Cybersecurity MASTER

UNIVERSITÉ Grenoble Alpes

Grenoble INP ensimag

Mater Cybersecurity
Master of Science in Informatics at Grenoble
Master Informatique / Master Mathématiques & Applications

Dynamic and Structured Secure Multi-party Computations

Anthony Martinez

<Defense Date>

Research project performed at Laboratoire Jean Kuntzmann

Under the supervision of:
Aude Maignan, Aude.Maignan@univ-grenoble-alpes.fr
Jean-Guillaume Dumas Jean-Guillaume.Dumas@univ-grenoble-alpes.fr

Defended before a jury composed of:
[Prof/Dr/Mrs/Mr] <first-name last-name>
[Prof/Dr/Mrs/Mr] <first-name last-name>
[Prof/Dr/Mrs/Mr] <first-name last-name>
[Prof/Dr/Mrs/Mr] <first-name last-name>

[June/September] 2022

Abstract

To do later

Contents

— 1 —
Introduction

Secure multi-party computing (SMC) is a subfield of cryptography with the goal of creating methods for parties to jointly compute a function over their inputs while keeping those inputs private.

For instance, consider two security agencies that wish to compare their lists of suspects without revealing their contents or an airline company that would like to check it's list of passengers against the list of people that are not allowed to go abroad.

The functionalities of interest thus include secure set-intersection, but also oblivious polynomial evaluation, secure equality of strings, approximation of a Taylor series, RSA key generation, oblivious keyword search, set membership, proof of data possession and more.

Initially developed in the context of cloud computing with client/server outsourcing protocols, it is nowadays largely used in more peer-to-peer setups with multiple players, or even in decentralized setups such as distributed ledger technology (blockchains).

The proposed methodology in this work is to focus on algebraic problems involving polynomial arithmetic and linear algebra. The main tools are secret sharing techniques and homomorphic cryptography and those need to be adapted to efficiently take into account modifications of the assumptions (dynamicity) during the protocols. Then the developed building blocks will be declined to give more efficient solutions to dynamic proof of retreivability systems for edge storage or decentralized storage networks, or also for private reputation systems and secure evaluation of decision forests.

First we will give an overview of the existing protocols and techniques and compare them with the protocol created by one of the LJK team, implemented in **Relic**.

We'll focus ourself on the **Zero-Knowledge Succinct Non-Interactive Argument of Knowledge** (zkSNARK) proof and **Fully Homomorphic Encryption** (FHE). The report will be divide in two parts, one for the zkSNARK and another one with the FHE where our results will be given.

Our problem is the following we want to evaluate a polynomial on a server and the server has to give a proof of it's result.

	Client	Communications	Server
Setup	Select the polynomial P and the evaluation point x	$\xrightarrow{P,x}$	
Eval		$\xleftarrow{y,\pi}$	Compute y=P[x] Compute π a proof of y
Verif	Check with π that y is correct		

We'll use the zkSNARK or FHE to create the proof and compare them on the following criteria :

- Computation time : how long it tooks to perform the evaluation and compute the proof
- Extra storage : the data e need to store to compute the proof
- Communication volume : the amount of communication needed
- Number of communication : how many communication we need to do to achieve the protocol

As a reminder **Zero-Knowledge Proof** enable a prover to convince a verifier that a statement is true without revealing anything else. It have 3 core property :

- **Completeness** : Given a statement and a witness, the prover can convince the verifier.
- **Soundness** : A malicious prover cannot convince the verifier of a false statement.
- **Zero-knowledge** : The proof does not reveal anything but the truth of the statement, in particular it doesn't reveal the prover's witness.

TODO add definitions and change report structure to be readable by someone who doesn't knwo zkSNARK

1.1 Definitions

This section give some definitions for the rest of the report.

Verifiable decryption: a verifiable decryption [CS03] is a primitive which can convince the verifier that the decrypted message is indeed from the corresponding ciphertext.

Rerandomizable encryption: a rerandomizable encryption [PR07] is a public-key encryption scheme where the ciphertext can be rerandomized, which can be viewed as a newly-encrypted ciphertext.

Additively-homomorphic encryption: an additively-homomorphic encryption is a primitive that allows computations on ciphertexts.

Commit-and-prove (CP) methodology. With a CP scheme one can prove statements of the form "$c_{ck}(x)$ contains x such that R(x,w)" where $c_{ck}(x)$ is a commitment. To see how the CP capability can be used for modular composition consider the following example of sequential composition in which one wants to prove that ∃w : z = h(x; w), where h(x; w) := g(f (x; w); w). Such a proof can be built by combining two CP systems Π_f and Π_g for its two building blocks, i.e., respectively f and g: the prover creates a commitment $c_{ck}(y)$ of y, and then uses Π_f (resp. Π_g) to prove that "$c_{ck}(y)$ contains y = f (x; w) (resp. contains y such that z = g(y; w))".

Quadratic Arithmetic Circuit (QAP) is a representation of an arithmetic circuit. If we have Q an arithmetic circuit that compute something with $a_{1...l}$ the outputs and inputs of the circuit, and $a_{l+1...m}$ the witness of the circuit (i.e the intermediate values). A QAP is a triplet set of polynomials $\{A_i[X], B_i[X], C_i[X]\}_{i=0}^{m}$, each A_i, B_i, C_i are of degree n, such that

$$(\sum_{i=0}^{m} a_i A_i)(\sum_{i=0}^{m} a_i B_i) = \sum_{i=0}^{m} a_i C_i$$

Now considering our QAP defined above with an n-degree polynomial Z[X] we say this accept a vector $x \in F^n$ such that Z(x) divide our equation above. If we have our set of polynomials $\{A_i[X], B_i[X], C_i[X]\}_{i=0}^{m}$ we can find Z with this method :
Generate an arbitrary set of n points $S \subseteq F$. Construct A(x), B(x), C(x) in such a way that $A(s_i)$,$B(s_i)$,$C(s_i)$ are the i-th rows of A, B, C respectively. The Z(x) is defined in such a way that $\forall s \in S : Z(s) = 0$.
Then we have :

$$(\sum_{i=0}^{m} a_i A_i(x)) * (\sum_{i=0}^{m} a_i B_i(x)) = \sum_{i=0}^{m} a_i C_i(x) \text{ mod } Z(x)$$

See [Pan] for more explanations.

Bilinear map is defined by seven elements $(p, G_1, G_2, G_T, e, g, h)$ such that :
- $e : G_1 * G_2 \rightarrow G_T$
- G_1, G_2, G_T are groups of prime order p
- g is a generator of G_1
- h is a generator of G_2
- e(g,h) is a generator of G_T
- $e(g^a, h^b) = e(g,h)^{ab}$

Zero-knowledge proof enable a prover to convince a verifier that a statement is true without revealing anything else. It have 3 core property :

- **Completeness** : Given a statement and a witness, the prover can convince the verifier.
- **Soundness** : A malicious prover cannot convince the verifier of a false statement.
- **Zero-knowledge** : The proof does not reveal anything but the truth of the statement, in particular it doesn't reveal the prover's witness.

— 2 —
zkSNARK

This chapter will present how zkSNARK works in general and we'll go deeper in a specific protocol with our adaptation to our problem.

2.1 zkSNARK overview

zkSNARK is a type of cryptographic proof protocol that reveal no information about the knowledge we try to prove. Here is the meaning of each letters.

- ZK for zero knowledge, the user's informations still remain confidential without compromising the security.
- S for succint, the test of the proof is supposed to be fast, with a small size.
- N for non-interactive, there is no constant interaction, communication or exchange between the prover and the verifier.
- ARK for argument of knowledge, the prover want to show to the verifier that he knows some knowledge.

A protocol should satisfy the 3 cores properties we defined above for zero-knowledge proof.

To construct the proof zkSNARK use polynomials evaluation, the idea is to convert our problem into a "Rank-1 Constraint System" **R1CS**. Then we convert the R1CS in a "Quadratic Arithmetic Circuit" (**QAP**) defined below with Lagrange interpolation. After that we evaluate our equation system on a point and check the equality.

Quadratic Arithmetic Circuit (QAP) is a representation of an arithmetic circuit. If we have Q an arithmetic circuit that compute something with $a_{1...l}$ the outputs and inputs of the circuit, and $a_{l+1...m}$ the witness of the circuit (i.e the intermediate values). A QAP is a triplet set of polynomials $\{A_i[X], B_i[X], C_i[X]\}_{i=0}^{m}$, each A_i, B_i, C_i are of degree n, such that

$$(\sum_{i=0}^{m} a_i A_i)(\sum_{i=0}^{m} a_i B_i) = \sum_{i=0}^{m} a_i C_i$$

Now considering our QAP defined above with an n-degree polynomial Z[X] we say this accept

a vector $x \in F^n$ such that Z(x) divide our equation above. If we have our set of polynomials $\{A_i[X], B_i[X], C_i[X]\}_{i=0}^m$ we can find Z with this method :
Generate an arbitrary set of n points $S \subseteq F$. Construct A(x), B(x), C(x) in such a way that $A(s_i)$,$B(s_i)$,$C(s_i)$ are the i-th rows of A, B, C respectively. The Z(x) is defined in such a way that $\forall s \in S : Z(s) = 0$.
Then we have :

$$(\sum_{i=0}^{m} a_i A_i(x)) * (\sum_{i=0}^{m} a_i B_i(x)) = \sum_{i=0}^{m} a_i C_i(x) \bmod Z(x)$$

See [Pan] for more explanations.
This two websites give a good example of how each transformations are done [But] and [Anob].

2.2 State of the art

A part of my work was to read some papers to get an idea of the existing schemes and techniques. This section give a summary of what I found.

2.2.1 Groth 16

[Gro] The aim of the algorithm is to prove to a verifier that we know a secret without revealing it. The algorithm will use a QAP to generate and check the proof that correspond to an equality of polynomials. This QAP need to be generated by yourself or a **trust party**.

If we think about an architecture client-server where the server want to prove to the client that he knows a secret. Then the client (or a trust party of the client) will have to generate a QAP to check this secret and send some keys to the server. The server will create the proof and send it back to the client. To finish the client will check the proof and accept or reject it.
This algorithm is sound, complet and can be zero-knowledge (ZK). And we'll use it for our protocol.

2.2.2 Groth16 agregation

[Anoa] this paper give a method to optimize the check of multiply answer with the Groth16 algorithm.

2.2.3 COCO

[Kos+] is designed to prove that someone else knows a knowledge. Where the knowledge is already knows by you. And it's supposed to provide a high level of snark circuit implementation, to be more user friendly. The repository jsnark implement their idea and give some examples. By the way to achieve this they cipher the witness inside the circuit in order to put a backdoor. With that if someone knows the witness it can prove it. And this encryption is interesting in our case.

For example with an RSA encryption the essential challenge is that the arithmetic operations are over integers mod n, where n is larger than the SNARK field of order p. They represent integers mod n as $\lceil \frac{\log_2 n}{m} \rceil$ m-bit elements. To multiply a pair of such integers z=xy mod n, they construct a circuit that verifies xy = qn + z, where q and z are $\lceil \frac{\log_2 n}{m} \rceil$ m-bit elements provided as witnesses by the prover. Their current implementation for big integers uses m = 64.

But as they said it's not efficient enough so they proposed a Diffie-Hellman key exchange via a SNARK-friendly field extension. Instead of relying on RSA as the main PKE scheme, they investigate another scheme based on the **Discrete-Logarithm** (DL) problem in Extension Fields, and use it for symmetric key exchange. Since p is only 254-bit prime, the DL problem in F_p will not be hard, therefore an extension F_{p^μ} will be used instead. The key exchange circuit has two generators in that case g, h $\leftarrow F_{p^\mu}$, where $\langle g \rangle = \langle h \rangle$ is a large multiplicative subgroup of order $q | p^\mu - 1$. They follow Lentra's guidelines for selecting q to be a factor of the μ-th cyclotomic polynomial θ_μ(x) when evaluated at x = p [Len97].

2.2.4 Lenstra

[Len97] from what I understand, they said that we can optimize calculations through our finite field by choosing a right basis. And they give a method to select this basis. I read this paper for a better understanding of COCO where they cite this technique.

Give a better description of the paper...

2.2.5 SAVER

[Lee+] give the opportunity to detach the encryption of our snark circuit. It's based on ElGamal Encryption. With this scheme we can convince a verifier that a clear message M is indeed from a corresponding ciphertext C. So it allow us to prove the correctness of the message without revealing the secret key.

Here are the different primitive of their scheme :
CRS $\leftarrow$ Setup(R) : takes an arbitrary relation R as an input, and outputs the corresponding common reference string CRS.
SK, P K, V K $\leftarrow$ KeyGen(CRS) : takes a CRS as an input, and outputs the corresponding secret key SK, public key P K, verification key V K.
π, CT $\leftarrow$ Enc(CRS, P K, M, $a_{n+1,l}$, $a_{l+1,m}$) : takes CRS, a public key P K, a message M = $m_1, ..., m_n$, a zk-SNARK statement $a_{n+1,l}$, and a witness $a_{l+1,m}$ as inputs, and outputs a proof π and a ciphertext CT = $(c_0, ..., c_n, \phi)$.
π' , C'_t $\leftarrow$ Rerandomize(PK, π, CT) : takes a public key PK, a proof π, a ciphertext CT as inputs, and outputs a new proof π' and a new ciphertext C'_t with fresh randomness.
0/1 $\leftarrow$ Verify_Enc(CRS, π, CT , $a_{n+1,l}$) : take a CRS, a proof π, a ciphertext CT and a statement $a_{n+1,l}$ as inputs, and outputs 1 if CT , $a_{n+1,l}$ is in the relation R, or 0 otherwise.
M, v $\leftarrow$ Dec(CRS, SK, VK, CT) : takes CRS, a secret key SK, a verification key VK, and a ciphertext CT = $(c_0, ..., c_n, \phi)$ as inputs, and outputs a plaintext M = $m_1, ..., m_n$ and a decryption proof v.
0/1 $\leftarrow$ Verify_Dec(CRS, VK, M, v, CT) : takes CRS, a verification key VK, a message M , a decryption proof v, and a ciphertext CT as inputs, and outputs 1 if M is a valid decryption of CT , or 0 otherwise.

2.3 Protocol

Now we have an idea of what are SNARK and how they work, our aim is to use them to create a remote polynomial evaluation. Where the server give a proof of the correctness of the computation. For this we'll use the groth16 algorithm implement in libsnark. All our results will be on this github repository. We'll detail the steps of the Groth16 algorithm to have a better understanding of it.

First we have to convert our polynomial into a **rank 1 constraint system** (R1CS) which will be convert in a QAP and then we'll create our keys and start our protocol.

2.3.1 Snark protocol in clear

If the client has a QAP with the polynomials $\{A_i[X], B_i[X], C_i[X]\}_{i=0}^{n}$ where he knows the value $a_{1...l,l+1...m}$ to solve it. Polynomials A_i, B_i and C_i are of degree n and Z is of degree n.
Let's call the QAP **R** such that $R = \{F, m, l, \{A_i, B_i, C_i\}_{i=0}^{m}, \{Z_i\}_{i=0}^{n}\}$
We define 3 methods Setup, Prove and Verify such that :
$Setup(R) \rightarrow (\sigma, \tau)$
$Prove(R, \sigma, a_{i...l}, a_{l+1...m}) \rightarrow \pi$
$Verify(R, \sigma, a_{i...l}, \pi) \rightarrow 0/1$

Setup function

```
Setup(R) :
    $\alpha \overset{\$}{\leftarrow} F^*$
    $\beta \overset{\$}{\leftarrow} F^*$
    $\gamma \overset{\$}{\leftarrow} F^*$
    $\delta \overset{\$}{\leftarrow} F^*$
    $s \overset{\$}{\leftarrow} F^*$
    $\tau = (\alpha, \beta, \gamma, \delta, s)$
    $\sigma = (\alpha, \beta, \gamma, \delta, \{s^i\}_{i=0}^{n-1})$
    return $(\tau, \sigma)$
```

Prove function

```
$Prove(R, \sigma, a_{i...l}, a_{l+1...m})$ :
    r$\overset{\$}{\leftarrow}$F
    k$\overset{\$}{\leftarrow}$F
    U = $\alpha + \sum_{i=0}^{m} a_i A_i(s) + r\delta$
    V = $\beta + \sum_{i=0}^{m} a_i B_i(s) + k\delta$
    We compute H(s) such that
```

$$(\sum_{i=0}^{m} a_i A_i(s))(\sum_{i=0}^{m} a_i B_i(s)) = \sum_{i=0}^{m} a_i C_i(s) + H(s)Z(s)$$

```
    $\mathbf{Aa} = \sum_{i=l+1}^{m} a_i A_i$
    $\mathbf{Ba} = \sum_{i=l+1}^{m} a_i B_i$
    $\mathbf{Ca} = \sum_{i=l+1}^{m} a_i C_i$
    $S = \frac{\beta \mathbf{Aa}(s) + \alpha \mathbf{Ba}(s) + \mathbf{Ca}(s)}{\delta}$
    W = $S + \frac{H(s)Z(s)}{\delta} + Uk + rV - rk\delta$
    $\pi = (U, V, W)$
    return $\pi$
```

Verify function

```
$\underline{Verify(R, \sigma, a_{i...l}, \pi):}$
    $\mathbf{Aa} = \sum_{i=0}^{l} a_i A_i$
    $\mathbf{Ba} = \sum_{i=0}^{l} a_i B_i$
    $\mathbf{Ca} = \sum_{i=0}^{l} a_i C_i$
    $Y = \frac{\beta \mathbf{Aa}(s) + \alpha \mathbf{Ba}(s) + \mathbf{Ca}(s)}{\gamma}$
    if $UV == \alpha\beta + Y\gamma + W\delta$
        return 1
    else
        return 0
```

See [Gro] for more explanations.

Proof of the equation

What we want is to check the following equality : $(\sum_{i=0}^{m} a_i A_i(s))(\sum_{i=0}^{m} a_i B_i(s)) = \sum_{i=0}^{m} a_i C_i(s) + H(s)Z(s)$
Detailled calculation for the if statement in verify function :
UV = $(\alpha + \sum_{i=0}^{m} a_i A_i(s) + r\delta)(\beta + \sum_{i=0}^{m} a_i B_i(s) + k\delta)$

= $\alpha\beta + \alpha(\sum_{i=0}^{m} a_i B_i(s)) + k\alpha\delta + \beta(\sum_{i=0}^{m} a_i A_i(s)) + (\sum_{i=0}^{m} a_i A_i(s))(\sum_{i=0}^{m} a_i B_i(s)) + k\delta(\sum_{i=0}^{m} a_i A_i(s)) + r\delta\beta + r\delta(\sum_{i=0}^{m} a_i B_i(s) + rk\delta\delta$

$\alpha\beta + Y\gamma + W\delta$ = $\alpha\beta + (\frac{\beta \sum_{i=0}^{l} a_i A_i(s) + \alpha \sum_{i=0}^{l} a_i B_i(s) + \sum_{i=0}^{l} a_i C_i(s)}{\gamma})\gamma + (S + \frac{H(s)Z(s)}{\delta} + Uk + rV - rk\delta)\delta$

= $\alpha\beta + \beta \sum_{i=0}^{l} a_i A_i(s) + \alpha \sum_{i=0}^{l} a_i B_i(s) + \sum_{i=0}^{l} a_i C_i(s) + (S + Uk + rV - rk\delta)\delta + H(s)Z(s)$

= $\alpha\beta + \beta \sum_{i=0}^{l} a_i A_i(s) + \alpha \sum_{i=0}^{l} a_i B_i(s) + \sum_{i=0}^{l} a_i C_i(s) + (\frac{\beta \sum_{i=l+1}^{m} a_i A_i(s) + \alpha \sum_{i=l+1}^{m} a_i B_i(s) + \sum_{i=l+1}^{m} a_i C_i(s)}{\delta} + (\alpha + \sum_{i=0}^{m} a_i A_i(s) + r\delta)k + r(\beta + \sum_{i=0}^{m} a_i B_i(s) + k\delta) - rk\delta)\delta + H(s)Z(s)$

= $\alpha\beta + \beta \sum_{i=0}^{l} a_i A_i(s) + \alpha \sum_{i=0}^{l} a_i B_i(s) + \sum_{i=0}^{l} a_i C_i(s) + \beta \sum_{i=l+1}^{m} a_i A_i(s) + \alpha \sum_{i=l+1}^{m} a_i B_i(s) + \sum_{i=l+1}^{m} a_i C_i(s) + ((\alpha + \sum_{i=0}^{m} a_i A_i(s) + r\delta)k + r(\beta + \sum_{i=0}^{m} a_i B_i(s) + k\delta) - rk\delta)\delta + H(s)Z(s)$

= $\alpha\beta + \beta \sum_{i=0}^{m} a_i A_i(s) + \alpha \sum_{i=0}^{m} a_i B_i(s) + \sum_{i=0}^{m} a_i C_i(s) + ((\alpha + \sum_{i=0}^{m} a_i A_i(s) + r\delta)k + r(\beta + \sum_{i=0}^{m} a_i B_i(s) + k\delta) - rk\delta)\delta + H(s)Z(s)$

= $\alpha\beta + \beta \sum_{i=0}^{m} a_i A_i(s) + \alpha \sum_{i=0}^{m} a_i B_i(s) + \sum_{i=0}^{m} a_i C_i(s) + (k\alpha + k \sum_{i=0}^{m} a_i A_i(s) + kr\delta + r\beta + r \sum_{i=0}^{m} a_i B_i(s) +$

$rk\delta - rk\delta)\delta + H(s)Z(s)$

$= \alpha\beta + \beta\sum_{i=0}^{m} a_iA_i(s) + \alpha\sum_{i=0}^{m} a_iB_i(s) + \sum_{i=0}^{m} a_iC_i(s) + k\delta\alpha + k\delta\sum_{i=0}^{m} a_iA_i(s) + kr\delta\delta + r\delta\beta + r\delta\sum_{i=0}^{m} a_iB_i(s) + H(s)Z(s)$

$\alpha\beta + \alpha(\sum_{i=0}^{m} a_iB_i(s)) + k\alpha\delta + \beta(\sum_{i=0}^{m} a_iA_i(s)) + (\sum_{i=0}^{m} a_iA_i(s))(\sum_{i=0}^{m} a_iB_i(s)) + k\delta(\sum_{i=0}^{m} a_iA_i(s)) + r\delta\beta + r\delta(\sum_{i=0}^{m} a_iB_i(s) + rk\delta\delta = \alpha\beta + \beta\sum_{i=0}^{m} a_iA_i(s) + \alpha\sum_{i=0}^{m} a_iB_i(s) + \sum_{i=0}^{m} a_iC_i(s) + k\delta\alpha + k\delta\sum_{i=0}^{m} a_iA_i(s) + kr\delta\delta + r\delta\beta + r\delta\sum_{i=0}^{m} a_iB_i(s) + H(s)Z(s)$

$\Leftrightarrow (\sum_{i=0}^{m} a_iA_i(s))(\sum_{i=0}^{m} a_iB_i(s)) = \sum_{i=0}^{m} a_iC_i(s) + H(s)Z(s)$

Back to client-server architecture

So now if we go back to our problem where our client want to know if the server has some knowledge :

Client	Server
Compute R Setup(R) $\rightarrow (\sigma, \tau)$ Send σ, a_i inputs and R to the server	
	The server compute with his knowledge the a_i outputs and the witness $Prove(R, \sigma, a_{i...l}, a_{l+1...m}) \rightarrow \pi$ Send π and a_i outputs to the client
Run $Verify(R, \sigma, a_{i...l}, \pi) \rightarrow 0/1$	

If the ouput is 1 the client knows that the server knows the witness and the outputs send by the server are correct. But if it's 0 the client knows that the server doesn't know the secret or has calculated a wrong output value. By the way this protocol is zero-knowledge on the witness (i.e someone who intercept the communication will not learn anything about the witness).

Problem

But we have a problem with these scheme. Since the server knows α, β, δ, x. He can cheat by this way :
Pick U,V over F at random
Compute W=$\frac{UV - \alpha\beta - \sum_{i=0}^{l}(a_i(\beta A_i(s) + \alpha B_i(s) + C_i(s)))}{\delta}$
Now our equality is :

$UV = \alpha\beta + \frac{\beta\sum_{i=0}^{l} a_iA_i(s) + \alpha\sum_{i=0}^{l} a_iB_i(s) + \sum_{i=0}^{l} a_iC_i(s)}{\gamma}\gamma + \frac{UV - \alpha\beta - \sum_{i=0}^{l}(a_i(\beta A_i(x) + \alpha B_i(x) + C_i(x)))}{\delta}\delta$

$UV = \alpha\beta + \beta\sum_{i=0}^{l} a_iA_i(s) + \alpha\sum_{i=0}^{l} a_iB_i(s) + \sum_{i=0}^{l} a_iC_i(s) + UV - \alpha\beta - \sum_{i=0}^{l}(a_i(\beta A_i(s) + \alpha B_i(s) + C_i(s)))$

$UV = \alpha\beta + \beta \sum_{i=0}^{l} a_i A_i(s) + \alpha \sum_{i=0}^{l} a_i B_i(s) + \sum_{i=0}^{l} a_i C_i(s) + UV - \alpha\beta - \sum_{i=0}^{l}(a_i(\beta A_i(s) + \alpha B_i(s) + C_i(s)))$

$0 = 0$ So the if statement in the verify function will return true even if the person who compute the proof didn't know the $a_{l+1..m}$ (the witness).

2.3.2 Snark protocol in ZK

Now we have our protocol and the idea behind that. We want to remove the problem identify above. To perform this we'll use pairing with elliptic curves. $G1 * G2 \rightarrow Gt$
Now let's define R like $R = \{p, G1, G2, Gt, e, g, h, l, A, B, C, Z\}$. With $(p, G1, G2, Gt, e, g, h)$ a bilinear map, with the following definition :

Bilinear map is defined by seven elements $(p, G_1, G_2, G_T, e, g, h)$ such that :
- $e : G_1 * G_2 \rightarrow G_T$
- G_1, G_2, G_T are groups of prime order p
- g is a generator of G_1
- h is a generator of G_2
- e(g,h) is a generator of G_T
- $e(g^a, h^b) = e(g,h)^{ab}$

For the same method Setup, Prove and Verify we just change the value of R and their output.

Setup function

```
Setup(R) :
    $\alpha \xleftarrow{\$} F^*$
    $\beta \xleftarrow{\$} F^*$
    $\gamma \xleftarrow{\$} F^*$
    $\delta \xleftarrow{\$} F^*$
    $s \xleftarrow{\$} F^*$
    $\tau = (\alpha, \beta, \gamma, \delta, s)$
    $\alpha_1 = [\alpha]_1$
    $\beta_1 = [\beta]_1$
    $\gamma_1 = [\gamma]_1$
    $\delta_1 = [\delta]_1$
    $\mathbf{S1} = \{[s^i]_1\}_{i=0}^{n-1}$
    $\mathbf{Sa1} = \{[\frac{\beta A_i(s)+\alpha B_i(s)+C_i(s)}{\delta}]_1\}_{i=l+1}^{m}$
    $\mathbf{Sb1} = \{[\frac{s^i Z_i(s)}{\delta}]_1\}_{i=0}^{n-1}$
    $\mathbf{Sc1} = \{[\frac{\beta A_i(s)+\alpha B_i(s)+C_i(s)}{\gamma}]_1\}_{i=0}^{l}$
    $\sigma 1 = (\alpha_1, \beta_1, \gamma_1, \delta_1, \mathbf{S1}, \mathbf{Sa1}, \mathbf{Sb1}, \mathbf{Sc1}, g, G_1, p)$
    $\beta_2 = [\beta]_2$
    $\gamma_2 = [\gamma]_2$
    $\delta_2 = [\delta]_2$
    $\mathbf{S2} = \{[s^i]_2\}_{i=0}^{n-1}$
    $\sigma 2 = (\beta_2, \gamma_2, \delta_2, \mathbf{S2}, h, G_2, p)$
    $\sigma = (\sigma 1, \sigma 2, A_i, B_i, C_i)$
    return $(\sigma, \tau)$
```

Only σ is sent to the person who want to prove something.

Prove function

```
$\underline{Prove(\sigma, a_{i\ldots l}, a_{l+1\ldots m}):}$
    r$\xleftarrow{\$}$F
    k$\xleftarrow{\$}$F
    $\mathbf{Aa} = \sum_{i=0}^{m} a_i A_i$
    $\mathbf{Ba} = \sum_{i=0}^{m} a_i B_i$
    $\mathbf{Ca} = \sum_{i=0}^{m} a_i C_i$
    U = $\alpha_1 \Pi_{i=0}^{m} \mathbf{S1}_i^{\mathbf{Aa}_i} (\delta_1)^r$
    **V1**= $\beta_1 \Pi_{i=0}^{m} \mathbf{S1}_i^{\mathbf{Ba}_i} (\delta_1)^k$
    **V2**= $\beta_2 \Pi_{i=0}^{m} \mathbf{S2}_i^{\mathbf{Ba}_i} (\delta_2)^k$
    We compute H(s) such that
        $$(\sum_{i=0}^{m} a_i A_i(s))(\sum_{i=0}^{m} a_i B_i(s)) = \sum_{i=0}^{m} a_i C_i(s) + H(s)Z(s)$$
    $S = \Pi_{i=l+1}^{m} (\mathbf{Sa1}_i)^{a_i}$
    W = $\frac{S(\Pi_{i=0}^{n} (\mathbf{Sb1}_i)^{H_i}) U^k \mathbf{V1}^k}{{\delta_1}^{rk}}$
    $\pi = (U, \mathbf{V2}, W)$
    return $\pi$
```

Verify function

```
$\underline{Verify(\sigma, a_{i\ldots l}, \pi):}$
    $Y = \Pi_{i=0}^{l} (\mathbf{Sc1}_i)^{a_i}$
    if $e(U, \mathbf{V2}) == e(\alpha_1, \beta_2) e(Y, \gamma_2) e(W, \delta_2)$
        return 1
    else
        return 0
```

Proof of the equality

What we want is to check the following equality :

$$(\sum_{i=0}^{m} a_i A_i(s))(\sum_{i=0}^{m} a_i B_i(s)) = \sum_{i=0}^{m} a_i C_i(s) + H(s)Z(s)$$

Here is the detailled calcul for the if statement in verify :
$e(U, \mathbf{V2}) = e((\alpha_1 \Pi_{i=0}^{m} \mathbf{S1}_i^{\mathbf{Aa}_i} (\delta_1)^r), (\beta_2 \Pi_{i=0}^{m} \mathbf{S2}_i^{\mathbf{Ba}_i} (\delta_2)^k))$

= $[(\alpha + \sum_{i=0}^{m} a_i A_i(s) + r\delta)(\beta + \sum_{i=0}^{m} a_i B_i(s) + k\delta)]_T$

$e(\alpha_1, \beta_2) e(Y, \gamma_2) e(W, \delta_2) = e(\alpha_1, \beta_2) e(\Pi_{i=0}^{l} (\mathbf{Sc1}_i)^{a_i}, \gamma_2) e(\frac{S(\Pi_{i=0}^{n} (\mathbf{Sb1}_i)^{H_i}) U^k \mathbf{V1}^k}{{\delta_1}^{rk}}, \delta_2)$

= $[\alpha\beta]_T [(\frac{\beta \sum_{i=0}^{l} a_i A_i(s) + \alpha \sum_{i=0}^{l} a_i B_i(s) + \sum_{i=0}^{l} a_i C_i(s)}{\gamma}) \gamma]_T [(S + \frac{H(s)Z(s)}{\delta} + Uk + rV - rk\delta)\delta]_T$

= $[\alpha\beta + (\frac{\beta \sum_{i=0}^{l} a_i A_i(s) + \alpha \sum_{i=0}^{l} a_i B_i(s) + \sum_{i=0}^{l} a_i C_i(s)}{\gamma}) \gamma + (S + \frac{H(s)Z(s)}{\delta} + Uk + rV - rk\delta)\delta]_T$

For our both side we have the same equation as the proof in 4.4 just in the g_T exponent.

Good point

We have solved our problem above, with the encryption of α, β, γ, s the server can't compute W as before :
W=$\frac{UV-\alpha\beta-\sum_{i=0}^{l}(a_i(\beta A_i(s)+\alpha B_i(s)+C_i(s)))}{\delta}$
or he has break the descrete logarithm problem in order to find s,β, α or γ.

2.3.3 Go back to our problem

Now we have this protocol with zkSNARKs we want something quite similar. Back to a connexion client-server the problem is the following. A client want to evaluate a polynomial on a point, but he want that the server achieve this and gave him a proof of the correctness of the result.
We have something like this :

Client	**Server**
P$\in F[X]$ a polynomial, we want to evaluate it on x $\in F$ Send P and x to the server	
	Compute y = P(x) Compute π a proof that y is correct Send π and y to the client
With π anyone can check that y is correct	

In order to compute the proof of our computation we will use the SNARK protocol with QAP explained above.

Protocol in clear

Client	**Server**
With P$\in F[X]$ Compute R corresponding to P Setup(R) $\rightarrow (\sigma, \tau)$ Send σ, R and the a_i input to the server	
	The server compute R with the a_i input to get the : a_i output and the witness Then he generate : $Prove(R, \sigma, a_{1...l}, a_{l+1...m}) \rightarrow \pi$ Send π, a_i ouput, to the client
Someone who want to check the proof run $Verify(R, \sigma, a_{i...l}, \pi) \rightarrow 0/1$	

If the ouput is 1 the client knows that the server computation is correct.

Protocol in ZK

Now we have our protocol let's imagine that the client don't wan't his polynomial P to be known by the server. So what we want is that the server don't learn anything about what he's currently computing.
A possible solution is to cipher the polynomials P before creating the R1CS with paillier encryption. It's the aim of the third chapter in zkSNARK.

Summarize of the first part of zkSNARK

	Client	**Communications**	**Server**
Setup	With P$\in F[X]$ Compute R corresponding to P $\alpha,\beta,\gamma,\delta,s \leftarrow^{\$} F^*$ $\alpha_1=[\alpha]_1,\beta_1=[\beta]_1,\gamma_1=[\gamma]_1$ $\delta_1=[\delta]_1,\mathbf{S1}=\{[s^i]_1\}_{i=0}^{n-1},$ $\mathbf{CisDelta}=\{[\frac{C_i(s)}{\delta}]_1\}_{i=l+1}^{m}$ $\mathbf{Sa1}=\{[\frac{\beta A_i(s)+\alpha B_i(s)}{\delta}]_1\}_{i=l+1}^{m},$ $\mathbf{Sb1}=\{[\frac{s^iZ_i(s)}{\delta}]_1\}_{i=0}^{n-1}$ $\beta_2=[\beta]_2,\gamma_2=[\gamma]_2,\delta_2=[\delta]_2,\mathbf{S2}=\{[s^i]_2\}_{i=0}^{n-1}$ crs=$(\alpha_1,\beta_1,\gamma_1,\delta_1,\mathbf{S1},$ $\mathbf{CisDelta},\mathbf{Sa1},\mathbf{Sb1},\beta_2,\gamma_2,\delta_2,\mathbf{S2},g,h,G_1,G_2)$ $S=\{g^{\frac{\beta x_i(s)+\alpha y_i(s)+z_i(s)}{\gamma}}\}_{i=0}^{n}$ $\mathbf{CisDeltaStart}=\{[\frac{C_i(s)}{\delta}]_1\}_{i=0}^{l}$ $\mathbf{Sc1}=\{[\frac{\beta A_i(s)+\alpha B_i(s)}{\gamma}]_1\}_{i=0}^{l}$ $\mathbf{Sc}=\{[C_is^i]_1\}_{i=0}^{n-1}$ vk=$(\alpha_1,\beta_2,S,\mathbf{CisDeltaStart},\gamma_2,\delta_2,\mathbf{Sc1},\mathbf{Sc})$	$\xrightarrow{crs,a_{input},R}$	
Eval		$\xleftarrow{\pi,a_{output}}$	Compute $a_{witness},a_{output}$ from R on a_{input} $r,k \overset{\$}{\leftarrow} F*$ $\mathbf{Aa}=\sum_{i=0}^{m}a_iA_i$ $\mathbf{Ba}=\sum_{i=0}^{m}a_iB_i$ $\mathbf{Ca}=\sum_{i=0}^{m}a_iC_i$ U = $\alpha_1\Pi_{i=0}^{m}\mathbf{S1}_i^{\mathbf{Aa}_i}(\delta_1)^r$ **V1**= $\beta_1\Pi_{i=0}^{m}\mathbf{S1}_i^{\mathbf{Ba}_i}(\delta_1)^k$ **V2**= $\beta_2\Pi_{i=0}^{m}\mathbf{S2}_i^{\mathbf{Ba}_i}(\delta_2)^k$ We compute H(s) such that $(\sum_{i=0}^{m}a_iA_i(s))(\sum_{i=0}^{m}a_iB_i(s))$ $=\sum_{i=0}^{m}a_iC_i(s)+H(s)Z(s)$ $S=\Pi_{i=l+1}^{m}(\mathbf{Sa1}_i)^{a_i}\mathbf{CisDelta}_i$ W = $\frac{\Pi_{i=0}^{n}S_i(\Pi_{i=0}^{n}(\mathbf{Sb1}_i)^{H_i})U^k\mathbf{V1}^k}{\delta 1^{rk}}$ $\pi=(U,W,\mathbf{V2})$

Verif	$Y=\Pi_{i=0}^{l}(\mathbf{Sc1}_i)^{a_i}\mathbf{CisDeltaStart}_i$
	if $e(U,\mathbf{V2})==e(\alpha_1,\beta_2)e(Y,\gamma_2)e(W,\delta_2)$: return 1
	else : return 0		

Evaluation of the polynomial

Setup function With p prime, g the generator of G1, h the generator of G2 and e such that e(g,h) is the generator of Gt. Let's call the QAP R such that $R=\{F,m,l,\{A_{i,j},B_{i,j},C_{i,j}\}_{i=0,j=0}^{i=m,j=n},\{Z_i\}_{i=0}^{n}\}$

```
Setup($1^\lambda, R$) :
    $\alpha \xleftarrow{\$} F*$
    $\beta \xleftarrow{\$} F*$
    $\gamma \xleftarrow{\$} F*$
    $\delta \xleftarrow{\$} F*$
    $\alpha_1 = [\alpha]_1$
    $\beta_1 = [\beta]_1$
    $\gamma_1 = [\gamma]_1$
    $\delta_1 = [\delta]_1$
    $\mathbf{S1} = \{[s^i]_1\}_{i=0}^{n-1}$
    $\mathbf{CisDelta} = \{[\frac{C_i(s)}{\delta}]_1\}_{i=l+1}^{m}$
    $\mathbf{Sa1} = \{[\frac{\beta A_i(s)+\alpha B_i(s)}{\delta}]_1\}_{i=l+1}^{m}$
    $\mathbf{Sb1} = \{[\frac{s^i Z_i(s)}{\delta}]_1\}_{i=0}^{n-1}$
    $\beta_2 = [\beta]_2$
    $\gamma_2 = [\gamma]_2$
    $\delta_2 = [\delta]_2$
    $\mathbf{S2} = \{[s^i]_2\}_{i=0}^{n-1}$
    crs=$(\alpha_1, \beta_1, \gamma_1, \delta_1, \mathbf{S1}, \mathbf{CisDelta}, \mathbf{Sa1}, \mathbf{Sb1}, \beta_2, \gamma_2, \delta_2, \mathbf{S2}, g, h, G_1, G_2)$
    $S = \{g^{\frac{\beta x_i(s)+\alpha y_i(s)+z_i(s)}{\gamma}}\}_{i=0}^{n}$
    $\mathbf{CisDeltaStart} = \{[\frac{C_i(s)}{\delta}]_1\}_{i=0}^{l}$
    $\mathbf{Sc1} = \{[\frac{\beta A_i(s)+\alpha B_i(s)}{\gamma}]_1\}_{i=0}^{l}$
    $\mathbf{Sc} = \{[C_i s^i]_1\}_{i=0}^{n-1}$
    vk=$(\alpha_1, \beta_2, S, \mathbf{CisDeltaStart}, \gamma_2, \delta_2, \mathbf{Sc1}, \mathbf{Sc})$
    return crs, vk
     Send crs to the prover and vk to the verifier.
```

Prove function We have $a_0 = 1$ it's a constant due to the R1CS constraint.

```
Prove(crs, $a_{1..l}$, $a_{l+1..m}$) :
    $r \xleftarrow{\$} F*$
    $k \xleftarrow{\$} F*$
    $\mathbf{Aa} = \sum_{i=0}^{m} a_i A_i$
    $\mathbf{Ba} = \sum_{i=0}^{m} a_i B_i$
    $\mathbf{Ca} = \sum_{i=0}^{m} a_i C_i$
    U = $\alpha_1 \Pi_{i=0}^{m} \mathbf{S1}_i^{\mathbf{Aa}_i} (\delta_1)^r$
    V1= $\beta_1 \Pi_{i=0}^{m} \mathbf{S1}_i^{\mathbf{Ba}_i} (\delta_1)^k$
    V2= $\beta_2 \Pi_{i=0}^{m} \mathbf{S2}_i^{\mathbf{Ba}_i} (\delta_2)^k$
    We compute H(s) such that
```

$$(\sum_{i=0}^{m} a_i A_i(s))(\sum_{i=0}^{m} a_i B_i(s)) = \sum_{i=0}^{m} a_i C_i(s) + H(s)Z(s)$$

```
    $S = \Pi_{i=l+1}^{m}(\mathbf{Sa1}_i)^{a_i}\mathbf{CisDelta}_i$
    W = $\frac{(\Pi_{i=0}^{n} S_i)(\Pi_{i=0}^{n}(\mathbf{Sb1}_i)^{H_i})U^k\mathbf{V1}^k}{\delta 1^{rk}}$
    $\pi = (U, W, \mathbf{V2})$
    return $\pi = (U, W, \mathbf{V2})$, $a_l$ // the one corresponding to the output.
```

Verify function $\underline{\text{Verify}(\text{vk}, a_{0..l-1}, a_l, \pi):}$

```
    $Y = \Pi_{i=0}^{l}(\mathbf{Sc1}_i)^{a_i}\mathbf{CisDeltaStart}_i$
    if $e(U, \mathbf{V2}) == e(\alpha_1, \beta_2)e(Y, \gamma_2)e(W, \delta_2)$ :
        return 1
    else
        return 0
```

2.3.4 Implementation in libsnark

The major difference is in the setup of the verification key and the provable key, where we don't send the A_i, B_i, C_i but the $A_i(s), B_i(s), C_i(s)$ with the R1CS constrainsts. And for the $\alpha, \beta, \sigma, \gamma$ they are already injected in the $A_i(s), B_i(s), C_i(s)$ above.

Give more details for the implementation

2.3.5 Results with libsnark

I've done some implementation with libsnark and here are my results compare to our protocol with pairing and paillier, for a polynomial in clear with libsnark :

Time of calculation

Time in seconds

	Libsnark			**Paillier 1024**			**Paillier 2048**		
Degree	Setup	Client	Server	Setup	Client	Server	Setup	Client	Server
256	0.1678	0.0249	0.1640	0.1824	0.0011	0.1638	0.6749	0.0017	0.2653
512	0.2946	0.0261	0.2843	0.3851	0.0011	0.3165	1.1360	0.0019	0.5072
1024	0.5177	0.0272	0.5551	0.6832	0.0011	0.6485	1.8827	0.0017	0.9836
2048	0.9732	0.0285	0.9687	1.3459	0.0011	1.2551	3.8696	0.0017	1.9426
4096	1.7896	0.0267	1.7556	2.7452	0.0011	2.5792	7.5359	0.0017	3.9480
8192	3.0986	0.02684	3.1388	5.5579	0.0011	5.3739	14.2259	0.0017	7.4721
16384	5.9548	0.0270	6.0528	10.6597	0.0011	10.4383	29.2171	0.0020	15.3933
32768	10.8519	0.0268	11.3400	21.3708	0.0011	20.3710	56.6373	0.0017	30.4662
65536	20.1288	0.0266	21.5019	41.8292	0.0011	41.0267	113.1845	0.0017	61.1323
131072	37.8404	0.0265	40.9986	83.8971	0.0011	82.3237	225.7390	0.0019	122.0703

Data to save

Data in bits

	Libsnark				
Degree	Client save after setup	Data send to server setup	Server save after setup	Eval data send	Response of data eval
256	3629	1 482 698	1 482 698	254	2548
512	3629	2 966 474	2 966 474	254	2548
1024	3629	5 934 026	5 934 026	254	2548
2048	3629	11 869 130	11 869 130	254	2548
4096	3629	23 739 338	23 739 338	254	2548
8192	3629	47 479 754	47 479 754	254	2548
16384	3629	94 960 586	94 960 586	254	2548
32768	3629	189 922 250	189 922 250	254	2548
65536	3629	379 845 578	379 845 578	254	2548
131072	3629	759 692 234	759 692 234	254	2548

	Relic				
Degree	Client save after setup	Data send to server setup	Server save after setup	Eval data send	Response of data eval
256	5376	722 944	722 944	256	768
512	5376	1 443 840	1 443 840	256	768
1024	5376	2 885 632	2 885 632	256	768
2048	5376	5 769 216	5 769 216	256	768
4096	5376	11 536 384	11 536 384	256	768
8192	5376	23 070 720	23 070 720	256	768
16384	5376	46 139 392	46 139 392	256	768
32768	5376	92 276 736	92 276 736	256	768
65536	5376	184 551 424	184 551 424	256	768
131072	5376	369 100 800	369 100 800	256	768

With our results we can see that libsnark use more memory to store necessary information for the protocol for the server side and less for the client side. But we have to take into account that our implementation of libsnark doesn't cipher our polynomial so anyone can see the polynomial we are currently evaluating. Adversely the instance of Relic cipher it.

— 3 —
FHE

Now we'll focus ourself on protocol using FHE encryption.

3.1 Efficiently Verifiable Computation on Encrypted Data

This paper describe a protocol using the "ring learning with error" **RLWE** which is a FHE [FGP].

3.1.1 LWE definition

The learning with errors (**LWE**) problem was introduced by Regev [Reg]. Defined as follow :
For security parameter λ let n = n(λ) be an integer dimension, let q = q(λ) $\geq$ 2 be an integer and let χ = $\chi(\lambda)$ $\geq$ 2 be a distribution over $\mathbb{Z}$. The $LWE_{n,q,\chi}$ problem is to distinguish the following two distributions :

In the first distribution, one samples (a_i, b_i) uniformly from $\mathbb{Z}_q^{n+1}$

In the second distribution one first draws $s \overset{\$}{\leftarrow} \mathbb{Z}_q^n$ uniformly and then samples $(a_i, b_i) \in \mathbb{Z}_q^{n+1}$ by sampling $a_i \overset{\$}{\leftarrow} \mathbb{Z}_q^n$ uniformly, $e_i \overset{\$}{\leftarrow} \chi$ and setting $b_i = \langle a, s \rangle + e_i$. The $LWE_{n,q,\chi}$ assumption is that the $LWE_{n,q,\chi}$ problem is infeasible.

3.1.2 RLWE definition

The ring learning with errors (**RLWE**) problem was introduced by Lyubaskevsky, Peikert and Regev [LPR]. Here is a simplified definition :
For security parameter λ, let f(x) = x^d + 1 where d = d(λ) is a power of 2. Let q = q(λ) $\geq$ 2 be an integer. Let R = $\mathbb{Z}$[x]/(f(x)) and let R_q = R/qR. Let χ = $\chi(\lambda)$ be a distribution over R. The $RLWE_{d,q,\chi}$ problem is to distinguish the following two distributions :

In the first distribution, one samples (ai, bi) uniformly from R_q^2.

In the second distribution, one first draws $s \overset{\$}{\leftarrow} R_q$ uniformly and then samples $(a_i, b_i) \in R_q^2$ by sampling $a_i \overset{\$}{\leftarrow} R_q$ uniformly, $e_i \overset{\$}{\leftarrow} \chi$, and setting $b_i = a_i s + e_i$. The $RLWE_{d,q,\chi}$ assumption is that the $RLWE_{d,q,\chi}$ problem is infeasible.

3.1.3 Homomorphic Encryption (HE) functions

We define the following functions for our RLWE.

HE.ParamGen function

Cyclotomic polynomial : In mathematics the n-th cyclotomic polynomial for any positive integer n is the unique irreducible polynomial with integer coefficients that is a divisor of $x^n - 1$ and is not a divisor of $x^k - 1$ for nay k<n. It's roots are all n-th primitive roots of unity $e^{2i\pi k/n}$. The n-th cyclotomic polynomial is equal to

$$\Phi_m(X) = \prod_{1 \leq k \leq n, gcd(k,n)=1} (x - e^{2i\pi k/n})$$

Given the security parameter λ we define the polynomial ring R = $\mathbb{Z}[X]/\Phi_m(X)$ where $\Phi_m(X)$ is the m-th cyclotomic polynomial in $\mathbb{Z}[X]$ whose degree n = $\varphi(m)$ is lower bounded by a function of the security parameter λ.
The message space M is the ring $R_p = R/pR = \mathbb{Z}_p[X]/\Phi_m(X)$. The ciphertext space is describe as follow, pick an integer q > p which is co-prime to p and define the ring $R_q = R/qR = \mathbb{Z}_q[X]/\Phi_m(X)$.
Ciphertexts can be thought of as polynomials in $\mathbb{Z}_q[X][Y]$ as follow : Encryption manipulated with addition : degree 1 in Y and degree (n - 1) in X. c$\in$ $\mathbb{Z}_q[X][Y]$ where c=$c_0 + c_1 Y$ with $c_0, c_1 \in R_q$
Encryption manipulated with multiplication : degree 2 in Y and degree 2(n - 1) in X. c$\in$ $\mathbb{Z}_q[X][Y]$ where c=$c_0 + c_1 Y + c_2 Y^2$ with $c_0, c_1, c_2 \in R_q$, $deg_X(c_i) = 2(n-1)$
We define 2 distributions :
$D_{\mathbb{Z}^n,\sigma}$ The discrete Gaussian with parameter σ it's a random variable over $\mathbb{Z}^n$ obtained from sampling $x \in \mathbb{R}^n$ with probability $e^{-\pi||x||_2/\sigma^2}$
ZO_n sample a vector x = ($x_1, ..., x_n$) with $x_i \in -1, 0, +1$ and Pr[$x_i = -1$] = 1/4, Pr[$x_i = +1$] = 1/4, Pr[$x_i = 0$] = 1/2

```
HE.ParamGen($\lambda$) :
    $D_{\mathbb{Z}^n,\sigma}$
    $ZO_n$
```

HE.KeyGen function

```
HE.KeyGen($1^\lambda$) :
    $a \xleftarrow{\$} R_q$
    $s, e \xleftarrow{\$} D_{\mathbb{Z}^n,\sigma}$
    $b = as + pe$
    $dk = s$
    $pk = (a, b)$
    return $dk, pk$
```

HE.Enc function

m is the message we want to cipher with $m \in R_q$

```
HE.Enc(pk, m) :
    $r \xleftarrow{\$} (ZO_n, D_{\mathbb{Z}^n,\sigma}, D_{\mathbb{Z}^n,\sigma})$
    $u = r[0], v = r[1], w = r[2]$
    $c_0 = bu + pw + m$
    $c_1 = au + pv$
    $c = c_0 + c_1 Y$
    return $c$
```

HE.Dec function

$c \in \mathbb{Z}_q[X][Y]$ is a ciphertext with $c = c_0 + c_1 Y + c_2 Y^2$, $c_i \in \mathbb{Z}_q[X]$

```
HE.Dec(dk, c) :
    for i in {0, 1, 2} :
        $c'_i = c_i \bmod \Phi_m(X)$
    $t = c'_0 - sc'_1 - s^2 c'_2$
    $res = t \bmod p$
    return $res$
```

3.1.4 Homomorphic hash function

We define the following functions for our homomorphic hash function.
- **H.KeyGen** generates the description of a function **H**
- **H** computes the function
- **H.Eval** allows to compute over $\mathbb{R}$

They propose a homomorphic hash whose key features is that it allows to "compress" an homomorphic encryption scheme by Brakerski and Vaikuntanathan (**BV**) [CG] cyphertext $\mu \in \mathbb{Z}_q[X][Y]$ into a single entry $v \in \mathbb{Z}_q$ in such a way that **H** is a ring homomorphism hence **H.Eval**(f, (**H**(γ_1),...,**H**(γ_t))) = **H**(f(γ_1,...,γ_t)).

Let q be a prime of λ bits, N, c be two integers of size at most polynomial in λ and let D=$\mu \in \mathbb{Z}[X][Y] : deg_X(\mu) = N, deg_Y(\mu) = c$

H.KeyGen function

```
H.KeyGen() :
    $\alpha, \beta \xleftarrow{\$} \mathbb{Z}_q$
    $k = (\alpha, \beta)$
    return $k$
```

H function

With $\mu \in D$ such that $\mu = \sum_{j=0}^{c} \mu_j Y^j$ the function **H** evaluate μ at Y=α over $\mathbb{Z}_q[X]$ and then evaluate $\mu(\alpha)$ at X=β

```
H(k, $\mu$) :
    $res = \sum_{j=0}^{c} \sum_{i=0}^{N} (\mu_j \alpha^j)_i \beta^i$
    return $res$
```

H.Eval function

On input two values $v_1, v_2 \in \mathbb{Z}_q$ and an operation f_g which is addition or multiplication

```
H.Eval($f_g, v_1, v_2$) :
    $res = f_g(v_1, v_2)$
    return $res$
```

3.1.5 Protocol

Now with our primitive defined above we can set the functions of our protocol as follow :

KeyGen function

With P=$\sum_{i=0}^{t} p_i X^i$ our polynomial and $p_i \in R_q$

```
KeyGen($p_0, ..., p_t, \lambda$) :
    Specify a group (G,.) of order q and a generator g
    $HE.ParamGen(\lambda)$
    $(dk, pk) = HE.KeyGen(1^\lambda)$
    $c, k_0, k_1 \xleftarrow{\$} \mathbb{Z}_q$
    $k = H.KeyGen()$ //Just a reminder k=$(\alpha, \beta)$
    for i in {0,...,t}
        $\gamma_i = HE.Enc(pk, p_i)$
        $T_i = c * H(k, \gamma_i) + k_1^i k_0$
        $G_{T,i} = g^{T_i}$
    $PK = (pk, G, g, \gamma_0, G_{T,0}, ..., \gamma_t, G_{T,t})$
    $SK = (pk, G, g, dk, c, k, k_0, k_1)$
    return (PK, SK)
```

ProbGen function

```
ProbGen(PK, x) :
    $\sigma_x = x$
    $\tau_x = x$
    return $(\sigma_x, \tau_x)$
```

Compute function

```
Compute(PK, $\sigma_x$) :
    $\gamma = \sum_{i=0}^{t} x^i \gamma_i$
    $G_t = \prod_{i=0}^{t}(G_{T,i})^{x^i}$
    $\sigma_y = (\gamma, G_t)$
    return $\sigma_y$
```

Verify function

```
Verify(SK,$\sigma_y, \tau_x$) : // Recall $\sigma_y = (\gamma, G_t)$ and $\tau_x = x$
    $a = ((\tau_x k_1)^{t+1} - 1)(\tau_x k_1 - 1)^{-1}$
    if $G_t == (g^{H(k,\gamma)})^c * (g^{k_0})^a$
        accept
        res = HE.Dec(dk,$\gamma$)
        return *res*
    else
        reject
```

Proof of the equality

We have the following equality $G_t == (g^{H(k,\gamma)})^c * (g^{k_0})^a$
$G_t = \prod_{i=0}^{t}(G_{T,i})^{x^i}$
$= \prod_{i=0}^{t}(g^{T_i})^{x^i}$
$= \prod_{i=0}^{t}(g^{c*H(k,\gamma_i)+k_1^i k_0})^{x^i}$
$= \prod_{i=0}^{t} g^{(c*H(k,\gamma_i)+k_1^i k_0)*x^i}$
$= \prod_{i=0}^{t} g^{c*H(k,\gamma_i)*x^i} g^{k_1^i k_0 x^i}$
$= g^{c\sum_{i=0}^{t} H(k,\gamma_i)*x^i} g^{\sum_{i=0}^{t} k_1^i k_0 x^i}$
Since **H** is a ring homomorphism we have $\sum_{i=0}^{t} H(k,\gamma_i)*x^i = \sum_{i=0}^{t} H(k, x^i\gamma_i)$ and $\sum_{i=0}^{t} H(k, x^i\gamma_i) = H(k,\gamma)$ so :
$= g^{cH(k,\gamma)} g^{((xk_1)^{t+1}-1)(xk_1-1)^{-1}}$
$= g^{cH(k,\gamma)} g^{a}$
We have our equality.

Protocol diagram

	Client	**Communications**	**Server**
Setup	P=$\sum_{i=0}^{t} p_i X^i$ $p_i \in R_q$ (SK, PK) $\leftarrow$ KeyGen(p_i, λ) (σ_x, τ_x) $\leftarrow$ ProbGen(PK, x)	$\xrightarrow{PK,\sigma_x}$	
Eval		$\xleftarrow{\sigma_y}$	$\sigma_y \leftarrow$ Compute(PK, σ_x)
Verif	Verify(SK,σ_y, τ_x)		

— 4 —
Conclusion

Some text…

Articles

[Len97] A. K. Lenstra. "Using cyclotomic polynomials to construct efficient discrete logarithm cryptosystems over finite fields". In: *Information Security and Privacy* (1997), pp. 126–138.

[CS03] Jan Camenisch and Victor Shoup. "Practical verifiable encryption and decryption of discrete logarithms". In: *Annual International Cryptology Conference* (2003), pp. 126–144.

[PR07] Manoj Prabhakaran and Mike Rosulek. "Rerandomizable rcca encryption". In: *Annual International Cryptology Conference* (2007), pp. 517–534.

[Anoa] Anonymous. "Practical Groth16 Aggregation". In: (). URL: `https://docs.zkproof.org/pages/standards/accepted-workshop4/proposal-aggregation.pdf`.

[Anob] Anonymous. "zkSNARKs: R1CS and QAP". In: (). URL: `https://risencrypto.github.io/zkSnarks/`.

[But] Vitalik Buterin. "How to construct a QAP". In: (). URL: `https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649`.

[CG] R. Canetti and J. A. Garay. "Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference". In: ().

[FGP] Dario Fiore, Rosario Gennaro, and Valerio Pastro. "Efficiently Verifiable Computation on Encrypted Data". In: (). URL: `https://eprint.iacr.org/2014/202.pdf`.

[Gro] Jens Groth. "On the Size of Pairing-based Non-interactive Arguments". In: (). URL: `https://eprint.iacr.org/2016/260.pdf`.

[Kos+] JAhmed Kosba et al. "COCO: A Framework for Building Composable Zero-Knowledge Proofs". In: (). DOI: `https://eprint.iacr.org/2015/1093.pdf`.

[Lee+] Jiwon Lee et al. "SAVER : SNARK-friendly, Additively-homomorphic, and Verifiable Encryption and decryption with Rerandomization". In: (). DOI: `https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-saver.pdf`.

[LPR] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On ideal lattices and learning with errors over rings". In: (). URL: `https://eprint.iacr.org/2012/230.pdf`.

[Pan] Alisa Pankova. “Succinct Non-Interactive Arguments from Quadratic Arithmetic Programs”. In: (). URL: `https://courses.cs.ut.ee/MTAT.07.022/2013_fall/uploads/Main/alisa-report`.

[Reg] Oded Regev. “On lattices, learning with errors, random linear codes, and cryptography”. In: (). URL: `https://cims.nyu.edu/~regev/papers/qcrypto.pdf`.