11 Homomorphic Encryption

11.1 Homomorphisms and privacy

1. I want to search a medical database for information about an STD without letting anyone know I have one.

2. I want to search for politically dangerous information in an authoritarian regime.

3. I want to store and search my data in the cloud without giving any internet companies access to it.

4. I want to make a database of genetics information available to researchers without allowing them to identify any specific person.

In all of these cases, I want to allow someone to do computations on my data, but I don't want them to actually have access to my data.

Definition 11.1. Let R, S be rings. We say a function $f : R \to S$ is a *homomorphism* if $f(x+y) = f(x) + f(y)$ and $f(xy) = f(x)f(y)$.

So we want an encryption algorithm that is homomorphic: we can encrypt the data, do a computation on it, and decrypt it, and we get the same result as just doing the computation on the original data.

11.2 Encryption with Ring-LWE

The algorithm presented here was presented in [Brakerski and Vaikuntanathan(2011)].

Our setup is very similar to last week's. We have $R_q = \mathbb{Z}/q\mathbb{Z}[x]/(x^N + 1)$ where N is a power of 2. We have a way of randomly generating small elements of the ring.

We can make a symmetric encryption algorithm out of this straightforwardly:

Algorithm 11.1. Key generation:

1. Alice generates a random small polynomial $s(x)$; this is the shared, symmetric key.

Encryption

1. Alice generates a random polynomial $a(x)$ from the entire ring, and a random small polynomial $e(x)$.

2. Her message is a string of bits, which she can think about as a polynomial $m(x)$ with coefficients either 0 or 1.

3. Alice computes $c_1(x) = -a(x)$ and $c_0(x) = a(x)s(x) + 2e(x) + m(x)$.

4. She transmits the ciphertext $(c_0(x), c_1(x))$.

Decryption:

1. Bob receives $(c_0(x), c_1(x))$.

2. He computes $c_0(x) + c_1(x)s(x)$.

3. He reduces modulo 2, and gets the message $m(x)$.

We see that $c_0(x) + c_1(x)s(x) = a(x)s(x) + 2e(x) + m(x) - a(x)s(x) = 2e(x) + m(x)$. When we reduce mod 2 we can ignore the $2e(x)$ term, and since all the coefficients of $m(x)$ are either 0 or 1, we get $m(x)$ back exactly.

Example 11.2. Let's take $N = 4$ and $q = 17$. We want to encrypt the message $(1, 0, 1, 0)$. We first encode this as $x^3 + x$.

We'll take as a key $s(x) = x^3 - x^2 + 2x$.

To encrypt, Alice generates a polynomial $a(x) = 3x^2+7x-5$, and a small error polynomial $e(x) = 2x^3 + x + 1$. She computes:

$$\begin{aligned} c_1(x) &= -a(x) = -3x^2 - 7x + 5 \\ c_0 &= a(x)s(x) + 2e(x) + m(x) \\ &= (3x^2 + 7x - 5)(x^3 - x^2 + 2x) + 4x^3 + 2x + 2 + x^3 + x \\ &= 3x^5 + 4x^4 - x^3 + 19x^2 - 7x + 2 \\ &= -x^3 + 2x^2 + 7x - 2. \end{aligned}$$

So Alice sends the message $(-x^3 + 2x^2 + 7x - 2, -3x^2 - 7x + 5)$.

When Bob receives this, he computes

$$\begin{aligned} c_0 + c_1 s &= -x^3 + 2x^2 + 7x - 2 + (-3x^2 - 7x + 5)(x^3 - x^2 + 2x) \\ &= -3x^5 - 4x^4 + 5x^3 - 17x^2 + 17x - 2 \\ &= 5x^3 + 3x + 2 \end{aligned}$$

and reducing mod 2 gives $x^3 + x$.

With a little tweaking, we can turn this into a public key algorithm.

Algorithm 11.2. Keygen:

1. Alice generates a random polynomial a_0 and random small polynomials s and e_0.
2. Alice computes $b_0 = as + 2e_0$. Her public key is (a_0, b_0).

Encryption:

1. Bob generates random small polynomials v, e_1, e_2.
2. He computes $a_1 = a_0 v + 2e_1, b_1 = b_0 v + 2e_2$.
3. He computes $c_0 = b_1 + m$ and $c_1 = -a_1$.
4. The ciphertext is (c_0, c_1).

Decryption:

1. Alice receives (c_0, c_1).
2. Alice computes $M = c_0 + sc_1$.
3. Alice reduces M mod 2 and gets the message m.

We see that

$$\begin{aligned} M &= c_0 + sc_1 \\ &= b_1 + m + sc_0 \\ &= (b_0 v + 2e_2) + m - sa_1 \\ &= a_0 sv + 2e_0 v + 2e_2 + m - sa_0 v - 2se_1 \\ &= m + 2(e_0 v + e_2 - se_1). \end{aligned}$$

Reducing mod 2 just leaves m.

Remark 11.3. Why is it important that the error terms are small, here? Because reducing mod q and mod 2 don't *actually* commute.

Suppose $q = 17$ and the message is $(1, 1, 1)$ or $m = x^2 + x + 1$, and suppose we have a large error $e = 8x^2 + 9x + 2$. Then $m + 2e = 17x^2 + 19x + 7 = 2x + 7$, and reducing mod 2 gives $m = 1$ and a message of $(0, 0, 1)$. If the error term adds up enough to be large relative to q then we can get the wrong sum when we mod out by 2 to recover m.

11.3 Somewhat Homomorhpic Encryption

So far this hasn't gotten us anything really new. But it turns out this encryption gives us a partial homomorphism.

For the rest of this lecture we'll just look at the symmetric version. This can all be upgraded into the public key version, just with more careful arithmetic.

Proposition 11.4. *This encryption scheme is additively homomorphic. That is,* $d(e(m) + e(m')) = m + m'$.

Proof. We encrypt m as (c_0, c_1) and $m'as(c_0', c_1')$. Then we have

$$(c_0, c_1) + (c_0', c_1') = (as + 2e + m + a's + 2e' + m', -a - a').$$

To decrypt we compute

$$\begin{aligned} c_0 + c_0' + (c_1 + c_1')s &= as + 2e + m + a's + 2e' + m' + (-a - a')s \\ &= m + m' + 2(e + e') \end{aligned}$$

so reducing mod 2 gives $m + m'$. Thus $d(e(m) + e(m')) = m + m'$ as desired.

□

Getting a multiplicative homomorphism is a bit trickier. We'd like to do the obvious thing and just multiply our points together. But

$$\begin{aligned} c_0c_0' &= (as + 2e + m)(a's + 2e' + m') \\ &= mm' + asm' + a'sm + aa's^2 + 2(ea's + 2ee' + em' + e'as + e'm). \end{aligned}$$

The last term is a perfectly reasonable error term, although we might worry that it gets too large. We have the product mm' that we're looking for, and we have asm' and $a'sm$ which we know how to deal with. But what do we do with the $aa's^2$ term?

In order to deal with this, we need to carry around some extra information.

Algorithm 11.3. A ciphertext will be a sequence of ring elements $\mathbf{c} = (c_0, \dots, c_d) \in R_q^{d+1}$. Each c_i is a polynomial mod q and mod $x^N + 1$.

We add ciphertexts pointwise. If two ciphertexts have different lengths (that is, different numbers of polynomials), we can pad the shorter one out with zeroes, since it will turn out that $(c_0, \dots, c_d, 0, \dots, 0)$ will decrypt to the same message as $(c_0, \dots, c_d)$. Then we have

$$\mathbf{c} + \mathbf{c}' = (c_0, \dots, c_d) + (c_0', \dots, c_d') = (c_0 + c_0', \dots, c_d + c_d').$$

Multiplication is more complex. If $\mathbf{c} = (c_0, \dots, c_d)$ and $\mathbf{c}' = (c'_0, \dots, c'_{d'})$, we do *not* pad with zeros. Instead we introduce a new symbolic variable v, and we write

$$\mathbf{c} = \sum_{i=0}^{d} c_i v^i = c_0 + c_1 v + \cdots + c_d v^d \in R_q[d].$$

Then we can compute

$$\mathbf{c} \times \mathbf{c}' = (\hat{c}_0, \dots, \hat{c}_{d+d'})$$

where

$$\left(\sum_{i=0}^{d} c_i v^i\right)\left(\sum_{i=0}^{d'} c'_i v^i\right) = \sum_{i=0}^{d+d'} \hat{c}_i v^i.$$

So how do we decrypt such a thing? Note that if we know the private key s then we also can compute s^i for any i. If our final ciphertext is $\mathbf{c} = (c_0, \dots, c_D) \in R_q^{D+1}$, we compute $\mathbf{s} = (1, s, \dots, s^D)$, and then compute the inner product

$$\langle \mathbf{c}, \mathbf{s} \rangle = \sum_{i=0}^{D} c_i s^i.$$

(This is really just substituting s for the symbolic variable v, but it's computationally nicer to set it up this way.)

When we reduce this inner product $\langle \mathbf{c}, \mathbf{s} \rangle$ mod 2, we get as output the plaintext m, as long as the error terms are sufficiently small.

We first want to check that this decryption algorithm works for the simple bits. But if we just encrypt a message $e(m) = (c_0, c_1)$, then this decryption algorithm gives us $\langle e(m), \mathbf{s} \rangle = c_0 + c_1 s$, which is the same as our old decryption algorithm. Addition also works pretty reasonably. The hard thing to check is that multiplication does what we want it to.

Proposition 11.5. *If $e(m) = \mathbf{c}$ and $e(m') = \mathbf{c}'$, then $d(\mathbf{c} \times \mathbf{c}') = mm'$, as long as the error is small.*

Proof. We have

$$\begin{aligned}
\mathbf{c} = (c_0, c_1) &= (as + 2e + m, -a) = as + 2e + m - av \\
\mathbf{c}' = (c'_0, c'_1) &= (a's + 2e' + m', -a') = a's + 2e' + m' - a'v \\
\mathbf{c} \times \mathbf{c}' &= mm' + asm' + a'sm + aa's^2 + 2(ea's + 2ee' + em' + e'as + e'm) \\
&\quad - (as + 2e + m)a'v - (a's + 2e' + m')av + aa'v^2. \\
\langle \mathbf{cc}', \mathbf{s} \rangle &= mm' + asm' + a'sm + aa's^2 + 2(ea's + 2ee' + em' + e'as + e'm) \\
&\quad - aa's^2 - 2ea's - ma's - aa's^2 - 2e'as - m'as + aa's^2 \\
&= mm' + 2(2ee' + em' + e'm).
\end{aligned}$$

When we reduce this mod 2 we get mm' as desired.

□

So why is this only "somewhat" homomorphic encryption? Notice the constraints of "as long as the error is small". Each multiplication step has the potential to increase the error dramatically; we go from an error of $2(e+e')$ to an error of $2(2ee'+em'+e'm)$. Since the error starts out small and the messsages are very small, we can manage a few multiplications, but if we do this too often, we will no longer be able to recover from the errors we have introduced.

Example 11.6. Alice already encrypted $m = x^3 + x$ to $e(m) = (-x^3 + 2x^2 + 7x - 2, -3x^2 - 7x + 5)$.

Now suppose Alice has another message $m' = x^2+x$. She uses the same $s(x) = x^3-x^2+2x$. She needs another polynomial $a'(x) = 4x^3 - 5x^2 + 2$ and small polynomial $e'(x) = 1$. Then we compute

$$\begin{aligned} c_1' &= -a' = -4x^3 + 5x^2 + 2 \\ c_0' &= a'(x)s(x) + 2e'(x) + m'(x) \\ &= -4x^6 + 9x^5 - 13x^4 + 17x^3 + 14x^2 + 10x + 6 \\ &= -8x^3 - 5x^2 - 3x + 6 \end{aligned}$$

Then we have

$$\begin{aligned} \mathbf{c} &= -x^3 + 2x^2 + 7x - 2 + (-3x^2 - 7x + 5)v \\ \mathbf{c}' &= -8x^3 - 5x^2 - 3x - 2 + (-4x^3 + 5x^2 + 2)v \\ \mathbf{cc}' &= 8x + 10x^2 + 3x^3 + (15 + 3x + 11x^2 + 15x^3)v + (11 + 2x + 14x^2 + 13x^3)v^2. \end{aligned}$$

To decrypt we replace each v with s, and get

$$\begin{aligned} \langle \mathbf{cc}', \mathbf{s} \rangle &= 8x + 10x^2 + 3x^3 + (15 + 3x + 11x^2 + 15x^3)(x^3 - x^2 + 2x) \\ &\quad + (11 + 2x + 14x^2 + 13x^3)(x^3 - x^2 + 2x)^2 \qquad = -4x^3 + 5x^2 + 3x - 1 \end{aligned}$$

Reducing mod 2 gives $x^2 + x + 1$.

We compute $mm' = x^5+x^4+x^3+x^2 = x^3+x^2-x-1$ and mod 2 this is x^3+x^2+x+1. So we've lost a little bit of information as we did this process. If we took a bigger q, we would be able to keep the error small enough through this point that we would get the correct answer.

11.4 Fully Homomorphic Encryption

There is a standard method developed by [Gentry et al.(2009)] of "bootstrapping" a somewhat homomorphic encryption scheme into a fully homomorphic encryption scheme. It's too complex to explain right now (if only we had another week!), but I can outline the basic idea.

In essence, we find a way to encrypt the private key itself inside our message. This requires us to have what's called KDM security or "key dependent message" security: the message should remain secure when the contents of the message depend on the key. It also requires some clever compression so that the key itself doesn't make things too big to maintain the somewhat-homomorphism property.

Once we can do this, we can bootstrap our encryption. The idea is to set it up so that for a given ciphertext, it's possible to reliably run the decryption process, and at least one more operation, while preserving the homomorphism. In this case, we can encrypt a message with its decryption circuit, and then run an operation on it. Then we can take that output, encrypt it with *its* decryption circuit, and run one more operation on it. By repeating this process, we can securely run as many operations as we want on the encrypted message, at the cost of having to re-encrypt it on a regular basis.

Currently no Fully Homomorphic Encryption is actually in use, because this procedure is far too computationally expensive to be practical. But researchers are working on getting it more and more efficient so that this will eventually be practically usable.

References

[Brakerski and Vaikuntanathan(2011)] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In *Annual cryptology conference*, pages 505–524. Springer, 2011.

[Gentry et al.(2009)] Craig Gentry et al. Fully homomorphic encryption using ideal lattices. In *Stoc*, volume 9, pages 169–178, 2009.

[Singh(2015)] Vikram Singh. A practical key exchange for the internet using lattice cryptography. Cryptology ePrint Archive, Report 2015/138, 2015. `http://eprint.iacr.org/2015/138`.