Foundations of Fully Homomorphic Encryption

Hilder Vitor Lima Pereira
Postdoctoral researcher at COSIC, KU Leuven

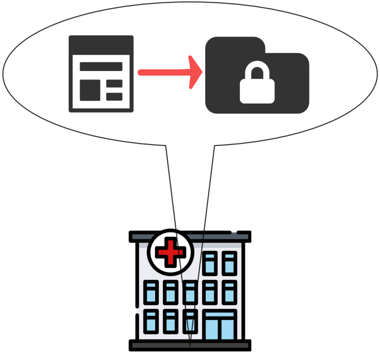
ISC Virtual Winter School

01 Feb 2023

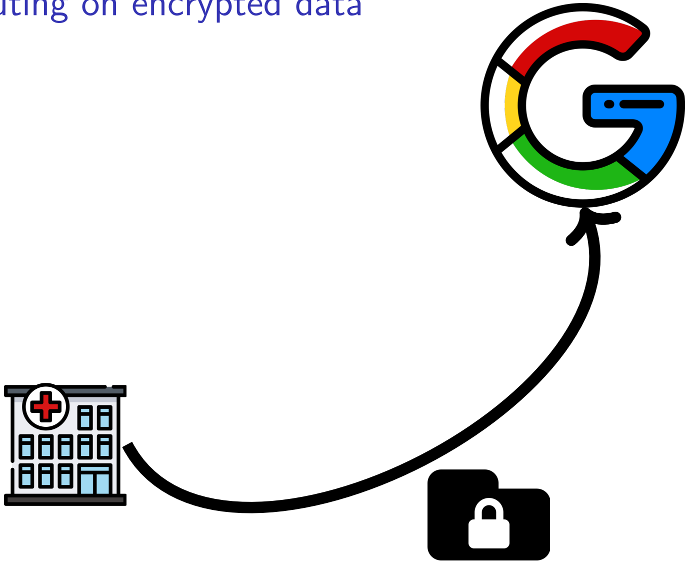
Table of Contents

Computing on encrypted data

Computing on encrypted data

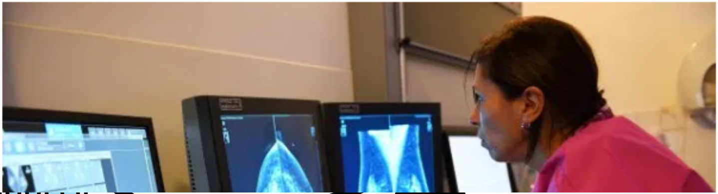


Computing on encrypted data



Computing on encrypted data

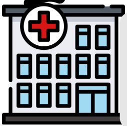
Artificial Intelligence Outperforms Doctors in Breast Cancer Diagnosis



How can the hospital use machine learning services provided by the cloud without revealing patients' data?

The cloud has to do something like

$$\bar{c} \leftarrow \mathrm{Eval}(\mathrm{pk}, f, c)$$



And the hospital:

$$f(x) = \mathrm{Dec}(\mathrm{sk}, \bar{c})$$

Fully Homomorphic Encryption (FHE)

Let Eval be a function that receives ciphertexts c_i's encrypting m_i's, a circuit C_f, and the public key pk, and outputs

$$c \leftarrow \mathrm{Eval}(\mathrm{pk}, C_f, c_1, ..., c_n)$$

such that

$$\mathrm{Dec}(\mathrm{sk}, c) = f(m_1, ..., m_n).$$

Let $\mathcal{E} = (\mathrm{KeyGen}, \mathrm{Enc}, \mathrm{Dec}, \mathrm{Eval})$ be an encryption scheme. We say that $\mathcal{E}$ is *fully* homomorphic if Eval is correct for all circuits.

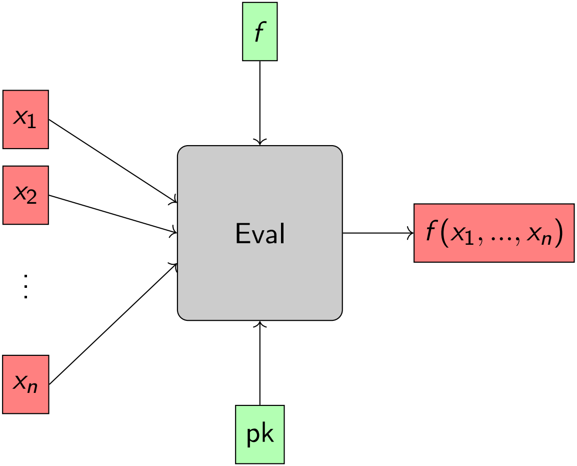


Figure: Homomorphic evaluation: red represents encrypted data.

“Trivial” applications of FHE

- Search on Google, DuckDuckGo, etc., without revealing the query nor the results.
- Use data analysis provided by the cloud without disclosing client’s data.
- Encrypting genomics data to simplify researcher’s access to them.

Applications of FHE in cryptography

- Reducing proof size in Non-interative Zero-Knowledge Proofs [GGI+15].
- One of the main tools in e-voting systems [CGGI16].
- Essential for efficient private information retrieval [MCR21].
- Key ingredient of compact deniable encryption [AGM21].

GGI+15 Craig Gentry, Jens Groth, Yuval Ishai, Chris Peikert, Amit Sahai, and Adam Smith, *Using Fully Homomorphic Hybrid Encryption to MinimizeNon-interative Zero-Knowledge Proofs*. In Journal of Cryptology 2015.

CGGI16 Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, Malika Izabachène, *A Homomorphic LWE Based E-voting Scheme*. In PQcrypto 2016.

MCR21 Haris Muhammad Mughees, Hao Chen, and Ling Ren, *OnionPIR: response efficient single-server PIR*. In ACM SIGSAC 2021.

AGM21 Shweta Agrawal, Shafi Goldwasser, and Saleet Mossel, *Deniable Fully Homomorphic Encryption from Learning with Errors*. In CRYPTO 2021.

Overview of FHE

Pros

- Very general and powerful
- Optimal 2-party secure computation
- Post-quantum secure

Cons

- Large ciphertext expansion (communication)
- It can be expensive for the client
- It is expensive for the server
- Hard to implement in practice

How FHE works

- Each FHE scheme offers some homomorphic operations (e.g., addition and multiplication)
- To evaluate f, we must represent f using the available homomorphic operations
- For example, $f(x) = x^2 + x$ would be

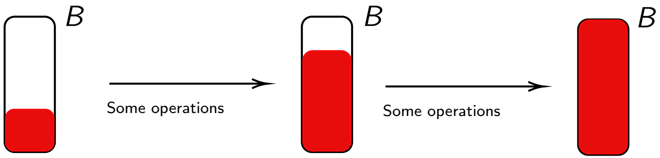
$$c' = \mathrm{HE.Mult}(c, c, \mathrm{pk}) \text{ then output } \mathrm{HE.Add}(c', c)$$

- Thus, homomorphic evaluation means executing a sequence of basic homomorphic operations.

How FHE works

Most remarkable property: ciphertexts are **noisy**

- Fresh ciphertexts (output by Enc) have very small noise
- Each homomorphic operation increases the noise
- If noise is larger than some bound B, then decryption fails



- So, the number of operations is limited...

So, we have the basic ingredients, but since the noise grows, we can only evaluate circuits with bounded depth...

Is there a way to turn "bounded" or somewhat homomorphic schemes in fully homomorphic encryption schemes?

We need a way to reduce the noise in the ciphertexts...

Bootstrapping

Gentry's idea: evaluate decryption function homomorphically!

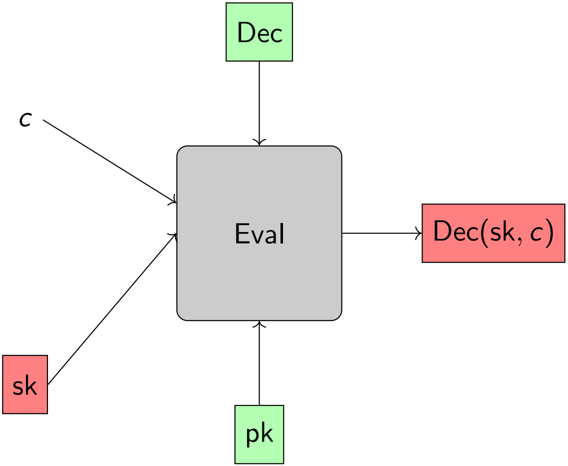


Figure: sk is encrypted. We obtain a new encryption of $m = \mathrm{Dec}(\mathrm{sk}, c)$.

Bootstrapping

Gentry's idea: evaluate decryption function homomorphically!

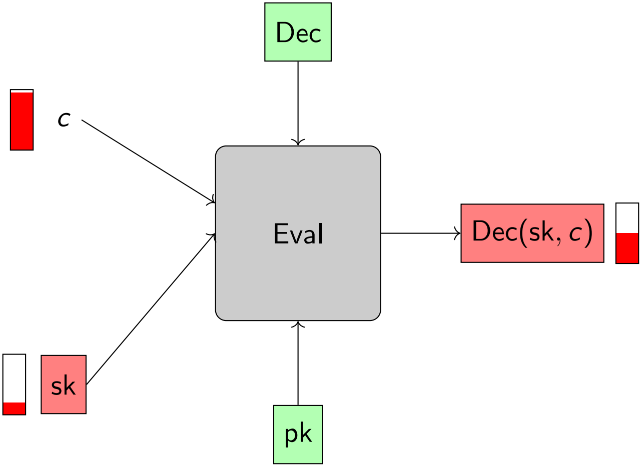


Figure: sk is encrypted. We obtain a new encryption of $m = \text{Dec}(\text{sk}, c)$.

Bootstrapping

(1) Perform some homomorphic operations
(2) Noise gets close to the limit
(3) Evaluate decryption homomorphically
(4) Go to (1)

- Bootstrapping is usually slow
- Bootstrapping requires a lot of key material

Table of Contents

Hardness assumption

(Most) FHE schemes are based on these two problems

- *learning with errors* problem (LWE)
- *ring learning with errors* problem (RLWE).

Learning with errors

- Fix a dimension n and modulus $q \in \mathbb{Z}$
- Let $\vec{s} \in \mathbb{Z}_q^n$ be a secret vector
- Now imagine you are given many random "multiples" of $\vec{s}$, that is,

$$(\vec{a}_i,\ b_i := \vec{a}_i \cdot \vec{s}) \in \mathbb{Z}_q^{n+1}$$

where $\vec{a}_i$ is uniformly sampled from $\mathbb{Z}_q^n$.

How can you recover $\vec{s}$?

Learning with errors

Define

$$A := \begin{pmatrix} - & \vec{a}_1 & - \\ - & \vec{a}_2 & - \\ & \vdots & \\ - & \vec{a}_n & - \end{pmatrix} \in \mathbb{Z}_q^{n\times n} \quad \text{and} \quad \vec{b} := \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \in \mathbb{Z}_q^n$$

Then we know that

$$A \cdot \vec{s} \equiv \vec{b} \pmod{q}$$

Thus, we can recover $\vec{s}$ by simply solving the linear system...

Learning with errors

Instead of publishing "multiples" of $\vec{s}$, we add some small errors:

$$(\vec{a}_i,\ b_i := \vec{a}_i \cdot \vec{s} + e_i) \in \mathbb{Z}_q^{n+1}$$

where $\vec{a}_i$ is uniformly sampled from $\mathbb{Z}_q^n$ and $e_i \in \mathbb{Z}$ is "small"

How can you recover $\vec{s}$?

Learning with errors

Instead of publishing "multiples" of $\vec{s}$, we add some small errors:

$$(\vec{a}_i,\ b_i := \vec{a}_i \cdot \vec{s} + e_i) \in \mathbb{Z}_q^{n+1}$$

where $\vec{a}_i$ is uniformly sampled from $\mathbb{Z}_q^n$ and $e_i \in \mathbb{Z}$ is "small"

How can you recover $\vec{s}$?

Now we know that

$$A \cdot \vec{s} + \vec{e} \equiv \vec{b} \pmod{q}$$

but both $\vec{s}$ and $\vec{e}$ are unknown.

Hardness assumption

(Most) FHE schemes are based on the *ring learning with errors* problem (RLWE).

- First, fix a power of two $N = 2^k$
- Define the ring $R = \mathbb{Z}[X]/\langle X^N + 1\rangle$
- That is, R is the set of polynomials modulo $X^N + 1$
- Then fix a positive integer q
- Define $R_q = R/qR = \mathbb{Z}_q[X]/\langle X^N + 1\rangle$
- So, R_q is the set of polynomials of degree less than N and coefficients modulo q
- Example: $N = 4$ and $q = 7$, then

$$R_q = \{a_0 + a_1 \cdot X + a_2 \cdot X^2 + a_3 \cdot X^3 : 0 \leq a_i \leq 6\}$$

The RLWE problem

Fix a secret polynomial $s \in R$

Let's say you are given multiples of s:

- Sample a_i uniformly from R_q
- Define $b_i := a_i \cdot s \bmod q$

You have many pairs $(a_i, b_i) \in R_q^2$.

Then it is easy to recover s with linear algebra
In particular, if some a_i is invertible, then $a_i^{-1} \cdot b_i \bmod q$ reveals s

But if we had $b_i = a_i \cdot s + e_i \bmod q$, then

$$a_i^{-1} \cdot b_i = s \quad + \quad \underbrace{a_i^{-1} \cdot e_i}_{\text{close to uniform}} \quad \bmod q$$

that is, we would not recover s like this...

The RLWE problem

Fix a secret polynomial $s \in R$

- Sample a_i uniformly from R_q
- Noise: small $e_i \leftarrow \chi$
- Let $b_i := a_i \cdot s + e_i \bmod q$

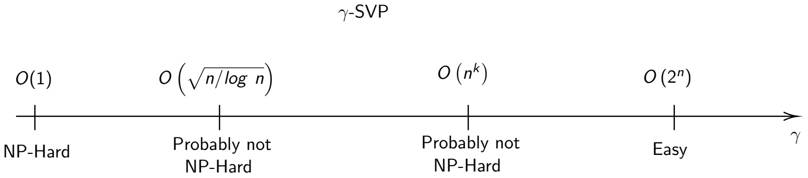
The RLWE hypothesis says that (a_i, b_i) is indistinguishable from uniform pairs of R_q^2

Hardness of (R)LWE

Theory

Worst-case to average-case reductions:

Solving (R)LWE with parameters n, Q allows us to solve γ-SVP, where $\gamma = \tilde{O}(Q/n)$.



Hardness of (R)LWE

Practice

- Pick parameters such that best attack takes exponential time
- Lattice estimator is used[1]
- Increasing n increases security
- Increasing q reduces security

[1] `https://github.com/malb/lattice-estimator`

Table of Contents

Using RLWE to encrypt

- The secret polynomial $s \in R$ is used as the secret key.
- We choose a plaintext modulus $t \in \mathbb{N}$
- RLWE samples (a_i, b_i) with $b_i := a_i \cdot s + t \cdot e_i \bmod q$ also look uniform

Using RLWE to encrypt

- The secret polynomial $s \in R$ is used as the secret key.
- We choose a plaintext modulus $t \in \mathbb{N}$
- RLWE samples (a_i, b_i) with $b_i := a_i \cdot s + t \cdot e_i \bmod q$ also look uniform
- If (a_i, b_i) is uniform, then $(a_i, b_i + m) \bmod q$ is also so
- In other words, $(a_i, b_i + m)$ hides the message m

$$\mathrm{Enc}_{\mathsf{sk}} : m \in R_t \mapsto (a_i, b_i := a_i \cdot s + t \cdot e_i + m) \in R_q^2$$

How to decrypt

Given $(a_i, b_i := a_i \cdot s + t \cdot e_i + m) \in R_q^2$ and the secret key s, compute

$$e^\star = b_i - a_i \cdot s \bmod q$$

then,

$$e^\star = t \cdot e_i + m \bmod q$$

How to decrypt

Given $(a_i, b_i := a_i \cdot s + t \cdot e_i + m) \in R_q^2$ and the secret key s, compute

$$e^\star = b_i - a_i \cdot s \bmod q$$

then,

$$e^\star = t \cdot e_i + m \bmod q$$

If $\|t \cdot e_i + m\|_\infty < q/2$, then

$$e^\star = t \cdot e_i + m$$

Output $e^\star \bmod t$

How to decrypt

Given $(a_i, b_i := a_i \cdot s + t \cdot e_i + m) \in R_q^2$ and the secret key s, compute

$$e^\star = b_i - a_i \cdot s \bmod q$$

then,

$$e^\star = t \cdot e_i + m \bmod q$$

If $\|t \cdot e_i + m\|_\infty < q/2$, then

This implies $\|e_i\|_\infty < \frac{q}{2t}$

$$e^\star = t \cdot e_i + m$$

Output $e^\star \bmod t$

Representing ciphertext as polynomial (of polynomials)

We can see a ciphertext as a polynomial $c(Y) \in R_q[Y]$:

$$\mathrm{Enc}_{\mathrm{sk}} : m \in R_t \mapsto c(Y) = c_0 + c_1 Y \in R_q[Y]$$

where

$$c_0 = a \cdot s + t \cdot e + m \in R_q$$

and

$$c_1 = -a \in R_q$$

Then

$$c(s) = t \cdot e + m$$

Noise growth

Now it is easy to see that homomorphic operations increase the noise:

$$d(Y) = c(Y) + \bar{c}(Y) \in R_q[Y]$$

Then,

$$\begin{aligned} d(s) &= c(s) + \bar{c}(s) \\ &= t \cdot e + m + t \cdot \bar{e} + \bar{m} \\ &= t \cdot (e + \bar{e}) + m + \bar{m} \end{aligned}$$

Thus, $d(Y)$ is an encryption of the sum, but with about twice the noise.

Homomorphic multiplication

Let $c(Y), \bar{c}(Y) \in R_q[Y]$
In principle, both have degree 1 on Y.

Multiplying them

$$d(Y) := c(Y) \cdot \bar{c}(Y) = d_0 + d_1 Y + d_2 Y^2 \in R_q[Y]$$

We can see that

$$\begin{aligned} d(s) &= c(s) \cdot \bar{c}(s) \\ &= (t \cdot e + m) \cdot (t \cdot \bar{e} + \bar{m}) \\ &= t \cdot (e t \bar{e} + e \bar{m} + \bar{e} m) + m \bar{m} \end{aligned}$$

Two problems:

- Noise growth $B \mapsto B^2$
- Ciphertext size is growing

Toy example of homomorphic evaluation

We want to compute the function $f(x, y) = (x + y)^4 \bmod t$
Start with $c(Y)$ and $\bar{c}(Y)$ with noise bounded by σ

1. Hom. Add: $d(Y) = c(Y) + \bar{c}(Y)$
2. Hom. Mul: $u(Y) = d(Y) \cdot d(Y) \in R_q[Y]$
3. Hom. Mul: $v(Y) = u(Y) \cdot u(Y) \in R_q[Y]$

Toy example of homomorphic evaluation

We want to compute the function $f(x,y) = (x+y)^4 \bmod t$
Start with $c(Y)$ and $\bar{c}(Y)$ with noise bounded by σ

1. Hom. Add: $d(Y) = c(Y) + \bar{c}(Y)$
2. Hom. Mul: $u(Y) = d(Y) \cdot d(Y) \in R_q[Y]$
3. Hom. Mul: $v(Y) = u(Y) \cdot u(Y) \in R_q[Y]$

Degree

1. 1
2. 2
3. 4

Toy example of homomorphic evaluation

We want to compute the function $f(x, y) = (x + y)^4 \bmod t$
Start with $c(Y)$ and $\bar{c}(Y)$ with noise bounded by σ

1. Hom. Add: $d(Y) = c(Y) + \bar{c}(Y)$
2. Hom. Mul: $u(Y) = d(Y) \cdot d(Y) \in R_q[Y]$
3. Hom. Mul: $v(Y) = u(Y) \cdot u(Y) \in R_q[Y]$

Noise growth

1. 2σ
2. $(2\sigma)^2$
3. $(4\sigma^2)^2 = 16\sigma^4$

Remember that we need final noise $< q/(2t)$, thus,

$$q \approx 32 \cdot t \cdot \sigma^4$$

Toy example of homomorphic evaluation

With $\sigma = 3.5$ and $t = 2^8$, we have

$$q \approx 32 \cdot t \cdot \sigma^4 = 1229312 \approx 2^{21}$$

Degree N of the cyclotomic polynomial is a free variable for now...
Then we choose a security level, e.g., $\lambda = 128$.
We plug (λ, σ, q) into the Lattice estimator and obtain $N = 1024$.

Our cyclotomic ring is

$$R_q = \mathbb{Z}_{2^{24}}[X]/\langle X^{1024} + 1\rangle$$

Problems with our homomorphic multiplication

We have a scheme homomorphic for additions and multiplications, but

- noise grows exponentially
- ciphertext size grows exponentially

Let's see how to solve the first problem...

Modulus switching

We saw that if $\|e_i\| \approx B$, then mult. produces $\|e_{mult}\| \approx B^2$.
The main idea is to somehow divide the ciphertexts by B, dividing also the noise.
At the end, we should have

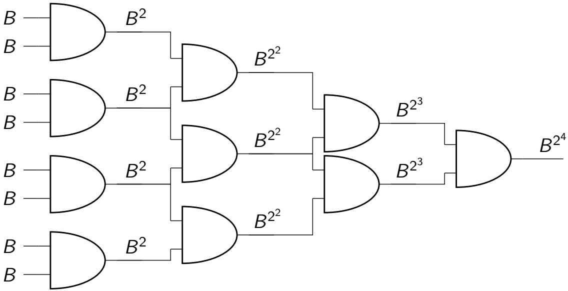
$$\left\|e'_{mult}\right\| \approx \|e_{mult}\| / B \approx B$$

but the modulus is also reduced, from Q to $\lfloor Q/B \rceil$

Modulus switching

What is the advantage of doing that?

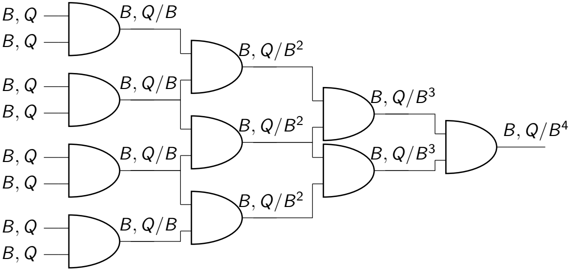
Consider the following circuit with multiplication gates



L levels $\Rightarrow$ final noise B^{2^L}

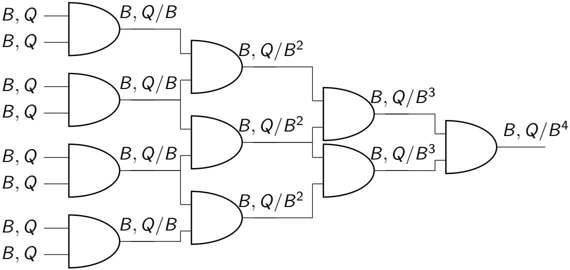
We need $Q > B^{2^L}$, thus $\log Q > \log(B^{2^L}) = 2^L \cdot \log(B)$ so, exponential in L

Now consider that we modswitch



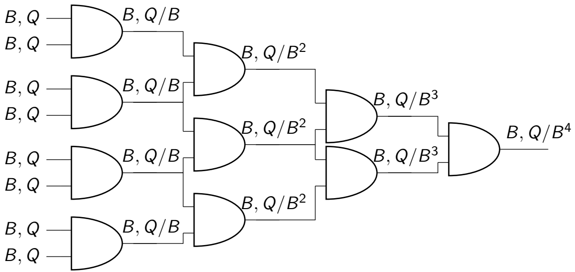
- L levels $\Rightarrow$ final noise B and final modulus Q/B^L

Now consider that we modswitch



- L levels $\Rightarrow$ final noise B and final modulus Q/B^L
- We need $Q/B^L > B$, thus $Q > B^{L+1}$

Now consider that we modswitch



- L levels $\Rightarrow$ final noise B and final modulus Q/B^L
- We need $Q/B^L > B$, thus $Q > B^{L+1}$
- Therefore $\log Q > \log(B^{L+1}) = (L+1) \cdot \log(B)$ so, linear in L

Modulus switching: how is it really done?

Consider ciphertexts of the form $(a, b) \in R_Q^2$ with

$$b = -a \cdot s + e + \Delta \cdot m$$

where $\Delta = \lfloor Q/t \rceil$

Modulus switching: how is it really done?

Consider ciphertexts of the form $(a, b) \in R_Q^2$ with

$$b = -a \cdot s + e + \Delta \cdot m$$

where $\Delta = \lfloor Q/t \rceil$

There are easy transformations between

$$b = -a \cdot s + t \cdot e + m \longleftrightarrow b = -a \cdot s + e + \Delta \cdot m$$

Modulus switching: how is it really done?

Consider ciphertexts of the form $(a, b) \in R_Q^2$ with

$$b = -a \cdot s + e + \Delta \cdot m$$

where $\Delta = \lfloor Q/t \rceil$

There are easy transformations between

$$b = -a \cdot s + t \cdot e + m \longleftrightarrow b = -a \cdot s + e + \Delta \cdot m$$

Let $Q = B^{L+1}$
Just define

$$\mathrm{ModSwt}(a, b) = (a', b') \in R_{Q'}^2$$

where

$$a' = \lfloor a/B \rceil \quad \text{and} \quad b' = \lfloor b/B \rceil$$

Modulus switching: how is it really done?

To see that $\mathrm{ModSwt}(a, b) = (a', b') = (\lfloor a/B \rceil, \lfloor b/B \rceil)$ is valid ciphertext, we want to check that

$$b' = -a' \cdot s + e' + (Q'/t) \cdot m$$

Modulus switching: how is it really done?

To see that $\mathrm{ModSwt}(a, b) = (a', b') = (\lfloor a/B \rceil, \lfloor b/B \rceil)$ is valid ciphertext, we want to check that

$$b' = -a' \cdot s + e' + (Q'/t) \cdot m$$

For any polynomial $u \in \mathbb{R}[X]$, we have

$$\lfloor u \rceil = u + \epsilon$$

where $\epsilon \in \mathbb{R}[X]$ and $\|\epsilon\| \leq 1/2$
Therefore, defining $Q' = Q/B$, we have

$$\begin{aligned} b' &= b/B + \epsilon \\ &= (-a \cdot s + e + (Q/t) \cdot m)/B + \epsilon \\ &= -(a/B) \cdot s + e/B + \epsilon + (Q'/t) \cdot m \end{aligned}$$

Modulus switching: how is it really done?

To see that $\mathrm{ModSwt}(a, b) = (a', b') = (\lfloor a/B \rceil, \lfloor b/B \rceil)$ is valid ciphertext, we want to check that

$$b' = -a' \cdot s + e' + (Q'/t) \cdot m$$

For any polynomial $u \in \mathbb{R}[X]$, we have

$$\lfloor u \rceil = u + \epsilon$$

where $\epsilon \in \mathbb{R}[X]$ and $\|\epsilon\| \leq 1/2$
Therefore, defining $Q' = Q/B$, we have

$$\begin{aligned} b' &= b/B + \epsilon \\ &= (-a \cdot s + e + (Q/t) \cdot m)/B + \epsilon \\ &= -(a/B) \cdot s + e/B + \epsilon + (Q'/t) \cdot m \end{aligned}$$

By writing $a' := \lfloor a/B \rceil = a/B + \epsilon'$, we have

$$b' = -a' \cdot s + \underbrace{e/B + \epsilon' \cdot s + \epsilon}_{\text{new noise } e'} + (Q'/t) \cdot m$$

Modulus switching: how is it really done?

Therefore, considering that $B|Q$,
$\mathrm{ModSwt}(a, b) = (a', b') := (\lfloor a/B \rceil, \lfloor b/B \rceil)$ outputs a valid ciphertext modulo $Q' = Q/B$ and with noise

$$\begin{aligned} \|e'\| &= \|e/B + \epsilon' \cdot s + \epsilon\| \\ &\leq \|e/B\| + \|\epsilon' \cdot s\| + \|\epsilon\| \\ &\leq \|e/B\| + \|s\| \cdot N/2 + 1/2 \end{aligned}$$

Modulus switching: how is it really done?

Therefore, considering that $B|Q$,
$\text{ModSwt}(a, b) = (a', b') := (\lfloor a/B \rceil, \lfloor b/B \rceil)$ outputs a valid ciphertext modulo $Q' = Q/B$ and with noise

$$\begin{aligned}\|e'\| &= \|e/B + \epsilon' \cdot s + \epsilon\| \\ &\leq \|e/B\| + \|\epsilon' \cdot s\| + \|\epsilon\| \\ &\leq \|e/B\| + \|s\| \cdot N/2 + 1/2\end{aligned}$$

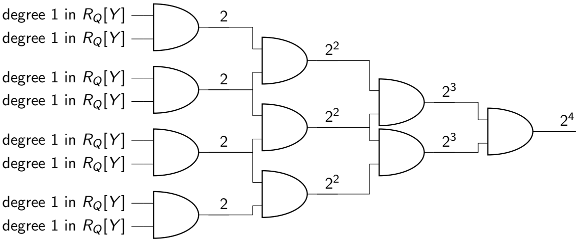
By using low-norm secret key s, we finally obtain

$$\|e'\| \approx \|e/B\|$$

as desired.

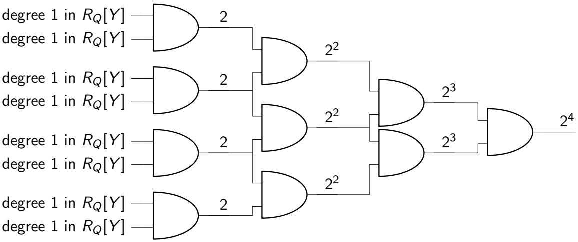
Remaining problem: ciphertext is not compact

We solved the issue about the exponential noise growth.
But we still have a problem with the size of the ciphertext, which grows when we multiply...



Remaining problem: ciphertext is not compact

We solved the issue about the exponential noise growth.
But we still have a problem with the size of the ciphertext, which grows when we multiply...



- L levels $\Rightarrow$ degree 2^L in Y
- Ciphertexts exponentially large: $(2^L + 1) \cdot N \cdot \log Q$ bits

Making ciphertexts compact

Main idea: somehow transform degree-two ctxt after mult into degree-one again

Making ciphertexts compact

Main idea: somehow transform degree-two ctxt after mult into degree-one again

Remember, after hom. mult we obtain

$$c(Y) = c_0 + c_1 \cdot Y + c_2 \cdot Y^2 \in R_Q[Y]$$

such that

$$c(s) = c_0 + c_1 \cdot s + c_2 \cdot s^2 = t \cdot e + m$$

If we could construct $c'(Y) = c'_0 + c'_1 \cdot Y$ as

$$c'_0 = c_0 + c_2 \cdot s^2 \quad \text{and} \quad c'_1 = c_1$$

then we would have

$$c'(s) = c(s)$$

that is, $c'(Y)$ would be a valid encryption of m, but with degree one, as desired.

Making ciphertexts compact

But we cannot publish s^2...

Making ciphertexts compact

But we cannot publish s^2...

First idea: publish an encryption of s^2: $\mathrm{rlk}(Y) \in R_Q[Y]$ such that $\mathrm{rlk}(s) = t \cdot \tilde{e} + s^2$

Making ciphertexts compact

But we cannot publish s^2...

First idea: publish an encryption of s^2: $\text{rlk}(Y) \in R_Q[Y]$
such that $\text{rlk}(s) = t \cdot \tilde{e} + s^2$

We have to assume circular security...

Making ciphertexts compact

But we cannot publish s^2...

First idea: publish an encryption of s^2: $\mathrm{rlk}(Y) \in R_Q[Y]$
such that $\mathrm{rlk}(s) = t \cdot \tilde{e} + s^2$

Now, given $c(Y) = c_0 + c_1 \cdot Y + c_2 \cdot Y^2 \in R_Q[Y]$, we can take compute

$$c'(Y) = c_2 \cdot \mathrm{rlk}(Y) \in R_Q[Y]$$

This should be an encryption of $c_2 \cdot s^2$...
Finally, compute

$$c_{mult}(Y) := c_0 + c_1 \cdot Y + c'(Y)$$

Now, we can see that

$$\begin{aligned} c_{mult}(s) &= c_0 + c_1 \cdot s + c'(s) \\ &= c_0 + c_1 \cdot s + c_2 \cdot s^2 + te' \\ &= t \cdot e + t \cdot e' + m \end{aligned}$$

Making ciphertexts compact

But we cannot publish s^2...

First idea: publish an encryption of s^2: $\mathrm{rlk}(Y) \in R_Q[Y]$
such that $\mathrm{rlk}(s) = t \cdot \tilde{e} + s^2$

Now, given $c(Y) = c_0 + c_1 \cdot Y + c_2 \cdot Y^2 \in R_Q[Y]$, we can take compute

$$c'(Y) = c_2 \cdot \mathrm{rlk}(Y) \in R_Q[Y]$$

This should be an encryption of $c_2 \cdot s^2$...
Finally, compute

$$c_{mult}(Y) := c_0 + c_1 \cdot Y + c'(Y)$$

Now, we can see that

However, $\|e'\| = \|c_2 \cdot \tilde{e}\| \approx Q$

$$\begin{aligned} c_{mult}(s) &= c_0 + c_1 \cdot s + c'(s) \\ &= c_0 + c_1 \cdot s + c_2 \cdot s^2 + te' \\ &= t \cdot e + t \cdot e' + m \end{aligned}$$

Making ciphertexts compact

OK... This idea looks promising...
So far, we have

- $c(Y) = c_0 + c_1 \cdot Y + c_2 \cdot Y^2 \in R_Q[Y]$ encrypting m
- a relinearization key $\mathrm{rlk}(Y)$ encrypting s^2

So far, we can

- multiply $\mathrm{rlk}(Y)$ by c_2
- obtain $c_{mult}(Y)$ of degree one encrypting m
- but noise of $c_2 \cdot \mathrm{rlk}(Y)$ is too big (basically Q)

Making ciphertexts compact

OK... This idea looks promising...
So far, we have

- $c(Y) = c_0 + c_1 \cdot Y + c_2 \cdot Y^2 \in R_Q[Y]$ encrypting m
- a relinearization key $\mathrm{rlk}(Y)$ encrypting s^2

So far, we can

- multiply $\mathrm{rlk}(Y)$ by c_2
- obtain $c_{mult}(Y)$ of degree one encrypting m
- but noise of $c_2 \cdot \mathrm{rlk}(Y)$ is too big (basically Q)

So, we need a way to multiply c_2 by $\mathrm{rlk}(Y)$ without increasing the noise of rlk that much...

Decomposing before mult to reduce noise

To avoid such noise growth, instead of multiplying by c_2 directly, we first decompose c_2 in some base (e.g., binary decomposition), then multiply by the digits we obtain...

Decomposing before mult to reduce noise

To avoid such noise growth, instead of multiplying by c_2 directly, we first decompose c_2 in some base (e.g., binary decomposition), then multiply by the digits we obtain...

- Fix a decomposition base B
- Let $\ell = \lceil \log_B(Q) \rceil$
- Define the "gadget vector" $\vec{g} = (B^0, B^1, ..., B^{\ell-1})$
- Decomp: $\forall\, a \in \mathbb{Z}_Q$, outputs

$$\vec{a} := (a_0, a_1, ..., a_{\ell-1}) \in \{0, ..., B-1\}^{\ell}$$

  such that

$$\vec{a} \cdot \vec{g} = \sum_{i=0}^{\ell-1} a_i B^i = a$$

Decomposing before mult to reduce noise

We can extend it to polynomials by decomposing each coefficient

$$\begin{aligned} \text{Decomp}\colon R_Q &\to R_B^{\ell} \\ a &\mapsto \vec{a} := (a_0, ..., a_{\ell-1}) : \vec{a} \cdot \vec{g} = a \end{aligned}$$

Decomposing before mult to reduce noise

We can extend it to polynomials by decomposing each coefficient

$$\begin{aligned} \mathrm{Decomp}\colon R_Q &\to R_B^{\ell} \\ a &\mapsto \vec{a} := (a_0, \ldots, a_{\ell-1}) : \vec{a} \cdot \vec{g} = a \end{aligned}$$

Notice, we can use this Decomp to multiply by a polynomial mod Q without increasing the noise up to Q...
If c encrypts m with noise $e \in R$, then $a_i \cdot c$ encrypts $a_i \cdot m$ with noise

$$\|a_i \cdot e\| \leq N \|a_i\| \cdot \|e\| \leq N \cdot B \cdot \|e\|$$

Thus, mult by multiplies the noise by $N \cdot B$ instead of Q.

Decomposing before mult to reduce noise

- Encrypt μ with the powers of the decomposition base B
- i.e., $\vec{c} := (c_0(Y), ..., c_{\ell-1}(Y))$ where $c_i(Y)$ encrypts $B^i \cdot \mu$
- Now, given $a \in R_Q$, decompose it: $\vec{a} := \mathrm{Decomp}(a)$
- Compute

$$c(Y) = \vec{a} \cdot \vec{c} = \sum_{i=0}^{\ell-1} a_i \cdot c_i(Y)$$

- Each $a_i \cdot c_i(Y)$ encrypts $\mu \cdot a_i B^i$, so $c(Y)$ encrypts

$$\mu \cdot \sum_{i=0}^{\ell-1} a_i B^i = \mu \cdot a$$

- If noise $c_i(Y) \leq V$, then noise of $c(Y)$ is $\approx \ell \cdot N \cdot B \cdot V$

Thus, we can define the relinearization key as

$$\vec{\mathsf{rlk}} = (\mathsf{rlk}_0(Y), ..., \mathsf{rlk}_{\ell-1}(Y))$$

where $\mathsf{rlk}_i(Y)$ encrypts $B^i \cdot s^2$

Thus, we can define the relinearization key as

$$\vec{\mathrm{rlk}} = (\mathrm{rlk}_0(Y), ..., \mathrm{rlk}_{\ell-1}(Y))$$

where $\mathrm{rlk}_i(Y)$ encrypts $B^i \cdot s^2$

Given $c(Y) = c_0 + c_1 \cdot Y + c_2 \cdot Y^2 \in R_Q[Y]$ encrypting m

- $\vec{u} := \mathrm{Decomp}(c_2)$
- $c'(Y) := \vec{u} \cdot \vec{\mathrm{rlk}}$ (enc of $c_2 \cdot s^2$ with small noise)
- Define

$$c_{mult}(Y) := c_0 + c_1 \cdot Y + c'(Y) \in R_Q[Y]$$

Thus, we can define the relinearization key as

$$\vec{\mathrm{rlk}} = (\mathrm{rlk}_0(Y), ..., \mathrm{rlk}_{\ell-1}(Y))$$

where $\mathrm{rlk}_i(Y)$ encrypts $B^i \cdot s^2$

Given $c(Y) = c_0 + c_1 \cdot Y + c_2 \cdot Y^2 \in R_Q[Y]$ encrypting m

- $\vec{u} := \mathrm{Decomp}(c_2)$
- $c'(Y) := \vec{u} \cdot \vec{\mathrm{rlk}}$ (enc of $c_2 \cdot s^2$ with small noise)
- Define

$$c_{mult}(Y) := c_0 + c_1 \cdot Y + c'(Y) \in R_Q[Y]$$

As discussed before,

$$c_{mult}(s) = c_0 + c_1 \cdot s + c'(s) = te + te' + m$$

but now, $\|e'\| \leq \ell NBV$ instead of $\|e'\| \approx Q$

Homomorphic multiplication

During key generation: produce relinearization key

$$\vec{\text{rlk}} = (\text{rlk}_0(Y), ..., \text{rlk}_{\ell-1}(Y))$$

where $\text{rlk}_i(Y)$ encrypts $B^i \cdot s^2$

Then, for every homomorphic multiplication we have two steps:
Multiplication itself:

$$\begin{array}{ccc} enc(m_0) & & \\ & \searrow & \\ & & \otimes \longrightarrow \begin{array}{c} (a', a, b) \in R_q^3 \\ \text{where } b := a' \cdot s^2 + a \cdot s + t \cdot e + m_0 \cdot m_1 \end{array} \\ & \nearrow & \\ enc(m_1) & & \end{array}$$

Homomorphic multiplication

During key generation: produce relinearization key

$$\vec{\mathsf{rlk}} = (\mathsf{rlk}_0(Y), ..., \mathsf{rlk}_{\ell-1}(Y))$$

where $\mathsf{rlk}_i(Y)$ encrypts $B^i \cdot s^2$

Then, for every homomorphic multiplication we have two steps:
Multiplication itself:

$$\begin{tikzcd} enc(m_0) \arrow[dr] & & \\ & \otimes \arrow[r] & (a', a, b) \in R_q^3 \\ enc(m_1) \arrow[ur] & & \end{tikzcd}$$

where $b := a' \cdot s^2 + a \cdot s + t \cdot e + m_0 \cdot m_1$

Relinearization (Key switching):

- uses $\vec{\mathsf{rlk}}$
- maps $(a', a, b) \in R_q^3$ back to a two-component ciphertext $(\bar{a}, \bar{b}) \in R_q^2$ encrypting $m_0 \cdot m_1$

Homomorphic multiplication

During key generation: produce relinearization key

$$\vec{\mathsf{rlk}} = (\mathsf{rlk}_0(Y), ..., \mathsf{rlk}_{\ell-1}(Y))$$

where $\mathsf{rlk}_i(Y)$ encrypts $B^i \cdot s^2$

Then, for every homomorphic multiplication we have two steps:
Multiplication itself:

$$\begin{array}{l} enc(m_0) \searrow \\ \qquad\qquad \otimes \longrightarrow \; (a', a, b) \in R_q^3 \\ enc(m_1) \nearrow \end{array}$$

where $b := a' \cdot s^2 + a \cdot s + t \cdot e + m_0 \cdot m_1$

Relinearization (Key switching):

- uses $\vec{\mathsf{rlk}}$
- maps $(a', a, b) \in R_q^3$ back to a two-component ciphertext $(\bar{a}, \bar{b}) \in R_q^2$ encrypting $m_0 \cdot m_1$

Usually, we also perform modulus switching

Recapitulation

Now we know how to construct a homomorphic scheme whose

- ciphertexts are compact (relinearization)
- noise grows slowly (modulus switching)

This is the base of schemes like BGV, CKKS, FV...

Recapitulation

Now we know how to construct a homomorphic scheme whose

- ciphertexts are compact (relinearization)
- noise grows slowly (modulus switching)

This is the base of schemes like BGV, CKKS, FV...

But we are encrypting polynomials...
Applications usually work with integers...
So, the final optimization: batching, aka SIMD, aka plaintext slots

Plaintext slots

Plaintext space of RLWE-based schemes: $R_t = \mathbb{Z}_t[X]/\langle X^N + 1\rangle$

But most applications do not use polynomials as the data type...

We can use the decomposition of $X^N + 1$ modulo t to represent the plaintext space in a more application-friendly way

Plaintext slots

For example,

$$X^4 + 1 = (X+2)(X+8)(X+9)(X+15) \bmod 17$$

Thus,

$$R_{17} = \frac{\mathbb{Z}_t[X]}{\langle X+2 \rangle} \times \frac{\mathbb{Z}_t[X]}{\langle X+8 \rangle} \times \frac{\mathbb{Z}_t[X]}{\langle X+9 \rangle} \times \frac{\mathbb{Z}_t[X]}{\langle X+15 \rangle} = \mathbb{Z}_t^4$$

So, instead of encrypting one "big" polynomial, we can encrypt 4 degree-0 polynomials (i.e., elements of $\mathbb{Z}_t$)

Plaintext slots

In general, $X^N + 1$ factors into S lower degree polynomials mod t

$$X^N + 1 = \prod_{i=1}^{S} f_i(X) \mod t$$

and we have S slots, i.e., we can encrypt a vector $(v_1, ..., v_S) \in \mathbb{Z}_t^S$
Then, homomorphic operations are applied to the slots in parallel:

Let $\vec{c}_u = \text{Enc}(u_1, ..., u_S)$ and $\vec{c}_v = \text{Enc}(v_1, ..., v_S)$

- $\text{HE.Add}(\vec{c}_u, \vec{c}_v) = \text{Enc}(u_1 + v_1, ..., u_S + v_S)$
- $\text{HE.Mult}(\vec{c}_u, \vec{c}_v, \text{rlk}) = \text{Enc}(u_1 \cdot v_1, ..., u_S \cdot v_S)$

Plaintext slots

In summary, with S slots, we can process S messages in parallel.

We call it SIMD (single instruction multiple data).

Plaintext slots

In summary, with S slots, we can process S messages in parallel.

We call it SIMD (single instruction multiple data).

Evaluating f homomorphically one single time yields

$$\mathrm{Enc}(f(u_1), ..., f(u_S))$$

Hence, the amortized running time is divided by S

Plaintext slots: rotations

On some applications, we need to combine values in different slots.

Plaintext slots: rotations

On some applications, we need to combine values in different slots.

FHE schemes typically also offer **slot rotation**:

Given an integer k and a key-switching key swk_k

$$\mathsf{HE.Rot}(\mathsf{Enc}(u_1, ..., u_S), k, \mathsf{swk}_k)$$

applies a shift rotation to $(u_1, ..., u_S)$ by k positions

Plaintext slots: rotations

On some applications, we need to combine values in different slots.

FHE schemes typically also offer **slot rotation**:

Given an integer k and a key-switching key swk_k

$$\mathsf{HE.Rot}(\mathsf{Enc}(u_1, ..., u_S), k, \mathsf{swk}_k)$$

applies a shift rotation to $(u_1, ..., u_S)$ by k positions

For example:

$$\mathsf{HE.Rot}(\mathsf{Enc}(u_1, u_2, ..., u_S), 1, \mathsf{swk}_1) = \mathsf{Enc}(u_2, u_3, ..., u_S, u_1)$$

$$\mathsf{HE.Rot}(\mathsf{Enc}(u_1, u_2, ..., u_S), 2, \mathsf{swk}_2) = \mathsf{Enc}(u_3, u_4, ..., u_1, u_2)$$

Plaintext slots: rotations

We have to plan ahead the rotations we want to execute

During the setup, we generate one (public) key-switching key swk_k for each k-wise rotation we need

Cost of homomorphic rotation:

- Run time: approximately same as HE.Mult
- Memory: each swk_k typically has around 30MB
- Noise: much less than HE.Mult

Example: computing inner product

- We want to compute $\vec{u} \cdot \vec{v} = \sum_{i=1}^{4} u_i \cdot v_i$
- Set at least 4 slots
- Start with ciphertexts $\mathrm{Enc}(u_1, \ldots, u_4)$ and $\mathrm{Enc}(v_1, \ldots, v_4)$
- Then HE.Mult gives us $\mathrm{Enc}(w_1, \ldots, w_4)$ where $w_i = u_i \cdot v_i$
- Rotate by 1 to get $\mathrm{Enc}(w_2, w_3, w_4, w_1)$
- Then HE.Add: $\mathrm{Enc}(w_1 + w_2, w_2 + w_3, w_3 + w_4, w_4 + w_1)$
- Rotate by 2: $\mathrm{Enc}(w_3 + w_4, w_4 + w_1, w_1 + w_2, w_2 + w_3)$
- Finally HE.Add: $\mathrm{Enc}(\vec{u} \cdot \vec{v}\ ,\ \vec{u} \cdot \vec{v}\ ,\ \vec{u} \cdot \vec{v}\ ,\ \vec{u} \cdot \vec{v})$
- This costs 1 HE.Mult and 2 HE.Rot.

Table of Contents

General approach to use FHE

- Identify the functions you want to compute
- Set parameters large enough to support those functions (or to support bootstrapping)
  - More complicated functions imply more noise, which implies larger parameters
  - Most libraries already have predefined sets of parameters
- Generate secret, public, relinearization and key-switching keys
- Send the server the encrypted data and the keys (except sk)
- The server will evaluate the functions using the available operations (e.g., HE.Mult, HE.Rot and bootstrapping)

Main schemes

Scheme	Data type	Slots	Bootstrapping	Key material
BGV/FV	$\mathbb{Z}_t^S$ for large t	Yes	Expensive	GB
CKKS	$\mathbb{R}^S$	Yes	Expensive	GB
TFHE/concrete	$\mathbb{Z}_t$ for small t	No	Cheap	MB
FINAL	$\mathbb{Z}_2$	No	Cheapest	MB

Of course, CKKS just supports "real numbers" up to some precision (say, 30 or 60 bits). Moreover, the homomorphic operations reduce the precision, so, output has much less precision than input.

Some libraries

	Schemes	User-friendly	Language
HElib	BGV, CKKS*	No	C++
OpenFHE	BGV*, FV*, CKKS, TFHE	Medium	C++
Lattigo	BGV*, FV*, CKKS	Yes	Go
SEAL	FV*, CKKS*	Yes	C++
concrete	(extended) TFHE	Yes	Rust
FINAL	FINAL	Yes	C++

Asterisk means that the scheme is implemented but without bootstrapping.

Thanks!

Any question or comment?

Please, feel free to contact!

`https://hilder-vitor.github.io`