The first major component of this thesis was selecting what homomorphic encryption scheme to implement. Before I go on, I first have to ask, is everyone here familiar with the basics of homomorphic encryption and bootstrapping?

If not:  
Homomorphic encryption means encryption on which functions can still be executed. In 2009 Gentry showed that Fully Homomorphic encryption, doing an arbitrary amount of operations on encrypted data is possible. He later 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, like this one. This box would Somewhat Homomorphic Encryption, because while the workers can do some operations, eventually the gloves break down and the workers have to bring the box back to Alice so that she can unlock the box, retrieve the finished jewels.  
  
The idea behind bootstrapping is to turn Somewhat Homomorphic Encryption in to Fully homomorphic encryption by performing the decryption, i.e. the unlocking of the box, while everything is under another encryption. In our analogy, this would be like locking our box, in another box which already has a key for our box inside of it. (Since we using public/private key encryption, we place things into a locked box with the public key). Then our workers can open the inner box and continue working on the jewels. For an arbitrary amount of boxes, we can thus compute an arbitrary amount of functions on data that stays encrypted. The fact that gloves wear out in our analogy is representative of the fact that the homomorphic encryption schemes uses some form of numerical noise to mask the message. As we do addition, and especially multiplication, the amount of noise increases. At some point, decryption (whose last step is usually a sort of rounding operation to remove noise) will no longer return the correct answer.

Just for clarity, I am going to call the person with secret key who encrypted the data originally the user, and the person executing the function on the encrypted data the server.

An example of homomorphic encryption can be seen in the Fan-Vercauteren scheme. This is based on the Ring Learning With Errors Problem, which states the following. For many vectors (ai,bi) ϵ (Rq, Rq), bi = ai \* s+ ei with ei being a randomly sample error term and s being our secret key and ai being a randomly sampled but known vector, it is not possible to find s (“not computationally feasible” (Roy et al.))

So in other words, we can use this term as a good public key for s since we can’t reverse engineer the secret from the public key. And this is the case if the server spends all day requesting public keys from the user to be able to guess the secret key.

Now this is an extremely simplified version of the Fan Vercauteren scheme, just to give an idea: we generate our cipher text from our public key and our message, in such a way that we can get our message back by multiplying the second part of the cipher text with s and adding it to the first part.

Now why do we use this scheme? Because if we add the 2 ciphertexts or multiply the 2 cipherteksts, we still get the correct value after decryption! (if we’ve taken our modulus high enough).

Addition is the most straightforward: when we add, we add the 2 errors of the ciphertexts together and error growth is minimal.

Multiplication is much less straightforward. I did not write down all the terms of the multiplication, but to summarize: afterwards we need to rescale with a certain factor, and relinearize.

We need to rescale because the actual algorithm is scaling with a certain factor, and if you multiply 2 values scaled with a factor delta, the result will be scaled by a factor delta² and thus we need to divide back by delta to get our original scaling.

Secondly, we need to be able to remove the secret key squared term from our result. We call this the linearization step, but I will not go into details here.

To understand how FV actually worked and get an idea for the size of the parameters I was going to be working with, I installed micrsoft SEAL, a library for homomorphic encryption. I tried implementing a simple function (for calculating the angle you have to fire something at to reach a certain distance, which means calculating an arcsin, which can be approximated using a taylor expansion). Because there is no bootstrapping, it is necessary to determine in advance what size of polynomial coefficients will be for every multiplication done to calculate the polynomial series. Additionally, as the TA’s told me and I also found out by trying, it is hard to floating point operations with integers, when rescaling has to be done after every multiplication, requiring another multiplication with a constant factor.

Therefore I switched to CKKS, which can be seen as a similar scheme designed for floating point operations.

To give an idea of what needs to be done: Firstly, the coefficient modulus of the polynomials must be chosen, so that there is enough noise budget to allow for the function that you want to execute. Then the scale (i.e. the precision of the floating point numbers) must also be chosen. Then public, private and realinerisation keys must be generated. Finally we can start doing multiplications and addition, but after every multiplication we must rescale and relinearize, and we need to do modulus switching as we go through the list of coefficient modulus primes that we generated to do our calculation.

This required about a 100 lines of code for a rather simpel formula. It executed fairly quickly, and both FV and CKKS are good at allowing Single Instruction, Multiple Data, but as can be seen from this example, they only allow for a limited amount of computations, and algorithms that are already out there and require a lot of instructions will thus not easily translate in to programs that can homomorphically execute data.

They are however good at doing homomorphic encryptions for a given, low-depth application. This can also be seen in a hardware acceleration for the Fan-Vercauteren scheme done by COSIC in 2018. This implementation could do FV very quickly, at 400 homomorphic multiplications per second, but is limited to a multiplicative depth of 4. This is enough for a lot of applications, but can limit its flexibillity in a general role.

# FHEW

FHEW, called that way as a reference to the Fastest Fourier Transform in the West library, is part of the so-called third generation schemes. 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 137miliseconds for NAND + bootstrap for 128 bit security on a intel i7. In other words, while only one NAND gate can be done at a time (later papers show parellisation 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.

FHEW is also based on RLWE. The slides I’m about to show are taken from a excellent, very visual presentation by the creators of FHEW, Léo Ducas and Daniele Micciancio. Our message is a single bit. If it is 0, we could represent it as some error, if it is 1, we could represent it as 256 with some error added to it. (512 is a usual value for q). In this case our maximum error is equal to +/- 128.

Now the representation that we actually use is one where there are 4 possible messages, with the message being multiplied by 128, or ¼ of the modulus q. We also have a smaller error of value 64 or q/8, so that there are values that are never reached by a given message and error. (these white spaces).

We can use this for addition. As you can see, the result now has more error than the beginning. Now if we consider every on the top left to be 1 and everything on the bottom right to be 0, we have a AND gate. To make this a NAND, we rotate by 5q/8, in other words we consider the bottom right to be 1 and the other 0 and this results in the results that we expect from a NAND gate for given input values.

Then we need to bootstrap: remember, bootstrappping is doing 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.

How do we bootstrap? Well before we start I’m going to try to give parameters of the values we are using as we go because I found it very difficult to get an idea of what was actually happening in the code as I read these papers and went over the code.

The version I’m using here is slightly different from the version that was discussed in the slides of Léo Ducas and Daniele Micciancio from 2015. This is an optimized version and is the version that I would like to implement in hardware. I’d like to implement this because of the advantages previously mentioned and because as far as I can tell, the only optimisation paper 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.

There is also NuFHE, which uses something similar to FHEW called TFHE with GPU acceleration and succeeded in a 100 times speed-up, bringing the cost of homomorphic encryption down to 0.13 ms/bit for binary gates (for FFT implementation). For NNT implementations it is 0.35 ms/bit <https://nufhe.readthedocs.io/en/latest/implementation_details.html>

Ok, now let’s explain the 2020 version of FHEW:

LWE ciphertexts look as follows and standard parameters are given. If we simply wanted to decrypt, we would calculate b-<a,s> and retreive our message through rounding, but this requires a secret key and gives us our result, which the server isn’t supposed to have.

Instead, we want to encrypt this scheme under another scheme so we can decrypt the LWE ciphertext and gain the benefits, namely noise removal from our ciphertext due to rounding, without making security pointless by having our decrypted result be unencrypted. While we remove the cipher text, we add some noise because of the operations that we are doing on our cipher text and because of the noise of the encrypted secret key.

Now there are 3 mains steps in any bootstrapping algorithm: First we initialize an Accumulator as b, then we update this b the product of our secret key and a, the first part of our LWE vector. Finally we extract our result, which should be an encryption of a rounding function applied to our b-as. That is what we want because we want to decrypt, round, but have this rounded result be encrypted so everything can be done on the server.

Now LWE is less efficient than RLWE so we take the input and turn it into RLWE. We also encrypt our secret keys using RGSW, which is a scheme that encrypts a message as 27 RLWE encryptions of 0, with message multiplied with shifted unity matrix 27 times. Q is a 27 bit prime. In our scheme, rather than multiplication with 2, we are multiplying with 23, but this is simply an optimisation.

If our input is a LWE value called v, then it becomes an encrypted Xv RLWE value. To compute a rounding function f (which also does our rotation for our NAND), we store all the function output for given the function inputs 0 to 255, and we choose our rounding function so that the output for a value more that 255 is equal to the negative output of value minus 255. Then we can initialize our accumulator to a polynomial with the function outputs as coefficients.

What we return, namely **(0, m)** is a noiseless RLWE with an **a** that’s equal to 0. But it is an RLWE encryption so we can do operations with this and add noise.

Now we come to accumulation. Seen from the LWE standpoint, we’re now going to subtract the inner product of a\*s from our b. a is public and in plaintext, but we are allowed to multiply a RGSW value by a plaintext a. We then “subtract” this a*i*\*si to our Accumulator by multiplying the RGSW and RLWE encryptions together. This is done via a special multiplicatio which returns an RLWE encryption of the Accumulator which, from a RLWE standpoint, rotates the functions by ai\*si , and which, from a LWE standpoint, looks like an subtraction with ai \* si. We then extract our result by taking the first value of our rotated functions, which is going to be our decrypted LWE whose error has been rounded of by our functions (and which has been rotated as required in advance). We would then have to do a key switching operation to complete our algorithm but…

The most performance challenging part for the server is of course the accumulation step. Every RLWE times RGSW operation is a vector times matrix operation. With the matrix in this case having dimension 1025\*6150 and the vector having dimension 1025.

# Numerical fourier transform hardware implementations

Roy: Gives explanation on security of LWE

Has a good explanation on the algorithms used