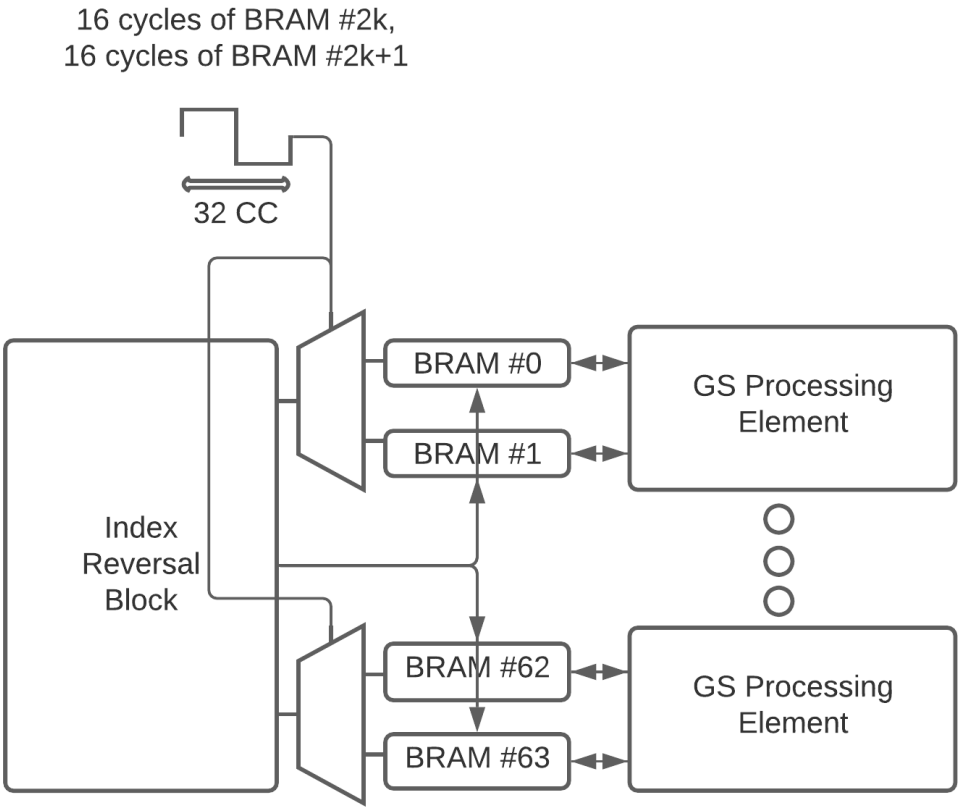


Figure 4.5: Bitreversal in the GS INTT

4.4.2 Overview of Index Bit Reversal

The difficulty with index bit-reversing is the organization of the memory because each BRAM is instantiated with only one read port and one write port. This means we must carefully consider which values are swapped in memory (an index bit-reverse operation is a series of swaps). We cannot swap two or more values residing in the same BRAM in the same clock cycle as this would require two reads from the same BRAM in one clock cycle. (BRAM's can be instantiated with two write and read ports in our FPGA architecture, but this requires doubling their size from 18 Kbit to 36 Kbit and having two read and two write ports would not make the bit-reversal problem much simpler, although it would allow bit-reversal in half the time).

A very simple implementation of bit-reversal would be to read out the values one by one, thereby guaranteeing that no values are read from the same BRAM, 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 equal to the number of BRAMs or 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 instances of 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 = 0x0100000000 = bitreverse(0x000000010). Thus, 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 BRAMs 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 [43]). 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 we propose is to read out the BRAMs 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 (which is good) but requires a complicated (but efficient) calculation of the read and write addresses.

The first thing to consider is that in Mert et al.'s implementation, the mapping of BRAM addresses and BRAM indices to coefficient indices changes for inputs and outputs. 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 :

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

$$\text{inputaddr} = i[8:5] \tag{4.3}$$

for values before an NTT or INTT commences and

$$\text{outputBRAM} = i[5:0] \tag{4.4}$$

$$\text{outputaddr} = i[9:6] \tag{4.5}$$

when the NTT or INTT has finished.
To solve the write limitation, we set:

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

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 element 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 multiplication by $\frac{1}{N}$ which is required for the INTT, adds a total of 80 cycles to the INTT run time of 250 cycles.

4.4.3 Creating CT NTT from GS NTT

As can be seen in Figure 4.4, 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. We simply change the datapath in Figure 4.2 (which represents a GS datapath) so that the multiplication comes before the addition and the subtraction. This converts the GS NTT to a CT NTT. Similar to the GS INTT, we modify the twiddle factors to support the *negatively wrapped convolution*.

As previously mentioned, the twiddle factor storage was not optimized for the CT NTT, as the optimization does not affect the area of the hardware for our parameter set. The calculation is given by Algorithm 6.

4.4.4 Non-NTT Components

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

The bootstrapping key multiplication requires either a separate block for multiplication and addition, or it can be integrated into CT NTT. A compromise between these two 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 instances of 27-bit adders.

Algorithm 6: Twiddle factor calculation for the CT NTT
Note: bitReverse is a function that writes a value in binary form $\pmod N$, then flips the value

Data: **R** ($= 2^{33}$), a factor of 2 provided by Mert for the internal Montgomery Multiplication
Result: **TwiddleFactor** containing ω for each BRAM and ψ so that $\psi^2 = \omega$

```
1  begin
2      for BRAM = 0, 1, ...2 * PE − 1 do
3          for j = 0, 1, ... log2(N) − 1 do
4              for k = 0, ..., PE − 1 do
5                  index = (BRAM + 2 * PE * k) * ⌊N * 2^(−j−1)⌋ ;
6                  exponent = bitReverse(2^j+index)
                    TwiddleFactor[BRAM][j][k] = ψ^exponent * R (mod Q);
7              end
8          end
9      end
10     Return TwiddleFactor;
11 end
```

Because we run both parts of the RGSW accumulator on the same datapath (multiplexed in time), the BRAMs of the CT NTT are extended to fit in the results of the bootstrapping 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 4.6). As previously mentioned, Mert under-utilizes the BRAM resources, so this does not increase the area taken up by the memory [43].

4.5 Memory Interface

The current rate-limiting step of the design is the memory interface. In order to perform the multiplication with the bootstrapping key in time with the CT NTT, we must be able to fetch $N * d_r * 2 = 8096$ elements in ≈ 900 cycles at 100 MHz it takes to complete half of one accumulator step. In other words, we would need a memory BW of

$$\frac{8 * N * f * \log_2(Q)}{\text{CC } 1/2 \text{ of Accumulation}} = \frac{8096 * 100 * 10^6 * 27}{900} = 24.3\text{Gbps} \tag{4.7}$$

This data rate can only be achieved if we store the key in the DDR4 memory at the PS side, and then build a 128-bit AXI interface at high enough clock frequency to communicate between the FPGA and this memory [42].

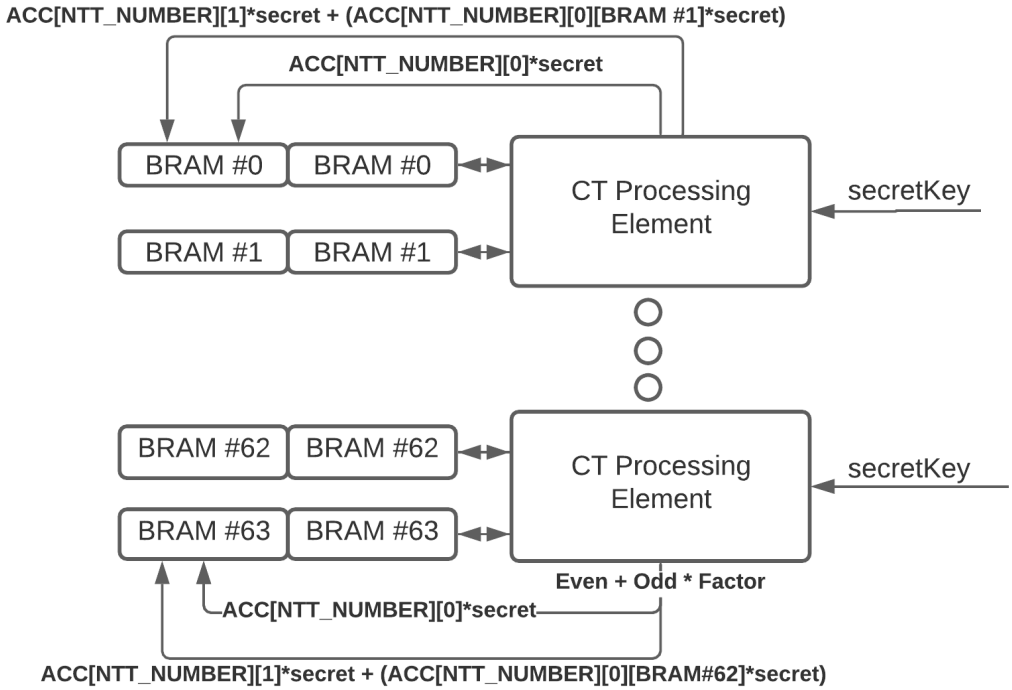


Figure 4.6: Bootstrapping key accumulation integrated in CT

Storing the entire bootstrapping key on the ZCU102 FPGA would not be possible at all, as it takes up a total of [24]:

$$4nNd_rB_rd_g \log_2(Q) = 10.4\text{Gigabits} = 1.3\text{GB} \tag{4.8}$$

This exceeds the maximum of 32.1 Megabits that can be generated[42].

Because the final design would be unlikely to fit a prototyping board such as the ZCU102, and making a very high-bandwidth DDR4 interface was not the main concern of this thesis, no attempt was made at designing a memory interface for the bootstrapping key. Instead, we verified the correctness of the computation blocks. In simulation, we used bootstrapping keys of the correct size to verify correctness. In synthesis, we preferred a bootstrapping key that was hardwired to the design, instead of reading from an external DDR4 memory. Because the multiplier used was the modular multiplier from the CT NTT, any bootstrapping key used in our design must be multiplied with the same factor R as our twiddle factors are multiplied with. Beyond that, the data in the bootstrapping key must be structured so that each BRAM receives the correct bootstrapping key at the correct clock cycle.

Task	Input	Add $a_i * s_i$ to **ACC**	Output	1/16th Bootstrap
CC	2052	3616	2049	114327

TABLE 4.1: Simulation run time

4.6 Results

4.6.1 Simulation Results

Because the bootstrapping key is in excess of 10 Gigabit, and storing the bootstrapping key as a text file for input into Verilog requires working with text files in excess of 10 GigaByte (as Verilog simulation requires text files for raw data input), it was decided to only run part of the bootstrapping process in simulation, with only 1/16th of the process being completed. The result of this simulation corresponded with the execution of 1/16th of the bootstrapping algorithm in PALISADE. The simulation gave us the following results for our processing speed: When extrapolating this to a full bootstrap, which consists of n (= 512) times adding $a_i * s_i$ (see Algorithm 3), we see that the full bootstrapping process executes in 1855493 clock cycles. Note that this number is a maximum, as accumulation will not be run when $c_j = 0$ (see line 4 of Algorithm 3).

4.6.2 Implementation Results

There are several important caveats attached to the implementation results. As previously mentioned, no bootstrapping key interface was created, so no area was taken up for the temporary buffering that a bootstrapping key interface might need. Secondly, the implemented design used slightly more configurable logic block resources than were available for use on the ZCU102. For this reason, half of the CT NTT's were cut from the design. The toy bootstrapping key that was employed to verify the correctness of the design was assumed to be zero for the half fed to these NTT's.

Lastly, while a BRAM interface was created and tested via simulation, our design was not able to write data to the BRAM interface, and as such, the functionality of the implementation with the BRAM interface could not be verified in the limited time available.

Note that the BRAM interface implemented is limited to providing the coefficients of the initial accumulator **ACC** and the c_j's of Algorithm 3. It would also allow for start and done "signals" (pre-determined memory places which are set to pre-determined values). Finally, it allows the resulting accumulator to be read out of memory from the software. It does not provide for an interface for the bootstrapping key, because of the high bandwidth specifications that such an interface would require.

When our implementation consisted of 2 NTT's and 1 INTT, we received the results described in Table 4.2. If we extrapolate from these results, and the results of

Frequency	WNS	LUT	FF	BRAM	DSP
100MHz	1.595 ns	133971	56565	146	768

Table 4.2: Implementation results for 1 INTT and 2 NTT run

LUT	FF	BRAM	DSP
223285	94275	240	1280

Table 4.3: Hypothetical implementation results for full implementation

the synthesis that did not fit on the device, the full design (minus the bootstrapping key memory) would use the resources described in Table 4.3.

4.6.3 Comparison to Other Implementations

There appear to be very few efforts to accelerate bootstrapping via hardware. A few efforts have centered on using GPU's to accelerate the bootstrapping step in FHEW [17], with a result of 110 ms execution time. Note that this result was achieved for an older, less efficient version of FHEW bootstrapping. The most recent variant of FHEW achieves bootstrapping in 137 ms [24] on a standard CPU. This hardware implementation, at 100MHz, succeeds in bootstrapping in 18.3 ms, a 7.5 times speed-up. Given that the individual NTT modules were able to run at 200 MHz, it is possible that the design, when implemented on the FPGAs that it would realistically target, would achieve bootstrapping in less than 10 ms.

4.7 Conclusion of Hardware Acceleration

By starting from Mert et al.'s NTT implementation, we created all the modules necessary for the successful acceleration of the FHEW bootstrapping process. After designing and implementing these modules, we implemented and simulated our design, and finally saw that while the design does not fit on the prototype board employed here, it would be able to execute the bootstrapping step in less than 20 ms on a larger design.

Chapter 5

Conclusion

We have designed a hardware accelerator for FHEW, a relatively simple Fully Homomorphic Scheme, so that with a fast memory interface, we can perform bootstrapping in less than 20 ms. To achieve this, we had to examine both the mathematics behind FHEW and the actual algorithm implemented in the PALISADE library, as well as the other homomorphic encryption schemes to understand why FHEW is a good candidate for homomorphic encryption. Using this understanding, the key multiplication in FHEW was accelerated through the Number Theoretic Transform. Many open source NTT HW implementations exist, but not all would be suitable for the purpose of accelerating FHEW. Mert et al.'s [23] implementation was chosen because it was the most suited to a FHEW hardware implementation. By modifying this design extensively, and creating separate implementations for the NTT and INTT steps, as well as integrating the computationally-intensive parts of the FHEW algorithm into the implementation, we were able to achieve a fast bootstrapping speed for a relatively non-optimized design.

5.1 Future Work

To turn the prototype created in this thesis into a working hardware implementation, an important first step would be the implementation of the design on a data accelerator card such as the U280. This would solve the current size constraint of the design. Secondly, a memory interface must be created which can provide the secret key at adequate speed for the FHEW bootstrapping, likely using pre-created IP such as DDR4 [42]. Thirdly, many optimisations can be created to make the design faster. The current clock frequency of 100 MHz is well below the 200 MHz that the NTT cores can be run at. Mert's design is "unoptimized" (see comments in [22]), and the critical path runs through DSP cores that have not been pipelined. Lastly, the inverse NTT, the signed digit decompose and the NTT are also used in TFHE, a scheme with a smaller secret key and faster execution. Therefore, the current FHEW implementation is from a hardware perspective very close to a TFHE implementation, which has a smaller secret key (but can have a longer execution time).

Using the optimized accumulation in hardware, software libraries such as PALISADE could call the accumulation design, and perform at least fifty, and probably over a hundred serial bit operations per second. From these bit operations, adders, multipliers, and all the basic functions required for computation can be created.

5.2 Applications

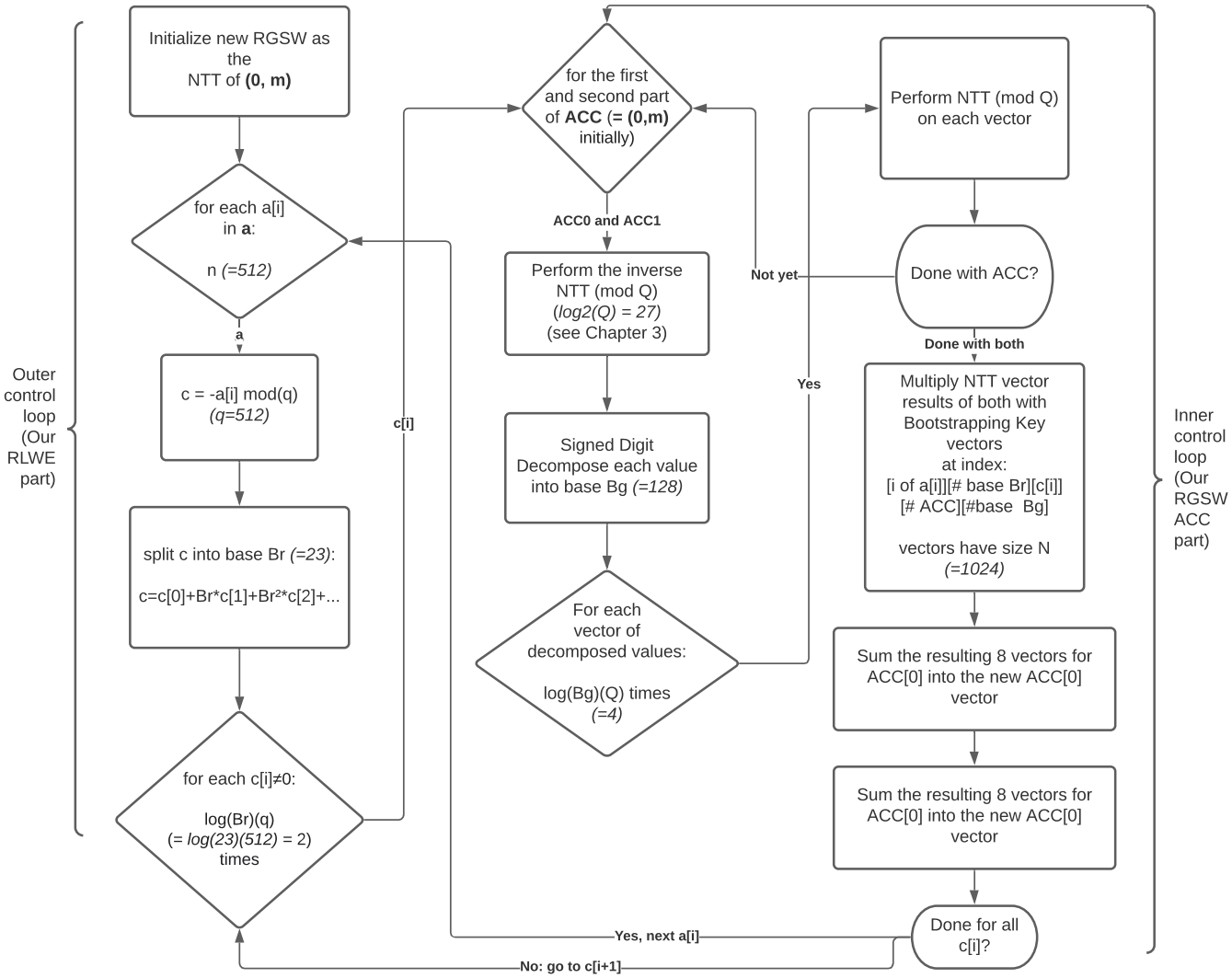
Applications for this hardware implementation can thus be found in any field that would benefit from Fully Homomorphic Encryption. While somewhat homomorphic encryption is much more efficient at executing low order polynomials, many applications cannot be constrained to only low order polynomials. For all those applications, (relatively) fast Fully Homomorphic Encryption, using functions created from (serially fast) single-bit operations, are the solution. Because our design is serially fast, the applications benefiting most from the hardware acceleration provided would be low-precision operations. Furthermore, because the hardware implementation at its basis accelerates RGSW $\times$ RLWE$'$ $\rightarrow$ RLWE multiplication, it can be used in any cryptography application where a significant number of these multiplications are performed.

Appendices

Appendix A

FHEW Algorithm flowchart

This flowchart was created from the algorithm found in [24]. 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.


Initialize new RGSW as the NTT of (0, m)
for each a[i] in a:
n (=512)
a
c = -a[i] mod(q) (q=512)
split c into base Br (=23):
c=c[0]+Br*c[1]+Br²*c[2]+...
for each c[i]≠0:
log(Br)(q) (= log(23)(512) = 2) times
c[i]
for the first and second part of ACC (= (0,m) initially)
ACC0 and ACC1
Perform the inverse NTT (mod Q) (log2(Q) = 27) (see Chapter 3)
Signed Digit Decompose each value into base Bg (=128)
For each vector of decomposed values:
log(Bg)(Q) times (=4)
Not yet
Yes
Perform NTT (mod Q) on each vector
Done with ACC?
Done with both
Multiply NTT vector results of both with Bootstrapping Key vectors at index: [i of a[i]][# base Br][c[i]] [# ACC][#base Bg]
vectors have size N (=1024)
Sum the resulting 8 vectors for ACC[0] into the new ACC[0] vector
Sum the resulting 8 vectors for ACC[0] into the new ACC[0] vector
Done for all c[i]?
Yes, next a[i]
No: go to c[i+1]
Outer control loop (Our RLWE part)
Inner control loop (Our RGSW ACC part)

Appendix B

Paper

Hardware Acceleration of FHEW

Jonas Bertels, Michiel Van Beirendonck, and Furkan Turan

***Abstract*—Fully Homomorphic Encryption allows operations to be done on encrypted data indefinitely. Both fully homomorphic and somewhat homomorphic schemes exist and are already in somewhat limited use, but the slow computation time, especially of Fully Homomorphic Encryption schemes, limits their adoption. In this paper, FHEW bootstrapping is accelerated in hardware. By modifying a fast hardware NTT implementation [1] with support for negatively wrapped convolution, large I/O ports to the NTT accelerator and an index bit-reversal block, the execution speed of FHEW bootstrapping can be increased by a factor of 7.5. This execution speed requires that a fast interface to the FHEW bootstrapping key is available.**

***Index Terms*—Homomorphic Encryption, Cryptography, Hardware Acceleration.**

I. Introduction

HOMOMORPHIC encryption means encryption where operations can still be done on encrypted data. Fast-working homomorphic encryption would have a wide-ranging impact on the financial sector [2], 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 [3] showed that Fully Homomorphic Encryption, i.e., doing an arbitrary amount of operations on encrypted data, is possible. A concept called "bootstrapping" is used to turn Leveled Homomorphic Encryption into Fully Homomorphic Encryption. This can be done by performing a decryption while both the bootstrapping key and the data which is being decrypted is encrypted under another encryption.

FHEW (the name is a reference to the Fastest Fourier Transform in the West library [4]) is part of the so-called third-generation schemes [5], [6]. 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(R) Core(TM) i7-9700 CPU [7]. While only one NAND gate can be done at a time, 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. Therefore, programmers do not have to worry about the number of operations they can do when using FHEW. Their only concern becomes building their program from the binary gates provided by the FHEW algorithm.

II. Preliminaries on the FHEW algorithm

We define $\mathbb{Z}_q$ as the ring of integers modulo q, and $\mathbb{Z}_q^n$ as n elements x so that $x \in \mathbb{Z}_q$. Lattice-based cryptography uses polynomial rings for various optimizations. The polynomials rings we use are defined as $R = \mathbb{Z}[x]/f(x)$. In practice, a popular choice for $f(x)$ is the $2d$-th cyclotomic polynomial, 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 $d = 2^m$.

We consider the discrete Gaussian sampling function $D_{\mathbb{Z},\sigma}$. From this Discrete Gaussian function, we can sample values $\chi = D^d_{\mathbb{Z},\sigma}$. We define $\odot$ as the element-wise multiplication between two different vectors or polynomials.

A. LWE

We define LWE with parameters: the dimension n, the message modulus $t \geq 2$ and the ciphertext modulus q [8]. We also define a randomized rounding function $\chi : \mathbb{R} \to \mathbb{Z}$. We use m to denote our messages and $m \in \mathbb{Z}_t$. The secret key is a vector and the secret key space is $\mathbb{Z}_q^n$, and the secret key may be chosen uniformly from this space. The encryption of m under $\mathbf{s}$ is then ([8], equation 2):

$$\mathrm{LWE}_s^{t/q}(m) = (\mathbf{a}, \chi(\mathbf{a} \odot \mathbf{s} + mq/t) \bmod q) \in \mathbb{Z}_q^{n+1} \quad (1)$$

We have $\mathbf{a} \leftarrow \mathbb{Z}_q^n$, which is chosen uniformly at random. When we define LWE less formally in the next section, we will assume that q is divisible by t so that:

$$\mathrm{LWE}_s^{t/q}(m) = (\mathbf{a}, \mathbf{a} \odot \mathbf{s} + mq/t + e \bmod q) \quad (2)$$

with error e "chosen according to fixed error distribution $\chi(0)$" [8].

B. RLWE

We also define RLWE, for a message $\tilde{m} \in R_q$ under a secret key $s \in R$ as [7]:

$$\mathrm{RLWE}_s(\tilde{m}) = (a, as + e + \tilde{m}) \quad (3)$$

with $a \leftarrow R_q$ (chosen uniformly at random) and $e \leftarrow \chi_\sigma^d$ with d our polynomial ring size as defined.

C. RLWE' and RGSW

We now introduce two more definitions based on RLWE, which we need when discussing FHEW. RLWE' is defined as [8] [7]:

$$\text{RLWE}'_s(m) = (\text{RLWE}_s(m), \text{RLWE}_s(B_g * m), \text{RLWE}_s(B_g^2 * m), ..., \text{RLWE}_s(B_g^{k-1} * m)) \quad (4)$$

with $B_g^k = q$. The reason for this definition is that we want a system that creates as little noise as possible when operations are completed. By breaking RGSW into bases, less noise is generated ([7], see section 5.1). Multiplications with a $d \in R_q$ so that $d = \sum_i B_g^i d_i$ are defined as:

$$(\odot) : R \times \text{RLWE}' \to \text{RLWE} : d \in R, \mathbf{c} \in \text{RLWE}' : d \odot (\mathbf{c}_0, ..., \mathbf{c}_{k-1}) = \sum_i d_i \mathbf{c}_i \quad (5)$$

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

$$(\odot') : R \times \text{RLWE}' \to \text{RLWE}' : d \odot \mathbf{C} = ((B_g^0 * d) \odot \mathbf{C}, (B_g^1 * d) \odot \mathbf{C}, ..., (B_g^{k-1} * d) \odot \mathbf{C}) \quad (6)$$

And finally, using these definitions, our RGSW scheme can be defined, which will allow multiplication of ciphertexts. We need this ability if we want to run our RLWE decryption while everything is encrypted under RGSW (this will be explained in-depth in the bootstrapping section).

$$\text{RGSW}_s(m) = (\text{RLWE}'_s(-s * m), \text{RLWE}'_s(m)) \quad (7)$$

To be able to use our secret key s encrypted under RGSW, we can multiply an RGSW ciphertext $\text{RGSW}(m_1) = (\mathbf{c}, \mathbf{c'})$ with an 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 [7]:

$$(a, b) \diamond (\mathbf{c}, \mathbf{c'}) = \langle (a, b), (\mathbf{c}, \mathbf{c'}) \rangle = a \odot \mathbf{c} + b \odot \mathbf{c}' \quad (8)$$

$$= a \odot \text{RLWE'}_s(-s * m_1) + b \odot \text{RLWE'}_s(m_1) \quad (9)$$

$$= \text{RLWE}_s((b - a * s) * m_1) = \text{RLWE}_s((m_0 + e_0) * m_1) \quad (10)$$

This RGSW $\times$ RLWE$'$ $\to$ RLWE can be turned into a RGSW $\times$ RLWE$'$ $\to$ RLWE$'$. In Algorithm 3, we will see how this multiplication is implemented. We will also see how the $\text{RGSW}_s(m) = (\text{RLWE}'_s(-s * m), \text{RLWE}'_s(m))$ is created through Algorithm 4. Briefly, each coefficient of the polynomial is broken down through a signed digit decomposition. Then a multiplication with an RLWE' vector occurs. As mentioned at the start of the section, using RLWE' instead of RLWE for the multiplication reduces the generated noise. Once the multiplication is completed, the results are summed.

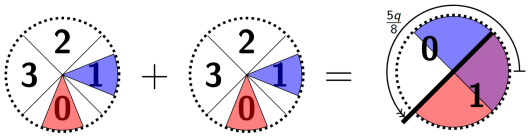


Fig. 1. NAND gate created by taking $\text{LWE}_s^{4/q}(\tilde{m})$ (with $e = q/16$) + $\text{LWE}_s^{4/q}(\tilde{m})$ (with $e = q/16$) = $\text{LWE}_s^{2/q}(\tilde{m})$ (with $e = q/8$) [9]

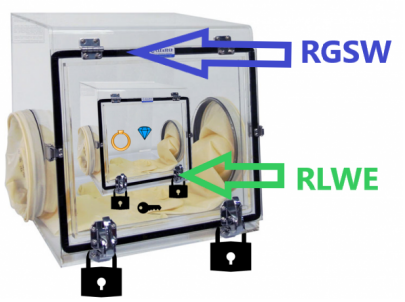


Fig. 2. Our LWE ciphertext is encrypted under an RGSW ciphertext so that it can be decrypted using LWE secret keys, which themselves have been encrypted under RGSW

III. FHEW (Fastest Homomorphic Encryption in the West)

A. Bootstrapping

The FHEW scheme is motivated by the need for fast bootstrapping [8]. This fast bootstrapping is necessary for "practical" Fully Homomorphic Encryption.

B. Introduction to Bootstrapping in FHEW

To bootstrap efficiently, the FHEW algorithm only performs one NAND gate before bootstrapping, or a similarly small binary gate operation. This causes us to go from $\text{LWE}_s^{4/q}(\tilde{m})$ to $\text{LWE}_s^{2/q}(\tilde{m})$, or in other words, from an LWE ciphertext deciphering to four possible messages with a maximum error of $q/16$ to a ciphertext deciphering to two possible messages with a maximum error of $q/8$ [9]. (see Figure 1).

Bootstrapping is done by executing decryption operations homomorphically, i.e., doing the decryption operations with an encrypted secret key. This reduces the noise of the $\text{LWE}_s^{2/q}(\tilde{m})$ ciphertext back down to the original level, completing our algorithm.

The encryption for the ciphertext is provided by RGSW. 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 (see Figure 2). We call the secret key encrypted under RGSW the bootstrapping key.

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). The Accumulation, which consists of n RGSW $\times$ RLWE$'$ $\to$ RLWE multiplications, is the

bottleneck of the FHEW algorithm.

Algorithm 1: Initialization[7]

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, ...q/2 − 1 do
        m_i = f(b − i);
        Note that f is our lookup table
    end
    m = Σ_{i<q/2} m_i * X^i = (m_0, m_1, ..., m_{q/2−1});
    Return (0, m);
end
```

Algorithm 2: A Single Accumulation Step[7]

Data: **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: **ACC**, updated with $a_i * s_i$

```
begin
    for j = 0, 1, ... log_{B_r}(q) − 1 do
        c_j = ⌊c/B_r^j⌋ mod B;
        if c_j > 0 then
            ACC ← ACC ⋄ Z_{j,c_j};
        end
    end
    Return ACC;
end
```

IV. Hardware Implementation

The current FHEW (software) implementation provided in the PALISADE library can execute one binary gate in approximately 140 ms [7]. For many applications, the execution of 7 binary gates per second is too slow to be practical, especially when an entire CPU must be dedicated to one binary gate.

Since we attempt to accelerate this FHEW software implementation, we use the parameter set (the STD128 set) for which timing results were available, and which fit the parameters of the available NTT accelerators best. Generally, these accelerators are designed for a ring depth of $N = 1024$ and for a modulus Q bit size of either 13.6 bit or around 32 bit. The parameters for which our hardware implementation was designed are given in Table I.

A. The Inner Control Loop

The most performance-sensitive part of Algorithm 3 lies in the inner control loop (lines 6-22). The inner loop can

Algorithm 3: Accumulation described in Algorithm 2 from an implementation perspective
Note: This describes the full algorithm, and is equal to Algorithm 2 being executed n (= 512) times

Data: **ACC**, made up of 2 vectors size N, **secretKey**, an array of $d_r B_r * 2 * (2d_g)$ values, **a** $= \{a_0, a_1, ..., a_{n-1}\}$
Result: **ACC**, updated with $a_i * s_i$ for $i = 0...n-1$

```
1  begin
2      for i = 0, 1, ...n − 1 do
3          for j = 0, 1, ... log_{B_r}(q) − 1 do
4              c_j = ⌊a_i/B_r^j⌋ mod B;
5              if c_j > 0 then
6                  for k = 0, 1 do
7                      CoefACC[k] = INTT(ACC[k]);
8                      {dcmp[k][d_g],.., dcmp[k][1],
                        dcmp[k][0]} = SignedDigitDecom-
                        pose(CoefACC[k]);
9                      for l = 0, 1, ...d_g do
10                         evalACC[k][l] =
                            NTT(dcmp[k][l]);
11                     end
12                 end
13                 for k = 0, 1 do
14                     ACC[k] = 0;
15                     for l = 0, 1, ...d_g − 1 do
16                         for m = 0, 1 do
17                             secretProduct =
                                evalACC[m][l] *
                                secretKey[k][l][m];
18                             ACC[k] = ACC[k] +
                                secretProduct;
19                         end
20                     end
21                 end
22             end
23         end
24     end
25     Return ACC;
26 end
```

be broken up into 4 parts: two Inverse Number Theoretic Transforms (INTTs) (line 7), Signed Digit Decomposition (line 8), eight ($= 2*d_g$) Number Theoretic Transforms (NTTs) (line 10) and RGSW bootstrapping key multiplication (line 17). Of these, the NTT's and INTT's are by far the most time-critical. The hardware accelerator is therefore based on a design for accelerating NTT's (see Mert et al. [1]). This design is highly parametric and allows for 3 parameters to be set:

- Ring Size N
- Modulus Size Q
- Processing Elements PE

The ring size and modulus size are found in the parameter set STD128 (see Table I). The number of processing elements determines the speed and (to a degree) the area of the design.

Algorithm 4: Signed Digit Decompose

Data: **CoefACC**, a vector of size N
Result: **dcmp[d_g], ..., dcmp[1], dcmp[0]**, which is the signed decomposition of each value in **CoefACC**

begin
  for $i = 0, 1, ...N-1$ **do**
    if ***CoefACC***$[i] \geq Q/2$ **then**
      $d =$ **CoefACC**$[i] - Q$;
    end
    else
      $d =$ **CoefACC**$[i]$;
    end
    for $j = 0, 1, ...d_r$ **do**
      $r = d \pmod{B}_g$;
      if $r \geq B_g/2$ **then**
        $r = r - B_g$;
      end
      $d = d - r$;
      $d = \lfloor d/B_r \rfloor$;
      if $r \geq 0$ **then**
        dcmp[j] $= r$;
      end
      else
        dcmp[j] $= Q - r$;
      end
    end
  end
  Return **ACC**;
end

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

TABLE I
PARAMETERS FOR STD128

For the purpose of this implementation, we chose a PE number of 32. The original datapath of one Processing Element is shown in Figure 3. From this, our hardware accelerator (Figure 4) is created. The Gentleman-Sande (GS) INTT block contains 32 instances of Figure 3, while the CT NTT block contains 32 instances of a modified datapath for the Cooley-Tukey NTT. The difference between the two algorithms is nicely summarized by Figure 5 and described in [10].

Our implementation extends the work of Mert et al. We add hardware support for bit-reversal of the indices (see Figure 6), signed digit decomposition and multiplication with the bootstrapping key (see Figure 7). This is described in the next sections.

B. Overview of Index Bit-Reversal

We consider the implementation of the Index Bit-Reversal step somewhat in-depth because it is required for a high-

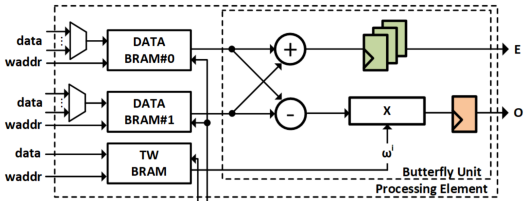


Fig. 3. Mert et al.'s datapath [1]

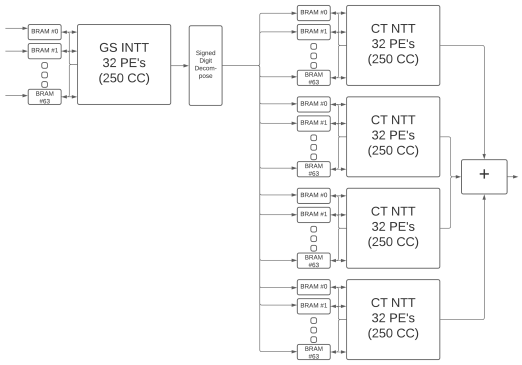


Fig. 4. Diagram of the RGSW accumulator implementation

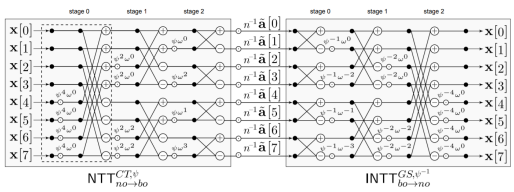


Fig. 5. Poppelmann et al.'s representation of the GS and CT NTT [10]

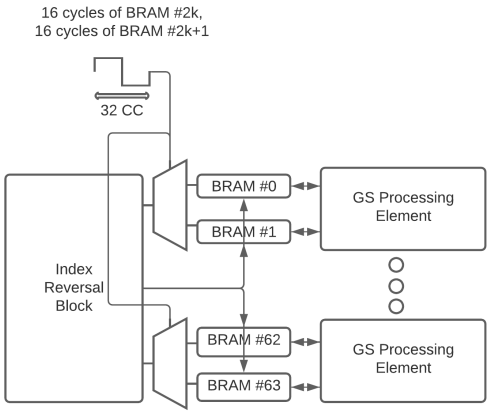


Fig. 6. Bitreversal in the GS INTT

performance polynomial multiplication.

The difficulty with the bit-reversing is the organization of the memory because each BRAM is instantiated with only one read port and one write port. This means we must carefully consider which values are swapped in memory (an index bit-reverse operation is a series of swaps). We cannot swap two or more values residing in the same BRAM in the same clock cycle as this would require two reads from the same BRAM in one clock cycle. (BRAM's can be instantiated with two write and read ports in our FPGA architecture, but this requires doubling their size from 18 Kbit to 36 Kbit and having two read and two write ports would not make the bit-reversal problem much simpler, although it would allow bit-reversal in half the time).

A very simple implementation of bit-reversal would be to read out the values one by one, thereby guaranteeing that no values are read from the same BRAM, 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 equal to the number of BRAMs or 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 instances of 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 = 0x0100000000 = bitreverse(0x0000000010). Thus, 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 BRAMs are also used in Mert et al. 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 [11]). 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 we propose is to read out the BRAMs 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 (which is good) but requires a complicated (but efficient) calculation of the read and write addresses.

The first thing to consider is that in Mert et al.'s implementation, the mapping of BRAM addresses and BRAM indices to coefficient indices changes for inputs and outputs. 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 :

$$\text{inputBRAM} = \{i[4:0], i[9]\} \tag{11}$$

$$\text{inputaddr} = i[8:5] \tag{12}$$

for values before an NTT or INTT commences and

$$\text{outputBRAM} = i[5:0] \tag{13}$$

$$\text{outputaddr} = i[9:6] \tag{14}$$

when the NTT or INTT has finished.
To solve the write limitation, we set:

$$\text{readaddr} = \text{BRAM}[5:2] + \text{cycle} \tag{15}$$

with cycle being the current clock cycle of the bit-reversal. In other words, it is a value that 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 element 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 multiplication by $\frac{1}{N}$ which is required for the INTT, adds a total of 80 cycles to the INTT run time of 250 cycles.

C. Non-NTT Components

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

The bootstrapping key multiplication requires either a separate block for multiplication and addition, or it can be integrated into CT NTT. A compromise between these two approaches was used in this design, 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 instances of 27-bit

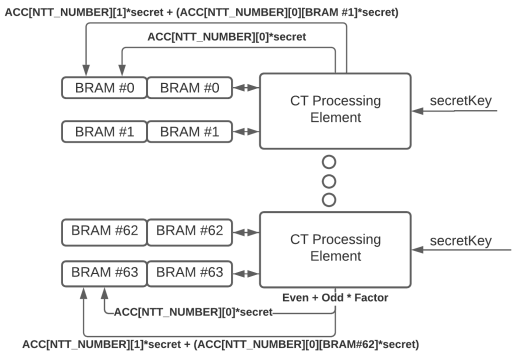


Fig. 7. Bootstrapping key accumulation integrated in CT

adders.

Because we run both parts of the RGSW accumulator on the same datapath (multiplexed in time), the BRAMs of the CT NTT are extended to fit in the results of the bootstrapping 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 7). As previously mentioned, Mert et al. under-utilizes the BRAM resources, so this does not increase the area taken up by the memory [11].

V. Memory Interface

The current rate-limiting step of the design is the memory interface. In order to perform the multiplication with the bootstrapping key in time with the CT NTT, we must be able to fetch $N * d_r * 2 = 8096$ elements in ≈ 900 cycles at 100 MHz it takes to complete half of one accumulator step. In other words, we would need a memory BW of

$$\frac{8 * N * f * \log_2(Q)}{\text{CC 1/2 of Accumulation}} = \frac{8096 * 100 * 10^6 * 27}{900} = 24.3\text{Gbps} \tag{16}$$

This data rate can only be achieved if we store the key in the DDR4 memory at the PS side, and then build a 128-bit AXI interface at high enough clock frequency to communicate between the FPGA and this memory [12].

Storing the entire bootstrapping key on the ZCU102 FPGA would not be possible at all, as it takes up a total of [7]:

$$4nNd_rB_rd_g \log_2(Q) = 10.4\text{Gigabits} = 1.3\text{GB} \tag{17}$$

This exceeds the maximum of 32.1 Megabits that can be generated[12].

Because the final design would be unlikely to fit a prototyping board such as the ZCU102, and making a very high-bandwidth DDR4 interface was not the main concern of this thesis, no attempt was made at designing a memory interface for the bootstrapping key. Instead, we verified the correctness of the computation blocks. In simulation, we used bootstrapping keys of the correct size to verify correctness. In synthesis, we preferred a bootstrapping key that was hardwired to the design, instead of reading from an external DDR4 memory.

Task	Input	Add $a_i * s_i$ to **ACC**	Output	1/16th Bootstrap
CC	2052	3616	2049	114327

TABLE II
Simulation run time

Because the multiplier used was the modular multiplier from the CT NTT, any bootstrapping key used in our design must be multiplied with the same factor R as our twiddle factors are multiplied with. Beyond that, the data in the bootstrapping key must be structured so that each BRAM receives the correct bootstrapping key at the correct clock cycle.

VI. Results

A. Simulation Results

Because the bootstrapping key is in excess of 10 Gigabit, it was decided to only run part of the bootstrapping process in simulation, with only 1/16th of the process being completed. The result of this simulation corresponded with the execution of 1/16th of the bootstrapping algorithm in PALISADE. The simulation gave us the following results for our processing speed: When extrapolating this to a full bootstrap, which consists of n $(= 512)$ times adding $a_i * s_i$ (see Algorithm 3), we see that the full bootstrapping process executes in 1855493 clock cycles. Note that this number is a maximum, as accumulation will not be run when $c_j = 0$ (see line 4 of Algorithm 3).

B. Implementation Results

There are several important caveats attached to the implementation results. As previously mentioned, no bootstrapping key interface was created, so no area was taken up for the temporary buffering that a bootstrapping key interface might need. Secondly, the implemented design used slightly more configurable logic block resources than were available for use on the ZCU102. For this reason, half of the CT NTT's were cut from the design. The toy bootstrapping key that was employed to verify the correctness of the design was assumed to be zero for the half fed to these NTT's.

Lastly, while a BRAM interface was created and tested via simulation, our design was not able to write data to the BRAM interface, and as such, the functionality of the implementation with the BRAM interface could not be verified in the limited time available.

Note that the BRAM interface implemented is limited to providing the coefficients of the initial accumulator **ACC** and the c_j's of Algorithm 3. It would also allow for start and done "signals" (pre-determined memory places which are set to pre-determined values). Finally, it allows the resulting accumulator to be read out of memory from the software. It does not provide for an interface for the bootstrapping key, because of the high bandwidth specifications that such an interface would require.

When our implementation consisted of 2 NTT's and 1 INTT, we received the results described in Table III. If we extrapolate

Frequency	WNS	LUT	FF	BRAM	DSP
100MHz	1.595 ns	133971	56565	146	768

TABLE III
IMPLEMENTATION RESULTS FOR 1 INTT AND 2 NTT RUN

LUT	FF	BRAM	DSP
223285	94275	240	1280

TABLE IV
HYPOTHETICAL IMPLEMENTATION RESULTS FOR FULL IMPLEMENTATION

from these results, and the results of the synthesis that did not fit on the device, the full design (minus the bootstrapping key memory) would use the resources described in Table IV.

VII. CONCLUSION

By starting from Mert et al.'s NTT implementation, we created all the modules necessary for the successful acceleration of the FHEW bootstrapping process. A fast index bit-reversal block was created, and the design was split into a separate NTT and INTT block to allow for negatively wrapped convolution and therefore efficient polynomial multiplication. Finally, the existing multipliers in the NTT implementation were reused for the multiplication with the bootstrapping key. After designing and implementing these modules, we implemented and simulated our design, and finally saw that while the design does not fit on the targeted prototype board, it would be able to execute the bootstrapping step in less than 20 ms on a larger design.

REFERENCES

[1] 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*, pp. 1–1, 2020.

[2] H.-T. Peng, W. W. Hsu, J.-M. Ho, and M.-R. Yu, "Homomorphic encryption application on financialcloud framework," in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2016, pp. 1–5.

[3] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, ser. STOC '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 169–178. [Online]. Available: https://doi.org/10.1145/1536414.1536440

[4] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[5] 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.

[6] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: Fast fully homomorphic encryption over the torus," Cryptology ePrint Archive, Report 2018/421, 2018, https://eprint.iacr.org/2018/421.

[7] D. Micciancio and Y. Polyakov, "Bootstrapping in fhew-like cryptosystems," *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 86, 2020. [Online]. Available: https://eprint.iacr.org/2020/086

[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] ——, "Fhew: Homomorphic encryption bootstrapping in less than a second1," University Lecture, 2015.

[10] 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.

[11] *UltraScale Architecture Memory Resources*, Xilinx, 3 2021, block Ram Summary.

[12] *ZCU102 Evaluation Board*, Xilinx, 06 2019, pS-Side: DDR4 SODIMM Socket.

Bibliography

[1] 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.

[2] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, page 309–325, New York, NY, USA, 2012. Association for Computing Machinery.

[3] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Tfhe: 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] C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. Cryptology ePrint Archive, Report 2013/340, 2013. https://eprint.iacr.org/2013/340.

[16] Y. Igarashi, T. Altman, M. Funada, and B. Kamiyama. *Computing: A Historical and Technical Perspective*. Taylor & Francis, 2014.

[17] M. S. Lee, Y. Lee, J. H. Cheon, and Y. Paek. Accelerating bootstrapping in fhew using gpus. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 128–135, 2015.

[18] X. Lei, R. Guo, F. Zhang, L. Wang, R. Xu, and G. Qu. Accelerating homomorphic full adder based on fhew 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.

[19] H. A. Leon Liu, S. S. Weber, N. F. Qiao, and A. J. Hartshorn. Number theoretic transform (ntt) fpga accelerator. Technical report, Worcester Polytechnic Institute, 100 Institute Road, Worcester MA 01609-2280 USA, May 2020.

[20] 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.

[21] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. Cryptology ePrint Archive, Report 2012/230, 2012. https://eprint.iacr.org/2012/230.

[22] Mert. Parametric ntt/intt hardware. https://github.com/acmert/parametric-ntt, 2020.

[23] 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.

[24] D. Micciancio and Y. Polyakov. Bootstrapping in fhew-like cryptosystems. *IACR Cryptol. ePrint Arch.*, 2020:86, 2020.

[25] 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.

[26] NuCypher. Nufhe, a gpu-powered torus fhe implementation. URL: https://nufhe.readthedocs.io/en/latest/, last checked on 2020-03-26.

[27] T. Oder and T. Güneysu. New hope. https://www.seceng.ruhr-uni-bochum.de/research/publications/implementing-newhope-simple-key-exchange-low-cost-/, 2017.

[28] 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.

[29] C. Peikert. A decade of lattice cryptography. Cryptology ePrint Archive, Report 2015/939, 2015. https://ia.cr/2015/939.

[30] 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.

[31] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6), Sept. 2009.

[32] D. Rijmenants. One-time pad. URL: http://users.telenet.be/d.rijmenants/en/onetimepad.htm, last checked on 2020-04-13.

[33] F. Roy. Heaws. https://github.com/KULeuven-COSIC/HEAT, 2020.

[34] S. Roy. Compact ring-lwe cryptoprocessor. https://gitlab.lrz.de/tueisec/post-quantum-crypto, 2014.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] Wang. qtesla. https://caslab.csl.yale.edu/code/qtesla-hw-sw-platform/, 2020.

[41] 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.

[42] Xilinx. *ZCU102 Evaluation Board*, 06 2019. PS-Side: DDR4 SODIMM Socket.

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