Introduction to mathematical cryptography
PCMI 2022 Undergraduate Summer School

Christelle Vincent

August 3, 2022

Contents

1 Computational complexity

As mentioned in the first lecture, the fundamental ideal behind cryptography is that there are operations that are **easy** to do but **hard** to undo. In this section we make this notion more precise.

1.1 How to measure complexity

In mathematics, an **algorithm** is a finite sequence of instructions or computations that, when performed, yield a result. For example, you might have learned to multiply integers in school by applying the so-called “schoolbook algorithm” for multiplication. Notice that here the algorithm is not just “multiplying two integers” but the specific process by which the answer to the multiplication problem is obtained. This might seem surprising, but there are other algorithms to multiply integers that have completely different steps to reach the same answer![1]

Given a specific algorithm, the **computational complexity** of that algorithm is how much resources it takes to accomplish the computation. The word “resources” is purposefully vague here: In this course we will almost exclusively talk about either time complexity or arithmetic complexity, which counts either how much time it takes for the algorithm to complete, or how many arithmetic operations must be performed to complete the algorithm. Another quantity that is often of interest is the amount of memory or storage necessary to perform a computation, but that will not come up for us except possibly in passing.

In the case of our example, the schoolbook algorithm for multiplication (or “schoolbook multiplication” for short), it’s perhaps most natural to consider the arithmetic complexity of the algorithm, by which here we mean the total number of additions and multiplications necessary to perform the computation. For example, to compute 15×6, we would:

1. Multiply $5 \times 6 = 30$, then
2. multiply $1 \times 6 = 6$, then
3. add $6 + 3 = 9$,

using 3 operations to obtain the answer 90.

Pretty quickly, it becomes clear that even if we always apply the schoolbook multiplication algorithm perfectly, the number of steps necessary to perform the computation depends on the specific numbers that we are multiplying. Numbers with more digits take a lot more

[1] If you are interested, you can look up the Karatsuba algorithm or the family of Toom-Cook algorithms, which are a generalization of the Karatsuba algorithm.

steps to multiply, but even two pairs of numbers with factors of the same size might not take exactly the same number of steps because of carries which introduce extra additions (here, by a "carry" I mean the addition in the problem 15×6, where the 3 tens from the units multiplication get added to the 6 tens from the tens multiplication). Since the number of arithmetic steps we need to perform depends on the numbers being multiplied, what could we possibly mean by the complexity of the whole algorithm?

After more experimentation with various numbers, we might notice that the main factor that influences the number of steps in a multiplication problem is the **size** of the numbers being multiplied. By this we mean that while adding the carries does introduce some extra operations here and there, "most" of the operations we perform are multiplications, so integers with the same numbers of digits require roughly the same number of operations to multiply.

This will turn out to be the case for the majority of the algorithms we talk about, so we will, from now on, always count the number of arithmetic steps of an algorithm as a function of the size of the input integers.

Definition 1.1. Let n be a positive integer. Then its **size** k is the number of digits in its decimal expansion. This is given by the expression

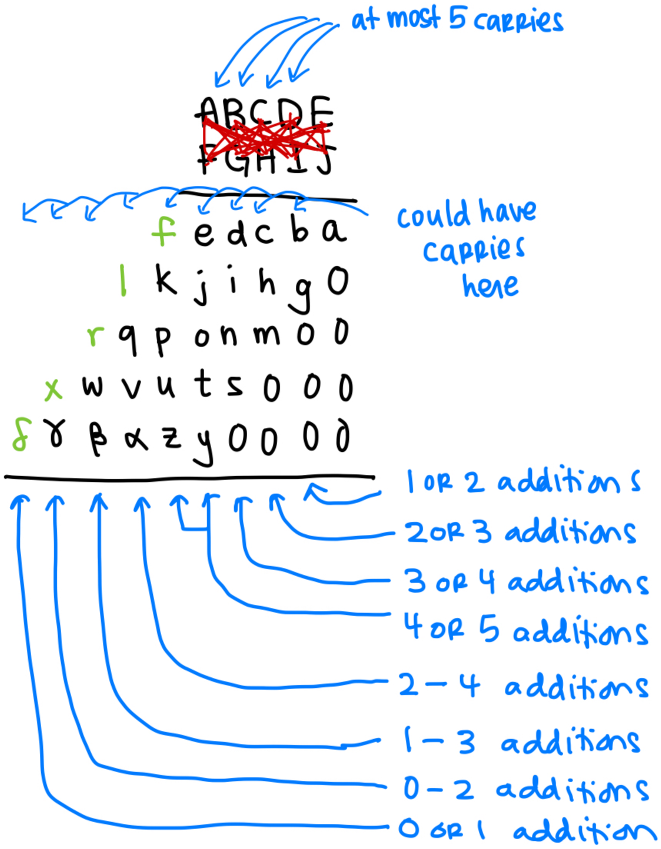
$$k = \lfloor \log_{10} n \rfloor + 1,$$

where $\lfloor \cdot \rfloor$ is the floor function. Often computing books use instead the number of bits in the binary expansion of the integer, which is given by the formula

$$\lfloor \log_2 n \rfloor + 1.$$

We will see that for our purposes we can use either notion of size interchangeably.

To see exactly how the number of steps in the schoolbook multiplication algorithm depends on the size of the integers being multiplied, let's consider multiplication of two five-digit integers. The image below shows what we mean:



Let's count the operations, noting the ones that would be done for any five-digit multiplication, and the ones that only occur sometimes:

- There are always twenty-five multiplications, as each digit must multiply each digit.
- There are also always at least sixteen additions in the column additions at the bottom.
- Carries during multiplications can add up to 5 additions for each of the four positions, so up to twenty additions.
- The column additions at the bottom could have up to thirteen additional additions.

In total there are therefore 41 operations that must always be performed, and up to 33 additional additions depending on carries. Therefore if $f(5)$ is the number of steps it takes to multiply two five-digit integers using schoolbook multiplication, we have

$$41 \leq f(5) \leq 74.$$

Compare this to multiplying two six-digit integers, when there are 61 operations that must always be performed, and up to 46 additional additions depending on carries. In general, if $f(k)$ is the number of steps that it takes to multiply two integers each with k digits using schoolbook multiplication, then we have

$$2k^2 - 2k + 1 \leq f(k) \leq 3k^2 - 1,$$

where

$$2k^2 - 2k + 1 = k^2 + \frac{k(k-1)}{2} + \frac{(k-1)(k-2)}{2}$$

and

$$3k^2 - 1 = k^2 + \frac{k(k-1)}{2} + \frac{(k-1)(k-2)}{2} + k(k-1) + k + 2(k-1).$$

Therefore one can say that multiplying two k-digit integers using schoolbook multiplication takes at least k^2 steps, but no more than $3k^2$ steps.

This is already pretty neat, but we do even say a bit more. This is because when k is very large, the difference between k^2 and $3k^2$ remains the same (one number is 3 times as big as the other). That's a "cost" that's built into our algorithm and which doesn't depend on the size of the numbers we are multiplying, and therefore we don't always want to keep track of it. It's a lot easier to remember that schoolbook multiplication takes "about" k^2 steps, since that's the number that will give you a sense of how long a multiplication problem will take.

To formalize what we mean by "about" k^2 steps, we need some notation. First we need a notion for a function that eventually becomes smaller than a multiple of another function:

Definition 1.2. Let f and g be two functions taking as input positive integers, and outputting positive integers. We will write this as $f, g \colon \mathbb{N} \to \mathbb{N}$. We write that

$$f \ll g$$

if there are positive constants c and C such that

$$f(k) \leq cg(k) \qquad \text{for all } k \geq C.$$

The expression "$f \ll g$" is read "f is less than less than g."

We will also need a notion for a function that eventually becomes bigger than a multiple of another function:

Definition 1.3. Let $f, g \colon \mathbb{N} \to \mathbb{N}$. We write that

$$f \gg g$$

if there are positive constants c and C such that

$$f(k) \geq cg(k) \qquad \text{for all } k \geq C.$$

The expression "$f \gg g$" is read "f is greater than greater than g."

These two notions together allow us to define when two functions are eventually "about the same magnitude:"

Definition 1.4. Let $f, g\colon \mathbb{N} \to \mathbb{N}$. We write that

$$f \sim g$$

if we have that

$$f \ll g \quad \text{and} \quad f \gg g.$$

The expression "$f \sim g$" is read "f is of the order of g."

There is often an easy way to determine if $f \sim g$:

Proposition 1.5. Let $f, g\colon \mathbb{N} \to \mathbb{N}$. Then if the limit

$$\lim_{k\to\infty} \frac{f(k)}{g(k)}$$

exists and is nonzero, $f \sim g$.

Now with this notation, we can say that if f is the number of steps it takes to multiply two k-digit integers, then $f \sim k^2$. Accordingly, we usually say that schoolbook multiplication can be accomplished in "quadratic time" since the number of steps f is of the order of a quadratic polynomial.

1.2 Easy and hard problems

Now that we have a way to express about how many steps it takes to perform a computation, we can talk about "easy" and "hard" problems. First, again here we must make precise that we cannot really talk about the complexity of solving a problem without having an algorithm in mind for solving that problem. However, we commonly will say that a problem is "easy" or "hard" depending on the number of steps it takes to solve the problem using the most efficient known algorithm.

In addition, here all of our problems are solved by algorithms, we assume that we can accomplish all the steps, so an "easy" problem is simply one that we can solve quickly, and a "hard" problem is one that takes an unreasonable amount of time to solve.

More precisely:

Definition 1.6. Let $f\colon \mathbb{N} \to \mathbb{N}$. We say that f grows **polynomially** if there are positive constants a and b such that

$$k^a \ll f(k) \ll k^b.$$

If the number of steps in an algorithm, when given as a function of the size of the inputs to the algorithm, grows polynomially, then we say that this algorithm is **fast**.

If a problem can be solved by a known fast algorithm, then we say that this problem is **easy**.

At the other end of the spectrum we have:

Definition 1.7. Let $f\colon \mathbb{N} \to \mathbb{N}$. We say that f grows **exponentially** if there are positive constants a and b such that

$$e^{ak} \ll f(k) \ll e^{bk}.$$

If the number of steps in an algorithm, when given as a function of the size of the inputs to the algorithm, grows exponentially, then we say that this algorithm is **slow**.
If the most-efficient known algorithm to solve a problem is slow, then we say that this problem is **hard**.

We will see that there is also an intermediate "speed" at which certain problems can be solved:

Definition 1.8. Let $f\colon \mathbb{N} \to \mathbb{N}$. We say that f grows **subexponentially** if for every positive constants a (no matter how big) and b (no matter how small) we have

$$k^a \ll f(k) \ll e^{bk}.$$

In other words, f is "larger" than any polynomial, but "smaller" than any exponential function.
If the most-efficient known algorithm to solve a problem has a number of steps that grows subexponentially, we usually still think of the problem as hard, but we must keep in mind that it is not exponentially hard.

2 DLP, Elgamal, and attacks

The first family of ciphers we will study rely on the difficulty of solving a certain problem, the discrete logarithm problem, in certain well-chosen groups.

2.1 The discrete log problem

Let G be a finite cyclic group. Then the **discrete logarithm problem** (DLP) in the group G is

Problem 2.1 (The discrete log problem)**.** Given a generator g of G and $h \in G$, find $0 \leq x < \#G$ such that

$$h = g^x.$$

The meaning of the name of this problem is the following: First, if g and h were real numbers, then we would write

$$h = g^x \quad \text{if and only if} \quad x = \log_g h,$$

where $\log_g$ is the real logarithm in base g. Therefore, it makes sense that the problem of finding x such that $h = g^x$ should be called a logarithm problem in general.

Secondly, the most "natural" topology on a finite group is the discrete topology, in which each singleton is an open set. This is in contrast to the usual topology on the real line, in which every open set around a point, no matter how small, contains as many points as there are in the real numbers. It turns out that this difference is crucial to make the logarithm problem hard, in the sense of Definition 1.7, in some groups. Indeed, in the discrete topology there is no good approximation to the logarithm given by a Taylor series, and therefore there might not always be a fast algorithm to solve the discrete log problem in a discrete group.

As a word of caution, as we saw in Monday's lecture, the discrete log problem is not always difficult for every discrete group G! If $G = \mathbb{Z}/n\mathbb{Z}$, equipped with addition modulo n as its operation, with generator $g = 1$, then the discrete log problem asks, given $h \in \mathbb{Z}/n\mathbb{Z}$, to find $0 \leq x < n$ such that

$$h \equiv x \pmod{n}.$$

This can be done quickly (in the sense of Definition 1.6) using long division. Indeed, if (as is likely) $h \in \mathbb{Z}/n\mathbb{Z}$ is given as an integer representing the equivalence class of h in $\mathbb{Z}/n\mathbb{Z}$, then x is simply the remainder of (the integer) h when it is divided by n.

2.2 The Elgamal cipher

If G is a group where the multiplication and inversion operations are fast but solving the discrete log problem is believed to be hard, then thanks to the fast exponentiation algorithm[2], the operation of raising g to a power x to obtain $h = g^x$ is fast, but the operation of recovering the exponent x from g and h is slow. We now explain how this can be leveraged into building a cipher.

Recall that the basic set-up in public key cryptography is that a person A would like to send a message to a person B on a public channel securely. To do that, person B (the recipient of the communication) must first establish private and public keys. Person A can then use the public key to **encrypt** a message to person B, turning the **plaintext** of the message into unintelligible **ciphertext** that can be shared on the public channel. Upon reception, person B can use their private key to **decrypt** the message.

Algorithm 2.2 (Elgamal key generation)**.** To prepare to receive a secure communication, person B:

1. First chooses a cyclic group G in which the DLP is hard, and a generator g for this group.

2. Then generates a **random secret** value x with $2 \leq x < \#G$. This value x is the secret key.

3. Finally computes $h = g^x$. The values G, g and h are the public key.

Take a moment to convince yourself that person B does not have to perform any hard computations to generate the keys. This is important, as we need every algorithm of the

[2] We unfortunately will not have time to study this algorithm more than we already have during XML. We will simply note that the naive way to exponentiate, as in the July 19 problem set, is **slow**, and therefore the existence of a fast exponentiation algorithm is crucial here.

cipher to be fast. Once the keys have been generated, person A can now encrypt a message for person B:

Algorithm 2.3 (Elgamal encryption)**.** Given an Elgamal public key (G, g, h), where G is a cyclic group generated by g and $h \in G$, and a plaintext $m \in G$, person A:

1. First generates a **random secret** value y with $2 \leq y < \#G$.
2. Then computes the ordered pair of ciphertexts

$$c_1 = g^y, \quad c_2 = m \cdot h^y,$$

   where here $\cdot$ is the group operation in G.

The ciphertexts are then sent to person B on the public channel.

Once again, take some time to confirm that person A does not have to perform any hard computations. In addition, notice that at no point does person A need knowledge of the secret key x. Upon reception of the ordered pair of ciphertexts c_1 and c_2, person B performs the following decryption algorithm to recover the message m:

Algorithm 2.4 (Elgamal decryption)**.** Given an Elgamal private key x corresponding to a public key (G, g, h), and an ordered pair of ciphertexts c_1, c_2, person B computes

$$m = c_1^{-x} \cdot c_2.$$

First, take a moment to verify that this equality is correct. Then once more we must make sure that person B does not have to perform any hard computations to decrypt the message: The quantity c_1^{-x} can be computed by first computing c_1^x, which is fast since person B knows the secret key x, and then taking the multiplicative inverse of this quantity. This is fast by assumption. Finally, multiplying by c_2 is also fast. We also note that to perform the decryption algorithm, person B does not need knowledge of person A's secret value y. This value is discarded by person A after the communication is complete.

2.3 Attacking the Elgamal cipher

If an adversary E can compute discrete logarithms in the group G, then adversary E will be able to recover the message m. Indeed, adversary E can obtain the secret key x from the public parameters g and h by computing a discrete logarithm, and then decrypt as easily as person B.[3] This is not the only problem that adversary E can solve to recover the message m.

Indeed, if adversary E can compute g^{xy} from the values g, $h = g^x$, and $c_1 = g^y$ (which are all public), then adversary E can recover the message by computing

$$m = g^{-xy} c_2.$$

[3] Note that adversary E could also obtain the secret value y from the parameters g and c_1 if they can compute logarithms in G; this also allows adversary E to obtain the message since then $m = c_2 \cdot h^{-y}$.

This problem, to recover g^{xy} from g, g^x, and g^y, is called the **Diffie-Hellman problem**. It is possible that this problem is easier to solve than the discrete logarithm problem, but we do not have at this time a better algorithm to solve the Diffie-Hellman problem than first solving the discrete log problem to obtain the exponents x and y to compute g^{xy}. Therefore we will focus on solving the discrete logarithm problem.

To figure out if an algorithm to solve the discrete problem is fast or slow, we must first define the size of the input to our problem. Here, the value which drives how difficult the discrete log problem is to solve is $\#G$, the order of the group G. Therefore, we will compute the number of steps to perform the algorithm as function of the size k of $\#G$. Remember from Definition 1.1 that we have

$$k \approx \log \#G,$$

where we can use the logarithm in any base $a > 1$ for our purposes.

2.4 Baby steps, giant steps

The provably fastest algorithm we have to solve the discrete logarithm problem in a general group G is called the **baby steps, giant steps algorithm** and it is due to Shanks. As we know, in specific groups G, such as $\mathbb{Z}/n\mathbb{Z}$ under addition, the discrete logarithm problem can sometimes be solved much faster with a special purpose algorithm that exploits properties of the specific group G. However, if we do not have any information about the group G other than it is cyclic, this is the fastest algorithm. Fortunately, the number of steps in this algorithm grows exponentially in $k \approx \log \#G$, so in a general group G solving the discrete logarithm problem is difficult.

The idea behind the algorithm is the following: Given a cyclic group G with generator g and $h = g^x$ for some unknown exponent $0 \leq x < \#G$, then for any integer N such that $\#G \leq N^2$, we can write

$$x = x_0 + Nx_1$$

with $0 \leq x_0, x_1 < N$.

Now think of "walking" between g and h by taking "baby steps" consisting of multiplication by g. For example, we can think of going between g and g^4 by taking three baby steps: one from g to g^2, a second from g^2 to g^3 and finally a third from g^3 to g^4. Of course if we take only baby steps, it will take $x - 1$ steps to "walk" from g to $h = g^x$. Taking only baby steps is a naive, brute force search, and can take as many as $\#G - 2$ steps.

To speed things up, we will also allow ourselves to take "giant steps" consisting of multiplication by g^N. Then we can "walk" from g to $h = g^x$ by first taking x_0 baby steps, and then x_1 giant steps. If we choose N^2 very close (but larger than!) $\#G$, then $N \approx \sqrt{\#G}$, so this allows us to "walk" from g to h in at most $2\sqrt{\#G}$ steps, since each of x_0 and x_1 is less than N.

We must now leverage this understanding into an algorithm that allows us to find x_0 and x_1 (and therefore the discrete logarithm x since $x = x_0 + Nx_1$). We do this by thinking of the baby steps and the giant steps meeting somewhere in the middle between g and h:

Walking “forward” from g with x_0 baby steps, we get to g^{x_0}, which is the same place we get if we walk “backwards” from h with x_1 giant steps.

Another way to think about this is the following, and this is what gives us the baby steps, giant steps algorithm:

1. First, adversary E computes and stores the values $1, g, g^2, g^3, \ldots, g^{N-1}$. We are guaranteed that g^{x_0} will appear among these values.

2. Then adversary E starts at h and take giant steps backwards, computing $hg^{-N}, hg^{-2N}, hg^{-3N}, \ldots$. At some point, after $x_1 < N$ giant steps, adversary E will reach one of the values $1, g, g^2, g^3, \ldots, g^{N-1}$ they have already computed. This reveals the values of x_0 and x_1 to adversary E.

All that remains is to estimate the complexity of this algorithm; to do so we consider one group operation to be one step. To compute the values $1, g, g^2, g^3, \ldots, g^{N-1}$, adversary E must perform approximately N or $\sqrt{\#G}$ group operations. This is already exponential in k, the size of $\#G$. Assuming that we can compute in G efficiently, computing g^{-N} can be done in time polynomial in k using fast exponentiation and then inverting the answer. Once this is done, adversary E has then to perform at most N or $\sqrt{\#G}$ steps. We therefore see that the computation takes approximately $2\sqrt{\#G} \sim 2^{k/2}$ steps in the worst case. (We note that it also requires storing approximately $2\sqrt{\#G} \sim 2^{k/2}$ elements of G in the worst case, so this algorithm is also exponential in storage complexity.)

2.5 The index calculus attack

In contrast with baby steps, giant steps, this attack is specifically for the situation where G is represented as $(\mathbb{Z}/p\mathbb{Z})^\times$. Since this attack uses more information, it is faster; it is a **subexponential** attack on the DLP on $(\mathbb{Z}/p\mathbb{Z})^\times$. The idea of the attack is to blur the distinction between elements of $\mathbb{Z}/p\mathbb{Z}$ and their least residue, and to use this to factor the least residues into small primes when possible. Here is how it goes:

Given a cyclic group G with generator g and $h = g^x$ for some unknown exponent $0 \leq x < \#G$, at the beginning of the algorithm, adversary E chooses a suitable bound B. In practice, B should be approximately $2^{\sqrt{\frac{\log p \log\log p}{2}}}$. Adversary E then computes and stores the set of primes that are strictly less than B:

$$\{\ell_1, \ell_2, \ldots, \ell_r\},$$

this set is called the **factor base**.

The first step of the algorithm is for adversary E to compute $\log_g \ell_i$ for each ℓ_i in the factor base. This is done in the following manner: For random exponents i, adversary E computes g_i, the least residue of $g^i \pmod{p}$, and checks if g_i is divisible **only** by primes in the factor base. If not, then g_i is discarded. If it is, then we have

$$g_i = \ell_1^{e_{\ell_1}(i)} \ell_2^{e_{\ell_2}(i)} \cdots \ell_r^{e_{\ell_r}(i)},$$

and therefore

$$i \equiv \log_g g_i \equiv e_{\ell_1}(i) \log_g \ell_1 + e_{\ell_2}(i) \log_g \ell_2 + \cdots + e_{\ell_r}(i) \log_g \ell_r \pmod{p-1}.$$

Once adversary E obtains r equations of this type, they can use them to solve for the values $\log_g \ell_j$ for each ℓ_j in the factor base.

Now that this is done, adversary E turns to the second step, which is to compute the least residue h_j of hg^j for random values of j. If any h_j is divisible only by elements in the factor base, say

$$h_j = \ell_1^{e_1} \ell_2^{e_2} \cdots \ell_r^{e_r},$$

then we have

$$\log_g(h_j) \equiv \log_g h + j \equiv e_1 \log_g \ell_1 + e_2 \log_g \ell_2 + \cdots + e_r \log_g \ell_r.$$

Since the values $\log_g \ell_i$ are known, this allows adversary E to recover the value of $\log_g h$.

2.6 Implications for security

We begin with a short primer on security in cryptography. Typically, someone implementing a cipher will have a certain target **security level** they would like to achieve. For example, common communications on the internet might need to be secure, but top secret government information might need to be "even more" secure.

The security level of an instance of a cipher is measured in bits, where saying that a certain instance offers k bits of security means that an adversary has to perform as least 2^k steps to perform the attack on the problem. 128 bits of security is considered suitable for the internet to keep information safe for 10 to 20 years, 192 bits is an intermediate level of security, and top secret information usually requires encryption with parameters that offer at least 256 bits of security.

Let us first consider the Elgamal cipher with generic group G. In this case, the best attack is baby steps, giant steps. As we saw, if $\#G$ has size k bits, this attack takes about $2^{k/2}$ steps to complete. Therefore, if we wish to obtain 128 bits of security, we must have $128 = \frac{k}{2}$, or $\#G$ of size 256 bits.

The analysis of the index calculus attack is more difficult because we did not count the steps it takes to perform the attack, and because in that case the exact constants play a role in how large the size of p must be to ensure 128 bits of security. We will just note that if $p-1$ can be factored into small prime factors, then we can use Sun Zi's Remainder Theorem to reduce the problem of computing discrete logarithms modulo $p-1$ to the problem of computing discrete logarithms modulo those prime factors, which is faster. Therefore when using DLP with $G = (\mathbb{Z}/p\mathbb{Z})^\times$, we recommend using p of size at least 1024 bits, and such that $p-1$ has at least one prime factor of size 256 to guard against the baby steps, giant steps attack.

3 Lattice-based cryptography

This week we will discuss ciphers based on the Learning With Errors problem, which we will formally introduce on Thursday. We begin on Tuesday by introducing a particularly simple cipher based on this idea before getting into the higher-dimension stuff.

3.1 A simple lattice-based cipher

In this section we present in detail one example of a lattice-based cryptographic scheme, which can be found in a 2010 article by van Dijk, Gentry, Halevi, and Vaikuntanathan entitled *Fully Homomorphic Encryption over the Integers.* The article is available on our website. Almost every single lattice-based encryption algorithm is a variation on Regev's original Learning With Errors scheme (the NTRU schemes are the exception), as is this one.

3.1.1 Our new remainder

Throughout our study of this encryption scheme, we will need to compute remainders that are a little bit different than those you might be used to.

Definition 3.1. Let a, b be integers with $b \neq 0$. Then we define the two integers $q_b(a)$ and $r_b(a)$ to be the unique integers such that

$$a = q_b(a)b + r_b(a), \quad \text{and} \quad -\frac{b}{2} < r_b(a) \leq \frac{b}{2}.$$

Note that this is different from the "usual" remainder, which we take to be between 0 (inclusively) and b (exclusively). One way to obtain this new, exciting remainder, is to compute the usual remainder, and then check if it is less than or equal to $\frac{b}{2}$. If so, then you are done, for this value of a the usual remainder and the value $r_b(a)$ agree. However, if the usual remainder is strictly greater than $\frac{b}{2}$, then subtract b to obtain $r_b(a)$. (Then $r_b(a)$ is negative.)

Example 3.2. Let $b = 7$. Then we will have that for any a, $-\frac{7}{2} < r_7(a) \leq \frac{7}{2}$, or, using the fact that $r_7(a)$ must be an integer:

$$r_7(a) \in \{-3, -2, -1, 0, 1, 2, 3\}.$$

If $a = 10$, we can write

$$10 = 1 \cdot 7 + 3.$$

Here we have that 3 is a valid value for $r_7(10)$, so $q_7(10) = 1$ and $r_7(10) = 3$.

If $a = 12$, we can write

$$12 = 1 \cdot 7 + 5.$$

Here 5 is not a valid value for $r_7(12)$; it's too big! So we must adjust the equation:

$$12 = 2 \cdot 7 - 2.$$

Now -2 is a valid value for $r_7(12)$, so we have that $q_7(12) = 2$ and $r_7(12) = -2$.

Note that the journal version of the article uses several notations for $r_b(a)$: They also write $[a]_b$ and $(a \bmod b)$ for the same number. We will stick to the notation $r_b(a)$ in these notes.

3.1.2 The private and public keys

For this description we will talk about "large," "medium," and "small" integers. The article gives a mathematical, technical definition for what these words must mean for the scheme to work, and we encourage you to read the article if you are curious. For a first acquaintance with the scheme we will not be more precise here. To get a sense of the size of the difference parameters, you can refer to the example in Section 3.1.5.

The **private key** is a medium-sized odd integer p. When we need it later, we will sometimes denote the private key, or secret key, by `sk`.

The **public key** is a list of $\tau + 1$ different integers which we will denote $x_0, x_1, \ldots x_\tau$, where τ is very small. All of these integers are of the form

$$x_i = pq_i + r_i,$$

where

- each q_i is chosen at random so it is likely to be very large (actually q_i is within an interval of the form $0 \leq q_i < \frac{2^\gamma}{p}$, where 2^γ is very big, so q_i could be small, but an interval of the form above contains way more large numbers than small numbers, so it's likely that q_i will be very large), and
- each r_i is chosen at random to be much smaller than p.

Furthermore, we will assume that the list has been ordered so that x_0 is its **largest element**, and we will assume that x_0 is odd and $r_p(x_0)$ is even. If that is not the case once we have generated our list of integers, we throw out the whole list, and start over until we obtain a list of integers of the form $x_i = pq_i + r_i$ with the largest one of them x_0 satisfying the conditions that x_0 is odd and $r_p(x_0)$ is even. That is a valid public key. When we need it later, we will sometimes denote the public key by `pk`.

Remark 3.3. We pause here to discuss the underlying "lattice" in lattice-based cryptography, and what we are learning when we are Learning With Errors. Very roughly speaking, a lattice L in $\mathbb{R}^n$ is a set of evenly spaced out points, such that 0 is in L and if ℓ_1 and ℓ_2 are in L, then $\ell_1 + \ell_2$ is in L. In the scheme we are presenting, we have $n = 1$, and the lattice L is the set of all multiples of p:

$$L = \{\ldots, -3p, -2p, -p, 0, p, 2p, 3p, \ldots\}.$$

We see that these points are evenly spaced out (there is always a distance of p between one element of the lattice and the next), $0 \in L$, and L is closed under addition since the sum of two multiples of p is a multiple of p.

If we look at the public key while thinking about our lattice, we see that the elements x_i are each (relatively) very close to an element of L. This is because each r_i is chosen to be much smaller than p, so $x_i = pq_i + r_i$ is close to $pq_i \in L$. In fact by how we chose the r_is, we know that the nearest lattice point (multiple of p) for any x_i is pq_i. So we think of the public key as almost a list of elements of L, except that each element has a small **error** r_i.

The difficult problem to recover the secret key p from the public key $x_0, x_1, \ldots, x_\tau$. In other words, the goal is to **learn** the shortest nonzero vector p in the lattice given some points that are almost in the lattice but have some **errors**. This is why the problem of recovering p from the values $x_0, x_1, \ldots, x_\tau$ is called Learning With Errors. The difficulty of the problem is as difficult as finding the shortest nonvector vector in a lattice, so this scheme is lattice-based.

3.1.3 Encryption

To encrypt a bit $m \in \{0, 1\}$, we chose a random subset of the elements $x_1, x_2, \ldots, x_\tau$ (note that we may not choose the largest element x_0) from the public key. Let S be the set that contains the indices of elements we have chosen at random; S is a random subset of $\{1, \ldots, \tau\}$.

We also choose a random small integer r, and the ciphertext encrypting m is

$$c = r_{x_0}\left(m + 2\sum_{i \in S} x_i + 2r\right).$$

We take some time to unpack this: What we have done is essentially add some noise to our value of m. Some of the noise are some randomly chosen values $2x_i$ for x_i in the public key (but $x_i \neq x_0$). These are approximately values in the lattice of multiples of p, so this noise has some structure. The rest of the noise is $2r$ for r a small integer. This is even, and a small number, which we will see is important during decryption.

3.1.4 Decryption

Given a ciphertext c, the decryption algorithm computes

$$m \equiv r_p(c) \pmod 2.$$

In other words, the bit m has the same parity as $r_p(c)$. Without knowledge of the private key, it is hard to compute the correct remainder to check its parity. The journal version of the article lists more attacks against this scheme in Section 5 and Appendix B but we will not cover them in class.

Proposition 3.4. This decryption algorithm recovers m correctly.

Proof. Remember that if $m \in \{0, 1\}$ is the plaintext, then an encryption of m is of the form

$$c = r_{x_0}\left(m + 2\sum_{i \in S} x_i + 2r\right).$$

Since this is a remainder modulo x_0, there exists an integer k such that

$$c = m + 2r + 2\sum_{i \in S} x_i - kx_0.$$

To see why this is true, imagine that given the integer $m + 2r + 2\sum_{i\in S} x_i$, you compute its remainder modulo x_0 by subtracting x_0 until you land in the interval $\left(-\frac{x_0}{2}, \frac{x_0}{2}\right]$.

Now remember that each x_i is of the form $pq_i + r_i$, and so we have

$$\begin{aligned} c &= m + 2r + 2\sum_{i \in S} x_i - kx_0 \\ &= m + 2r + 2\sum_{i \in S} (pq_i + r_i) - k(pq_0 + r_0) \\ &= m + 2r + 2\sum_{i \in S} r_i - kr_0 + p(2\sum_{i \in S} q_i - kq_0) \end{aligned}$$

Now if

$$2r + 2\sum_{i \in S} r_i - kr_0$$

is small, then we will have that

$$r_p(c) = m + 2r + 2\sum_{i \in S} r_i - kr_0.$$

More precisely, since p is odd, we have that

$$r_p(c) = m + 2r + 2\sum_{i \in S} r_i - kr_0.$$

if

$$\left| 2r + 2\sum_{i \in S} r_i - kr_0 \right| < \frac{p}{2} - 1.$$

(We subtract 1 because we do add the message bit m to this in the ciphertext so we don't want that extra 1 to be what tips this over.) This is why we need to be careful when we define what it means for the values of r_i to be "much smaller" than p and for τ to be small. Note that if τ is small, and x_0 is the largest of the values of x_i, then k will be small too (less than τ), so kr_0 is small. You can read a discussion of the acceptable and suggested parameter sizes in Section 3 of the journal version of the article.

We assume thus that our values have been chosen so that

$$r_p(c) = m + 2r + 2\sum_{i \in S} r_i - kr_0.$$

We must now show that $r_p(c) \equiv m \pmod 2$, which is equivalent to showing that

$$2r + 2\sum_{i \in S} r_i - kr_0 \equiv 0 \pmod 2.$$

In turns this reduces to showing that

$$kr_0 \equiv 0 \pmod 2.$$

But recall that x_0 is chosen so that $r_p(x_0)$ is even. Since $x_0 = pq_0 + r_0$, and r_0 is very small in absolute value compared to p, we have that $r_p(x_0) = r_0$. We now see why we had to assume that $r_p(x_0)$ is even, and the proof is done. □

Remark 3.5. Now that we know why $r_p(x_0)$ had to be even, you might wonder why x_0 had to be odd. Recall that

$$c = m + 2r + 2\sum_{i\in S} x_i - kx_0$$

for some integer k.

If x_0 were even, the ciphertext would always have the same parity as m ($c \equiv m \pmod 2$). Since x_0 is part of the public key, it would be easy to notice this vulnerability and to decrypt every single message without knowing p.

If x_0 is odd, then c is either the same parity as m or not, depending if k is even or odd. Given the value of the remainder of $m + 2r + 2\sum_{i\in S} x_i$ modulo x_0, it is thought to be very hard to recover the set S (or the random values of x_i that were chosen to obscure m), so it is hard to compute k. Therefore it is random if c has the same parity as m or not and the parity of c does not give an attacker any information about m.

3.1.5 An example

In this section we give an example of an instance of this cipher. For reference, the parameters were chosen for to achieve 3 bits of security ($\lambda = 3$ in the notation of the article). Recall that for an instance that is secure in real life, we should use $\lambda = 128$ or $\lambda = 256$ to have 128 or 256 bits of security. The further parameters for this instance are

- $\rho = \lambda = 3$: this is the number of binary digits in the values of the noise r_i chosen at random to create the public key,
- $\rho' = 2\lambda = 6$: this is the number of binary digits in the value of the noise r chosen at random to encrypt a bit,
- $\eta = 12$: this is the number of binary digits in the value of the secret key p,
- $\gamma = 33$: this is the maximum number of binary digits in the product pq_i, where q_i is chosen at random to create the public key,
- $\tau = \gamma + \lambda = 36$: this is one less than the number of values x_i in the public key.

For the private key we must choose an odd integer p between $2^{\eta-1} = 2048$ and $2^\eta = 4096$. We choose $p = 3011$.

To create the public key, we choose random values of q_i and r_i to form $x_i = pq_i + r_i$. According to our parameters, we will have

$$0 \leq q_i \leq 2,852,851,$$

and

$$-7 \leq r_i \leq 7.$$

Here

$$\frac{2^\gamma}{21} = 2852851.076...$$

gives us the upper bound for the values of q_i, and r_i must be strictly between -2^ρ and 2^ρ.

Here is a valid public key for the private key $p = 3011$:

$$\begin{gathered} x_0 = 8454204503, 3327341689, 1349786140, 2796047723, 7830075393, 6761697318, \\ 4923797967, 2282744485, 2700505680, 2574555543, 4536432818, 5853763387, \\ 6757138668, 8182482829, 8130443719, 6817659754, 2897777368, 2859480454, \\ 8404833136, 2986869839, 2216788527, 3241154823, 7130084136, 898925021, \\ 4384274941, 7507585242, 1921632240, 783868684, 6288094121, 6833000810, 6364802355, \\ 2855280111, 3534432240, 2013660441, 6553649254, 7582742811, 7411341636. \end{gathered}$$

To encrypt a bit $m \in \{0,1\}$, we must choose a random number r strictly between $-2^{\rho'} = -64$ and $2^{\rho'} = 64$ (notice that the "noise" in the encryption is twice as large in binary digits as the noise in the public key elements), and a random set of elements x_i from the public key, but avoiding x_0. The ciphertext is

$$c = r_{x_0}\left(m + 2r + 2\sum_{i \in S} x_i\right).$$

Suppose that we wish to encrypt the bit $m = 0$. Then we would choose a small number r at random, say $r = -10$, and a random list of elements from the public key (but avoiding x_0), say the numbers

$$2216788527, 1349786140, 783868684, 6553649254, 8404833136.$$

We first compute

$$m + 2r + 2\sum_{i \in S} x_i = 38617851462$$

and then the ciphertext is the remainder modulo x_0 of 38617851462

$$c = r_{x_0}(38617851462) = -3653171053.$$

To decrypt this same ciphertext ($c = -3653171053$), we just compute $r_{3011}(c)$, and check if it's even or odd. If it's even, the message was $m = 0$, and if it's odd, the message was $m = 1$:

$$r_{3011}(-3653171053) = -28.$$

This is even so the message was $m = 0$.

3.2 Learning With Errors

Now that we have seen a simple cipher based on the Learning With Errors (LWE from now on) problem, we are ready to study LWE in more detail.

3.2.1 The LWE problems and definitions

To agree with the notation in the literature, from now on $q \in \mathbb{Z}$ will be a **prime number**.

In our simple cipher, we needed to generate **random small** numbers for the noise r_i in the public key. This was done by setting a bound of what we meant by "small" (the authors suggest using noise that is smaller than 2^λ in absolute value, where λ is the security parameter) and choosing the noise r_i uniformly randomly in the interval $(-2^\lambda, 2^\lambda)$. For more complicated ciphers, we will use a slightly more complicated random distribution:

Definition 3.6. Let $\sigma > 0$ and $B \in \mathbb{Z}$ be fixed. Then the **truncated discrete Gaussian distribution on the integers with variance σ^2 and constrain** $-B \leq x \leq B$ is given by the following probability distribution on the integers:

$$P(X = x) = \begin{cases} \dfrac{\exp\left(\dfrac{-x^2}{2\sigma^2}\right)}{\displaystyle\sum_{-B \leq y \leq B, y \in \mathbb{Z}} \exp\left(\dfrac{-y^2}{2\sigma^2}\right)} & \text{if } -B \leq x \leq B, x \in \mathbb{Z} \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

As the name suggests, you should think of this distribution as a normal distribution centered at 0 with variance σ^2, but the only values obtained are integers between $-B$ and B. The variance σ^2 essentially tells you what counts as "small."

Definition 3.7. Let q be a prime, $\sigma > 0$, and n a positive integer be fixed, and choose a vector $\vec{s} \in \mathbb{F}_q^n$. An **LWE$_{q,\sigma,\vec{s}}$ pair** is a tuple $(\vec{a}, b) \in \mathbb{F}_q^n \times \mathbb{F}_q$ such that

- $\vec{a} \in \mathbb{F}_q^n$ is drawn uniformly at random, and
- $b = \vec{a} \cdot \vec{s} + e$ for e drawn from a truncated discrete Gaussian distribution on the integers with variance σ^2 and constraint $-\lfloor \frac{q}{2} \rfloor \leq x \leq \lfloor \frac{q}{2} \rfloor$.

Here $\vec{a} \cdot \vec{s}$ denotes the usual dot product.

There are two main problems related to Learning With Errors:

Definition 3.8. Given m LWE$_{q,\sigma,\vec{s}}$ pairs

$$(\vec{a}_1, b_1), \ldots, (\vec{a}_m, b_m)$$

the **(search) LWE problem** asks to recover the vector $\vec{s}$.

And:

Definition 3.9. Given m pairs

$$(\vec{a}_1, b_1), \ldots, (\vec{a}_m, b_m)$$

with $(\vec{a}_i, b_i) \in \mathbb{F}_q^n \times \mathbb{F}_q$, the **decision LWE problem** asks to determine if the pairs are $\text{LWE}_{q,\sigma,\vec{s}}$ for some σ and $\vec{s}$, or if the values $\vec{a}_i$ and b_i were drawn uniformly at random from $\mathbb{F}_q^n$ and $\mathbb{F}_q$, respectively.

If one can solve the search LWE problem, then one can solve the decision LWE problem with high probability: Indeed, feed the pairs $(\vec{a}_i, b_i)$ into the algorithm that solves for $\vec{s}$, then check the distribution of the values

$$\vec{a}_i \cdot \vec{s} - b_i.$$

If these values are distributed according to a truncated Gaussian distribution, then the pairs were LWE. If these values are uniformly distributed, then the pairs were not LWE. The reason why this allows us to solve the decision problem with high probability is that it is possible to get very unlucky such that the pairs are actually LWE but the values $\vec{a}_i \cdot \vec{s} - b_i$ look uniformly distributed. Using more pairs $(\vec{a}_i, b_i)$, the probability of our being unlucky can get as small as we wish. We say that the decision LWE problem **reduces** to the search LWE problem (with high probability).

Interestingly, one can show that the search LWE problem also reduces to the decision LWE problem if q is bounded by a polynomial in n. (This constraint is so that the problem can be solved in polynomial time.) Indeed we can recover the coordinates of $\vec{s}$ one at a time in the following way: Given a set of $\text{LWE}_{q,\sigma,\vec{s}}$ pairs $(\vec{a}_1, b_1), \ldots, (\vec{a}_m, b_m)$, make a guess $g \in \mathbb{F}_q$ for the value of the jth coordinate of $\vec{s}$. Then, from the pairs you were given, form the new pairs

$$(\vec{a}_i + (0, 0, \ldots, 0, r_i, 0, \ldots 0), b_i + r_i g),$$

where r_i appears in the jth coordinate, and r_i is drawn uniformly at random from $\mathbb{F}_q$. If the guess g is correct, then these pairs are still $\text{LWE}_{q,\sigma,\vec{s}}$ pairs, and if the guess g is incorrect, the pairs are uniformly distributed. Thanks to our ability to solve the decision LWE problem, we can therefore guess all of the coordinates of $\vec{s}$ one by one in polynomial time.

The security of the cipher we will present is based on the hardness of the *decision* LWE problem. When q is polynomial in n, this is equivalent to the hardness of the search LWE problem, but that's not the case when q is exponential in n.

3.2.2 More lattice problems

In this section we highlight more lattice problems and their relationship to the LWE problems. But first, we define what a lattice is:

Definition 3.10. A **lattice** $L \subset \mathbb{R}^n$ is a discrete subgroup (under addition) of $\mathbb{R}^n$ of rank n. More practically, every lattice in $\mathbb{R}^n$ is the $\mathbb{Z}$-span of some basis $\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n$ of $\mathbb{R}^n$. In that case, we say that the set $\{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n\}$ **generates** the lattice L, or that it forms a **basis** for L.

Lattices have many problems:

Definition 3.11. Given a basis $\vec{v}_1, \vec{v}_2, \dots, \vec{v}_n$ for a lattice L, the **shortest vector problem (SVP)** asks to give the shortest nonzero vector $\vec{v}$ belonging to the lattice. In other words, let

$$\lambda(L) = \min_{0 \neq \vec{v} \in L} \|\vec{v}\| . \tag{2}$$

Then give a vector $\vec{v} \in L$ with $\|\vec{v}\| = \lambda(L)$.

A variation on this problem is this:

Definition 3.12. Fix a constant $\beta > 0$. Given a basis $\vec{v}_1, \vec{v}_2, \dots, \vec{v}_n$ for a lattice L, the **GapSVP$_\beta$ problem** asks to decide if the shortest vector of L has length less than or equal to 1, or strictly greater than β. In other words, with $\lambda(L)$ as in equation (2), decide if $\lambda(L) \leq 1$ or $\beta < \lambda(L)$.

Note that in the GapSVP problem, we promise that the case of $1 < \lambda(L) \leq \beta$ will not be asked. Or if it is asked, we allow ourselves to give the wrong answer or to refuse to answer.

In *Public-key cryptosystems from the worst-case shortest vector problem*, Peikert shows that if $q \geq 2^{n/2}$, the search LWE problem is at least as hard as the GapSVP problem in the worst case. Unfortunately since we do not have a reduction of search to decision in this case, this does not say anything about the hardness of the decision LWE problem.

For moduli q polynomial in n, the search LWE problem (and therefore the decision LWE problem) is at least as hard as a slightly different problem called ζ-to-γ-GapSVP. When using a Gaussian error distribution, Regev showed (in a personal communication to Peikert reported in *loc. cit.* that search and decision LWE are equivalent when q is a product of small primes (but q itself can be exponential in n, so GapSVP reduces to the seach LWE poblem), so this is another instance when a hard problem reduces to the decision LWE problem.

None of these reductions are exactly what we would like (reducing a hard problem to solving decision LWE). In 2013, in *Classical hardness of learning with errors* by Brakerski, Langlois, Peikert, Regev and Stehlé, it was shown that if q is polynomial in n, then the GapSVP problem in dimension $\sqrt{n}$ reduces to the decision LWE problem in dimension n.

For fun, here is one more problem:

Definition 3.13. Fix $\beta > 0$ and q a prime. Given an $n \times m$ matrix A with entries in $\mathbb{F}_q$, the **short integer solution (SIS$_\beta$) problem** asks to find a nonzero vector $\vec{z} \in \mathbb{Z}^m$ such that

1. $\|\vec{z}\| \leq \beta$, and
2. $A\vec{z} \equiv 0 \pmod{q}$.

Note that to guarantee that there exists such a $\vec{z}$, one can require $\sqrt{n \log q} \leq \beta$ (but at the same time, so the problem is not trivial, one should require $\beta < q$ so that $\vec{z} = (q, 0, 0, \dots, 0)$ is not a solution), and $m \geq n \log q$.

Proposition 3.14. The decision LWE problem reduces to the short integer solution problem with high probability.

Proof. Suppose that we have a list of m pairs $\{(\vec{a}_i, b_i)\}$ and want to determine if they are LWE. Suppose further that for any matrix and any value β, we can solve the short integer solution problem efficiently. Then we can solve the decision LWE problem as follows:

We begin by forming a large number of subsets $T_j \subset \{1, 2, \ldots, m\}$, $j = 1, 2, \ldots, N$, say. For each j, we form the matrix

$$A_j = \begin{pmatrix} \vec{a}_{k_1} \\ \vec{a}_{k_2} \\ \cdots \\ \vec{a}_{k_\ell} \end{pmatrix},$$

whose rows are the elements $\vec{a}_k$ for $k \in T_j$, and then use the SIS solver to give a short integer solution $\vec{z}_j$ to the equation $A_j^T \vec{z}_j \equiv 0 \pmod{q}$.

Then for each j we form the vector

$$\vec{B}_j = \begin{pmatrix} b_{k_1} \\ b_{k_2} \\ \cdots \\ b_{k_\ell} \end{pmatrix},$$

and compute $\vec{B}_j^T \vec{z}_j$. If the values $\vec{B}_j^T \vec{z}_j$ are distributed according to a truncated discrete Gaussian distribution, then we conclude that the pairs were LWE. If the values $\vec{B}_j^T \vec{z}_j$ are uniformly distributed in $\mathbb{Z}/q\mathbb{Z}$, then we conclude that the pairs were not LWE.

The reason why this works is the following: If the pairs were LWE all along, then we have that for each j

$$\vec{B}_j = A_j \vec{s} + \vec{E}_j,$$

where $\vec{E}_j$ is the vector of errors. In that case then, we have that

$$\begin{aligned} \vec{B}_j^T \vec{z}_j &= (A_j \vec{s} + \vec{E}_j)^T \vec{z}_j \\ &= (\vec{s}^T A_j^T + \vec{E}_j^T) \vec{z}_j \\ &= \vec{s}^T A_j^T \vec{z}_j + \vec{E}_j^T \vec{z}_j \\ &= \vec{E}_j^T \vec{z}_j, \end{aligned}$$

since $A_j^T \vec{z}_j \equiv 0 \pmod{q}$. Recall that we have that

$$\vec{E}_j \cdot \vec{z}_j \leq \left\| \vec{E}_j \right\| \|\vec{z}_j\|,$$

and $\left\| \vec{E}_j \right\|$ is smaller in absolute value than $\frac{q}{4}$ with high probability. Then if $\|\vec{z}_j\|$ is small enough (which we can control since we control the value of β), we will see a small value of $\vec{E}_j \cdot \vec{z}_j$ with high probability and therefore very often.

If the pairs are not LWE, then $\vec{B}_j^T \vec{z}_j$ is just a uniformly chosen random vector whose dot product is taken with a short vector. This will not be as short, and should sometimes be big, so with high probability we will see larger values. □

3.2.3 Regev's cipher based on LWE

We now present the first cipher based on the hardness of LWE, proposed by Regev in 2005 in *On Lattices, Learning with Errors, Random Linear Codes, and Cryptography.*

Algorithm 3.15 (Regev LWE key generation)**.** Given n a positive integer, choose m a positive integer, q a prime, and a probability distribution χ on $\mathbb{F}_q$:

1. Choose a vector $\vec{s} \in \mathbb{F}_q^n$ uniformly at random. This is the secret key.
2. For $i = 1, \ldots, m$, choose m vectors $\vec{a}_i \in \mathbb{F}_q^n$ uniformly at random and independently of each other.
3. For $i = 1, \ldots, m$, choose m values $e_i \in \mathbb{F}_q$ from the distribution χ and independently from each other.
4. The public key is the set of pairs $(\vec{a}_i, b_i)$ for $i = 1, \ldots, m$, where for each i

$$b_i = \vec{a}_i \cdot \vec{s} + e_i.$$

When choosing vectors of length n, Regev recommends the following choices:

- $n^2 \leq q \leq 2n^2$,
- $m = (1 + \varepsilon)(n + 1) \log_2 q$ for any $\varepsilon > 0$, and
- χ is a truncated discrete Gaussian with standard deviation $\sigma = \dfrac{q}{\sqrt{2\pi n} \log^2 n}$. (Please note that this corrects the mistake I made in class where I didn't originally multiply by q.)

Algorithm 3.16 (Regev LWE encryption)**.** Given a Regev LWE public key $\{(\vec{a}_i, b_i)\}$, one can encrypt one bit, 0 or 1. In either case, choose a random subset $T \subseteq \{1, 2, \ldots, m\}$, then:

- To send the bit 0, send the pair $(\vec{a}, b)$, where

$$\vec{a} = \sum_{i \in T} \vec{a}_i, \quad \text{and} \quad b = \sum_{i \in T} b_i;$$

- To send the bit 1, send the pair $(\vec{a}, b)$, where

$$\vec{a} = \sum_{i \in T} \vec{a}_i,$$

as before, but

$$b = \left\lfloor \frac{q}{2} \right\rfloor + \sum_{i \in T} b_i.$$

The hardness of the decision LWE problem comes in here: When the bit sent is 0 the pair sent is an LWE pair, and when it is 1 the pair is not LWE. More precisely, Regev shows that if someone can distinguish between encryptions of 0 and 1, then they can solve the decision LWE problem for a non-negligible fraction of all $\vec{s}$.

Finally we present the decryption algorithm:

Algorithm 3.17 (Regev LWE decryption)**.** Given the secret key $\vec{s}$, upon reception of a pair $(\vec{a}, b)$, compute

$$b - \vec{a} \cdot \vec{s}.$$

If this value is closer to 0 than to $\lfloor \frac{q}{2} \rfloor$, the bit sent was 0 with high probability, and if this value is closer to $\lfloor \frac{q}{2} \rfloor$ than to 0, the bit sent was 1 with high probability.

More precisely, we have the following result:

Lemma 3.18 (Regev LWE correctness)**.** Let $\delta > 0$, and $k \in \{0, 1, \ldots, m\}$. Let $\{e_i\}$ be a set of k independently chosen random values from the distribution χ. If the probability that

$$\sum_{i=1}^{k} e_i$$

is closer to 0 than to $\lfloor \frac{q}{2} \rfloor$ is greater than $1 - \delta$ for every $k \in \{0, 1, \ldots, m\}$, then the probability of decryption error is at most δ.

With the parameters chosen by Regev, it can be shown that the probability of an error in decryption is negligible as a function of n.

3.3 Ring-LWE

Unfortunately LWE encryption is very inefficient, requiring a public key of size about n^2 and a ciphertext of size n to send a single bit. We present here a modification of LWE that is more efficient, called Ring Learning With Errors, which was first introduced in 2013 by Lyubashevsky, Peikert and Regev in *On ideal lattices and learning with errors over rings.*

3.3.1 Algebraic preliminaries

A **number field** K is a field containing $\mathbb{Q}$ that is finite dimensional as a vector space over $\mathbb{Q}$. The dimension of K over $\mathbb{Q}$ is called the **degree** of K. If K has degree n over $\mathbb{Q}$, then every element $\alpha \in K$ satisfies a monic irreducible polynomial $f \in \mathbb{Q}[x]$, called the **minimal polynomial** of α. The **ring of integers** $\mathcal{O}_K$ of K is the set of $\alpha \in K$ such that the minimal polynomial of α has coefficients in $\mathbb{Z}$. (This set is indeed a ring.)

If K has degree n over $\mathbb{Q}$, then there are n different injective field homomorphisms $K \to \mathbb{C}$. Those whose image is contained in $\mathbb{R}$ are called **real embeddings** of K and those whose image is not contained in $\mathbb{R}$ are called **complex embeddings** of K. All of the complex embeddings of K come in **conjugate pairs**: If $\tau \colon K \to \mathbb{C}$ is a complex embedding of K, then so is $\bar{\tau} = \bar{\cdot} \circ \tau$, where $\bar{\cdot}$ denotes complex conjugation on $\mathbb{C}$.

Suppose that K has s_1 real embeddings $\sigma_1, \ldots, \sigma_{s_1}$ and s_2 pairs of complex embeddings, with a representative of each pair given by $\tau_1, \ldots, \tau_{s_2}$ (so that $n = s_1 + 2s_2$). Then the **canonical embedding**[4] of K is the map sending $\alpha \in K$ to

$$(\sigma_1(\alpha), \ldots, \sigma_{s_1}(\alpha), \sqrt{2}\,\mathrm{Re}(\tau_1(\alpha)), \sqrt{2}\,\mathrm{Im}(\tau_1(\alpha)), \ldots, \sqrt{2}\,\mathrm{Re}(\tau_{s_2}(\alpha)), \sqrt{2}\,\mathrm{Im}(\tau_{s_2}(\alpha))) \in \mathbb{R}^n.$$

The image of $\mathcal{O}_K$ under the canonical embedding, $\Lambda = \sigma(\mathcal{O}_K)$ is a lattice in $\mathbb{R}^n$.

3.3.2 Polynomial-LWE

We begin by explaining a particularly straightforward adaptation of LWE to the number field setting, which is mentioned by Lyubashevsky, Peikert, and Regev in *On ideal lattices and learning with errors over rings*. This cipher is now commonly called "polynomial learning with errors" or PLWE.

If K is a number field of degree n, there is a **primitive element** $\gamma \in K$ such that

$$K = \mathbb{Q}(\gamma) = \{a_0 + a_1\gamma + \cdots + a_{n-1}\gamma^{n-1} : a_i \in \mathbb{Q}\}.$$

By "clearing the denominator" of γ, we may even assume that $\gamma \in \mathcal{O}_K$. However, it is **not the case in general** that if $K = \mathbb{Q}(\gamma)$ for $\gamma \in \mathcal{O}_K$, then

$$\mathcal{O}_K = \mathbb{Z}[\gamma] = \{a_0 + a_1\gamma + \cdots + a_{n-1}\gamma^{n-1} : a_i \in \mathbb{Z}\}.$$

When there is such a choice of γ, we say that $\mathcal{O}_K$ is **monogenic**. In this section we will suppose that this is the case throughout.

In this case, there is a simple way to draw a "small" element of $\mathcal{O}_K$ at random: For some suitable $\sigma > 0$, choose n elements $e_j \in \mathbb{Z}$, $j = 0, \ldots, n-1$, independently at random from a discrete Gaussian distribution with variance σ^2, and form the element

$$e = e_0 + e_1\gamma + \cdots + e_{n-1}\gamma^{n-1}.$$

Now fix a prime $q \in \mathbb{Z}$. We have then that the elements of the quotient ring $R_q = \mathcal{O}_K/q\mathcal{O}_K$ are of the form

$$\bar{e}_0 + \bar{e}_1\bar{\gamma} + \cdots + \bar{e}_{n-1}\bar{\gamma}^{n-1}$$

for $\bar{e}_j \in \mathbb{Z}/q\mathbb{Z}$ and $\bar{\gamma}$ a representative of the coset $\gamma + q\mathcal{O}_K$. We can draw a "small" element of R_q at random by first drawing a small element of $\mathcal{O}_K$ according to the procedure in the preceding paragraph, and then reducing it modulo q. From now on this is what we will mean by drawing a small element of R_q at random.

We are now ready to describe the PLWE cipher:

Algorithm 3.19 (PLWE key generation)**.** Given a number field K of degree n with monogenic ring of integers generated by γ, a prime q, and a standard deviation $\sigma > 0$ for the error distribution:

[4] Though it is said to be canonical, which might connote the fact that it is unique, several slightly different maps are called the "canonical embedding" in the literature. Usually it does not really matter which specific canonical embedding we choose.

1. Draw a small element $s \in R_q$ at random. This is the secret key.
2. Draw an element $a \in R_q$ uniformly at random, a small element $e \in R_q$ and form the pair
$$(a, b = as + e) \in R_q^2.$$
This is the public key.

Algorithm 3.20 (PLWE encryption)**.** To send an n-bit message whose jth bit is $m_j \in \{0, 1\}$, given the public key $(a, b) \in R_q^2$, we view the message as

$$m = m_0 + m_1\bar{\gamma} + \cdots + m_{n-1}\bar{\gamma}^{n-1} \in R_q,$$

then choose three random small elements of R_q denoted r, e_1, e_2 and send the pair $(u, v) \in R_q^2$, where
$$u = ar + e_1 \quad \text{and} \quad v = br + e_2 + \left\lfloor \frac{q}{2} \right\rfloor m.$$

Algorithm 3.21 (PLWE decryption)**.** Upon reception of a ciphertext $(u, v) \in R_q^2$, we compute
$$v - us = (re - se_1 + e_2) + \left\lfloor \frac{q}{2} \right\rfloor m.$$
When σ is chosen appropriately, the coefficients of the element $re - se_1 + e_2$ as a polynomial in $\bar{\gamma}$ are smaller than $\frac{q}{4}$, so the message m can be recovered by rounding the coefficients of $v - us$ as a polynomial in $\bar{\gamma}$ to either 0 or $\lfloor \frac{q}{2} \rfloor$, whichever is closest. If the jth coefficient rounds to 0, the jth sent bit was 0, and if it rounds to $\lfloor \frac{q}{2} \rfloor$, the jth sent bit was 1.

The security of this cipher depends on the decision RLWE problem (which is similar to the decision LWE problem, with $\vec{a}$ replaced by a and $\vec{s}$ replaced by a small value s), since the public key (a, b) and the ciphertext (u, v) appear to have been drawn uniformly at random, and therefore do not leak information about the message. Note that here we are using a theorem showing that LWE and RLWE with small secret s is just as safe as their counterparts with secret chosen uniformly at random, and a theorem giving the equivalence of search and decision RLWE, which is valid in particular if K is Galois over $\mathbb{Q}$.

3.3.3 A RLWE cipher

In this section we present a modification of the cipher presented by Lyubashevsky, Peikert, and Regev in *A toolkit for Ring-LWE cryptography.* It is perhaps no surprise at this point that we must first define some error distributions we will need:

Definition 3.22. Let $\vec{\nu} \in \mathbb{R}^n$ be a vector, and $\sigma > 0$. Then the **continuous Gaussian distribution** on $\mathbb{R}^n$ (centered at 0) with standard deviation σ is given by the probability distribution function
$$D_\sigma(\vec{x}) = \frac{1}{(2\pi\sigma^2)^{n/2}} \exp\left(\frac{-\|\vec{x}\|}{2\sigma^2} \right),$$
where $\|x\|$ is the Euclidean norm on $\mathbb{R}^n$.

Definition 3.23. Let $\Lambda \subset \mathbb{R}^n$ be a lattice, $\vec{\nu} \in \mathbb{R}^n$ be a vector, and $\sigma > 0$. Then the **discretization of the Gaussian distribution with standard deviation** σ to the lattice coset $\vec{\nu} + \Lambda$ is the distribution whose elements are drawn in the following way:

1. Draw a vector $\vec{x}$ from a Gaussian distribution with variance σ^2 on $\mathbb{R}^n$.
2. Round $\vec{x}$ to an element $\vec{y}$ of the set $\vec{\nu} + \Lambda$ that is "not too far" from $\vec{x}$. We refer to the original paper, Section 2.4.2 for three valid ways to define "not too far."

We can now describe the cipher:

Algorithm 3.24 (Key generation for an RLWE cipher)**.** Let K be a number field of degree n with ring of integers R, and $p \in \mathbb{Z}$ be a prime. Fix an integer $\ell \geq 2$, real numbers $r \geq 2nq^{(n+2)/n\ell}$ and $\sigma > \frac{1}{\sqrt{2\pi}}$, and finally $q \in \mathbb{Z}$ a prime distinct from p with $q \geq \sigma\sqrt{2\pi n \log n(r^2\ell + 1)}$. All of this information is available publicly. Then

- Let $a_0 = -1 \in R/qR$, and choose $\ell - 1$ values $a_i \in R/qR$, $i = 1, \ldots, \ell - 1$, independently and uniformly at random.
- Fix $x_\ell = 1$, and draw ℓ values x_i, $i = 0, \ldots, \ell - 1$, from a discrete Gaussian distribution on the integers centered at 0 with standard deviation r.
- Compute the value
$$a_\ell = -\sum_{i=0}^{\ell-1} a_i x_i,$$
where here the x_is have been reduced modulo q.

Then the secret key is the vector $\vec{x} = (x_1, \ldots, x_\ell) \in \mathbb{Z}^\ell$, and the public key is the vector $\vec{a} = (a_1, \ldots, a_\ell) \in (R/qR)^\ell$.

Algorithm 3.25 (Encryption for an RLWE cipher)**.** Given an instance of this RLWE cipher, to encrypt the value $\mu \in R/pR$,

- Draw ℓ values $e_0, \ldots, e_{\ell-1}$ from the discretization of the Gaussian distribution on $\mathbb{R}^n$ with standard deviation $p\sigma$ to the lattice $p\Lambda$, where $\Lambda = \sigma(R)$, and form the vector $\vec{e} = (e_1, \ldots, e_{\ell-1})$.
- Draw one value e_ℓ from the discretization of the Gaussian distribution on $\mathbb{R}^n$ with standard deviation $p\sigma$ to the lattice coset $\mu + p\Lambda$.

The ciphertext is
$$\vec{c} = e_0\vec{a} + \vec{e} \in (R/qR)^\ell.$$

Algorithm 3.26 (Decryption for an RLWE cipher)**.** Given a ciphertext $\vec{c}$, compute the dot product $\bar{d} = \vec{c} \cdot \vec{x} \in R/qR$. Fix $\{b_j\}$ a "good" basis for R, write $b_j \equiv \bar{b}_j \pmod{qR}$, and express

$$\bar{d} = \sum_{j=1}^{n} \bar{d}_j \bar{b}_j,$$

with $\bar{d}_j \in \mathbb{Z}/q\mathbb{Z}$. For each $\bar{d}_j$, let $d_j \in \mathbb{Z}$ be the unique integer congruent to $\bar{d}_j$ modulo q and such that $-\frac{q}{2} \leq d_j < \frac{q}{2}$. Then

$$\mu \equiv d = \sum_{j=1}^{n} d_j b_j \pmod{pR}$$

with high probability (given the parameters we have chosen).

The security of this cipher is based on the hardness of the decision RLWE problem for K. Indeed the authors show that the public key $\vec{a}$ is indistinguishable from a random vector, so no information can be gained from it. Furthermore, if the ith entry of the ciphertext $\vec{c}$ is denoted c_i and a_i is the ith entry of the public key $\vec{a}$, the pairs (a_i, c_i) are then RLWE pairs with two different error distributions, and the authors show that we cannot distinguish the first $\ell - 1$ pairs from the last (whose error distribution contains the plaintext information).

4 Fully homomorphic encryption

A **fully homomorphic** encryption scheme is one that allows the performance of computations on encrypted data, in such a way that the outcome of the computation on the encrypted data is the encryption of the string that would have been the outcome of the computation if had been performed on the unencrypted data.

More formally, a fully homomorphic encryption scheme is a usual cipher (with key generation, encryption and decryption algorithms) along with an "evaluation" operation, which allows to evaluate a circuit on the encrypted data without access to the secret key. After one evaluates a circuit on the encrypted data, the output of the evaluate function is a valid ciphertext which can be decrypted to obtain the outcome of the circuit if it had been evaluated on the unencrypted data.

The idea of fully homomorphic encryption is as old as public-key cryptography. Indeed Rivest, Adleman, and Dertouzos introduced the notion in 1978, the same year RSA, the first publicly published public key cryptosystem, was presented. At the time, it was already understood that RSA was **homomorphic**: RSA respects multiplication (see Example 4.1). However it was not until 2009 that Gentry suggested the first *fully* homomorphic encryption scheme. We will explain the connection between homomorphic and fully homomorphic encryption in Section 4.1

Example 4.1. Recall the RSA encryption algorithm: Given a public key (N, e), a plaintext m is encrypted to the ciphertext

$$c \equiv m^e \pmod{N}.$$

Consider now two plaintexts m_1, m_2 and their product $m_1 m_2 \equiv m \pmod N$. Then the encryption of m

$$c_m \equiv m^e \pmod N$$

is equal to the product of the encryptions of m_1 and m_2:

$$\begin{aligned} c_m &\equiv m^e \pmod N \\ &\equiv (m_1 m_2)^e \pmod N \\ &\equiv m_1^e m_2^e \pmod N \\ &\equiv c_{m_1} c_{m_2} \pmod N. \end{aligned}$$

4.1 Homomorphisms and boolean circuits

As mentioned above, an encryption algorithm is **homomorphic** if it supports the evaluation of arbitrary circuits consisting only of one gate. That gate is usually addition or multiplication, but for example the Goldwasser-Micali cryptosystem allowed evaluation of an arbitrary number of XOR gates (exclusive or gates).

In 1978 when the idea of fully homomorphic encryption was first introduced, "fully homomorphic" meant that the encryption algorithm supports the evaluation of arbitrary circuits consisting of two gates, addition and multiplication. In mathematics, a map that respects addition and multiplication is called a (ring) homomorphism; this is where the phrase "fully homomorphic" comes from. As a note, a homomorphic encryption algorithm corresponds in mathematics to a group homomorphism; for mathematicians all homomorphisms are homomorphisms, though they may be homomorphism for different objects such as groups (which have one operation) or rings (which have two operations). In computer science "homomorphic" corresponds to respecting one operation and "fully homomorphic" to respecting two operations.

It was quickly understood, however, that if one thinks of bits as elements of $\mathbb{F}_2$, that is, if one defines bit addition as

$$0 + 0 = 0, \quad 0 + 1 = 1 + 0 = 1, \quad 1 + 1 = 0$$

and bit multiplication as

$$0 \cdot 0 = 0, \quad 0 \cdot 1 = 1 \cdot 0 = 0, \quad 1 \cdot 1 = 1,$$

then a fully homomorphic encryption scheme would allow the evaluation of arbitrary circuits on the encrypted bits.

Indeed, one can show that using these two binary gates (addition and multiplication), one can create any Boolean circuit we wish. This is because together, the AND and NOT gates are **functionally complete**: using AND and NOT allows the construction of any circuit. It thus suffices to show that addition and multiplication allow us to recreate the AND and NOT gates.

Recall that the AND gate has the following truth table:

Input	Output
00	0
01	0
10	0
11	1

This is exactly the same as the multiplication gate! Therefore if we have an encryption algorithm that respects multiplication, it will respect the AND gate.

The NOT gate just flips the bit; it changes 0 to 1 and 1 to 0. But notice that this is exactly what adding the bit 1 does:

$$0 + 1 = 1 \quad \text{and} \quad 1 + 1 = 0.$$

Therefore if the encryption algorithm respects addition, by adding the encryption of the bit 1 to any bit, we will have applied the NOT gate.

I'm not sure exactly when the definition of "fully homomorphic" changed from meaning "respects two operations" to "allows the evaluation of arbitrary circuits." But now that is what it means. Some fully homomorphic encryption schemes will show that their encryption algorithm respects the NAND gate, instead of respecting addition and multiplication. This is because by itself the NAND gate is functionally complete, and therefore if the encryption respects the NAND gate it will respect any circuit. (So in particular, if an encryption algorithm respects the NAND gate, it will automatically respects addition and multiplication, since they are Boolean circuits.) The NAND gate has the following truth table:

Input	Output
00	1
01	1
10	1
11	0

It is called NAND because it is the same as first doing AND and then doing NOT to the output.

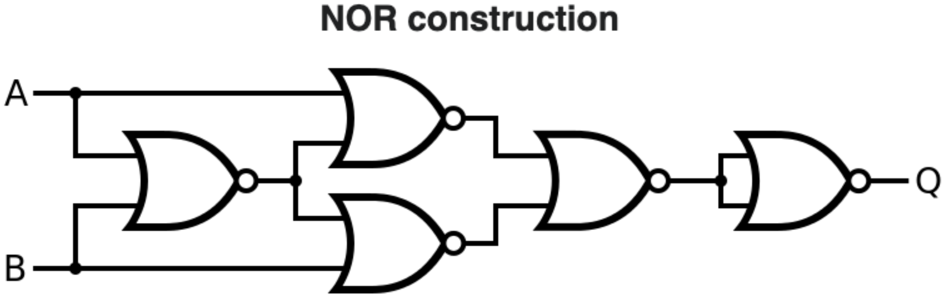
4.2 A brief history of homomorphic encryption

As we mentioned above, the idea of fully homomorphic encryption was put forth as early as 1978, and at that time it was already known that the RSA encryption scheme was homomorphic. However, at that time there was no construction of a fully homomorphic scheme, and it was not clear whether or not it would be possible to design such a scheme.

Over the years many homomorphic schemes were designed, but the next big step was taken twenty years later, when the Sander-Young-Yung system was published in 1999. This scheme did allow the evaluation of both addition and multiplication, but the number of multiplications allowed was limited.

To discuss this notion further, we introduce the concept of the **depth** of a circuit. The depth of a given circuit is the maximum number of gates that a given bit needs to travel through, or the longest path between the data input and the data output.

For example, consider this illustration of how to implement the XOR (exclusive or) gate using only NOR gates (the NOR gate is a universal gate, just like the NAND gate; it can be used by itself to create any circuit) which I found on Wikipedia:



The depth of this circuit is 4, because the maximum number of NOR gates that a bit has to go through is 4 (notice that on the outside some copies of the bits only go through 3 NOR gates, but the depth is the maximum number of gates).

The reason the depth of a circuit is important when studying fully homomorphic encryption schemes is because we know that to be secure, a homomorphic scheme cannot be deterministic: The encryption of a message must be random in some way, otherwise an attacker could use the operation(s) on the encrypted data to recover the original plaintext. So far, every fully homomorphic encryption scheme works by adding some “noise” to the plaintext as part of the encryption. As long as the amount of noise is not too large, the decryption algorithm can recover the plaintext.

However, it is a characteristic of every known fully homomorphic scheme so far that while addition does not increase the noise too much, multiplication does. If a circuit performs too many multiplications on the input data, when we apply the circuit to the encrypted data the multiplications will add too much noise and at the end the decryption algorithm will fail to recover the correct answer in plaintext.

The first encryption scheme that allowed two operations, the one proposed by Sander, Young, and Yung in 1999, had an amount of noise that grew exponentially with the number of multiplications. For this reason, to keep the amount of noise low enough for decryption to be correct, one could only apply circuits of logarithmic depth to the encrypted data. An encryption scheme that supports the evaluation of both addition and multiplication, as long as the circuit depth is logarithmic in the number of input bits is called **somewhat homomorphic**.

The next step in the development of homomorphic encryption was taken in 2005 by Boneh, Goh and Nissim. Their encryption scheme allowed an arbitrary number of additions but only one multiplication. However, this was an improvement on the Sander-Young-Yung since the noise did not increase exponentially with the multiplication, which greatly improved the efficiency of the scheme.

Finally in 2009, Gentry showed how to **bootstrap** a somewhat homomorphic encryption scheme so that it becomes fully homomorphic. This was the topic of Monday’s lecture, and we refrain from repeating the information here. A good read on this topic, from which my presentation was inspired is the article *Computing arbitrary functions of encrypted data* by Craig Gentry, which I have posted on our website. In these notes we will simply say that the bootstrapping operation allows one to "refresh" the encrypted data to reduce the noise. In other words, after performing some multiplications, which have made the noise increase, one refreshes the ciphertext using the bootstrapping operation so that the noise is reduced to allow for more multiplications.

We note that the bootstrapping operation is incredibly time-consuming. This has been improved in the past 10 years, but overall bootstrapping is still considered expensive. Because of this, researchers have developed **leveled fully homomorphic** encryption schemes, which allow the evaluation of circuits of bounded depth without causing decryption errors. The difference between somewhat homomorphic encryption and leveled fully homomorphic encryption is that the depth of the circuits that can be evaluated correctly is independent of the size of the input for a level fully homomorphic scheme. In applications where the number of multiplications to be performed is known in advance, it is usually more efficient to use a leveled fully homomorphic encryption algorithm than a fully homomorphic encryption algorithm.