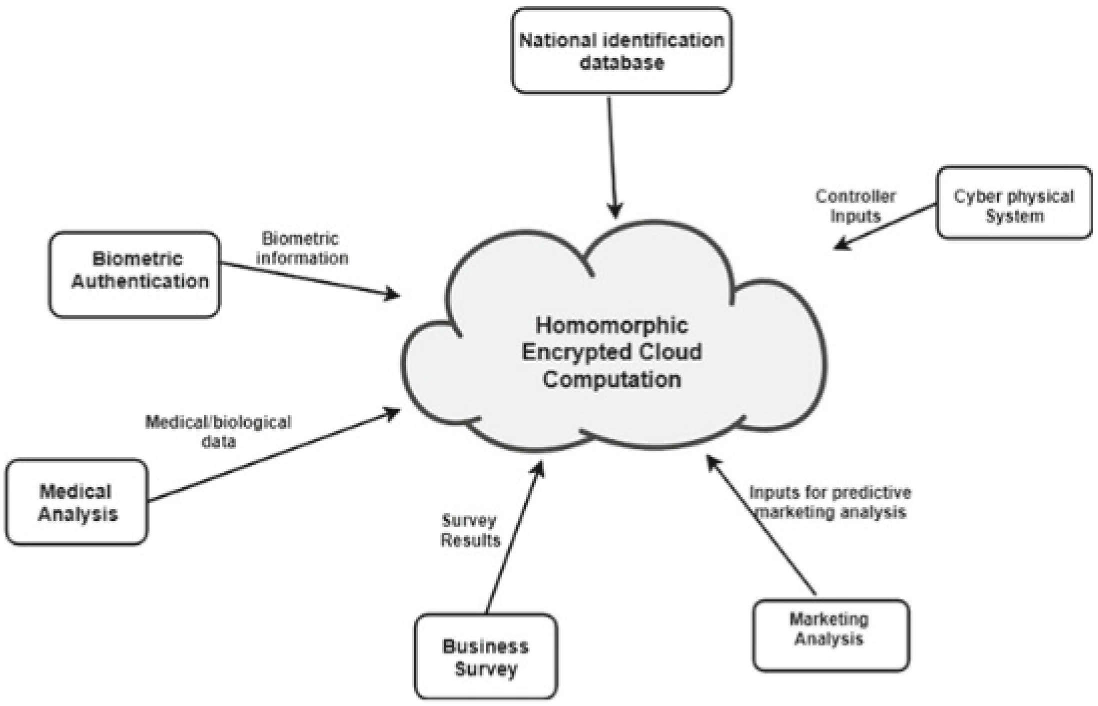
1. Homomorphic Encryption

- https://www.youtube.com/watch?v=umqz7kKWxyw - Good talk

Applications

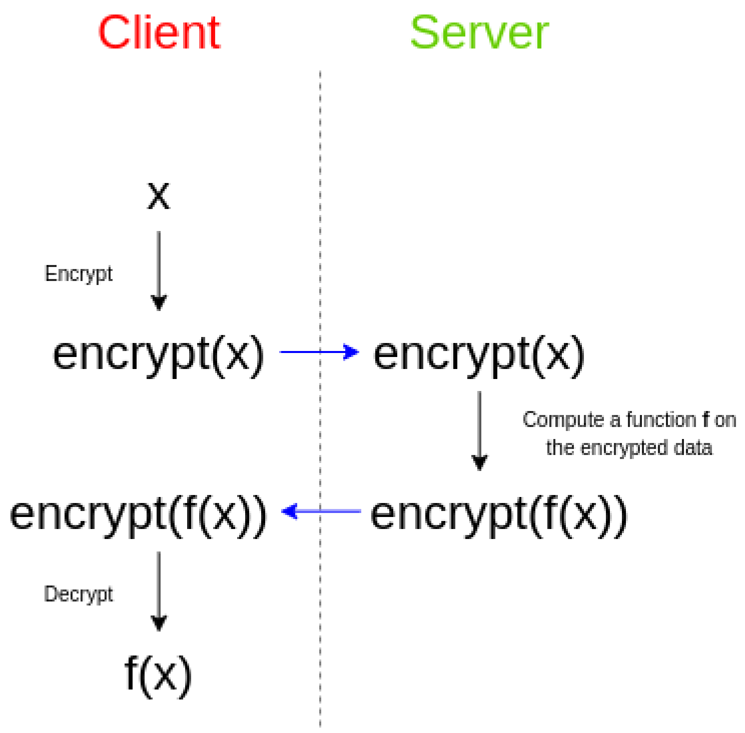
Usually homomorphic encryption is easier explained through applications. So let's go through some examples.



1. Outsourcing storage and computations

- Let A be a company that wants to store data in the cloud C. A doesn't want the cloud provider C to see the sensitive data. Therefore he encrypts the information.
- *Problem*: A wants to use the information (do computations on it, apply a function f on them) without locally retrieving it and decrypting it (defeating the purpose of storing it in the cloud).
- *Solution*: Using homomorphic encryption, the cloud provider can process the information in the encrypted form.

- Ex: finance, medical apps, consumer privacy in ads.



2. Private queries

- Let D be a database provider that holds a big database.
- *Problem* : The client A wants to retrieve a query without the database provider D learning which query it is.
- *Solution*: Homomorphic encryption lets us encrypt the index of the record. Then we can use this encrypted index to fetch and return the encrypted result to the client. Then the client can decrypt the record.

Limitations of HE

- Encrypted output - No one can make sense of it unless decrypted.
- All inputs must be encrypted under the same key.
- No integrity guarantees - You cannot check if the result was obtained by the computation being carried out or by encrypting the same value (fresh encryption).

Definitions

Intuition

- A user has a function f and some inputs $(m_1, ...m_n)$.
- He wants to compute $m^* = f(m_1, ...m_n)$ but instead of computing it directly, he wants to encrypt $(m_1, ...m_n) \to (c_1, ...c_n)$ and do the computation on the ciphertext, obtaining a result which eventually decrypts to $m^* = f(m_1, ...m_n)$.

Homomorphic encryption -- Definition

Let `(KeyGen, Encrypt, Decrypt, Evaluate)` be a tuple of procedures $(KeyGen, E, D, V)$
Let $f \in \mathcal{F}$ be a function in a family of functions. This function is also usually called a *circuit* in a family of circuits $C \in \mathcal{C}$.

- $(sk, pk) \leftarrow KeyGen(1^\lambda, 1^d)$ - key generation, λ is a security parameter, d is a functionality parameter (degree of the polynomial or depth of the circuit).
- $c_i \leftarrow E(pk, m_i)$ - Encryption of a message m_i. -- known as **fresh ciphertexts**.
- $c^* = V(pk, f, c_1, ...c_n)$ - Evaluate the function V on the ciphertexts. -- known as **evaluated ciphertexts**

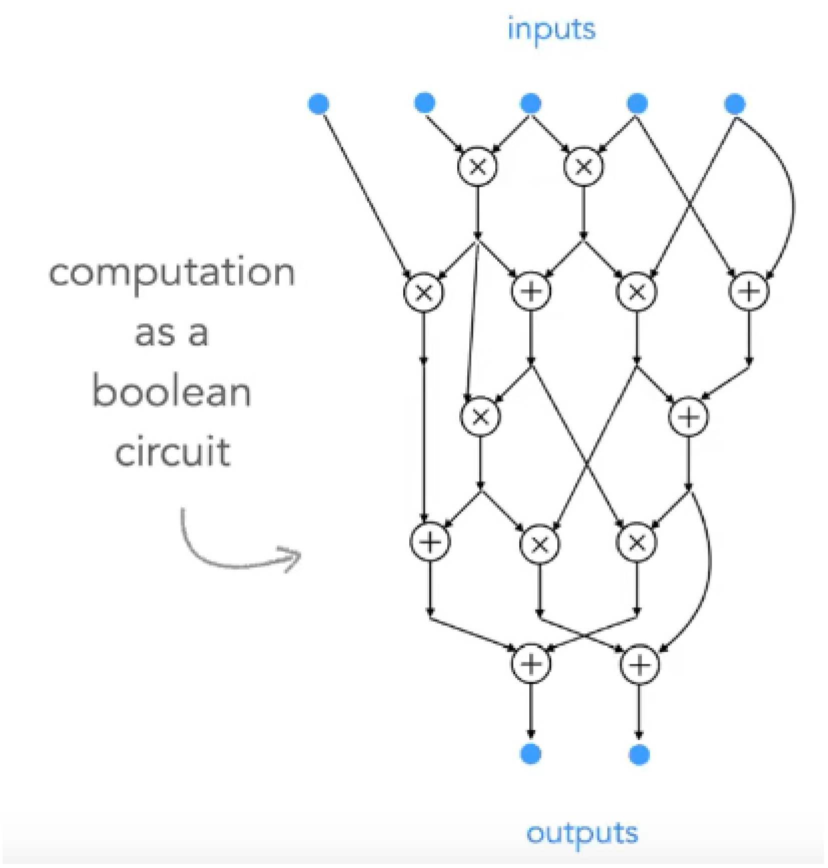
Corectness
Correctly decrypt both fresh and evaluated ciphertexts

$$D(sk, c^*) = m^* = f(m_1, ..., m_n)$$

Circuits?
The 2 important operations we care about are addition and multiplication, because with them we can construct everything. Homomorphic encryption schemes started out by encrypting bits, so the equivalent operations for additon and multiplication on bits, `xor` and `and`, are used.

$$\oplus = + \bmod 2$$
$$\otimes = \times \bmod 2$$

So, by extenting to many bits, with `xor` and `and` you can represent any computation using boolean circuits.



To convey the fact that a circuit supports both `xor` and `and` usually a `nand` gate is used, because `nand` gates are composed by 1 multiplication and 1 addition and they are logically complete (you can build any circuit with them).

$$\text{nand}(a, b) = a \times b + 1 \bmod 2$$

Properties and definitions

Security

The security is the classic semantic security definition of indistinguishability. This is given by the (E, D) algorithms. This means that the encryption is randomized and there are many encryptions of the same thing. In a deterministic setting an attacker may perform a dictionary attack against our encryption.

Strong homomorphism

All ciphertexts should have the same distribution. Evaluated ciphertexts should "look" the same as fresh ciphertexts.

- **Note**: Sometimes, even the party that generated the keys shouldn't distinguish between the ciphertext types. This is ensured by the strong homomorphism property, but not ensured by just requiring *computational indistinguishability* between fresh and evaluated ciphertexts.

Compactness - Intuitive definition

The complexity of decrypting the result c^* should be independent of the complexity of the function that we want to compute f.

Ex: we could just append the description of the function f to the ciphertext. However as f becomes more complicated the ciphertext becomes bigger and more convoluted to decrypt. This is what compactness ensures against.
However, sometimes we can allow a *slight* increase in complexity.

Circuit privacy

The ciphertext generated by V does not reveal anything about the function f that it evaluates, beyond the output value, even to the party that generated the keys.

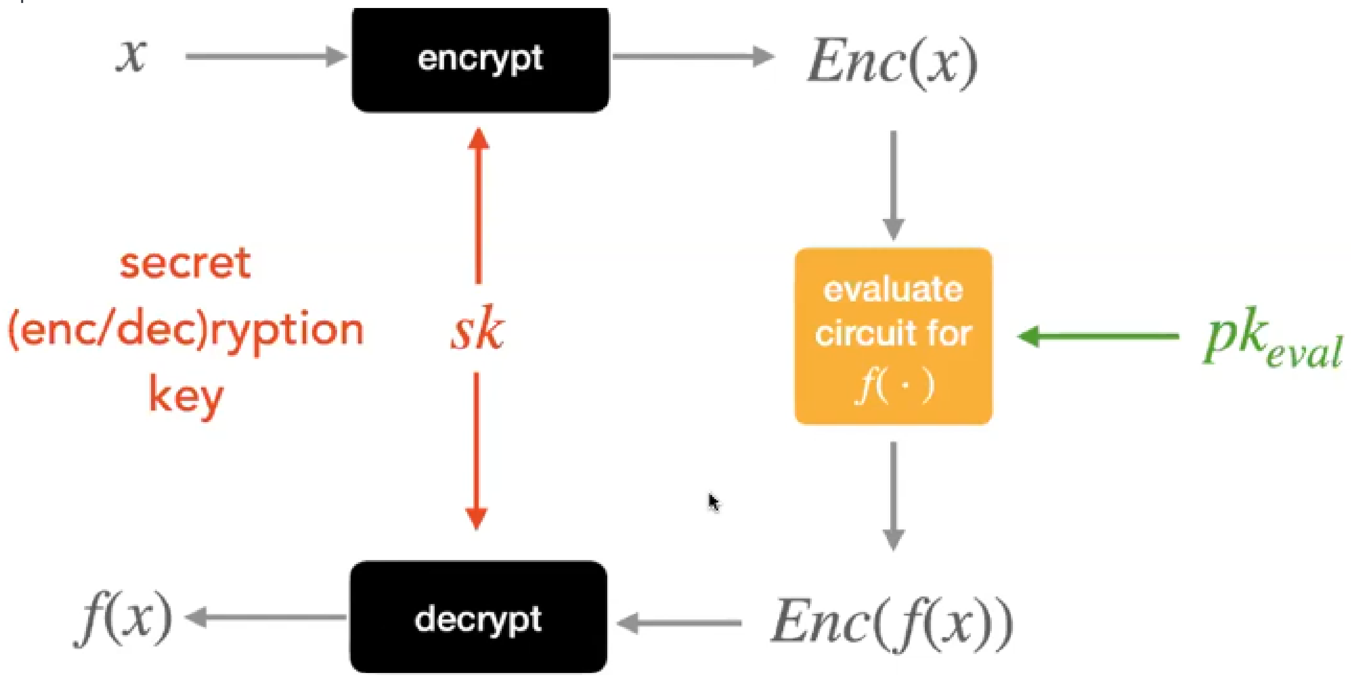
Multi hop HE

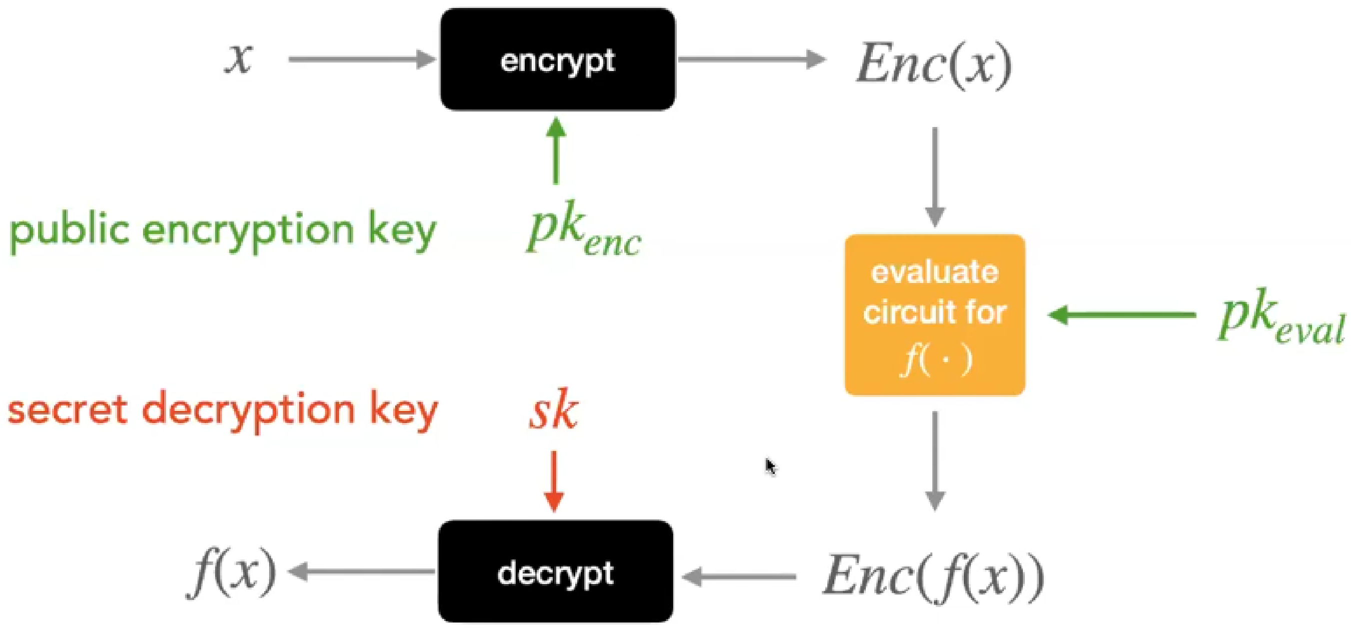
Idea: use evaluate on its own output multiple times.

From secret key to public key HE

- Any compact, multi hop secret key FHE scheme can be transformed into a compact, multi hop public key FHE

- Equivalent constructions





Classification

- https://vitalik.ca/general/2020/07/20/homomorphic.html

The HE can be classified based on the type of functions f that it supports.

Partially homomorphic encryption

Given $E(m_1)$, and $E(m_2)$ you can do limited operations. Ex: Only addition or multiplication.

Somewhat homomorphic encryption

Limited number of multiplications (Circuits of a maximum depth)

Given $E(m_1), ..., E(m_n)$ you can compute $E(f(m_1, ...m_n))$ where f is a polynomial of a limited degree.

Leveled homomorphic encryption

Considering the fact that multiplications are much more expensive than addition, leveled homomorphic encryption schemes allow circuits of a maximum depth.

Fully homomorphic encryption

Unlimited multiplications and additions.

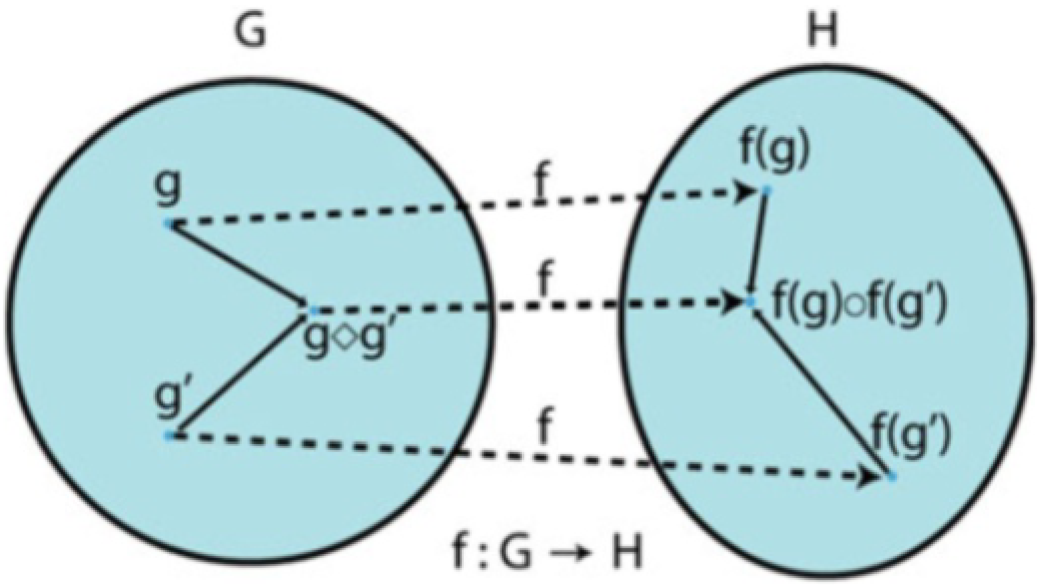
Homomorphisms

Homomorphism -- Definition

Let G, G' be groups. A Homomorphism is a map $f : G \to G'$ with the following property:

$$f(xy) = f(x)f(y)\ \forall x, y, \in G$$

- Homomorphisms preserve structure



Homomorphisms can be represented with commutative diagrams.
Example
$x \mapsto e^x$ is a homomorphism from the multiplicative to the additive group.
$f(x+y) = e^{x+y} = e^x \cdot e^y = f(x) \cdot f(y)$

$$\begin{tikzcd}
\mathbb{R}\times\mathbb{R} \arrow[r, "x+y"] \arrow[d, "{f(x),f(y)}"'] & \mathbb{R} \arrow[d, "f(x+y)"'] \\
\mathbb{R}^+\times\mathbb{R}^+ \arrow[r, "f(x)\cdot f(y)"] & \mathbb{R}^+
\end{tikzcd}$$

In the commutative diagram above you start from top left and it doesn't matter which path you take to end up bottom right.

For the following commutative diagram let $\mathcal{M}$ be the set of messages, $\mathcal{C}$ be the set of ciphertexts. It's commutative because the order you apply the decryption function and f doesn't matter.

$$\begin{tikzcd}
\mathcal{C}^n \arrow[r, "{V(pk,f,\cdot,\dots,\cdot)}"] \arrow[d, "{D(sk,\cdot,\dots,\cdot)}"'] & \mathcal{C} \arrow[d, "{D(sk,\cdot)}"'] \\
\mathcal{M}^n \arrow[r, "{f(\cdot,\dots,\cdot)}"] & \mathcal{M}
\end{tikzcd}$$

We can think of the top level as the *ciphertext level* and at the bottom level as the *plaintext level*.

Noise

Like said in the security argument above, we need our encryption to be randomized. This means that $c = E(m)$ is not quite correct, a better writing would be $c = E(m, r)$ where r is a random string of bits.

In current FHE schemes this random encryption is achieved by adding noise somewhere in the scheme. This noise hides the message.

However adding noise comes at a cost. Each homomorphic operation (add or mult) adds noise. When doing additions the noise is usually addded (not a big increase in noise). However multiplications multiply the noise, making them very expensive. Besides, as you have deeper circuits (more complicated fs) the noise will grow exponentially.
If the noise exceeds some threshold decryption is **no longer correct**.

Without using extra techniques this noise framework can only create **somewhat HE** schemes.

Bootstrapping

Bootstrapping is a way to use the FHE scheme to remove / reset the noise. It was introduced in Gentry's thesis and intuitively it can be understood as *recryption*.

Intuition:
Bootstrapping works by encrypting the ciphertext again (becoming doubly encrypted) and by *homomorphically decrypting* the first encryption. This way the first encryption's noise is gone (because the decryption removed it) and we remain with the small noise (because we only encrypted once) of the 2nd encryption.

$$\xrightarrow{E(sk,\cdot)} \boxed{x} \xrightarrow{\text{operations}} \underbrace{\boxed{y}}_{\text{noisy}} \xrightarrow{E(sk,\cdot)} \boxed{\boxed{y}} \xrightarrow[\text{homomorphic}]{D(sk,\cdot)} \underbrace{\boxed{y}}_{\text{not noisy}} \rightarrow$$

For bootstrapping to work the scheme must be able to homomorphically evaluate its **own decryption** circuit and one extra `nand` gate.

Using this technique we can transform somewhat HE schemes into FHE.
Bootstrapping was introduced by Gentry in 2009. He devloped FHE for encrypting bits evaluating the $\oplus$ and $\otimes$ (add and mul) operations.

The problem with the initial scheme was that the denoising was very slow. Gentry proposed that at least every multiplication you'd want to do a bootstrap. This means that the size of the noise is constant.
The oposing idea was to give a "big enough" noise budget and keep the scheme SHE.

Leveled HE

Soon after the 2nd generation of FHE schemes emerged: leveled HE schemes. The main gist of these schemes is that the ciphertext's noise has a level up to a maximum noise level.

So 2 ciphertexts c_0, c_1 may have 2 possible noise levels $l_0, l_1 \in \{1, 2, ...L\}$ where L is the maximum allowed noise. Then when computing a ciphertext c^*

- any addition would take the maximum of the input noise levels -- $l^* = \max(l_0, l_1)$
- any multiplication would take the maximum of the noise levels and increment it -- $l^* = \max(l_0, l_1) + 1$

When we exceeded the noise level, so when $l^* > L$, we would get an **incorrect decryption**.
When needed we can reset the noise level by bootstrapping

$$l^* \rightarrow \boxed{\text{bootstrap}} \rightarrow 1$$

Remark:

- We don't need to send the ciphertext to a level 1 noise, we can set it to whatever lower level we may want.
- We want to be smart when making circuits. For example if we have the equation $a \cdot b \cdot c \cdot d$ and we evaluate it as $(((a \cdot b) \cdot c) \cdot d)$ we would get a depth of 3. However if we evaluate it as $((a \cdot b) \cdot (c \cdot d))$ we get a depth of 2

$$\underbrace{(\underbrace{(\underbrace{(a \cdot b)}_{1} \cdot c)}_{2} \cdot d)}_{3} \quad \text{vs} \quad \underbrace{(\underbrace{(a \cdot b)}_{1} \cdot \underbrace{(c \cdot d)}_{1})}_{2}$$

Improvements

- This meant that the noise grew linearly with the circuit (not exponentially). This meant that you don't have to resort to bootstrapping as often.
- Another improvement was that there appeared schemes where you could operate on numbers $\bmod p$, not only on bits $\bmod 2$.

Modulus switching / FHE without bootstrapping

Resources

- Wikipedia entry

High level explanations

- openmined video

- blogpost
- bitdefender blog
-