Chapter 1

Introduction to homomorphic encryption

1.1 An introduction to Encryption

Let's say it's 1943 and you are the head of a resistance movement that helps downed pilots escape. To get the pilots out of Europe, you need to be able to communicate to Britain to be able to arrange a pick-up in Spain. The only way to do this quickly and reliably is via the 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 on Figure 1.1 and were used to turn the message, which we call "plain-text" into a jumble of seemingly meaningless letters. This process of turning plain-text messages into messages that cannot be understood by eavesdroppers is called encryption. Only the receiver in Britain, with the only copy of the one-time pad in the world, would be able to turn the encrypted message back into the original message.

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

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

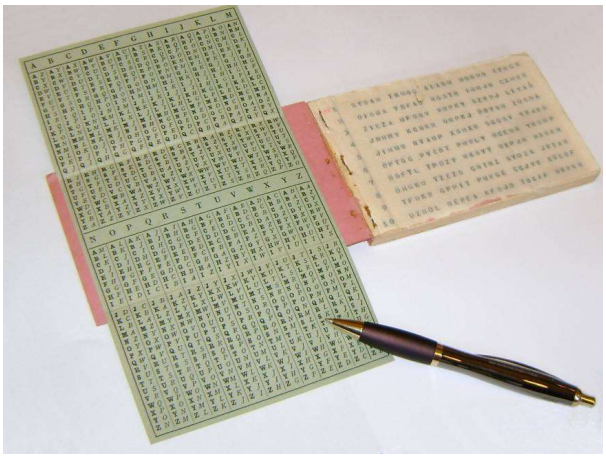

Figure 1.1: One Time Pad [13]

has encrypted his message, he cannot reverse this process.

It is as if Alice has 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 (a key to that safe that the receiver can use). If you are the receiver and you've chosen a secret key, you should be able to create a public key from this secret key. However the opposite should obviously be impossible! After all, the public key is being given away to anyone and should not give (much) information about the secret key. Given enough time, you might be able to reverse engineer the public key into a secret key, so we call a scheme "secure" (i.e. good enough) if it takes so long and requires so many resources that it might as well be impossible. Impossible meaning that the required time to break the secret key is on the order a small country dedicating all its computers for millennia for high enough security needs.

The schemes that were used over the last 50 years did a pretty good job of this. 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. The scheme approximately consists of taking the secret key and (part of) the public key, multiplying them together and then adding some random error. The result of this operation doesn't contain enough information to reverse engineer the secret key , and so the result can be distributed to anyone safely. In addition to being resistant to quantum attacks, LWE has another neat feature: it can be used for Homomorphic Encryption.

1.3 Homomorphic Encryption

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

There are 2 types of homomorphic encryption: Somewhat Homomorphic Encryption and Fully Homomorphic Encryption. Somewhat Homomorphic Encryption allows for a limited number of computations, Fully Homomorphic Encryption allows for an infinite number of computations. In 2009 Gentry [7] 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. She creates a box that allows the workers to manipulate the jewels. This box represents Somewhat 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.[2]

Figure 1.2: The jewel box [2]

The “box” in this analogy is of course our encryption scheme. “Alice” is the user while the workers represent a server that can do operations (such as addition, multiplication or others) on the encrypted data without being able to access it. However, because the “box” in our case is the encryption scheme “Learning With Errors”, every operation that is done will increase the amount of error added to the final result. In the beginning, this error is small, meaning that we can still recover our result simply by rounding. At some point, the number of operations will cause the noise to exceed the threshold at which we can remove it. At that point, our “worker” (i.e. the server which is doing the operations) has to stop or the output it gives will be wrong.

The idea behind "bootstrapping" is to turn Somewhat 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.

Then our workers can open the inner 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. 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, unless of course we bootstrap in time to reduce the noise by performing a decryption.

1.4 Encryption Libraries and homomorphic encryption schemes

Because Homomorphic Encryption would be so widely applicable, 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/ etc ....

We can split the homomorphic schemes that are currently the focus of most research into 2 (or 3) generations. The first generation consists of older schemes that are so inefficient they are no longer being used. The second generation consists of schemes like the Fan-Vercauteren [6] scheme, which is primarily used for Somewhat Homomorphic Encryption. The second-generation schemes allow for multiplication and addition but have a very compute-intensive bootstrapping process. The third generation schemes only allow for 1 gate (such as a AND/OR gate), and bootstrap immediately, but the bootstrapping happens relatively fast ($\tilde{1}$00ms)[9]. In this master thesis, I have accelerated one of the third generation schemes, namely Fastest Homomorphic Encryption in the West (FHEW) but it is still worth to first consider the Fan Vercautren scheme.

1.4.1 Fan-Vercauteren: usually Somewhat Homomorphic Encryption

FV has been the subject of hardware acceleration by multiple groups [10] [16] [14]. It is also implemented in the homomorphic encryption libraries SEAL and PALISADE among others.
The problem with the second generations 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 [14]. 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) is part of the so-called third-generation schemes[4] [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 actually be done within a reasonable amount of time, namely 137 milliseconds for NAND + bootstrap for 128-bit security on an intel i7 [9]. 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, no limit on the functions that can be executed.

Because of its flexibility, a FHEW scheme that operated fast enough would have a large advantage in usability over the FV 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 back to the client. The entire FHEW scheme can be treated as a black box by those using it, and it is for this reason that I chose this scheme to accelerate.

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: general purpose electronics needs to occupy itself with reading the instructions that are sent to it. Specialized electronics don't need to do this, since they would always get the same instructions, and as such don't need to waste time on understanding what they have to execute and can dedicate all their resources to execute very specific operations. In this thesis, we use a FPGA, which is a form of *programmable* electronics, where different electrical components are wired together according to a netlist that can be programmed from a computer.

The main advantage of hardware acceleration is that it can provide a speedup of a factor 10 to a factor 100 for a given task.[14] The disadvantage is that by its very nature, it requires specialized electronics to be used. Those electronics might

one day end up in a consumer device, such as laptop, smartphone, or in the case of homomorphic encryption, a server, but this is unlikely to occur soon, and requires significant investment. Hardware requires a new chip to become part of the server hardware, and so would normally only start to function once the servers are replaced with new servers that contain the chip. 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 those investments pointless. Software is much easier to deploy, and requires no expensive investment. However, recently web hosting companies such as Amazon Web Services have started making servers with programmable FPGA's available [15], 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 problems faced by the homomorphic encryption schemes.