Fully Homomorphic Encryption!!

Noah Stephens-Davidowitz

June 9, 2023

1 Fully Homomorphic Encryption

1.1 Homomorphisms of encryption schemes

Remember that all of the encryption schemes that we've seen so far have some sort of homomorphism or malleability. For example, in the Goldwasser-Micali scheme, a ciphertext has the form $y^m x^2 \bmod N$, where $y \in \overline{\mathsf{QR}}_N$ is a quadratic non-residue. So, if we know two ciphertexts $c_1 := y^{m_1} x_1^2 \bmod N$ and $c_2 := y^{m_2} x_2^2 \bmod N$ (and the public key), we can compute $c_3 := c_1 \cdot c_2 \bmod N = y^{m_1 \oplus m_2} x_3^2 \bmod N$, which is an encryption of $m_1 \oplus m_2$.[1] More generally, suppose that we know many ciphertexts $c_1 := y^{m_1} x_1^2 \bmod N, \ldots, c_\ell := y^{m_\ell} x_\ell^2 \bmod N$ which are encryptions of the plaintexts $m_1, \ldots, m_\ell \in \{0,1\}$. Then, we can compute $c^* := c_1 c_2 \cdots c_\ell \bmod N = y^{m_1 \oplus m_2 \oplus \cdots \oplus m_\ell} (x^*)^2 \bmod N$, which is an encryption of the plaintext $m_1 \oplus \cdots \oplus m_\ell$.

More generally still, a *homomorphism* of an encryption scheme is an efficient algorithm that takes as input (possibly the public key and) ciphertexts $c_1, \ldots, c_\ell$, which are encryptions of the plaintext $m_1, \ldots, m_\ell \in \{0,1\}$, and outputs a ciphertext c_f corresponding to the plaintext $f(m_1, \ldots, m_\ell)$ for some function $f : \{0,1\}^\ell \to \{0,1\}$. Of course, some functions f are not very interesting. E.g., the function $f(m_1, m_2, \ldots, m_\ell) = m_1$ is not very interesting in this context, nor is $f(m_1, \ldots, m_\ell) = 1$. But, the function $f(m_1, \ldots, m_\ell) = m_1 \oplus \cdots \oplus m_\ell$ from above *is* interesting.

When we do this, we refer to it as *homomorphically evaluating* f, and when there is a non-trivial f for which this is possible, we say that the encryption scheme is *homomorphic*. In the special case when $\ell = 1$, we instead say that the scheme is *malleable*. Homomorphisms are potentially a security threat (though a homomorphic encryption scheme can still be semantically secure, it might not be good if an adversary can produce an encryption of $m_1 \oplus m_2$ given encryptions of m_1 and m_2…), but for some applications, they are also extremely useful.

1.2 "Any efficiently computable function"—representing functions with circuits

Below, we will build a *fully homomorphic* encryption scheme, that is, an encryption scheme that allows us to efficiently homomorphically compute *any* efficiently computable function $f : \{0,1\}^\ell \to \{0,1\}$. (Notice that we need to restrict ourselves to efficiently computable functions! In particular, the complexity of the procedure of encrypting, applying the homomorphism, and then decrypting, must be at least the complexity of f.)

[1] Here, $x_3 = y x_1 x_2 \bmod N$ if $m_1 = m_2 = 1$. Otherwise, $x_3 = x_1 x_2 \bmod N$. In particular, while it may look like we get an encryption of $m_1 + m_2$, rather than $m_1 \oplus m_2$ in the above scheme, notice that the plaintext m is only defined modulo 2 in the Goldwasser-Micali scheme. I.e., if we "encrypt $m + 2$ to get $y^{m+2} x^2 \bmod N$," this is equivalent to encrypting m using $y^m (xy)^2 \bmod N$.

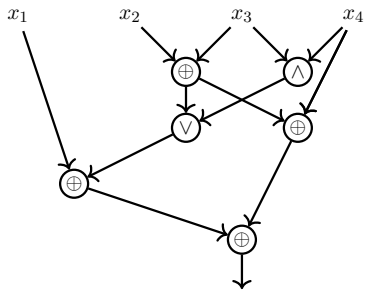

Figure 1: A circuit with four-bit input and depth 3. If f is the function computed by this circuit, then you can check that, e.g., $f(1,0,1,0) = 1$ and $f(1,1,1,1) = 0$. (Thanks to Sasha Golovnev for the figure.)

To formalize this, we need to take a brief detour to decide how we will even represent an arbitrary efficiently computable function f. For our purposes, it will be convenient to assume that f is described as a *circuit*. Formally, a circuit is a directed acyclic graph, where every node has in-degree 2, except for n special *input nodes*, which have in-degree zero. All non-special nodes are labeled with a *gate* $G : \{0,1\}^2 \to \{0,1\}$, e.g. $\oplus$ or AND or OR or NAND. One of these nodes is designated as the output. The value of the circuit on input $x_1, \ldots, x_n$ is computed as follows. We assign the values $x_1, \ldots, x_n \in \{0,1\}$ to the input vertices, and then do the following repeatedly. Label each vertex whose parents do have assignments, $b_1, b_2 \in \{0,1\}$ with $G(b_1, b_2)$, where G is the gate corresponding to the vertex. After doing this repeatedly, we will eventually assign a value to the output node. This is the output of the circuit.

The *depth* of a circuit is the length of the longest path from an input vertex to the output vertex. See Figures 1 and 2.

But, the details of this model of computation will be largely unnecessary from our perspective. Intuitively, a circuit is just a "bit-level description" of a function. It can be interpreted as describing a computation in the form "first take x_i and x_j and XOR them together; then take the result y and AND it with x_k; then take that same result y and XOR it with x_1; then take the results of the previous two operations and take the OR of them…" The Cook-Levin theorem tells us that any efficient algorithm can be represented as a polynomial-sized circuit, and that this representation can be computed efficiently.

Furthermore, the operations $\oplus$ and AND are a complete set of gates. That is, any circuit can be efficiently converted into an equivalent circuit where all gates are either AND or $\oplus$. (The way that I described it, this isn't *exactly* true because using only AND and $\oplus$, we cannot, e.g., represent any function f such that $f(0,\ldots,0) = 1$. But, we always assume that we have access to *constants*—that is, we can include an additional input vertex that is always set to zero, and an additional input vertex that is always set to one. It is in this model that AND and $\oplus$ are universal.) So, we will assume that our functions are represented by circuits consisting entirely of AND and $\oplus$ gates. (It is sometimes convenient to think of AND gates as multiplication gates, and we do so quite a bit later.)

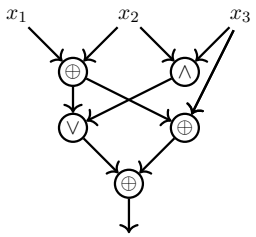

Figure 2: A circuit that takes 3 bits as input and has depth 3. If f is the function computed by this circuit, then you can check that, e.g., $f(1,0,1) = 0$ and $f(1,1,1) = 0$. (Thanks to Sasha Golovnev for the figure.)

1.3 Definition of fully homomorphic encryption

A fully homomorphic encryption scheme is an encryption scheme that is augmented with an additional efficient algorithm Eval, which takes as input a description of a function f (as a circuit, as described above) and ciphertexts $c_1 = \mathsf{Enc}(m_1), \ldots, c_\ell = \mathsf{Enc}(m_\ell)$ (and possibly the public key as well), and outputs a single ciphertext c^* that encodes $f(m_1, \ldots, m_\ell)$, i.e., $\mathsf{Dec}(c^*) = f(m_1, \ldots, m_\ell)$.

Formally, we will have two separate definitions, one for secret-key encryption and one for public-key encryption. We take the message space to be $\{0,1\}$ for convenience.

Definition 1.1. *A* fully homomorphic secret-key encryption scheme *consists of four efficient algorithms* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ *with the following properties.*

- $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is a correct and semantically secure secret-key encryption scheme with message space* $\{0,1\}$*.*
- **(Homomorphism.)** *For any function* $f : \{0,1\}^\ell \to \{0,1\}$ *and any plaintexts* $m_1, \ldots, m_\ell \in \{0,1\}$*,*

$$\Pr_{k \leftarrow \mathsf{Gen}(1^n)}[c_1 \leftarrow \mathsf{Enc}(k, m_1), \ldots, c_\ell \leftarrow \mathsf{Enc}(k, m_\ell),\ \mathsf{Dec}(k, \mathsf{Eval}(f, c_1, \ldots, c_\ell)) = f(m_1, \ldots, m_\ell)] = 1 \ .$$

- **(Compactness.)** *For any function* $f : \{0,1\}^\ell \to \{0,1\}$ *and any* $m_1, \ldots, m_\ell \in \{0,1\}$*,*

$$\Pr_{k \leftarrow \mathsf{Gen}(1^n)}[c_1 \leftarrow \mathsf{Enc}(k, m_1), \ldots, c_\ell \leftarrow \mathsf{Enc}(k, m_\ell), c^* \leftarrow \mathsf{Eval}(f, c_1, \ldots, c_\ell),\ |c^*| = |c_1| = \cdots = |c_\ell|] = 1 \ ,$$

where $|c^*|$ *is the bit length of* c^**. In other words, the length of the output of* Eval *is the same as the length of the output of the encryption algorithm.*

The last condition, compactness, might seem a bit strange. But, something like this is necessary to make the definition non-trivial. For example, let $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be *any* semantically secure secret-key encryption scheme, and consider the algorithm $\mathsf{Eval}(f, c_1, \ldots, c_\ell) = (f, c_1, \ldots, c_\ell)$. If

we modify the decryption algorithm to Dec' such that $\mathsf{Dec}'(f, c_1, \ldots, c_\ell) = f(\mathsf{Dec}(c_1), \ldots, \mathsf{Dec}(c_\ell))$ (and $\mathsf{Dec}'(c) = \mathsf{Dec}(c)$), then the scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}', \mathsf{Eval})$ satisfies all of the other requirements for a fully homomorphic cryptosystem. But, it is not very interesting, precisely because it is *not* compact.

By requiring compactness, we are intuitively requiring the Eval algorithm to "do something interesting," since it cannot, e.g., simply write down the function f if the description of f is too large. (Compactness also implies that there is a fixed polynomial $p(n)$ such that Dec runs in time at most $p(n)$ when called on the output of Eval, independent of f. E.g., if f can only be computed in time $n^{100} \cdot p(n)$, then "Eval must do most of the computation itself.")

One could imagine various stronger definitions. E.g., one could demand that c^* "has the same form as c_i" (e.g., that c^* is computationally indistinguishable from $\mathsf{Enc}(k, 0)$). The notion of function hiding is one way to formalize this, and the scheme that we show here will satisfy a very weak notion of function hiding. But, the above definition is simpler and still interesting. There are weaker notions too: e.g., one can simply require that there be some fixed polynomial bound on the length of c^* (independent of ℓ and f).

The public-key definition is essentially the same except that (1) the underlying encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is now a public-key scheme; and (2) we now explicitly give the Eval algorithm access to the public key.

Definition 1.2. *A* fully homomorphic public-key encryption scheme *consists of four efficient algorithms* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ *with the following properties.*

- $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is a correct and semantically secure public-key encryption scheme with message space* $\{0, 1\}$.
- **(Homomorphism.)** *For function* $f : \{0,1\}^\ell \to \{0,1\}$ *and any plaintexts* $m_1, \ldots, m_\ell \in \{0, 1\}$,

$$\Pr_{(sk,pk)\sim\mathsf{Gen}(1^n)}[c_i \leftarrow \mathsf{Enc}(pk, m_i),\ \mathsf{Dec}(sk, \mathsf{Eval}(pk, f, c_1, \ldots, c_\ell)) = f(m_1, \ldots, m_\ell)] = 1 \ .$$

- **(Compactness.)** *There exists some polynomial* $p(n)$ *such that for any function* $f : \{0,1\}^\ell \to \{0,1\}$ *and any* $m_1, \ldots, m_\ell \in \{0, 1\}$,

$$\Pr_{(sk,pk)\sim\mathsf{Gen}(1^n)}[c_i \leftarrow \mathsf{Enc}(pk, m_i), c^* \leftarrow \mathsf{Eval}(pk, f, c_1, \ldots, c_\ell),\ |c^*| = |c_i|] = 1 \ ,$$

where $|c^*|$ *is the bit length of* c^*.

Remark (Why only one-bit output?)**.** *Notice that, while we formally defined the function* $f : \{0,1\}^\ell \to \{0,1\}$ *to output only a single bit, this immediately extends to functions* $F : \{0,1\}^\ell \to \{0,1\}^k$ *that output many bits. In particular, we can define* $F := (f_1, f_2, \ldots, f_m)$*, so that* f_i *computes the* i*th bit of* F*. Notice that the* f_i *are still efficiently computable, and by running* Eval *with* $f_1, \ldots, f_m$*, we can homomorphically evaluate* F*. (Of course, this might not be the most efficient way to do this.) Therefore, this definition is essentially equivalent to the definition that allows* f *to output many bits. (On the homework, we use the many-bit definition.)*

1.4 Use case: delegation

Suppose there is a giant company named, e.g., Nile, which sells a service called Nile Web Services (NWS), which allows customers to use their computational power. In particular, customers can upload their data $M = (m_1, \ldots, m_\ell) \in \{0,1\}^\ell$ (e.g., their genome or a list of their clients, represented as some bit string) to NWS's servers, specify some function $f : \{0,1\}^\ell \to \{0,1\}$, and receive in response $f(M)$.

Of course, we might not be comfortable just sending M in the clear to NWS! E.g., if M consists of sensitive medical records. Fortunately, with FHE, we do not have to. Specifically, if $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ is a fully-homomorphic encryption scheme (it does not matter in this case whether it is a public-key scheme or a secret-key scheme), then we can instead send $c_1 \leftarrow \mathsf{Enc}(m_1), \ldots, c_\ell \leftarrow \mathsf{Enc}(m_\ell)$ to NWS, together with f. NWS can then respond with $c^* \leftarrow \mathsf{Eval}(f, c_1, \ldots, c_\ell)$, and we can simply decrypt c^* to get $f(M)$.

Notice that this would be useless without compactness! In particular, for this to be useful, it must be the case that encrypting M and decrypting c^* are more efficient than computing $f(M)$ ourselves. Compactness (together with the fact that Dec is efficiently computable) guarantees that this is the case for a sufficiently complicated efficiently computable function f (e.g., a function that can be computed in time n^{100} but not n^{99}).

This is called *delegation*. In fact, there are many more considerations that go into delegation that we will not get into here. E.g., how do we confirm that the result that we get is really $f(M)$? Perhaps NWS was lazy (or malicious) and instead computed $f'(M)$! (We could compute $f(M)$ ourselves to check, but this is not very useful. Instead, we would like some way to confirm the computation in time that is independent of the complexity of f.)

2 Constructing FHE! GSW encryption

Now that we have the definition, we can try to actually construct fully homomorphic encryption. I hope it's clear that this task is difficult! It was originally suggested by Rivest, Adleman, and Dertouzos [RAD78] all the way back in 1978. But, the first construction was only found in 2008, by Craig Gentry [Gen09]—30 years later. (It's my understanding that many people thought that this was impossible before Gentry's work, though I might be wrong about that.)

Gentry's original construction was rather complicated, so we will see a more modern form due to Gentry, Sahai, and Waters [GSW13], called GSW encryption. The construction is based on the LWE assumption that we saw in the previous lecture. In fact, the encryption scheme itself (ignoring the Eval function for now) is not too different from the Regev encryption scheme that we saw in that lecture. There are two main differences:

- While Regev ciphertexts were vectors, GSW ciphertexts are matrices. You can think of GSW ciphertexts as essentially concatenations of many Regev ciphertexts of the same underlying plaintext (though this is not strictly true because of the other difference), in the same way that a matrix can be viewed as a concatenation of many vectors.
- Instead of multiplying the message by $\lfloor q/2 \rfloor$ like we did in Regev encryption, we will multiply the message by some very cleverly chosen *gadget matrix* $\boldsymbol{G}$.

First, we will describe the scheme without specifying $\boldsymbol{G}$, and without specifying Eval. Then, we will work to construct Eval and derive the properties of $\boldsymbol{G}$ that we need in the process.

The scheme has public parameters q, m, N, $\sigma \ll q/m$ and $\boldsymbol{G} \in \mathbb{Z}_q^{N \times (n+1)}$ where $N := (n+1)\lfloor \log q \rfloor$ (all of which depend on the security parameter n).

- $\mathsf{Gen}(1^n)$: Sample $\boldsymbol{s} \sim \mathbb{Z}_q^n$, $\boldsymbol{A} \sim \mathbb{Z}_q^{m \times n}$, and $\boldsymbol{e} \sim [-\sigma, \sigma]^m$.

  For convenience, we take $sk := \boldsymbol{t} := \begin{pmatrix} \mathbf{s} \\ -1 \end{pmatrix} \in \mathbb{Z}_q^{n+1}$ (i.e., we add an extra coordinate -1 to the secret $\boldsymbol{s}$) and public key $pk := \mathbf{B} := (\boldsymbol{A}, \boldsymbol{A}\boldsymbol{s} + \boldsymbol{e} \bmod q) \in \mathbb{Z}_q^{m \times (n+1)}$ (i.e., we add an extra column $\boldsymbol{A}\boldsymbol{s} + \boldsymbol{e}$ to the matrix $\boldsymbol{A}$). This is exactly the same as Regev encryption, just with some extra notation. Notice that we have set things up so that $\mathbf{B}\boldsymbol{t} = -\boldsymbol{e} \bmod q$, i.e., the secret key is a vector $\boldsymbol{t}$ such that $\mathbf{B}\boldsymbol{t} \bmod q$ is small.

- $\mathsf{Enc}(pk, \mu \in \{0,1\})$: Sample $\boldsymbol{R} \sim \{0,1\}^{N \times m}$ and output $\boldsymbol{C} := \boldsymbol{R}\boldsymbol{B} + \mu \boldsymbol{G} \in \mathbb{Z}_q^{N \times (n+1)}$.

- $\mathsf{Dec}(sk, \boldsymbol{C})$: Compute $\boldsymbol{d} := \boldsymbol{C}\boldsymbol{t} \in \mathbb{Z}_q^N$. Output 1 if at least one of the coordinates of $\boldsymbol{d}$ is larger than, say, $q/10$ in absolute value, and 0 otherwise.

To understand correctness, notice that $\boldsymbol{d} = \boldsymbol{C}\boldsymbol{t} = \boldsymbol{R}\boldsymbol{B}\boldsymbol{t} + \mu \boldsymbol{G}\boldsymbol{t} = -\boldsymbol{R}\boldsymbol{e} + \mu \boldsymbol{G}\boldsymbol{t} \bmod q$. Notice that the coordinates of $\boldsymbol{R}\boldsymbol{e}$ are of size at most $\sigma m \ll q$. Therefore, for the scheme to be correct, we need some guarantee that at least one coordinate in $\boldsymbol{G}\boldsymbol{t} \bmod q$ is always larger than, e.g., $q/5$. Since the last coordinate of $\boldsymbol{t}$ is -1 by definition, it suffices if $\boldsymbol{G}$ has at least one row of the form $(0, 0, 0, \ldots, 0, r)$ for some $r \approx q/2$. Our $\boldsymbol{G}$ will in fact satisfy this, so the scheme is correct.

Furthermore, notice that, for correctness, we did not need $\boldsymbol{R}$ to have coordinates in $\{0,1\}$. We could have instead taken, e.g., $\boldsymbol{R} \in \{-B, \ldots, B\}^{N \times m}$ for any $B \ll q/(\sigma m)$. I.e., $\boldsymbol{R}$ only needs to be "small." This will be important soon.

The security proof for this scheme, assuming Decisional LWE, is identical to the proof for Regev encryption. In particular, when we proved security of the Regev scheme, it really came down to proving that $\boldsymbol{R}\boldsymbol{B}$ is computationally indistinguishable from a uniformly random matrix in $\mathbb{Z}_q^{N \times (n+1)}$ (though in the Regev case we only looked at a single row, i.e., we took $N = 1$). And, this is clearly enough to imply security here.

2.1 Thinking about Eval

Now, let's start to think about how the Eval function could possibly behave. Specifically, let's see how to take two ciphertexts $\boldsymbol{C}_1 := \boldsymbol{R}_1\boldsymbol{B} + \mu_1\boldsymbol{G} \bmod q$ and $\boldsymbol{C}_2 := \boldsymbol{R}_2\boldsymbol{B} + \mu_2\boldsymbol{G} \bmod q$ and use them to compute $\boldsymbol{C}_\oplus := \boldsymbol{R}_\oplus\boldsymbol{B} + (\mu_1 \oplus \mu_2)\boldsymbol{G} \bmod q$ for some "small" matrix $\boldsymbol{R}_\oplus$ *and* $\boldsymbol{C}_\times := \boldsymbol{R}_\times\boldsymbol{B} + \mu_1\mu_2\boldsymbol{G} \bmod q$ for some "small" matrix $\boldsymbol{R}_\times$. Intuitively, if we can accomplish this, then we might hope to compute an arbitrary circuit with $\oplus$ and $\mathsf{AND} = \times$ gates. (It is not quite sufficient to show how to compute such a $\boldsymbol{C}_\oplus$ and $\boldsymbol{C}_\times$, because in general we have no guarantee that we can repeat this procedure many times. In particular $\boldsymbol{R}$ can get larger each time we apply these operations, and eventually it could become too large to decrypt. This will be a major issue that we will need to deal with later.)

First, consider $\boldsymbol{C}_+ := \boldsymbol{C}_1 + \boldsymbol{C}_2$. Notice that $\boldsymbol{C}_+ = (\boldsymbol{R}_1 + \boldsymbol{R}_2)\boldsymbol{B} + (\mu_1 + \mu_2)\boldsymbol{G} = \boldsymbol{R}_+\boldsymbol{B} + (\mu_1 + \mu_2)\boldsymbol{G} \bmod q$. This is kind of like what we want. In particular, $\boldsymbol{R}_+$ is not much larger than $\boldsymbol{R}_1$ and $\boldsymbol{R}_2$, so we can declare that to be "small." But, $\mu_1 + \mu_2$ is not necessarily the same as $\mu_1 \oplus \mu_2$. However, notice that $\mu_1 \oplus \mu_2 = \mu_1 + \mu_2 - 2\mu_1\mu_2$. So, suppose we already knew how to compute

$\boldsymbol{C}_\times$, then we could compute

$$\boldsymbol{C}_\oplus := \boldsymbol{C}_+ - 2\boldsymbol{C}_\times = (\boldsymbol{R}_+ - 2\boldsymbol{R}_\times)\boldsymbol{B} + (\mu_1 \oplus \mu_2)\boldsymbol{G} = \boldsymbol{R}_\oplus \boldsymbol{B} + (\mu_1 \oplus \mu_2)\boldsymbol{G} \ ,$$

where $\boldsymbol{R}_\oplus := \boldsymbol{R}_+ - 2\boldsymbol{R}_\times$ is not much larger than $\boldsymbol{R}_+$ and $\boldsymbol{R}_\times$.

So, "all" we need to do is to show how to compute $\boldsymbol{C}_\times$. We will then worry about how quickly $\boldsymbol{R}$ grows when we perform these operations, to try to allow ourselves to perform these operations many times.

2.2 Choosing G wisely

Given what we've done so far, I think it's natural to assume that $\mathbf{C}_\times$ will have the form $\boldsymbol{C}_\times = I_{\boldsymbol{G}}(\boldsymbol{C}_1)\boldsymbol{C}_2$ for some function $I_{\boldsymbol{G}} : \mathbb{Z}_q^{N\times(n+1)} \to \mathbb{Z}_q^{N\times N}$. (The standard notation for what I'm calling $I_{\boldsymbol{G}}$ is $\boldsymbol{G}^{-1}$, but this notation has some problems. In particular, the standard notation makes it look like $I_{\boldsymbol{G}}$ is a linear function, but it is not.)

Let's see what properties we want $I_{\boldsymbol{G}}$ to have. Expanding out the definition of $\boldsymbol{C}_\times$, we have

$$\boldsymbol{C}_\times = I_{\boldsymbol{G}}(\boldsymbol{C}_1)\boldsymbol{R}_2\boldsymbol{B} + \mu_2 I_{\boldsymbol{G}}(\boldsymbol{C}_1)\boldsymbol{G} \ .$$

So, it would suffice if $I_{\boldsymbol{G}}$ had two properties: (1) $I_{\boldsymbol{G}}(\boldsymbol{C}_1)$ should be "small," e.g., $I_{\boldsymbol{G}}(\boldsymbol{C}_1) \in \{0,1\}^{N\times N}$; *and* (2) $I_{\boldsymbol{G}}(\boldsymbol{C}_1)\boldsymbol{G} = \mathbf{C}_1 = \boldsymbol{R}_1\boldsymbol{B} + \mu_1\boldsymbol{G}$. I.e., $I_{\mathbf{G}}$ is kind of like an inverse of $\mathbf{G}$, in the sense that $I_{\mathbf{G}}(\mathbf{C})\mathbf{G} = \mathbf{C}$. Then, we would have

$$\mathbf{C}_\times = (I_{\mathbf{G}}(\mathbf{C}_1)\mathbf{R}_2 + \mu_2\mathbf{R}_1)\mathbf{B} + \mu_1\mu_2\mathbf{G} = \mathbf{R}_\times\mathbf{B} + \mu_1\mu_2\mathbf{G} \ ,$$

as needed.

It turns out that there is a very natural function $I_{\boldsymbol{G}}$ and matrix $\boldsymbol{G} \in \mathbb{Z}_q^{N\times(n+1)}$ that together satisfy these properties. We take $N := (n+1)(\lfloor \log_2 q \rfloor + 1)$, and we take $I_{\boldsymbol{G}}$ to be the "bit decomposition function." I.e., recall that each row of $\boldsymbol{C}_1$ is a vector in $\mathbb{Z}_q^{n+1}$. $I_{\boldsymbol{G}}$ operates on each such row, taking such vectors and converting them into vectors in $\{0,1\}^N$ by writing out each coordinate in binary. (Conventionally, we write the lower-order bits first in this representation.) E.g., if $q = 15$ and $n = 2$, then

$$I_{\mathbf{G}}(2,5,9) = (0,1,0,0,1,0,1,0,1,0,0,1)$$

because 2 maps to $(0,1,0,0)$, 5 maps to $(1,0,1,0)$ and 9 maps to $(1,0,0,1)$.

Then, the gadget matrix $\boldsymbol{G}$ is given by

$$\boldsymbol{G} := \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 2 & 0 & 0 & \cdots & 0 \\ 4 & 0 & 0 & \cdots & 0 \\ 8 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 2^{\lfloor \log q \rfloor} & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 2 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 2^{\lfloor \log q \rfloor} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ 0 & 0 & 0 & \cdots & 2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 2^{\lfloor \log q \rfloor} \end{pmatrix}$$

(If you are comfortable with tensor products, then $\boldsymbol{G} := \boldsymbol{I} \otimes \boldsymbol{g}$, where $\boldsymbol{g} := (1, 2, \ldots, 2^{\lfloor \log q \rfloor})$ and $\boldsymbol{I}$ is the identity matrix.) Notice that $I_{\boldsymbol{G}}(\boldsymbol{C}_1)\boldsymbol{G} = \boldsymbol{C}_1$. (Notice also that $\boldsymbol{G}$ has a row of the form $(0, \ldots, 0, 2^{\lfloor \log q \rfloor - 1})$, which we needed earlier for correctness. In some sense, $I_{\boldsymbol{G}}$ is an inverse of $\boldsymbol{G}$, but notice that $I_{\boldsymbol{G}}$ is not a linear function, so it is not what we would normally think of as an inverse of $\boldsymbol{G}$.) In other words, $\boldsymbol{G}$ is the linear transformation that maps bit representations to integers.

In particular,

$$\mathbf{C}_\times = (I_{\mathbf{G}}(\mathbf{C}_1)\mathbf{R}_2 + \mu_2\mathbf{R}_1)\mathbf{B} + \mu_1\mu_2\mathbf{G} = \mathbf{R}_\times\mathbf{B} + \mu_1\mu_2\mathbf{G} \bmod q \ .$$

Since $I_{\boldsymbol{G}}(\boldsymbol{C}_1) \in \{0,1\}^{N \times N}$ is small and $\mathbf{R}_2 \in \{0,1\}^{N \times m}$ is small, the matrix $\boldsymbol{R}_\times$ is "small." Specifically, the coordinates of $\mathbf{R}_\times = I_{\mathbf{G}}(\boldsymbol{C}_1)\boldsymbol{R}_2$ lie in the interval $[-N-1, N+1]$. We will therefore still be able to decrypt correctly as long as $N \ll q/(m\sigma)$. Below, we will pay much more attention to the size of $\boldsymbol{R}_\times$.

2.3 Controlling the noise and leveled fully homomorphic encryption

So, we have shown how to homomorphically compute a single XOR operation or a single AND operation while maintaining the correctness of the ciphertext. In order to compute an entire circuit, we would like to perform these operations repeatedly. E.g., we would like to homomorphically compute the XOR of $\mathbf{C}_1$ and $\mathbf{C}_2$ to obtain $\mathbf{C}_\oplus$, and the AND of $\mathbf{C}_3$ and $\mathbf{C}_4$ to obtain $\mathbf{C}_\times$, and then we might wish to, e.g., homomorphically compute the AND of $\mathbf{C}_\oplus$ and $\mathbf{C}_\times$, and so on.

Our analysis above shows that, if we start out with fresh ciphertexts $\boldsymbol{C}_1, \boldsymbol{C}_2$ and compute $\boldsymbol{C}_\times$ or $\boldsymbol{C}_\oplus$, these operations work. But, after we have applied these operations, the size of $\mathbf{R}$ has grown. Remember that our decryption algorithm converts a ciphertext $\boldsymbol{C} = \boldsymbol{R}\boldsymbol{B} + \mu\boldsymbol{G} \bmod q$ into $\boldsymbol{R}\boldsymbol{e} + \mu\boldsymbol{G}\boldsymbol{t} \bmod q$. We call the term $\boldsymbol{R}\boldsymbol{e}$ the "noise," and we can decrypt if the coordinates of the noise are all in the range, e.g., $\{-q/10, \ldots, q/10\}$.

We therefore want to control the size of the noise, and in particular, to keep the noise small even if we perform many homomorphic operations like the ones described above. To that end, let's use the notation $\|\boldsymbol{R}\|_\infty := \max_{i,j} |R_{i,j}|$, i.e., $\|\boldsymbol{R}\|_\infty$ is the maximal absolute value of a coordinate in $\boldsymbol{R}$. Notice that we can decrypt the ciphertext $\boldsymbol{RB} + \mu\boldsymbol{G}$ if $\|\boldsymbol{R}\|_\infty \leq q/(10m\sigma)$. So, we want to study $\|\boldsymbol{R}_\oplus\|_\infty$ and $\|\boldsymbol{R}_\times\|_\infty$, where $\boldsymbol{R}_\oplus := \boldsymbol{R}_1 + \boldsymbol{R}_2 - 2\boldsymbol{R}_\times$ and $\boldsymbol{R}_\times := I_{\mathbf{G}}(\boldsymbol{C}_1)\boldsymbol{R}_2 + \boldsymbol{R}_1$.

Recalling that $I_{\boldsymbol{G}}(\boldsymbol{C}_1) \in \{0,1\}^{N\times N}$, we have

$$\|\boldsymbol{R}_\times\|_\infty \leq N\|\boldsymbol{R}_2\|_\infty + \|\boldsymbol{R}_1\|_\infty \ ,$$

and

$$\|\boldsymbol{R}_\oplus\|_\infty \leq \|\boldsymbol{R}_1\|_\infty + \|\boldsymbol{R}_2\|_\infty + 2\|\boldsymbol{R}_\times\|_\infty \leq 3\|\boldsymbol{R}_1\|_\infty + (2N+1)\|\boldsymbol{R}_2\|_\infty \ .$$

So, the noise increases by a factor of roughly $N \approx n\log q$ for every operation. Therefore, if we need to evaluate a circuit of depth d, the noise will grow by a factor of roughly $N^d \approx (n\log q)^d$. This means that we can homomorphically evaluate any circuit of depth d such that $(n\log q)^d \ll q/(m\sigma)$.

As it turns out, LWE is not secure if $q/\sigma \gtrsim 2^{n/\operatorname{poly}\log(n)}$. This means that with this approach, we can't hope to handle $d \gtrsim n/\operatorname{poly}\log(n)$. In other words, we can only compute circuits with slightly sublinear depth in the security parameter n. This is called *leveled fully homomorphic encryption*, or sometimes *somewhat homomorphic encryption*, and it is already *very* useful! Many functions of interest can be written as low-depth circuits, and we can always just increase the security parameter n if we want to compute deeper circuits. However, this is not quite an FHE scheme because of this subtle restriction on the depth.

2.4 Going from leveled FHE to FHE using bootstrapping!

To get true fully homomorphic encryption, we need one last idea, called bootstrapping. This idea is due to Gentry. It is also completely insane!

Gentry's idea was to compute the *decryption function* homorphically! Specifically, for a ciphertext $\boldsymbol{C}$, let $f_{\boldsymbol{C}}(sk) := \mathsf{Dec}(sk, \boldsymbol{C})$. Notice that we think of $\boldsymbol{C}$ as hard-wired into the function $f_{\boldsymbol{C}}$ and sk as the input. This is important!

Now, let $\boldsymbol{C}_1^* := \mathsf{Enc}(pk, sk_1), \ldots, \boldsymbol{C}_\ell^* := \mathsf{Enc}(pk, sk_\ell)$ be encryptions of the bits of the secret key! Then, consider what happens when we run $\boldsymbol{C}' := \mathsf{Eval}(f_{\boldsymbol{C}}, \boldsymbol{C}_1^*, \ldots, \boldsymbol{C}_\ell^*)$. Take a second to stare at this, and you will see that $\mathsf{Dec}(sk, \boldsymbol{C}') = \mathsf{Dec}(sk, \boldsymbol{C})$. In other words, $\boldsymbol{C}'$ is an encryption of the same plaintext as $\boldsymbol{C}$! In particular, if $\boldsymbol{C} := \boldsymbol{RB} + \mu\boldsymbol{G}$ with $\|\boldsymbol{R}\|_\infty < q/(10\sigma m)$, then $\boldsymbol{C}' := \boldsymbol{R}'\boldsymbol{B} + \mu\boldsymbol{G} \bmod q$ for the same plaintext μ. The process of converting a ciphertext to another in this way is known as *bootstrapping*.

But, how big is $\boldsymbol{R}'$? Well, $\boldsymbol{C}_1^*, \ldots, \boldsymbol{C}_\ell^*$ are fresh encryptions (i.e., they are the direct output of the encryption algorithm; not Eval), so they all have noise level $\|\boldsymbol{R}^*\|_\infty \leq 1$. We then homomomorphically evaluated the decryption function $f_{\boldsymbol{C}}$ on these. So, our final noise level will be $\|\boldsymbol{R}'\|_\infty \approx (n\log q)^{d_{\mathsf{Dec}}}$, where d_{Dec} is the depth of the decryption circuit $f_{\boldsymbol{C}}$. Notice that this noise level does not depend on the noise level of $\boldsymbol{C}$! So, if $\boldsymbol{C}$ has noise larger than $(n\log q)^{d_{\mathsf{Dec}}}$, then bootstrapping gives us a lower-noise ciphertext $\boldsymbol{C}'$ of the same plaintext!

Gentry made the (crazy!) observation that the decryption function $f_{\boldsymbol{C}}(sk)$ is relatively simple—it just multiplies $sk = \boldsymbol{t}$ by $\boldsymbol{C}$ modulo q and checks if the result is big or small. He was therefore able to show that the depth d_{Dec} of a circuit needed to compute this function can be made small enough to make this noise level manageable.

Therefore, in order to compute an arbitrarily deep circuit f, we can do the following. In addition to the standard public key $\boldsymbol{B}$, the key generation algorithm also releases $\boldsymbol{C}^*_1, \ldots, \boldsymbol{C}^*_\ell$, encryptions of the bits of the secret key. To homomorphically evaluate f, we start to compute the circuit as described in the previous section until we reach depth d for which $N^d \approx q/(10\sigma m)$—i.e., until we reach a depth where the noise is nearly the largest noise that can still be decrypted. Once we reach that depth, we run Gentry's crazy bootstrapping procedure on the ciphertexts corresponding to each gate at that depth, giving us new encryptions of the same plaintexts with smaller noise level! Then, we continue to compute the circuit on the new ciphertexts until the noise level approaches $q/(10\sigma m)$, at which point we again run the bootstrapping procedure. Proceeding in this way, we eventually compute all of f, without ever letting the noise grow larger than $q/(10\sigma m)$!

Not so fast: circular security. Notice that, in order for this bootstrapping process to work, the public key must include $\boldsymbol{C}^*_1, \ldots, \boldsymbol{C}^*_\ell$, which are encryptions of the secret key! Unfortunately, we do not know how to prove the security of this new scheme under the LWE assumption. Instead, we have to assume what's known as *circular security*, which says that, well, the scheme remains secure even if we publish encryptions of the bits of the secret key. (With some work, one can write this as an assumption that looks like a funny variant of LWE.) As far as we know, this is true, but we do not know how to prove that this strange assumption holds under LWE.

Better leveled FHE from bootstrapping. The form of leveled FHE that we achieved above is rather unsatisfying because it fails at some depth d, which depends only on n, σ, q, and m. It would be better if we could choose any depth $d = \mathrm{poly}(n)$ that we like and generate (sk, pk) depending on d such that we can homomorphically evaluate any circuits of depth d (but no further). This isn't exactly *fully* homomorphic encryption because we can't compute any circuit, but it's pretty close.

To do this, we can again use bootstrapping, except instead of just having one public key pk and one secret key sk, we generate many $(sk_1, pk_1), \ldots, (sk_d, pk_d)$. We also include in the public key $\boldsymbol{C}^*_{i,1} := \mathsf{Enc}(pk_{i+1}, sk_{i,1}), \ldots, \boldsymbol{C}^*_{i,\ell} := \mathsf{Enc}(pk_{i+1}, sk_{i,\ell})$ for all i. I.e., we encrypt the secret key sk_i using the public key pk_{i+1}. This lets us get around the circular security issue because sk_i is unrelated to pk_{i+1}. So, a simple hybrid argument shows that the resulting scheme is semantically secure.

Then, to homomorphically evaluate a circuit of depth at most d, we do the same bootstrapping trick as before, except now the first time that we bootstrap we use $\boldsymbol{C}^*_{1,1}, \ldots, \boldsymbol{C}^*_{1,\ell}$. Notice that this will convert our ciphertexts from encryptions under pk_1 to encryptions under pk_2. Then, the second time that we bootstrap, we use $\boldsymbol{C}^*_{2,1}, \ldots, \boldsymbol{C}^*_{2,\ell}$, etc. In general, after the ith bootstrap, we will have encryptions under the pk_{i+1}, and the noise level will still remain relatively small.

This allows us to have arbitrarily good *leveled* fully homomorphic encryption from LWE—without needing circular security and without needing to mess with the security parameter n.

2.5 A note on (im)practicality

While the above scheme is polynomial-time computable, it is *not* very practical. In practice, we must take $n \gg 100$ in order to have any hope of security, and q must be quite large as well. This means that we will need millions of bits to represent the ciphertext $\boldsymbol{C}$ corresponding to a single plaintext bit μ.

In particular, this means that computing f homomorphically will be slower than computing f directly by a factor of over one million. It's hard to imagine a situation in which one would want to delegate such a computation with such large overhead.

And, the bootstrapping procedure requires us to homomorphically compute a function $f_{\boldsymbol{C}}$ consisting of millions of gates. That's… not ideal.

Fortunately, there are tricks to make this much more efficient. Perhaps the most interesting trick is to use a variant of the above scheme in which many of the matrices, like $\boldsymbol{A}$ and $\boldsymbol{R}$, are not chosen uniformly at random, but are instead chosen to be a random matrix from some family with a succinct description. If done properly, this idea leads to the notion of Ring-LWE, and it ends up saving us a factor of roughly n in the size of ciphertexts and in the running time of the encryption, decryption, and evaluation algorithms. Another important trick is to pack more than one bit into every ciphertext.

With enough tricks, FHE becomes quite practical—so much so that you can find many different implementations online, including implementations of this crazy bootstrapping idea.

References

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, 2009. 5

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encription from Learning with Errors: conceptually-simpler, asymptotically-faster, attribute-based. In *CRYPTO*, pages 75–92, 2013. 5

[RAD78] R Rivest, L Adleman, and M Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978. 5