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

Dynamic and Structured Secure Multi-party Computations

Anthony Martinez

<27-28/06/2022>

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:
[Mr] <Pernet Clément>

[June/September] 2022

Abstract

This report gives a client server protocol, based on SNARK technology, which allows to perform a remote polynomial evaluation on the server with a proof that the result is correct. Our aim was to make this protocol dynamic (i.e. the client can change the coefficient of its polynomial without rerunning the protocol from scratch), and to give the possibility for the client to cipher the coefficients of its polynomial to hide its data from the server.

The chapter 1 gives an introduction to understand our problem. The second chapter will describe our protocol based on Groth16 with SNARK where the coefficients of the client polynomial are in clear, the computation time with this protocol is interesting. The third one give a protocol that cipher the coefficients of the client polynomial with the Paillier's cryptosystem, in order to hide the client data from the server. Where we found that this protocol can't be used because it takes too much time to compute. And the chapter 4 is an update of the protocol given in the chapter 2 where we made it dynamic for the client, but the coefficients are in clear too.

Contents

— 1 —
Introduction

1.1 Context of this work

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

For example, consider an instance (which we'll call client) that wants to compute if an element is part of a set. For this it can create a polynomial composed of the elements of this set and evaluate this polynomial at a point. According to the result we can know if the evaluation point is part of the set or not.

If we take the following set: E=$\{$**e1**,**e2**,**e3**,**e4**$\}$ And we construct a polynomial that is equal to zero each time that the evaluation point is part of the set $P(x) = (\mathbf{e1}-x)(\mathbf{e2}-x)(\mathbf{e3}-x)(\mathbf{e4}-x)$. Now if we want to know if a number **x** is part of this set we can test the equality $P(x) == 0$, if it's equal to true then **x** is in our set.

A client could have some problem if its set is composed of a large number of elements with large numbers, then its polynomial will also be large and the computation time may become too long for him. So he can be tempted to ask to a server (i.e. another party) to perform the computation for him, but in this case how the client can be sure that the server has done the right computation. Or that the server didn't try to read its data.

To solve this the client could be interested to cipher the coefficients of its polynomial, in order to hide them from the server. Or to have a protocol that gives a proof that the server's computation is correct.

This is the goal of this report, give a protocol were the client could have a proof that the computation of the server is correct and where he can hide its data. And more generally were a party can ask to another party to perform a computation with a proof of the result and a possibility to hide its data.

1.2 Definition of a secret polynomial evaluation

In the case where the client wants to make a calculation to a server to obtain a result it will have different requirements for the server:

- The client wants a proof that the result given by the server is the right one
- If the data are sensitive, the calculation made by the server must not allow it to learn anything about the client's data
- The time it takes for the client to verify the proof of the server should be as short as possible
- The proof must be dynamic, i.e. any change made on the client polynomial must be done on the server side without having to start the protocol from scratch
- More generally, the proof can be verified publicly by anyone who has access to it

It's those types of problem to which we wish to bring solutions in this work.

1.3 How we want to solve a secret polynomial evaluation

The proposed methodology in this work is to focus on algebraic problems involving polynomial arithmetic and linear algebra. The main tools are zero knowledge proof techniques and homomorphic cryptography and those need to be adapted to efficiently take into account modifications of the assumptions (dynamicity) during the protocols.

Definition 1 ***Zero-Knowledge Proof*** *enables a prover to convince a verifier that a statement is true without revealing anything else. It has three 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 doesn't reveal anything but the truth of the statement, in particular it doesn't reveal the prover's witness*

Definition 2 ***Homomorphic encryption*** *is a form of encryption that permits users to perform computations on its encrypted data without decrypting it. We define an encryption system composed of two functions* ***E****(x) and* ***D****(x) where the function* ***E*** *allows to encrypt a message and the function* ***D*** *allows to decrypt a message. We define two types of homomorphic property:*

- ***Additive homomorphic****: if for two plaintexts* ***m1*** *and* ***m2*** *we have* $\boldsymbol{E(m1) \bigoplus E(m2) = E(m1 + m2)}$
- ***Multiplication homomorphic****: if for two plaintexts* ***m1*** *and* ***m2*** *we have* $\boldsymbol{E(m1) \bigodot E(m2) = E(m1 \cdot m2)}$

Among the existing techniques we will use and compare the Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (**zkSNARK**) proof, we'll explain how it works later, and Fully Homomorphic Encryption (**FHE**), our encryption system has the two homomorphic properties listed in definition 2. And the idea is to develop a protocol using those techniques to answer our problems listed in section 1.2.

1.4 Formalization of a secret polynomial evaluation

The environment we are going to use is composed of a client and a server, where the client wants to do a polynomial evaluation on this server. We can consider three types of server:

- **Honest**: the server gives the good result and don't try to do anything with the given data
- **Honest but curious**: it gives the good result but it'll try to learn things about data
- **Untrusted**: it'll try to cheat by giving bad results and analyse our data

These different types of servers require different types of protocols, e.g. an honest server doesn't require a proof when it sends us the result while a dishonest server does. For the rest of this report we will consider that the server will be either **honest but curious** or **untrusted**.

On the client side we consider two types:

- **Clear data**: the client doesn't care if the server sees his data and just want the good result
- **Hidden data**: the client wants the good result and the server don't learn anything about his data

These different types of servers and clients will influence our conclusions about our results.

1.5 Proposed solution for a secret polynomial evaluation

To solve our problem we will create a protocol to do our polynomial evaluation on a server. This protocol must allow the client to have a proof that the result given by the server is correct, if possible to make this proof verifiable by any entity having access to it, as well as the possibility of encrypting its data to hide it from the server and if possible to make this protocol dynamic. We define our protocol as dynamic if the client can change a coefficient of its polynomial without having to run the protocol again from the beginning. It will be composed of three steps:

- **Setup**: the client or a trust party will generate the necessary elements for the protocol to run
- **Eval**: the client sends to the server the evaluation point of the polynomial, the server must compute the result and generate the proof that it will send back to the client
- **Verif**: the client checks with the proof, sent to the Eval part by the server, if the result is correct

Table 1.1: Representation of the steps of our protocol

	Client	**Verifier**	**Communications**	**Server**
Setup	The client has a polynomial P Generation of the set of elements α needed for the proof		$\xrightarrow{P,\alpha}$	
Eval	Select **x** the point on which we want to evaluate **P**		$\xrightarrow{x}$ $\xleftarrow{y,\pi}$	Compute **y**=P(**x**) Compute π with α a proof of **y**
Verif	Check with π that **y** is correct If the proof is correct the server has give the good result otherwise the client must not trust the server anymore			

1.6 Evaluation measures of a protocol in this report

To be able to compare our protocols between them and to know which ones are advantageous according to the techniques we are going to use, we have defined evaluation criteria:

- **Computation time**: how long it takes to perform each part of the protocol
- **Extra storage**: the data we need to store during the protocol
- **Communication volume**: the amount of communication required for the protocol
- **Number of communication turns**: how many communication turns we need to perform in order to achieve the protocol

1.7 Existing protocols "for polynomial evaluation"

Before starting to create or change protocols we looked for existing ones, to see if we could use them or adapt them to our needs. For some of them we felt that they were not adapted to our needs like SAVER [Lee+], for this one the data could be encrypted, but the decryption was done by calculating the discrete log. In our case we want the coefficients of our polynomial to be part of a large set where the discrete log will be impossible to calculate. So this protocol is not practicable . Some other protocols seem's interesting, but we haven't had the time to work on them like FIORE 2014 [FGP] and FIORE 2020.

To get an idea of the protocols we have studied and their specificities, we have made a table with those criteria:

- **Polynomial coefficient clear/ciphered**: in the protocol, are the coefficients of our polynomial encrypted or not
- **Input clear/ciphered**: is the evaluation point of our polynomial encrypted or not in the protocol
- **Verification public/private**: the verification of the proof given by the server in the protocol can be done publicly or not
- **Method**: what is the technique used in this protocol to create and verify the proof

- **Library**: is there a library that implements this protocol and if so which one
- **Dynamic**: is this protocol dynamic

Table 1.2: Summary of the different protocols we have studied with their specificity

Protocol	Polynomial coefficient clear/ciphered	Input clear/ciphered	Verification Public/private	Method	Library	Dynamic
snarkGroth16	clear	clear	public	SNARK	libsnark	No
COCO	ciphered	clear	public	SNARK	jsnark	No
SAVER	clear	ciphered	public	SNARK		No
FIORE 2014	ciphered	clear	private	FHE	homomorphic-authent	No
					tication-library	No
FIORE 2020	ciphered	clear		FHE and SNARK		No
VESPO	ciphered	clear	private	Paillier and Pairing		Yes

1.8 Plan of this report

The objective of this report is to compare SNARK-based protocols with the VESPO protocol.

In a first chapter the snarkGroth16 protocol will be detailed, with our adaptation to make a polynomial evaluation. We'll give our algorithms with the results we obtained during our tests.

As you will see the protocol of our first chapter doesn't allow to encrypt the coefficients of our polynomial, so the client can't hide its data to the server. The second chapter aims to encrypt the coefficients of our polynomial while applying the snarkGroth16 protocol. We will also detail our algorithms with the results we obtained.

And finally, chapter 3 will explain how to make our protocol dynamic.

— 2 —

ZKSNARK with Groth16

In this chapter we focus on the Groth16 protocol using zkSNARK, which we'll call snark-Groth16 protocol, presented in table 1.2 and we will present how it works in detail. Next we will create a protocol based on snarkGroth16 to generate a proof of the result computed by the server, and we'll present our contributions to this protocol.

2.1 Technique used by snarkGroth16

snarkGroth16 uses zkSNARK to generate a proof, it's a type of cryptographic proof protocol that reveals no information about the knowledge we try to prove. Here is the meaning of each letters:

- **ZK** for **Zero Knowledge**, the user's information 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

2.1.1 How zkSNARK works

In order to construct the proof, zkSNARK use polynomials evaluation, the idea is to convert our problem into a "Rank-1 Constraint System" **R1CS**.

Definition 3 *A **R1CS** is a sequence of groups of three vectors (**u**, **v**, **w**) of size **m**. And the solution to an **R1CS** is a vector **s**, where **s** must satisfy the equation "(**s**·**u**) (**s**·**v**) - (**s**·**w**) = 0" where "·" is the dot product.*

For our case, our problem is to do a polynomial evaluation, if we take the polynomial **P**, of degree **d**, that we want to evaluate at point **r**. To perform the evaluation we use Horner's method.

Definition 4 ***Horner's method****, if we have a polynomial* **P** *of degree* **d** *with coefficients* $[p_i]_{i=0}^{d}$*. So* ***P****(x)=*$p_0 + p_1 * x + ... + p_d * x^d$*.*
We can write our polynomial as follows ***P****(x)=*$p_0 + x(p_1 + x(..(p_{d-1} + p_d * x)..))$*.*

Example 1 *So if we want to evaluate* ***P*** *at point* ***r*** *we can create the following* ***R1CS****, with* a_i *the differents variable in our* ***R1CS*** *and* a_1 *corresponding to the input (i.e.* a_1*=****r****) and* a_2 *corresponding to the output (i.e.* a_2*=****P(r)****). And each vector* ***u****,****v*** *and* ***w*** *corresponding to the* a_i *variable [*a_0*,* a_1*, ...* a_m*], for every equations:*

Table 2.1: Our set of equations with their corresponding **u**, **v**, **w**

Equations	**u**	**v**	**w**
$a_3 \leftarrow p_d * a_1$	$[p_d$, 0, 0, ..., 0]	[0, 1, 0, ..., 0]	[0, 0, 0, 1, 0, ..., 0]
$a_4 \leftarrow (a_3 + p_{d-1}) * a_1$	$[p_{d-1}$, 0, 0, 0, 1, ..., 0]	[0, 1, 0, ..., 0]	[0, 0, 0, 0, 1, 0, ..., 0]
..	..	..	..
$a_{d+2} \leftarrow (a_{d+1} + p_1) * a_1$	$[p_1$, 0, ..., 1, 0]	[0, 1, 0, ..., 0]	[0, 0, ..., 1]
$a_2 \leftarrow (a_{d+2} + p_0) * 1$	$[p_0$, 0, 0, ..., 1]	[1, 0, 0, ..., 0]	[0, 0, 1, 0, ..., 0]

With this construction we need to store all our a_i *variables with i ranging from 0 to* ***d****+2, with a particularity that* a_0 *is a constant equal to 1. So our vectors* ***u****,****v*** *and* ***w*** *will have size* ***d****+3, and we'll have* ***d****+1 set of triplet (****u****,****v****,****w****). So the number of values we will need to store for an* ***R1CS*** *evaluating a polynomial of degree* ***d*** *at a point* ***r*** *is (3(****d****+1))(****d****+3).*

Then we convert the **R1CS** into a "Quadratic Arithmetic Program" (**QAP**) with Lagrange interpolation.

Definition 5 ***Lagrange interpolation****: given a set of k + 1 data points* $(x_0, y_0), (x_1, y_1), .., (x_k, y_k)$ *where no two* x_j *are the same. The interpolation polynomial in the Lagrange form is a linear combination* $L(x) = \sum_{j=0}^{k} y_j l_j(x)$ *of lagrange basis polynomial* $l_j(x) = \prod_{0 \leq m \leq k, m \neq j} \frac{x - x_m}{x_j - x_m}$ *where* $0 \leq j \leq k$*.*

Example 2 *This example is just to show how the Lagrange interpolation is done. For this we consider an* ***R1CS*** *with a set of triples (****u****,****v****,****w****), and we put* α *the set of vectors* ***u*** *with* u_0 *the first* ***u*** *vector,* u_1 *the second... to* u_d*. Let us say that* α*=* $\begin{bmatrix} u_0 \\ u_1 \\ \cdots \\ u_d \end{bmatrix} = \begin{bmatrix} u_{0,0}, u_{0,1}, \ldots u_{0,m} \\ u_{1,0}, u_{1,1}, \ldots u_{1,m} \\ \cdots \\ u_{d,0}, u_{d,1}, \ldots u_{d,m} \end{bmatrix}$ *Then we take each column of* α *and convert it into a polynomial of degree* ***d****, so we will have a total of* ***m****+1 polynomials of degree* ***d****. Let's define our polynomial set A with* A_0 *the polynomial corresponding to colone 0,* A_1 *the polynomial corresponding to colone 1, up to* A_m*. And we define* A_j *= LagrangeInterpolation((0,* $u_{0,j}$*), (1,* $u_{1,j}$*), ..., (d,* $u_{d,j}$*)). In a similar way we define B with all the vectors* ***v*** *and C with all the vectors* ***w****.*

Definition 6 ***Quadratic Arithmetic Program (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, a_0 the constant value 1, 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 off degree n, such that*

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

In comparison with the dimensions of our **R1CS** given in example 1, the number of polynomials A, B, C corresponds to the number of variables in our **R1CS**, i.e. **d**+3 with **d** the degree of our polynomial **P**. And the degree of our polynomial A_i, B_i, C_i is **d**-1.

If we consider a **QAP** with a set of polynomials $\{A_i(X),\ B_i(X),\ C_i(X)\}_{i=0}^m$ as given in definition 6, and a set $[a_i]_{i=0}^m$ that correspond to a potential solution. To check this solution we take t $\in F_{d^\circ \leq n}[X]$, where t = $(\sum_{i=0}^m a_i A_i(x)) * (\sum_{i=0}^m a_i B_i(x)) - \sum_{i=0}^m a_i C_i(x)$. And we divide t by the polynomial Z = (x-1)(x-2)...(x-n) $\in F_{d^\circ \leq n}[X]$ and we check that this division has no remainder. We define the polynomial H =t/Z $\in F_{d^\circ \leq n}[X]$.
To have more explanation you can read this paper [Pan]. And there are also these two websites that give good examples [But] and [Ano].

2.1.2 Principle of the snarkGroth16 protocol

[Gro] The goal of the protocol is to know if a person knows a secret without revealing it. In the case where a client has a secret and wants to know if a server knows this secret. Using the snarkGroth16 protocol, it will generate a QAP that is verified by knowing the secret **s**, then it will create a "verification key" **vk** and a "proof key" **pk**. With these two keys we can generate and verify a proof. Any person possessing the key **vk** can verify the proof generated with the key **pk**. This key pair must be generated by the client or a trusted party. Then the client sends the key **pk** to the server and the server generates a proof with **pk** and **s** that it'll send to the client. Then the client will be able to check if the server knows **s** or not. This algorithm is sound, complete and can be zero-knowledge (ZK). And we'll use it for ours protocol.

Table 2.2: Informal definition of the snarkGroth16 protocol

	Client	**Verifier**	**Communications**	**Prover**
Setup	We have a secret **s** Construct the **QAP Q** that check the secret **s** Generate **vk** and **pk** from Q		$\xrightarrow{\mathbf{pk},x,\mathbf{Q}}$	
Eval			$\xleftarrow{\pi}$	Compute the proof π with **pk** and Q
Verif	check with **vk** if π is correct We know that the prover knows the secret **s** Otherwise we suppose that he didn't know it			

2.2 Design of our protocol based on snarkGroth16

Section 2.1 helped us understand how SNARKs work. Using this technology, we will implement a protocol that allows a client to evaluate a polynomial on a server, with a proof that the result is correct.

To be able to implement this protocol and conduct tests we will use a library called libsnark [lab], and create our protocol over it. Our programs and results will be available on the github snark-protocol-experiment directory [rus].

We will detail the snarkGroth16 protocol in section 2.2.1, then we will talk about our implementation in section 2.2.2 and finally we will detail the results we obtained with our protocol in section 2.2.3 according to the criteria we defined in section 1.6.

2.2.1 snarkGroth16 protocol algorithm

The client has a polynomial **P**$\in F_{d^\circ \leq d}[X]$, of degree **d** $\in \mathbb{Z}$ with the coefficient $[p_i]_{i=0}^{d}$ where $p_i \in F$. And he wants to evaluate his polynomial at some point on a server.

First the client must generate the **R1CS** corresponding to **P**, then using this **R1CS** it will generate a **QAP** composed of the polynomials $\{A_i(X), B_i(X), C_i(X)\}_{i=0}^{m}, Z(X)$ with $A_i(X), B_i(X), C_i(X), Z(X) \in F_{d^\circ \leq n}[X]$ (with m=d+4 and n=d-1 if we generate the **R1CS** as in the example 1), where the client knows the value $a_{0..l,l+1...m} \in F$, where $l, m \in \mathbb{N}$, to solve it.

Recall that a_0 correspond to the constant 1, the $a_{1..l}$ correspond to the inputs and outputs of our **R1CS** and the $a_{l+1...m}$ correspond to the intermediate values that we will call witness.

And as we are doing a polynomial evaluation we have one input and one output in our **R1CS**, so l is 2 with a_1 the input of our **R1CS** and a_2 the output.

For the snarkGroth16 protocol we also need a bilinear map that we will use to generate our keys.

Definition 7 ***Bilinear map*** *is defined by seven elements* $(p, G_1, G_2, G_T, e, g, h)$ *such that:*

- G_1*,* G_2*,* G_T *are groups of prime order p, with p*$\in \mathbb{N}$
- $e : G_1 \times G_2 \to G_T$
- *g is a generator of* G_1
- *h is a generator of* G_2
- *e(g,h) is a generator of* G_T
- $e(g^a, h^b) = e(g,h)^{ab}$

Afterwards we will call **R** the variable containing our **QAP** and a bilinear map as given in definition 7, so $\mathbf{R} = \{p, e, g, h, l, m, \{A_i(X), B_i(X), C_i(X)\}_{i=0}^{m}, Z(X)\} \in \mathbb{N} \times (G_1 \times G_2 \to G_T) \times G_1 \times G_2 \times \mathbb{N} \times \mathbb{N} \times ((F_{d^\circ \leq n}[X])^m)^3 \times F_{d^\circ \leq n}[X]$. Using the variable **R** the snarkGroth16 protocol defines three functions Setup, Prove and Verify such that:

1. Setup(**R**) → (**pk**, **vk**)

Input: $\mathbf{R} \in \mathbb{N} \times (G_1 \times G_2 \to G_T) \times G_1 \times G_2 \times \mathbb{N} \times \mathbb{N} \times (F_{d^\circ \leq n}[X])^m)^3 \times F_{d^\circ \leq n}[X]$

Output: (**pk**, **vk**)

- **pk** $\in G_1^4 \times G_1^{n+1} \times G_1^{m-l} \times G_1^{n+1} \times G_2^3 \times G_2^{n+1}$
- **vk** $\in G_1^{l+1} \times G_1 \times G_2^3$

Description: This function will be called by the client or a trusted party to generate the **pk** and **vk** keys that will be used in the rest of the protocol

2. Prove(**R**, **pk**, $a_{0...l}, a_{l+1...m}) \to \pi$

Input: (**R**, **pk**, $a_{0...l}, a_{l+1...m}$)

- $\mathbf{R} \in \mathbb{N} \times (G_1 \times G_2 \to G_T) \times G_1 \times G_2 \times \mathbb{N} \times \mathbb{N} \times (F_{d^\circ \leq n}[X])^m)^3 \times F_{d^\circ \leq n}[X]$
- **pk** $\in G_1^4 \times G_1^{n+1} \times G_1^{m-l} \times G_1^{n+1} \times G_2^3 \times G_2^{n+1}$
- $a_{0...l}$ with $a_i \in F$
- $a_{l+1...m}$ with $a_i \in F$

Output: $\pi \in G_1 \times G_2 \times G_1$

Description: This function will be called by the server to prove that it knows all the a_i

3. Verify(**R**, **vk**, $a_{0...l}$, π) $\to$ True/False

Input: (**R**, **vk**, $a_{0...l}$, π)

- $\mathbf{R} \in \mathbb{N} \times (G_1 \times G_2 \to G_T) \times G_1 \times G_2 \times \mathbb{N} \times \mathbb{N} \times (F_{d^\circ \leq n}[X])^m)^3 \times F_{d^\circ \leq n}[X]$
- **vk** $\in G_1^{l+1} \times G_1 \times G_2^3$
- $a_{0...l}$ with $a_i \in F$
- $\pi \in G_1 \times G_2 \times G_1$

Output: True or False $\in \mathbb{B}$

Description: Anyone who wants to check the computation and the proof generated by the function **Prove** can call this function. Output True if π = Prove(**R**, **pk**, $a_{0...l}, a_{l+1...m}$) else False.

Here are the algorithms of the different functions specified above. To simplify the writing of the algorithms when we use our bilinear map we will note:

- g^a as $[a]_1$
- h^a as $[a]_2$
- $e(g^a, h^b)$ as $[ab]_T$

Algorithm 1 Setup(**R**)

```
Input: $\mathbf{R} = \{p, e, g, h, l, m, \{A_i(X), B_i(X), C_i(X)\}_{i=0}^{m}, Z(X)\}$:
    • p,l,m $\in \mathbb{N}$
    • e $\in (G_1 \times G_2 \rightarrow G_T)$
    • g $\in G_1$
    • h $\in G_2$
    • $\{A_i(X), B_i(X), C_i(X)\}_{i=0}^{m} \in (F_{d^\circ \leq n}[X])^m$
    • $Z(X) \in F_{d^\circ \leq n}[X]$
Output: (pk, vk)
    • pk $\in G_1^4 \times G_1^{n+1} \times G_1^{m-l} \times G_1^{n+1} \times G_2^3 \times G_2^{n+1}$
    • vk $\in G_1^{l+1} \times G_1 \times G_2^3$
Ensure: Verify(R, vk, $a_{0...l}$, Prove(R, pk, $a_{0...l}, a_{l+1...m}$)) will output True if the $a_i$ solves the
    QAP in R else it'll output False
    $\alpha, \beta, \gamma, \delta, s \xleftarrow{\$} F^*$
    $\alpha_1 = [\alpha]_1$, $\beta_1 = [\beta]_1$, $\gamma_1 = [\gamma]_1$, $\delta_1 = [\delta]_1 \in G_1$
    $\mathbf{S1} = \{[s^i]_1\}_{i=0}^{n} \in G_1^{n+1}$
    $\mathbf{Sa1} = \{[\frac{\beta A_i(s) + \alpha B_i(s) + C_i(s)}{\delta}]_1\}_{i=l+1}^{m} \in G_1^{m-l}$
    $\mathbf{Sb1} = \{[\frac{s^i Z(s)}{\delta}]_1\}_{i=0}^{n} \in G_1^{n+1}$
    $\mathbf{Sc1} = \{[\frac{\beta A_i(s) + \alpha B_i(s) + C_i(s)}{\gamma}]_1\}_{i=0}^{l} \in G_1^{l+1}$
    $\beta_2 = [\beta]_2$, $\gamma_2 = [\gamma]_2$, $\delta_2 = [\delta]_2 \in G_2$
    $\mathbf{S2} = \{[s^i]_2\}_{i=0}^{n} \in G_2^{n+1}$
    vk= $(\mathbf{Sc1}, \alpha_1, (\gamma_2, \beta_2, \delta_2)) \in G_1^{l+1} \times G_1 \times G_2^3$
    pk= $((\alpha_1, \beta_1, \gamma_1, \delta_1), \mathbf{S1}, \mathbf{Sa1}, \mathbf{Sb1}, (\beta_2, \gamma_2, \delta_2), \mathbf{S2}) \in G_1^4 \times G_1^{n+1} \times G_1^{m-l} \times G_1^{n+1} \times G_2^3 \times G_2^{n+1}$
    return (vk, pk) $\in G_1^{l+1} \times G_1 \times G_2^3 \times G_1^4 \times G_1^{n+1} \times G_1^{m-l} \times G_1^{n+1} \times G_2^3 \times G_2^{n+1}$
```

After the client setup can send the **pk** key to the prover and publish **vk**.

Algorithm 2 Prove(**R**, **pk**, $a_{0...l}, a_{l+1...m}$)

Input: (**R**, **pk**, $a_{0...l}, a_{l+1...m}$)

- **R** $= \{p, e, g, h, l, m, \{A_i(X), B_i(X), C_i(X)\}_{i=0}^{m}, Z(X)\}$:
  - p,l,m $\in \mathbb{N}$
  - e $\in (G_1 \times G_2 \to G_T)$
  - g $\in G_1$
  - h $\in G_2$
  - $\{A_i(X), B_i(X), C_i(X)\}_{i=0}^{m} \in (F_{d^\circ \leq n}[X])^m$
  - $Z(X) \in F_{d^\circ \leq n}[X]$
- **pk**$= ((\alpha_1, \beta_1, \gamma_1, \delta_1), \mathbf{S1}, \mathbf{Sa1}, \mathbf{Sb1}, (\beta_2, \gamma_2, \delta_2), \mathbf{S2})$:
  - $(\alpha_1, \beta_1, \gamma_1, \delta_1) \in G_1^4$
  - $\mathbf{S1} \in G_1^{n+1}$
  - $\mathbf{Sa1} \in G_1^{m-l}$
  - $\mathbf{Sb1} \in G_1^{n+1}$
  - $(\beta_2, \gamma_2, \delta_2) \in G_2^3$
  - $\mathbf{S2} \in G_2^{n+1}$
- $a_{0...l}$, $a_{l+1...m}$ with $a_i \in F$

Output: $\pi \in G_1 \times G_2 \times G_1$

Ensure: If (**vk**, **pk**) == Setup(**R**) then Verify(**R**, **vk**, $a_{0...l}$, Prove(**R**, **pk**, $a_{0...l}, a_{l+1...m}$)) will output True if the a_i solves the **QAP** in **R** else it'll output False

```
r, k$\xleftarrow{\$}$F
$\mathbf{Aa} = \sum_{i=0}^{m} a_i A_i \in F_{d^\circ \leq n}[X]$
$\mathbf{Ba} = \sum_{i=0}^{m} a_i B_i \in F_{d^\circ \leq n}[X]$
$\mathbf{Ca} = \sum_{i=0}^{m} a_i C_i \in F_{d^\circ \leq n}[X]$
U = $\alpha_1 (\Pi_{i=0}^{n} \mathbf{S1}^{\mathbf{Aa}_i})(\delta_1)^r \in G_1$
V1= $\beta_1 (\Pi_{i=0}^{n} \mathbf{S1}^{\mathbf{Ba}_i})(\delta_1)^k \in G_1$
V2= $\beta_2 (\Pi_{i=0}^{n} \mathbf{S2}^{\mathbf{Ba}_i})(\delta_2)^k \in G_2$
We compute $t \in F_{d^\circ \leq n}[X]$ such that t = $(\sum_{i=0}^{m} a_i A_i)(\sum_{i=0}^{m} a_i B_i) - \sum_{i=0}^{m} a_i C_i$
And then $H = t/Z \in F_{d^\circ \leq n}[X]$
$S = \Pi_{i=l+1}^{m} (\mathbf{Sa1}_i)^{a_i} \in G_1$
W = $\frac{S(\Pi_{i=0}^{n} (\mathbf{Sb1}_i)^{H_i}) U^k \mathbf{V1}^k}{{\delta_1}^{rk}} \in G_1$
$\pi = (U, \mathbf{V2}, W) \in G_1 \times G_2 \times G_1$
return $\pi$
```

```
Algorithm 3 Verify($\mathbf{R}$, $\mathbf{vk}$, $a_{0...l}$, $\pi$)
    Input: ($\mathbf{R}$, $\mathbf{vk}$, $a_{0...l}$, $\pi$)
    • $\mathbf{R} = \{p, e, g, h, l, m, \{A_i(X), B_i(X), C_i(X)\}_{i=0}^{m}, Z(X)\}$:
        – p,l,m $\in \mathbb{N}$
        – e $\in (G_1 \times G_2 \rightarrow G_T)$
        – g $\in G_1$
        – h $\in G_2$
        – $\{A_i(X), B_i(X), C_i(X)\}_{i=0}^{m} \in (F_{d^\circ \leq n}[X])^m$
        – $Z(X) \in F_{d^\circ \leq n}[X]$
    • $\mathbf{vk}$= $(\mathbf{Sc1}, \alpha_1, (\gamma_2, \beta_2, \delta_2))$:
        – $\mathbf{Sc1} \in G_1^{l+1}$
        – $\alpha_1 \in G_1$
        – $(\gamma_2, \beta_2, \delta_2) \in G_2^3$
    • $a_{0...l}$ with $a_i \in F$
    • $\pi = (U, \mathbf{V2}, W)$:
        – $U \in G_1$
        – $\mathbf{V2} \in G_2$
        – $W \in G_1$
    Output: True or False $\in \mathbb{B}$
Ensure: If ($\mathbf{vk}$, $\mathbf{pk}$) == Setup($\mathbf{R}$) and $\pi$==Prove($\mathbf{R}$, $\mathbf{pk}$, $a_{0...l}, a_{l+1...m}$) then Verify($\mathbf{R}$, $\mathbf{vk}$, $a_{0...l}$,
    $\pi$) will output True if the $a_i$ solves the QAP in $\mathbf{R}$ else it'll output False
$Y = \Pi_{i=0}^{l}(\mathbf{Sc1}_i)^{a_i} \in G_1$
if $e(U, \mathbf{V2}) == e(\alpha_1, \beta_2)e(Y, \gamma_2)e(W, \delta_2)$ then
    return True
else
    return False
end if
```

Proof that the protocol is sound

Let us show that the protocol composed of algorithms 1, 2 and 3 verifies the equality of our **QAP** which is the equation:

$$(\sum_{i=0}^{m} a_i A_i(s))(\sum_{i=0}^{m} a_i B_i(s)) = \sum_{i=0}^{m} a_i C_i(s) + H(s)Z(s) \tag{2.1}$$

Now let's detail the calculation of the if statement in the function Verify:
$e(U,\mathbf{V2}) == e(\alpha_1,\beta_2)e(Y,\gamma_2)e(W,\delta_2)$

$$\begin{aligned}
e(U,\mathbf{V2}) &= e((\alpha_1\Pi_{i=0}^{m}\mathbf{S1}_i^{\mathbf{Aa}_i}(\delta_1)^r),(\beta_2\Pi_{i=0}^{m}\mathbf{S2}_i^{\mathbf{Ba}_i}(\delta_2)^k)) \\
&= [(\alpha+\sum_{i=0}^{m}a_iA_i(s)+r\delta)(\beta+\sum_{i=0}^{m}a_iB_i(s)+k\delta)]_T \\
&= [(\alpha+\sum_{i=0}^{m}a_iA_i(s)+r\delta)(\beta+\sum_{i=0}^{m}a_iB_i(s)+k\delta)]_T \\
&= [\alpha\beta+\alpha(\sum_{i=0}^{m}a_iB_i(s))+k\alpha\delta+\beta(\sum_{i=0}^{m}a_iA_i(s))+(\sum_{i=0}^{m}a_iA_i(s))(\sum_{i=0}^{m}a_iB_i(s))+k\delta(\sum_{i=0}^{m}a_iA_i(s))+r\delta\beta+r\delta(\sum_{i=0}^{m}a_iB_i(s)+rk\delta\delta]_T
\end{aligned} \tag{2.2}$$

$$\begin{aligned}
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_iA_i(s)+\alpha\sum_{i=0}^{l}a_iB_i(s)+\sum_{i=0}^{l}a_iC_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_iA_i(s)+\alpha\sum_{i=0}^{l}a_iB_i(s)+\sum_{i=0}^{l}a_iC_i(s)}{\gamma})\gamma+(S+\frac{H(s)Z(s)}{\delta}+Uk+rV-rk\delta)\delta]_T \\
&= [\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)+(S+Uk+rV-rk\delta)\delta+H(s)Z(s)]_T \\
&= [\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)+(S+Uk+rV-rk\delta)\delta+H(s)Z(s)]_T \\
&= [\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)+(\frac{\beta\sum_{i=l+1}^{m}a_iA_i(s)+\alpha\sum_{i=l+1}^{m}a_iB_i(s)+\sum_{i=l+1}^{m}a_iC_i(s)}{\delta} \\
&\quad +(\alpha+\sum_{i=0}^{m}a_iA_i(s)+r\delta)k+r(\beta+\sum_{i=0}^{m}a_iB_i(s)+k\delta)-rk\delta)\delta+H(s)Z(s)]_T \\
&= [\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)+\beta\sum_{i=l+1}^{m}a_iA_i(s)+\alpha\sum_{i=l+1}^{m}a_iB_i(s)+\sum_{i=l+1}^{m}a_iC_i(s) \\
&\quad +((\alpha+\sum_{i=0}^{m}a_iA_i(s)+r\delta)k+r(\beta+\sum_{i=0}^{m}a_iB_i(s)+k\delta)-rk\delta)\delta+H(s)Z(s)]_T \\
&= [\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)+((\alpha+\sum_{i=0}^{m}a_iA_i(s)+r\delta)k+r(\beta+\sum_{i=0}^{m}a_iB_i(s)+k\delta)-rk\delta)\delta+H(s)Z(s)]_T \\
&= [\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\alpha+k\sum_{i=0}^{m}a_iA_i(s)+kr\delta+r\beta+r\sum_{i=0}^{m}a_iB_i(s)+rk\delta-rk\delta)\delta+H(s)Z(s)]_T \\
&= [\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)]_T
\end{aligned} \tag{2.3}$$

If we test the equality of our equations 2.6 and 2.7 we obtain:

$$\begin{aligned}
&[\alpha\beta+\alpha(\sum_{i=0}^{m}a_iB_i(s))+k\alpha\delta+\beta(\sum_{i=0}^{m}a_iA_i(s))+(\sum_{i=0}^{m}a_iA_i(s))(\sum_{i=0}^{m}a_iB_i(s))+k\delta(\sum_{i=0}^{m}a_iA_i(s))+r\delta\beta+r\delta(\sum_{i=0}^{m}a_iB_i(s)+rk\delta\delta]_T = [\alpha\beta+\beta\sum_{i=0}^{m}a_iA_i(s) \\
&\quad +\alpha\sum_{i=0}^{m}a_iB_i(s)+\sum_{i=0}^{m}a_iC_i(s)+k\delta\alpha+k\delta\sum_{i=0}^{m}a_iA_i(s)+kr\delta\delta+r\delta\beta+r\delta\sum_{i=0}^{m}a_iB_i(s)+H(s)Z(s)]_T \\
&\Rightarrow [(\sum_{i=0}^{m}a_iA_i(s))(\sum_{i=0}^{m}a_iB_i(s))=\sum_{i=0}^{m}a_iC_i(s)+H(s)Z(s)]_T
\end{aligned}$$

Our result corresponds to the one we expected. One could be tempted to think that the snarkGroth16 protocol can work without the bilinear map, but in this case the server will know the values of α,β,δ,s. And by using them the server can build a false proof that would make any verifier believe that it knows the secret. More exactly the server must choose the values of U, V and W in this way to build a false proof:

Pick randomly U,V $\in F$ Compute W=$\frac{UV-\alpha\beta-\sum_{i=0}^{l}(a_i(\beta A_i(s)+\alpha B_i(s)+C_i(s)))}{\delta}$ The if statement in the function Verify(**R**, **vk**, $a_{0...l}$, π) without the bilinear map is equal to $UV == \alpha\beta + Y\gamma + W\delta$. So we have:

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

Therefore our condition in our if statement will be true and the Verify(**R**, **vk**, $a_{0...l}$, π) function will return the value true, so every verifier will think that the server knows the secret. In conclusion without the bilinear map the snarkGroth16 protocol could not work.

Table 2.3: Summarizes of the snarkGroth16 protocol process

	Client	**Communications**	**Prover**
Setup	With P$\in F[X]$ Compute R corresponding to P R$= \{(p,g,h,e) \in \mathbb{Z} \times G_1 \times G_2 \times G_T$ $l,m, \in \mathbb{Z}, \{A_i,B_i,C_i\}_{i=0}^{m}, \in F_{d^\circ \leq n+1}[X]$ $Z \in F_{d^\circ \leq n+1}[X]\}$ $\alpha,\beta,\gamma,\delta,s \leftarrow^{\$} F^*$ $\alpha_1 = [\alpha]_1, \beta_1 = [\beta]_1, \gamma_1 = [\gamma]_1, \delta_1 = [\delta]_1 \in G_1$ $\mathbf{S1} = \{[s^i]_1\}_{i=0}^{n} \in G1^n$, $\mathbf{Sa1} = \{[\frac{\beta A_i(s)+\alpha B_i(s)+C_i(s)}{\delta}]_1\}_{i=l+1}^{m} \in G_1^{m-(l+1)}$ $\mathbf{Sb1} = \{[\frac{s^i Z(s)}{\delta}]_1\}_{i=0}^{n} \in G_1^n$ $\mathbf{Sc1} = \{[\frac{\beta A_i(s)+\alpha B_i(s)+C_i(s)}{\gamma}]_1\}_{i=0}^{l} \in G_1^l$ $\beta_2 = [\beta]_2, \gamma_2 = [\gamma]_2, \delta_2 = [\delta]_2 \in G_2$, $\mathbf{S2} = \{[s^i]_2\}_{i=0}^{n} \in G_2^n$ $pk = ((\alpha_1,\beta_1,\gamma_1,\delta_1), \mathbf{S1}, \mathbf{Sa1}, \mathbf{Sb1}, (\beta_2,\gamma_2,\delta_2)$ $, \mathbf{S2}) \in G_1^4 \times G_1^n \times G_1^{m-(l+1)} \times G_1^n \times G_2^3 \times G_2^n)$ $vk = (\mathbf{Sc1}, \alpha_1, (\gamma_2,\beta_2,\delta_2)) \in G_1^l \times G_1 \times G_2^3$ The client has to store *vk* and can discard everything else	$\xrightarrow{pk,a_{input},R}$	
Eval		$\overset{\pi}{\leftarrow}$	With $pk = ((\alpha_1,\beta_1,\gamma_1,\delta_1)$ $, \mathbf{S1}, \mathbf{Sa1}, \mathbf{Sb1}, (\beta_2,\gamma_2,\delta_2)$ $, \mathbf{S2}) \in (G_1^4 \times G_1^{n+1} \times G_1^{m-l-1} \times$ $G_1^{n+1} \times G_2^3 \times G_2^{n+1})$ And R$= \{(p,g,h,e) \in \mathbb{Z} \times G_1 \times G_2 \times G_T$ $l,m, \in \mathbb{Z}, \{A_i,B_i,C_i\}_{i=0}^{m}, \in F_{d^\circ \leq n+1}^{m}[X]$ $Z \in F_{d^\circ \leq n+1}[X]\}$ With a_{input} compute all the a_i $r,k \overset{\$}{\leftarrow} F*$ $\mathbf{Aa} = \sum_{i=0}^{m} a_i A_i \in F_{d^\circ \leq n+1}[X]$ $\mathbf{Ba} = \sum_{i=0}^{m} a_i B_i \in F_{d^\circ \leq n+1}[X]$ $\mathbf{Ca} = \sum_{i=0}^{m} a_i C_i \in F_{d^\circ \leq n+1}[X]$ U = $\alpha_1(\Pi_{i=0}^{n} \mathbf{S1}_i^{\mathbf{Aa}_i})(\delta_1)^r \in G_1$ **V1**= $\beta_1(\Pi_{i=0}^{n} \mathbf{S1}_i^{\mathbf{Ba}_i})(\delta_1)^k \in G_1$ **V2**= $\beta_2(\Pi_{i=0}^{n} \mathbf{S2}_i^{\mathbf{Ba}_i})(\delta_2)^k \in G_2$ We compute $t \in F_{d^\circ \leq n+1}[X]$ such that t = $(\sum_{i=0}^{m} a_i A_i)(\sum_{i=0}^{m} a_i B_i) - \sum_{i=0}^{m} a_i C_i$ And then $H = t/Z \in F_{d^\circ \leq n+1}[X]$ $S = \Pi_{i=l+1}^{m} (\mathbf{Sa1}_i)^{a_i} \in G_1$ W = $\frac{S(\Pi_{i=0}^{n}(\mathbf{Sb1}_i)^{H_i})U^k \mathbf{V1}^k}{{\delta_1}^{rk}} \in G_1$ $\pi = (U, \mathbf{V2}, W) \in (G_1 \times G_2 \times G_1)$
Verif	**This function can be run by any verifier** With $\pi = (U, \mathbf{V2}, W) \in (G_1 \times G_2 \times G_1)$ $Y = \Pi_{i=0}^{l} (\mathbf{Sc1}_i)^{a_i} \in G_1$ if $e(U, \mathbf{V2}) == e(\alpha_1,\beta_2)e(Y,\gamma_2)e(W,\delta_2)$: return true else: return False		

In summary, with the snarkGroth16 protocol we can check if a person knows a secret and have a proof of it thanks to algorithms 1, 2 and 3. What we are interested in is significantly different, we need the server to compute the result of our polynomial evaluation and prove that this result is correct. Our adaptation of the protocol with is implementation will be given in chapter 2.2.2.

2.2.2 Implementation of our protocol using libsnark

This part is used to explain some of the code we have implemented to simulate our protocol, it's based on the libsnark library and it can be found on the github snark-protocol-experiment directory [rus].

First we will give the specifications of the functions and objects that we use in libsnark, and then our functions.

Specification of the functions and objects we use in libsnark

To use the snarkGroth16 protocol libsnark has created objects representing the different elements

1. **r1cs_ppzksnark_constraint_system** which store our **R1CS**
2. **r1cs_ppzksnark_proving_key** which store our proving key **pk**
3. **r1cs_ppzksnark_verification_key** which store our verification key **vk**
4. **r1cs_ppzksnark_keypair** which store our two keys object above
5. **r1cs_ppzksnark_proof** which store the proof generated by the function **Prove**
6. **r1cs_ppzksnark_primary_input** which store the $a_{1..l}$
7. **r1cs_ppzksnark_auxiliary_input** which store the $a_{l+1..m}$
8. **protoboard** which will contain our **R1CS** and all the a_i
9. **pb_variable** which will refer to a variable of our R1CS
10. **r1cs_variable_assignment** contains some **pb_variable** with their assignment

And for each of the algorithms we wrote in section 2.2.1, libsnark contains a corresponding function:

1. Algorithm 1: r1cs_ppzksnark_generator(**R1CS**) $\rightarrow$ **keypair**

   Input: R1CS: r1cs_ppzksnark_constraint_system

   Output: keypair: r1cs_ppzksnark_keypair

   Description: This function will be called by the client or a trusted party to generate the **keypair** keys that will be used in the rest of the protocol

2. Algorithm 2: r1cs_ppzksnark_prover(**pk**, **inp**, **aux**) $\rightarrow \pi$

   Input: (**pk**, **inp**, **aux**)

   - **pk**: r1cs_ppzksnark_proving_key
   - **inp**: r1cs_ppzksnark_primary_input
   - **aux**: r1cs_ppzksnark_auxiliary_input

   Output: π: r1cs_ppzksnark_proof

Description: This function will be called by the server to prove that it knows all the **inp** and **aux** with **inp** = $a_{1..l}$ and **aux** = $a_{l+1..m}$

3. Algorithm 3: r1cs_ppzksnark_verifier_strong_IC(**vk**,$\mathbf{a_{1..l}}$, π) → True/False

   Input: (**vk**,$\mathbf{a_{1..l}}$, π)

   - **vk**: r1cs_ppzksnark_verification_key
   - $\mathbf{a_{1..l}}$: r1cs_ppzksnark_primary_input
   - π: r1cs_ppzksnark_proof

   Output: $\mathbb{B}$

   Description: Anyone who wants to check the computation and the proof generated by the function **Prove** can call this function. Output True if π = Prove(**R**, **pk**, $a_{0...l}, a_{l+1...m}$) else False.

With these different functions we have created new ones to simulate our protocol.

Specification of the functions we use to simulate our protocol

In the case of libsnark our bilinear map is created with the elliptic curve **bn128**, each of the coefficients of our polynomial will be in this set that we will call G_{ec128} later.

First we have created a function to generate a polynomial of given degree **d** with it's coefficients in the group G_{ec128}. Then with the obtained result we generate an **R1CS** with Horner's method, as in example 1. And finally we added another function allowing us to calculate the result of our polynomial evaluation with the intermediate values of the **R1CS**. Here are our functions with their specifications:

1. create_polynomials(**d**) → **poly**

   Input: **d**: integer

   Output: **poly**: vector<G_{ec128}>

   Description: Generate our polynomial with **d** coefficients in the group G_{ec128}

2. create_constraint_horner_method(**polynomial**, **ptb**, **d**) → (**in**,**out**)

   Input: (**polynomial**, **ptb**, **d**)

   - **polynomial**: vector<G_{ec128}>
   - **ptb**: protoboard
   - **d**: integer

   Output: (**in**, **out**)

   - **in**: pb_variable
   - **out**: pb_variable

   Description: Generate all our constraints for our **R1CS** in the protoboard **ptb** and give the two variables **in** and **out**. Where **in** will be equal to the evaluation point of our polynomial and **out** to the result.

3. compute_polynomial_witness_output(**ptb**, **pk**) → **var_assignement**

   Input: (**ptb**, **pk**)

   - **ptb**: protoboard
   - **pk**: r1cs_ppzksnark_proving_key

   Output: **var_assignement**: r1cs_variable_assignment

   Description: Calculates for each of our variables pb_variable, present in the protoboard **ptb**, their values which will also be stored in the protoboard **ptb**. Then with the **primary_input()** function we can get the $a_{1..l}$ and with the **auxiliary_input()** function we get the $a_{l+1..m}$.

With the functions already present in libsnark and those we have just added we have created a function to simulate the protocol. The function is in the file "libsnark-experiment/src/ snark_polynomial_in_clear.cpp" in the github directory and is called "test_polynomial_in_clear(integer **d**)", which corresponds to algorithm 4. We only take as parameter the degree of the polynomial **d** because the rest (the coefficients of the polynomial and the evaluation point) are chosen randomly.

Algorithm 4 test_polynomial_in_clear(**d**)

```
Input: d: Integer
Output: True if the result of the server is the good one, False otherwise
Description: simulate the protocol snarkGroth16 on a random polynomial with a random point of evaluation and output true if the calcul of the server is correct else return false.
▷ //————Client phase————//
▷ Create the protoboard which will store our R1CS
protoboard ptb = new Protoboard()
▷ Create a random polynomial
vector<$G_{ec}$> poly ← create_polynomials(d)
▷ Create the R1CS in the protoboard
pb_variable in, pb_variable out ← create_constraint_horner_method(polynomial, ptb, d)
in $\xleftarrow{\$}$ $G_{ec}$    ▷ Get a random value for in, which correspond to our evaluation point of our polynomial
▷ Generate the keys pk and vk in the keypair
r1cs_ppzksnark_keypair keypair ← r1cs_ppzksnark_generator(ptb.constraint_system)
▷ //————Server phase————//
▷ Compute the witness of our R1CS (i.e the $[a_i]_{i=l+1}^{m}$) and the output $a_2$
r1cs_variable_assignment var_assignement ← compute_polynomial_witness_output( ptb, pk)
▷ Compute the proof of our result
r1cs_ppzksnark_proof $\pi$ ← r1cs_ppzksnark_prover( pk, ptb.primary_input(), ptb.auxiliary_input())
▷ //————Client phase————//
▷ Check if the proof is correct
$\mathbb{B}$ res ← r1cs_ppzksnark_verifier_strong_IC( vk, ptb.primary_input(), $\pi$)
if res == True then
    return true
else
    return false
end if
```

With this simulation function we have made tests, with as evaluation criteria those we have stated in section 1.6. Our results are given in section 2.2.3.

2.2.3 Results obtained with our simulation function

With the simulation function described in the algorithm 4 we have done some tests. The objective is to see the dependency between our different evaluation criteria given in section 1.6 (i.e. ths computation time, the extra storage, the communication volume and the number of communication turns) with the size of our polynomial (i.e. the degree of our polynomial). To know if our results are interesting or not we compared them with the VESPO protocol created by a team of researchers at the LJK, and implemented with the RELIC library. Their protocol does a polynomial evaluation in the Paillier cipher, for the measurements we use a Paillier cipher of size 2048bits.

For the theoretical complexity of our protocol we have a linear computation time in the size of the circuit, and we have the calculation of an FFT which makes our complexity in order of magnitude $O(n\log(n))$. And for the verifier we are in constant time.

For our results we are supposed to have a constant computation time on the client side during the **Verif** step and a linear evolution according to the degree of our polynomial for the **Setup**. On the prover side we should also have a linear evolution according to the degree of our polynomial for the **Eval** step.

Table 2.4: Computing time during the different steps of the protocol in seconds

	Our protocol based on libsnark			**VESPO with Paillier 2048**		
Polynomial degree	Setup (s)	Verify (s)	Eval (s)	Setup (s)	Verify (s)	Eval (s)
256	0.043857	0.004686	0.049905	0.504179	0.003312	0.395885
512	0.089097	0.004759	0.091626	0.721449	0.003283	0.779790
1024	0.131321	0.004608	0.149774	0.704225	0.003228	1.555976
2048	0.249322	0.004501	0.269804	1.063072	0.003335	3.091791
4096	0.460878	0.004473	0.494810	1.970152	0.003364	6.230299
8192	0.941089	0.004512	0.934293	3.118222	0.003391	12.380993
16384	1.851514	0.004466	1.764115	5.445983	0.003401	24.751030
32768	4.003796	0.004963	4.182594	10.875640	0.003360	49.515388
65536	8.202355	0.005257	9.081271	20.922072	0.003375	98.753132
131072	13.875263	0.004648	16.357264	41.569178	0.003431	199.38868

Focusing on the results obtained with our protocol, we can see that the Setup and Eval phases grow linearly with the degree of our polynomial for the computation time. While the Verify phase the calculation time is constant. If we compare our protocol with that of VESPO:

- During the Verify phase our protocol takes slightly longer than VESPO
- During the Setup and Eval phases our protocol is faster with a factor of 2 for the Eval and a factor of 5 for the Setup phase

Now we are interested in the data to be stored and sent, again comparing our protocol with VESPO. We suuposed that the volume of data to be stored on the client side is constant as well as the data to be sent and received from the server during the Eval. On the other hand, during the setup phase, the volume of data to be sent and stored on the server side increases in a linear way according to the degree of our polynomial.

With the results given in table 2.5 and 2.6 we can see that our protocol stores less data on the client side than VESPO but on the server side it is the opposite. With all our results one could be tempted to think that our protocol based on libsnark would be better than VESPO's. But contrary to ours, the VESPO protocol encrypts the coefficients of the client's polynomial. So our protocol is only applicable if the client does not need to hide its data. In addition to that our protocol is not dynamic so the Setup phase needs to be re-run every time we change a coefficient of our polynomial. In conclusion our protocol is interesting only in particular cases that the client must be aware of.

Table 2.5: Data flow during our protocol based on libsnark

	Our protocol based on libsnark				
Polynomial degree	Client data save after setup (bits)	Data send to server for setup (bits)	Server, data save after setup (bits)	Eval data send (bits)	Response of data for eval (bits)
256	3629	914 940	914 940	254	2548
512	3629	1 828 860	1 828 860	254	2548
1024	3629	3 656 700	3 656 700	254	2548
2048	3629	7 312 380	7 312 380	254	2548
4096	3629	14 623 740	14 623 740	254	2548
8192	3629	29 246 460	29 246 460	254	2548
16384	3629	58 491 900	58 491 900	254	2548
32768	3629	116 982 780	116 982 780	254	2548
65536	3629	233 964 540	233 964 540	254	2548
131072	3629	467 928 060	467 928 060	254	2548

Table 2.6: Data flow during VESPO protocol

	VESPO with Paillier 2048				
Polynomial degree	Client data save after setup (bits)	Data send to server for setup (bits)	Server data save after setup (bits)	Eval data send (bits)	Response of data for eval (bits)
256	5376	722 944	722 944	256	2304
512	5376	1 443 840	1 443 840	256	2304
1024	5376	2 885 632	2 885 632	256	2304
2048	5376	5 769 216	5 769 216	256	2304
4096	5376	11 536 384	11 536 384	256	2304
8192	5376	23 070 720	23 070 720	256	2304
16384	5376	46 139 392	46 139 392	256	2304
32768	5376	92 276 736	92 276 736	256	2304
65536	5376	184 551 424	184 551 424	256	2304
131072	5376	369 100 800	369 100 800	256	2304

2.2.4 Conclusion

This chapter shows us that the snark technology is interesting and deserves to be studied. There are however improvements to be made on the data encryption problem and the dynamicity of our protocol. Part 3 of this report will try to solve the data encryption problem and part 4 the dynamicity problem.

— 3 —
Jsnark

In this chapter we always pose in the scheme where a client wants to do a polynomial evaluation on a server with a proof that the result is correct. At the end of chapter 2 of this report we found a problem on the fact that our data (the coefficients of our polynomial) were not encrypted. The objective of this chapter is to try to find a solution to this problem and to see if this solution is feasible or not.

The [Kos+] paper offers a solution to encrypt our data in our **R1CS**. They have created a tool to generate a **R1CS** with modular operations on large integers, this tool is on the github directory [ako].

3.1 Encryption system we will use

To encrypt the coefficients of our polynomial we will use the **Paillier's cryptosystem**, which is an asymmetric algorithm for public key cryptography.

Definition 8 ***Paillier's cryptosystem*** *is composed of a encryption function* ***E(m)*** *(with* ***m*** *a plaintext we want to cipher) and a decryption function* ***D(c)*** *(with* ***c*** *a ciphertext we want to decipher). The key generation works as follow:*

- *Choose two large prime numbers* ***p*** *and* ***q*** *randomly and independently of each other, such that gcd(pq,(p-1)(q-1))=1. This property is assured if both primes are of equal length.*
- *Compute* ***n****=pq and* $\lambda = lcm(p-1, q-1)$ *with lcm the Least Common Multiple.*
- *Select random integer* ***g*** *where* $g \in \mathbb{Z}^*_{n^2}$
- *Ensure* ***n*** *divides the order of* ***g*** *by checking the existence of the following modular multiplicative inverse:* μ = *(**L**(*$g^\lambda \mod n^2))^{-1} \mod n$ *where function* ***L*** *is defined as* ***L****(x)=*$\frac{x-1}{n}$

Encryption function ***E(m)****:*

- *Let* ***m*** *be a message to be ciphered where* $0 \leq \mathbf{m} < \mathbf{n}$
- *Select random* ***r*** *where* $0 \leq \mathbf{r} < \mathbf{n}$
- *Compute the ciphertext* ***c*** *as* ***c****=*$g^m r^n \mod n^2$

*Decryption function **D(c)**:*

- *Let **c** be the ciphertext to decipher where **c**$\in \mathbb{Z}^*_{n^2}$*
- *Compute the plaintext message **m** as m=**L**($c^\lambda \mod n^2)\mu \mod n$*

*We have **m=D(E(m))**. And this cryptosystem is additive and multiplicative homomorphic:*

- ***Additive homomorphic**: if for two plaintexts **m1** and **m2** we have:*
  - ***D(E**(m_1,r_1)**E**$(m_2,r_2) \mod n^2) = m_1 + m_2 \mod n$*
- ***Multiplication homomorphic**: if for two plaintexts **m1** and **m2** we have:*
  - ***D(E**$(m_1,r_1)^{m_2} \mod n^2) = m_1 m_2 \mod n$*
  - ***D(E**$(m_2,r_2)^{m_1} \mod n^2) = m_1 m_2 \mod n$*

3.2 Polynomial evaluation with Paillier's cryptosystem

With our polynomial **P** of degree **d** with the coefficients $[\mathbf{p_i}]_{i=0}^d$ such that: $p_0 + p_1 x + .. + p_d x^d$. With the Horner's method we have: $p_0 + x(p_1 + x(..x(p_{d-1} + xp_d)))$

Using the Paillier's cryptosystem give in definition 8, we define c_j the ciphertext coefficient of p_j where $c_j = \mathbf{E}(p_j)$. Our objective is to do our polynomial evaluation of **P** at point **k** with the Horner's method in the ciphers. We make the following computation:

res=$(c_0 * (c_1(..(c_{d-1}(c_d^k \mod n^2) \mod n^2)^k \mod n^2..)^k \mod n^2)^k \mod n^2) \mod n^2$

And we have **P(k)**=**D**(res)

Now we want to create an **R1CS** that performs this calculation with a 2048bit Paillier encryption. So each of our c_j's are large numbers and we need to do modulo operation. For that we used functions of jsnark which allowed us to create this **R1CS**. The details of our implementation are given in chapter 3.3.

3.3 Creation of an R1CS containing the coefficients of our encrypted polynomial with Jsnark

In this part we will explain how our program works to create our **R1CS** containing the coefficients of our polynomial encrypted with the 2048 bits Paillier cryptosystem. Our code is on the github snark-protocol-experiment directory [rus].

Here is a list of the different objects in jsnark that we used in our program:

1. **LongElement** is an object that handles some operations like modulo, multiplication, addition for long integer. With that we can create the different operation in our **R1CS**
2. **Wire** is an object to link a wire of our **R1CS** to a value. A list of wire can correspond to a **LongElement**

And with these objects we have created the following obejcts:

1. AdditionLongElementPaillier(**paillierModulus**, **a**, **b**, **paillierKeyBitLength**, **desc**) → **result**

   Input: (**paillierModulus**, **a**, **b**, **paillierKeyBitLength**, **desc**)

   - **paillierModulus**: LongElement
   - **a**: Wire[]
   - **b**: Wire[]
   - **paillierKeyBitLength**: int
   - **desc**: String...

   Output: **result**: Wire[]

   Description: Create the **R1CS** that perform the addition with Paillier's modulus, where **result** = **a+b** mod **paillierModulus**. With **paillierKeyBitLength** corresponding to the size in bits of the LongElement

2. HornerPolynomialEvalPaillier(**paillierModulus**, **coefficients**, **x**, **paillierKeyBitLength**, **desc**) → **result**

   Input: (**paillierModulus**, **coefficients**, **x**, **paillierKeyBitLength**, **desc**)

   - **paillierModulus**: LongElement
   - **coefficients**: ArrayList<Wire[]>
   - **x**: BigInteger
   - **paillierKeyBitLength**: int
   - **desc**: String...

   Output: **result**: Wire[]

   Description: Generate the **R1CS** to perform the evaluation of the polynomial **P** of degree **d**, with all it's coefficients $\mathbf{p_i}$ ciphered with the Paillier's cryptosystem of modulo **paillierModulus** in the array **coefficients**, at the point **x**. And the result is stored in **result**

3. ModPowLongElementPaillier(**paillierModulus**, **a**, **b**, **paillierKeyBitLength**, **desc**) → **result**

   Input: (**paillierModulus**, **a**, **b**, **paillierKeyBitLength**, **desc**)

   - **paillierModulus**: LongElement
   - **a**: Wire[]
   - **b**: BigInteger
   - **paillierKeyBitLength**: int
   - **desc**: String...

   Output: **result**: Wire[]

   Description: This object creates the corresponding **R1CS** that perform the modular exponentiation with LongElement, where **result** = $\mathbf{a^b}$ mod **paillierModulus**

4. ModPowPaillier(**paillierModulus**, **a**, **b**, **paillierKeyBitLength**, **desc**) → **result**

Input: (**paillierModulus**, **a**, **b**, **paillierKeyBitLength**, **desc**)

- **paillierModulus**: LongElement
- **a**: Wire[]
- **b**: int
- **paillierKeyBitLength**: int
- **desc**: String...

Output: **result**: Wire[]

Description: This object creates the corresponding **R1CS** that perform the modular exponentiation with the exponent as integer, **result** = $\mathbf{a^b} \mod$ **paillierModulus**

5. MultiplicationLongElementPaillier(**paillierModulus**, **a**, **b**, **paillierKeyBitLength**, **desc**) → **result**

   Input: (**paillierModulus**, **a**, **b**, **paillierKeyBitLength**, **desc**)

   - **paillierModulus**: LongElement
   - **a**: Wire[]
   - **b**: Wire[]
   - **paillierKeyBitLength**: int
   - **desc**: String...

   Output: **result**: Wire[]

   Description: Generate the corresponding **R1CS** that perform the multiplication in paillier's with LongElement, **result=ab** $\mod$ **paillierModulus**

6. Paillier_KeyPair(**privateKey**, **publicKey**, **upperBound**)

   Input: (**privateKey**, **publicKey**, **upperBound**)

   - **privateKey**: Paillier_PrivateKey
   - **publicKey**: Paillier_PublicKey
   - **upperBound**: BigInteger

   Description: Store the private key **privateKey** and the public key **publicKey** of our paillier's cryptosytem. The upper bound is null by default.

7. Paillier_PublicKey(**n**, **nSquared**, **g**, **bits**)

   Input: (**n**, **nSquared**, **g**, **bits**)

   - **n**: BigInteger
   - **nSquared**: BigInteger
   - **g**: BigInteger
   - **bits**: int

   Description: Store the public key of our Paillier's cryptosystem with **bits** equal to 1024 or 2048

8. Paillier_PrivateKey(**lambda**, **preCalculatedDenominator**)

   Input: (**lambda**, **preCalculatedDenominator**)

   - **lambda**: BigInteger

- **preCalculatedDenominator**: BigInteger
- **d**: integer

Description: Store the private key of our paillier's cryptosystem

9. Paillier_keyPairBuilder

   Description: It's a constructor used to generate a Paillier_KeyPair with the method **generateKeyPair**()

To be able to use each of the objects we have created we need to use an object given by jsnark called CircuitGenerator with the following functions:

1. generateCircuit()

   Description: Generate the **R1CS**

2. genInputEval()

   Prerequisite: The function generateCircuit() need to be called before

   Description: Add the input value to the **R1CS**

3. evalCircuitWithoutInput()

   Prerequisite: You have to run the function genInputEval() before.

   Description: Evaluate the circuit to find the witness and output

4. evalCircuit()

   Prerequisite: The function generateCircuit() need to be called before

   Description: This function evaluate the circuit, it means that it calculate the witness and output of the **R1CS**.

5. prepFilesPaillier()

   Prerequisite: We have to call the function genInputEval() before

   Description: Create files with the **R1CS** when we have not calculated the witness and output

6. writeAllPailliersCoefsWithN(**coeff_in_paillier**, BigInteger **x**, **publicKey**)

   Input: (**coeff_in_paillier**, BigInteger **x**, **publicKey**)

   - **coeff_in_paillier**: ArrayList<BigInteger>
   - **x**: BigInteger
   - **publicKey**: Paillier_PublicKey

   Description: Create files which contain all our polynomial coefficients **coeff_in_paillier**, our evaluation point **x** and the pailleir's public key **publicKey**

7. prepFiles()

   Prerequisite: We need to call the function evalCircuit() before

Description: Create files with the **R1CS** with the witness and output.

With the objects whose specifications we have given above we'll create our protocol. As a reminder, our goal is that a client wishing to perform a polynomial evaluation on a server can obtain the result with a proof. And what we want to add in this part is the encryption of the polynomial coefficients, so that the server doesn't have access to the client data.

3.4 Algorithm of our protocol

We have written two algorithms in jsnark, the first one is used to create the **R1CS** corresponding to the evaluation of the polynomial with the coefficients encrypted with the Paillier's cryptosystem (see HornerPolynomialPaillierGenerator in annex) and the second one is used to compute the intermediate values and the result of the **R1CS** (see HornerPolynomialPaillierFromFileGenerator in annex) .

With our first code given in appendix we generate 3 files:

1. Horner_polynomial_eval_big_int_paillier_gadgetWithoutEval.arith: file used by libsnark for the Setup function it contains our **R1CS** constraints

2. Horner_polynomial_eval_big_int_paillier_gadgetWithoutEval.in: file used by libsnark for the Setup function, it contains the value of the variable in our **R1CS**

3. AllPailliersCoeffAndInformation.out: this file will be send to the server for it's computation, it contains the coefficients of our polynomial, ciphered with the Paillier's cryptosystem.

And the second one generate 2 files:

1. Horner_polynomial_eval_big_int_paillier_gadget_input_file.arith: file used by libsnark for the Prove function, it contains our **R1CS** constraints

2. Horner_polynomial_eval_big_int_paillier_gadget_input_file.in: file used by libsnark for the Prove function, it contains the value of the variable in our **R1CS**

And we wrote another algorithm in libsnark to run our version of snarkGroth16 on the **R1CS** generated with jsnark. This algorithm is very similar to algorithm 4, the only two differences are when creating the polynomial in libsnark that we read from the files generated with the first code:

1. Horner_polynomial_eval_big_int_paillier_gadgetWithoutEval.arith

2. Horner_polynomial_eval_big_int_paillier_gadgetWithoutEval.in

And in the calculation of the intermediate values of our R1CS that we do from the files generated with the second code:

1. Horner_polynomial_eval_big_int_paillier_gadget_input_file.arith

2. Horner_polynomial_eval_big_int_paillier_gadget_input_file.in

The conversion of the data stored in our files in R1CS with the values that it can contain is done with the object CircuitReader in libsnark :

1. CircuitReader(**file1**, **file2**, **ptb**)

   Input: (**file1**, **file2**, **ptb**)

   - **ptb**: protoboard
   - **file1**: File, it contains our **R1CS** constraints
   - **file2**: File, it contains the value of the variable in our **R1CS** (it's not a problem if some value are not given)

   Description: Generate the **R1CS** contained in the files **file1** with the variables value contained in the file **file2**. And store the **R1CS** in the protoboard **ptb**. If the file **file2** contains the witness of the **R1CS** they'll be stored in the protoboard **ptb** too.

Algorithm 5 test_polynomial_in_clear_with_jsnark(**fileFromClient1**, **fileFromClient2**, **fileFromServer1**, **fileFromServer2**)

Input: (**fileFromClient1**, **fileFromClient2**, **fileFromServer1**, **fileFromServer2**)

- **fileFromClient1**: File, it contains the **R1CS** constraints
- **fileFromClient2**: File, it contains the value of the input in our **R1CS**
- **fileFromServer1**: File, it contains the **R1CS** constraints
- **fileFromServer2**: File, it contains the value of the variable in our **R1CS**, input, output an the witness

Output: True if the result of the server correspond to the ciphered polynomial evaluation, False otherwise

Description: Simulate the protocol snarkGroth16 on the **R1CS** generated with our code in jsnark return true if the server gives the good result else return false.

```
▷ //————Client phase————//
▷ Generate the protoboard which will contain our R1CS
protoboard ptbClient = new Protoboard()
▷ Create the R1CS in the ptbClient with the file of the client
CircuitReader(fileFromClient1, fileFromClient2, ptbClient)
▷ Generate the keys pk and vk
r1cs_ppzksnark_keypair keypair $\leftarrow$ r1cs_ppzksnark_generator(ptbClient.constraint_system)
▷ //————Server phase————//
▷ Generate the protoboard which will contain our R1CS with the intermediate value and the result
protoboard ptbServer = new Protoboard()
▷ Create the R1CS in the ptbServer with the file of the server
CircuitReader(fileFromServer1, fileFromServer2, ptbServer)
▷ Generate the proof of the result contained in the ptbServer
r1cs_ppzksnark_proof $\pi$ $\leftarrow$ r1cs_ppzksnark_prover( pk, ptbServer.primary_input(), ptbServer.auxiliary_input())
▷ //————Client phase————//
▷ Check if the server's proof is correct
$\mathbb{B}$ res $\leftarrow$ r1cs_ppzksnark_verifier_strong_IC(vk, ptbClient.primary_input(), $\pi$)
if res == True then
    return true
else
    return false
end if
```

With the algorithm 5 and the 2 jsnark algorithms given in annex (HornerPolynomialPaillierGenerator,HornerPolynomialPaillierFromFileGenerator) we can simulate our protocol and do similar tests as in section 2.2.3.

3.5 Results with jsnark

Our test protocol consists of 3 steps:

- Run the algorithm HornerPolynomialPaillierGenerator
- Run the algorithm HornerPolynomialPaillierFromFileGenerator
- Run the algorithm 5

For the theoretical complexity we expect to be almost linear with respect to the degree of our polynomial, but quadratic with respect to the size of our Paillier cryptosystem $\mathrm{O}(\log^2(N) * d)$, with N the size of our Paillier's cryptosystem and d the degree of our polynomial.

Table 3.1: Computing time during the different steps of the protocol in seconds

	Our protocol based on libsnark					
	Paillier 1024			**Paillier 2048**		
Polynomial degree	Setup (s)	Verify (s)	Eval (s)	Setup (s)	Verify (s)	Eval (s)
1	1.114	0.009	0.362	2.41913	0.009	0.955576
2	118.408908	0.009	71.986391	404.638173	0.009	321.18834
4	351.608624	0.009	228.616748	1179.890857	0.009	1010.255126
6	568.728423	0.009	370.837077	1958.586091	0.009	1636.967579
8	764.745023	0.009	502.289528	We didn't run for 8	because it tooks to much time	
10	994.470295	0.009	637.652533			

Table 3.2: Detailed computation time for Pailler 1024 for different protocol steps

Polynomial degree	Setup in jsnark (s)	Setup in libsnark (s)	Verify (s)	Eval in jsnark (s)	Eval in libsnark (s)
1	1.114	0.009	0.362	2.41913	0.009
2	8.011	120.897581	0.008795	9.451	49.489633
4	14.689	235.561558	0.008875	18.735	99.197377
6	25.120	336.539282	0.008761	31.016	143.545923
8	29.215	431.182645	0.008840	35.484	195.972166
10	36.154	549.644333	0.008918	48.615	245.565576

We can see that our computation time increases more than linearly with the degree of our polynomial, and we are quadratic with the size of the Paillier cipher. We think that the fact that our protocol has a computation time that is not linear with the degree of the polynomial is due of the memory space we consume during execution.

We don't even need to compare ourselves to the **VESPO** protocol to see that our protocol cannot be used in practice, our times are much too long for the degree of our polynomial. We explain this gap with libsnark because of the method we used, to implement the modular arithmetic we split our encrypted coefficients (size 1024 or 2048 according to the Paillier cryptosystem) into packets of 8 bits, so the number of constraints of our **R1CS** is really big, for 1 coefficients in our polynomial we have 4672 constraints, 578257 constraints for 2 and 1150932 for 3 (with Paillier's 2048) compare to 8 constraints in snarkGroth16 protocol for 3 coefficients.

3.6 Conclusion

To conclude it's possible to encrypt the coefficients of our polynomial and to do a polynomial evaluation on a remote server with a proof that the calculation is correct. But the solution we found cannot be used, because the computation time is much too long.

A solution could be to change the homomorphic encryption system to one with smaller integers as Castagnos-Laguuillaumie ones (DDH), or to change our implementation in jsnark to break our integers into packets larger than 8 bits to reduce the number of constraints in the **R1CS**, but we can't have packets with a size more than 32 bits because of our pairing modulus (i.e. a multiplication with 2 packets of size more than 32 bits will exceeds the modulus and our result will be wrong). But even with larger packets it seem's that we can only divide our computation time by a factor 16.

— 4 —

snarkGroth16 protocol dynamics

The objective of this section is to be able to make the protocol of our chapter 2 dynamic, more precisely if the client having a polynomial **P** wishes to change one of its coefficients it must be able to do it in logarithmic computation time. So it must be possible to change the key **pk** and **vk** of the snarkGroth16 protocol in logarithmic computation time.

4.1 Theoretical change in the snarkGroth16 protocol

A client has a polynomial **P**, of degree **d**, with coefficients $[p_i]_{i=0}^{d}$, and he wants to change one of his coefficient p_j to p'_j.

4.1.1 Change in the QAP corresponding to our polynomial P

With the construction of our **R1CS** as done in example 1 we can see that all the coefficients of our polynomial **P** are in the vectors **u** at index 0 (this property changes depending on our construction).

Then when we generate our QAP (with the construction of example 2) all our coefficients are in the polynomial A of index 0 where $A_0 = LagrangeInterpolation((0, u_{0,0}), (1, u_{1,0}), ..., (d, u_{d,0}))$.

With our definition 5 if we define A_0 by the vector of its coefficients $a_{0,i}$ then each coefficients of our polynomial **P**, p_i has an impact on each values of the $a_{0,i}$.

If we want to change one of the coefficients of our polynomial **P**, p_j into another p'_j, with A_0 in the canonical basis, we have to change all our $a_{0,i}$ which implies to do a lagrange interpolation again, which depends on the degree of our polynomial **d**. So we are not in logarithmic time but linear with respect to the degree of our polynomial, therefore we need to find another approach. Where our goal is to minimize the update function.

For this we can change the basis, if we consider the polynomial A_0 as the set of p_i coefficients in the Lagrange polynomial basis, we can write $A_0 = \sum_{j=0}^{d} p_j l_j(x)$ with $l_j(x) = \prod_{0 \leq i \leq d, i \neq j} \frac{x-i}{j-i}$. With this equality for A_0 we can say that A_0 is equal to the set of $[p_i]_{i=0}^{d}$ in the Lagrange polynomial basis, we'll note the Lagrange polynomial basis as $[]_{LP}$. So $A_0 = \begin{bmatrix} p_d \\ \cdots \\ p_1 \\ p_0 \end{bmatrix}_{LP}$

If we go back to the case where the client wants to change one of the coefficients of $\mathbf{P}$, p_j to p'_j in this case our new polynomial $A'_0 = A_0 + (p'_j - p_j)l_j(x)$, then $A'_0 = \begin{bmatrix} p_d \\ \dots \\ p'_j \\ \dots \\ p_0 \end{bmatrix}_{LP}$. So our change of coefficient no longer depends on the degree of our polynomial $\mathbf{P}$ if we know the polynomial $l_j(x)$. Now our goal is to find a form of the polynomial $l_j(x)$ that can be easily calculated.

Definition 9 *New definition of the function $l_j(x)$. We choose to take the primitive roots ω of the unit of order **d+1**, rather than the points m to make our interpolation so $l_j(x) = \prod_{0\leq i\leq d, i\neq j} \frac{x-\omega^i}{\omega^j-\omega^i}$. Therefore our polynomial $\prod_{0\leq i\leq d}(x-\omega^i) = x^{d+1}-1$ because $(\omega^i)^{d+1} = 1$ so:*

$$l_j(x) = \prod_{0\leq i\leq d, i\neq j} \frac{x-\omega^i}{\omega^j-\omega^i} = \frac{x^{d+1}-1}{x-\omega^j} * \prod_{0\leq i\leq d, i\neq j} \frac{1}{\omega^j-\omega^i} = \frac{x^{d+1}-1}{x-\omega^j} * v_j \tag{4.1}$$

Where $v_j = \prod_{0\leq i\leq d, i\neq j} \frac{1}{\omega^j-\omega^i}$. With this method we can compute the polynomial $l_j(x)$ in a logarithmic computation time.

So we can compute $A'_0(s) = A_0(s) + (p'_j - p_j)l_j(s)$ with the function $l_j(x)$ describe in equation 4.1 in a logarithmic computation time. Since the value of s is supposed to be known only by the client the calculation $(p'_j - p_j)l_j(s)$ need to be done by the client.

4.1.2 Change in the keys of our snarkGroth16 protocol

Now we want to update the keys **pk** and **vk** generated previously with our protocol snarkGroth16. We have:

- **vk**= $(\mathbf{Sc1}, \alpha_1, (\gamma_2, \beta_2, \delta_2)) \in G_1^{l+1} \times G_1 \times G_2^3$ with:
  - $\mathbf{Sc1} = \{[\frac{\beta A_i(s)+\alpha B_i(s)+C_i(s)}{\gamma}]_1\}_{i=0}^{l} \in G_1^{l+1}$
- **pk**= $((\alpha_1, \beta_1, \gamma_1, \delta_1), \mathbf{S1}, \mathbf{Sa1}, \mathbf{Sb1}, (\beta_2, \gamma_2, \delta_2), \mathbf{S2}) \in G_1^4 \times G_1^{n+1} \times G_1^{m-l} \times G_1^{n+1} \times G_2^3 \times G_2^{n+1}$
  - $\mathbf{S1} = \{[s^i]_1\}_{i=0}^{n} \in G_1^{n+1}$
  - $\mathbf{Sa1} = \{[\frac{\beta A_i(s)+\alpha B_i(s)+C_i(s)}{\delta}]_1\}_{i=l+1}^{m} \in G_1^{m-l}$
  - $\mathbf{Sb1} = \{[\frac{s^i Z(s)}{\delta}]_1\}_{i=0}^{n} \in G_1^{n+1}$
  - $\mathbf{S2} = \{[s^i]_2\}_{i=0}^{n} \in G_2^{n+1}$

After changing our A_0 polynomial in our **QAP** we need to change each element of our keys that contain A_0. We have to change the value of **Sc1** which contains A_0, so if we have A'_0 the new polynomial we update **Sc1** such that $\mathbf{Sc1}' = \{\mathbf{Sc1}_0 * [\frac{\beta(A'_0(s)-A_0(s))}{\gamma}]_1, \{[\frac{\beta A_i(s)+\alpha B_i(s)+C_i(s)}{\gamma}]_1\}_{i=1}^{l}\}$, with $\mathbf{Sc1}_0 = [\frac{\beta A_0(s)+\alpha B_0(s)+C_0(s)}{\gamma}]_1$. To change **Sc1** in our key **vk** we need to store β, γ and s, which are supposed to be secret for the server so the client has to do the computation.

4.1.3 Protocol snarkGroth16 with the update

To perform the update in the snarkGroth16 protocol we have created a function Update, that compute the new $\mathbf{Sc1}'_0$ and the value $\Delta = (p'_j - p_j)l_{j+1}(s)$.

Now the server stored the polynomial $\{A_i, B_i, C_i\}_{i=0}^{m}$ evaluated at point s (i.e. $\{A_i(s), B_i(s), C_i(s)\}_{i=0}^{m}$), and the client need to store the values of s, β, γ, $A_0(s)$ and the coefficients of its polynomial. For the coefficients a solution could be that the server store them with a merkle tree (where the merkle tree will be used to check if the coefficients sends by the server is the good one this technic is used in the protocol **VESPO**) and the client ask them to the server when he want to made a change.

Algorithm 6 Update(**previousCoef**, **newCoef**, **index**, $A_0(s)$, **vk**)

Input: (**previousCoef**, **newCoef**, **index**, $A_0(s)$, **vk**= $(\mathbf{Sc1}, \alpha_1, (\gamma_2, \beta_2, \delta_2))$):

- previousCoef, newCoef $\in F$
- index $\in \mathbb{N}$
- $A_0(s) \in F$
- $\mathbf{Sc1} \in G_1^{l+1}$
- $\alpha_1 \in G_1$
- $(\gamma_2, \beta_2, \delta_2) \in G_2^3$

Output: ($\mathbf{Sc1}'_0$, $\Delta = (p'_j - p_j)l_{j+1}(s)$)

- $\mathbf{Sc1}'_0 \in G_1$
- $\Delta = (p'_j - p_j)l_{j+1}(s) \in F$

Description: The client want to changes the coefficient of index **index** with the previous value **previousCoef** to the new value **newCoef**, so he generate the value Δ to change the $A_0(s)$ and the value $\mathbf{Sc1}'_0$ to change the key **vk**.

```
Ensure: With R'= $\{(p,g,h,e) \in \mathbb{Z} \times G_1 \times G_2 \times G_T, l, m, \in \mathbb{Z}, \{\{A_0 + \Delta, B_0, C_0\}, \{A_i, B_i, C_i\}_{i=1}^{m}\}, \in F_{d^\circ \leq n+1}^{m}[X], Z \in F_{d^\circ \leq n+1}[X]\}$, $\mathbf{Sc1}' = \{\mathbf{Sc1}'_0, \{[\frac{\beta A_i(s) + \alpha B_i(s) + C_i(s)}{\gamma}]_1\}_{i=1}^{l}\}$ and vk'= $(\mathbf{Sc1}', \alpha_1, (\gamma_2, \beta_2, \delta_2)) \in G_1^{l+1} \times G_1 \times G_2^3$ we have Verify(R', vk', $a_{0\dots l}$, Prove(R', pk, $a_{0\dots l}, a_{l+1\dots m}$)) will output True if the $a_i$ solves the QAP in R' else it'll output False
$\Delta = (p'_j - p_j)l_{j+1}(s) \in F$    ▷ The function $l_j(x)$ is given in the definition 9 and the implementation is detailled in libsnark
$\mathbf{Sc1}'_0 = \mathbf{Sc1}_0 * [\frac{\beta(A'_0(s) - A_0(s))}{\gamma}]_1$
return ($\mathbf{Sc1}'_0$, $\Delta$) $\in G_1 \times F$
```

Algorithm 6 describes the update function, which we can add to the snarkGroth16 protocol.

Table 4.1: snarkGroth16 with the update

	Client	**Communications**	**Prover**
Setup	With P$\in F[X]$ Compute R corresponding to P R$= \{(p,g,h,e) \in \mathbb{Z} \times G_1 \times G_2 \times G_T$ $l,m, \in \mathbb{Z}, \{A_i,B_i,C_i\}_{i=0}^{m}, \in F_{d^\circ \leq n+1}[X]$ $Z \in F_{d^\circ \leq n+1}[X]\}$ $\alpha,\beta,\gamma,\delta,s \xleftarrow{\$} F^*$ $\alpha_1 = [\alpha]_1, \beta_1 = [\beta]_1, \gamma_1 = [\gamma]_1, \delta_1 = [\delta]_1 \in G_1$ $\mathbf{S1} = \{[s^i]_1\}_{i=0}^{n} \in G1^n,$ $\mathbf{Sa1} = \{[\frac{\beta A_i(s)+\alpha B_i(s)+C_i(s)}{\delta}]_1\}_{i=l+1}^{m} \in G_1^{m-(l+1)}$ $\mathbf{Sb1} = \{[\frac{s^i Z(s)}{\delta}]_1\}_{i=0}^{n} \in G_1^n$ $\mathbf{Sc1} = \{[\frac{\beta A_i(s)+\alpha B_i(s)+C_i(s)}{\gamma}]_1\}_{i=0}^{l} \in G_1^l$ $\beta_2 = [\beta]_2, \gamma_2 = [\gamma]_2, \delta_2 = [\delta]_2 \in G_2,$ $\mathbf{S2} = \{[s^i]_2\}_{i=0}^{n} \in G_2^n$ R'$= \{(p,g,h,e) \in \mathbb{Z} \times G_1 \times G_2 \times G_T$ $l,m, \in \mathbb{Z}, \{A_i(s),B_i(s),C_i(s)\}_{i=0}^{m}, \in F$ $Z \in F_{d^\circ \leq n+1}[X]\}$ $pk = ((\alpha_1,\beta_1,\gamma_1,\delta_1),\mathbf{S1},\mathbf{Sa1},\mathbf{Sb1},(\beta_2,\gamma_2,\delta_2)$ $,\mathbf{S2}) \in G_1^4 \times G_1^n \times G_1^{m-(l+1)} \times G_1^n \times G_2^3 \times G_2^n)$ $vk = (\mathbf{Sc1},\alpha_1,(\gamma_2,\beta_2,\delta_2)) \in G_1^l \times G_1 \times G_2^3$ The client store β,γ, s, $A_0(s)$, vk and the coefficients of its polynomial, he can discard the rest	$\xrightarrow{pk,a_{input},R}$	
Update	The client want to change p_j to p'_j $\Delta = (p'_j - p_j)l_{j+1}(s) \in F$ $\mathbf{Sc1}'_0 = \mathbf{Sc1}_0 * [\frac{\beta(A'_0(s)-A_0(s))}{\gamma}]_1$ return $\mathbf{Sc1}'_0, \Delta$ $\mathbf{Sc1}' = \{\mathbf{Sc1}'_0, \{[\frac{\beta A_i(s)+\alpha B_i(s)+C_i(s)}{\gamma}]_1\}_{i=1}^{l}\}$ **vk'**$= (\mathbf{Sc1}',\alpha_1,(\gamma_2,\beta_2,\delta_2))$	$\xrightarrow{\Delta}$	$A'_0(s) = A_0(s) + \Delta \in F$ R'$= \{(p,g,h,e) \in \mathbb{Z} \times G_1 \times G_2 \times G_T$ $l,m, \in \mathbb{Z}, \{\{A'_0,B_0,C_0\},\{A_i,B_i,C_i\}_{i=1}^{m}\}, \in F_{d^\circ \leq n+1}^{m}[X]$ $Z \in F_{d^\circ \leq n+1}[X]\}$

4.2 Implementation in libsnark

For this part we used the same functions and the same objects that we created in part 2.2.2 and we added some others:

1. update_constraint_horner_method(**newCoefValue**, **ptb**, **index**, **d**)

   Input: (**newCoefValue**, **ptb**, **index**, **d**)

   - **newCoefValue**: G_{ec128}
   - **ptb**: protoboard
   - **index**: integer
   - **d**: integer

   Description: This function update the **R1CS** contained in the protoboard **ptb**. By changing the coefficients of index **index** in our polynomial of degree **d** with its new value **newCoefValue**.

2. update_proving_key_compilation(**R1CS**, **lastCoefValue**, **newCoefValue**, **indexCoef**, **d**, **evaluationPoint**, β, γ, **lastKeyPair**) → **newKey**

   Input: (**newCoefValue**, **ptb**, **index**, **d**)

   - **R1CS**: r1cs_ppzksnark_constraint_system
   - **lastCoefValue**: G_{ec128}
   - **newCoefValue**: G_{ec128}
   - **indexCoef**: integer
   - **d**: integer

- **evaluationPoint**, β, γ: G_{ec128}
- **lastKeyPair**: r1cs_ppzksnark_keypair

Output: **newKey**: r1cs_ppzksnark_keypair

Description: This function update the previous key **lastKeyPair** and return the new one **newKey**.

4.2.1 Results obtained with our simulation of the update

Table 4.2: Computing time of the update function depending of the polynomial degree

	Update function results			
Polynomial degree	Update of the polynomial coefficients (s)	Update of the R1CS (s)	Update of the keys (s)	Total (s)
256	0.000053	0.000010	0.000422	0.000484
512	0.000046	0.000011	0.000499	0.000556
1024	0.000038	0.000010	0.000578	0.000626
2048	0.000046	0.000012	0.000787	0.000845
4096	0.000053	0.000023	0.001240	0.001316
8192	0.000044	0.000021	0.002174	0.002240
16384	0.000043	0.000024	0.003919	0.003985
32768	0.000041	0.000026	0.008289	0.008355
65536	0.000039	0.000038	0.016577	0.016654
131072	0.000035	0.000030	0.028823	0.028888

4.3 Conclusion

With our method we can make the snarkGroth16 protocol dynamic. But we didn't find the security proof yet this is our goal for the rest of my internship. We have to do a good implementation too and give benchmarks.

— 5 —
Conclusion

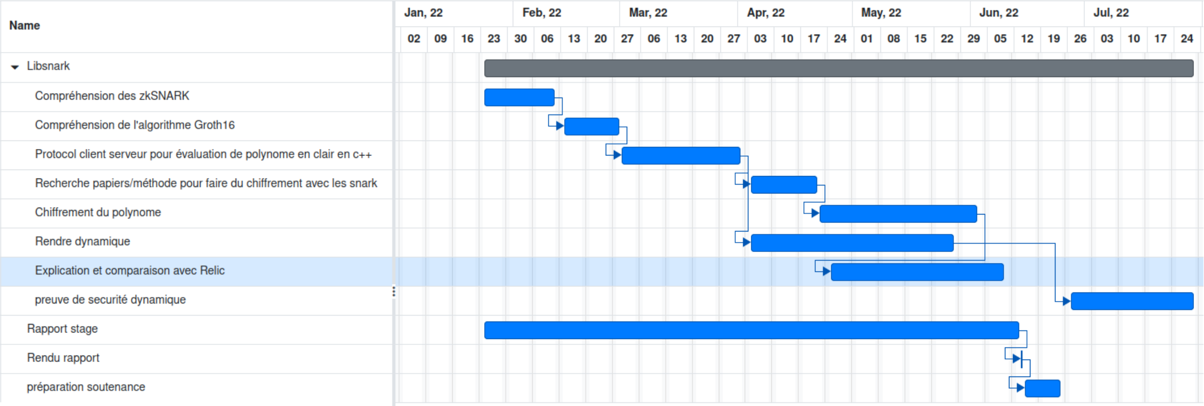
During this internship we were able to begin to understand how snark technology works and how to use it. Our goal was to create a protocol to perform a polynomial evaluation.

In particular in the first part of this report we were able to understand the snarkGroth16 algorithm, and we were able to adapt it to do our polynomial evaluation. By comparing it with the protocol created by the LJK VESPO team we could see that our protocol was faster for the server provided that the client doesn't want to encrypt the coefficients of its polynomial.

Then we tried to solve the problem concerning the encryption of the coefficients of our polynomial. For that we used a library allowing to create a modular arithmetic in our R1CS, it was the object of our second part. After having implemented our circuit we made tests to see if our protocol was practicable, our results show that for the moment our technique is not usable.

Then in our last part we tried to solve our other problem concerning the dynamicity of our protocol. Our goal was to make our first protocol, with the coefficients of our polynomial in clear, dynamic. We succeeded in proving that this was possible, but we still need to do the security proof.

To summarize the course of the internship we have added a gantt in the report :



To conclude the snark technology is interesting and seems promising but there is still progress to make concerning the encryption of the client data during a protocol.

— 6 —
Annex

6.1 HornerPolynomialPaillierGenerator code

This code is in the directory "snark-protocol-experiment/jsnark/JsnarkCircuitBuilder/src/examples/generators/paillier/".

```java
public class HornerPolynomialPaillierGenerator {
    private Integer number_of_coef;
    private Paillier_keyPairBuilder keygen;
    private Paillier_KeyPair keyPair;
    private Paillier_PublicKey publicKey;
    private ArrayList<BigInteger> coeff_in_clear;
    private ArrayList<BigInteger> coeff_in_paillier;
    //Evaluation point of our polynomial
    private BigInteger x;
    private long endTimeCalculation;

    public HornerPolynomialPaillierGenerator(Integer coef_number) {
        //Initialize all our variables
        ...
    }

    public BigInteger getRandomBigInteger() {
        //Generate a random ineger of bit size 254.
        ...
    }

    public void createCoefficients(){
        //Create number_of_coef random coefficients in clear in the list coeff_in_clear and ciphered in the list  coeff_in_paillier
        ...
    }

    public BigInteger paillierResCiphered(){
        //Compute the results with Horner's method of our polynomial (with the ciphered coefficients)
        ...
    }

    public void genCircuit() throws Exception{
        BigInteger paillierModulusValue = this.publicKey.getnSquared();
        int paillierModulusSize = paillierModulusValue.bitLength();
        //The following object will be used to create our R1CS
        CircuitGenerator generator = new CircuitGenerator("Horner_polynomial_eval_big_int_paillier_gadget") {
            ArrayList<Wire[]> coefficientsInR1CS;
            Wire[] cipherText;
            LongElement paillierModulus;
            HornerPolynomialEvalPaillier hornerPolynomialEvalPaillier;
            @Override
            protected void buildCircuit() {
                paillierModulus = createLongElementInput(paillierModulusSize);
                //Initialize the list of our coefficients in our R1CS
                coefficientsInR1CS = new ArrayList<>();
                for(int j = 0; j < coeff_in_paillier.size(); j++){
                    Wire[] coefficientWire = createProverWitnessWireArray(coeff_in_paillier.
                        get(j).toByteArray().length);
                    for(int i = 0; i < coeff_in_paillier.get(j).toByteArray().length; i++){
                        coefficientWire[i].restrictBitLength(8);
                    }
                    coefficientsInR1CS.add(coefficientWire);
                }

                //Call of our object that will generate the R1CS with the Horner's method
                hornerPolynomialEvalPaillier = new HornerPolynomialEvalPaillier(
                    paillierModulus, coefficientsInR1CS, x, paillierModulusSize);

                //Get the output of our R1CS
                Wire[] cipherTextInBytes = hornerPolynomialEvalPaillier.getOutputWires();

                // Reconstruct the output
                cipherText = new WireArray(cipherTextInBytes).packWordsIntoLargerWords(8, 8);
                makeOutputArray(cipherText,
                        "Output cipher text");
            }

            @Override
            public void generateSampleInput(CircuitEvaluator evaluator) {
                //Set the value of our coefficients in our R1CS
                for(int j = 0; j < coeff_in_paillier.size(); j++){
                    byte[] array = coeff_in_paillier.get(j).toByteArray();
                    for(int i = 0; i < array.length; i++){
                        long num = array[array.length - i - 1] & 0xff;
                        evaluator.setWireValue(coefficientsInR1CS.get(j)[i], num);
                    }

                }
                //Set the value of our Paillier's modulus
                evaluator.setWireValue(this.paillierModulus, paillierModulusValue,
                            LongElement.CHUNK_BITWIDTH);
            }
        };
        //Generate our R1CS circuit
        generator.generateCircuit();
        //Add the input in our R1CS
        generator.genInputEval();
        //Generate the file for libsnark
        generator.prepFilesPaillier();
        //Generate file that will be used by the server to compute the witness
        generator.writeAllPailliersCoefsWithN(this.coeff_in_paillier, this.x, this.publicKey);

        ...
    }
}
```

6.2 HornerPolynomialPaillierFromFileGenerator code

This code is in the directory "snark-protocol-experiment/jsnark/JsnarkCircuitBuilder/src/examples/generators/paillier/"

```
public class HornerPolynomialPaillierFromFileGenerator {
    private ArrayList<BigInteger> coeff_in_paillier;
    private BigInteger x;
    private BigInteger nSquared;

    public HornerPolynomialPaillierFromFileGenerator() {
```

```java
        //Initialisation of our variables
        ...
    }

    public void readAllPailliersCoefsWithN() {
        //Read all coefficients and the input from the file send by the client
        ...
    }

    public BigInteger paillierResCiphered(){
        //Compute the result of the polynomial evaluation with the Horner's method
        ...
    }

    public void genCircuit(){
        BigInteger paillierModulusValue = this.nSquared;
        int paillierModulusSize = paillierModulusValue.bitLength();

        //The following object will be used to create our R1CS
        CircuitGenerator generator = new CircuitGenerator("Horner_polynomial_eval_big_int_paillier_gadget_input_file") {
            ArrayList<Wire[]> coefficientsInR1CS;

            Wire[] cipherText;
            LongElement paillierModulus;

            HornerPolynomialEvalPaillier hornerPolynomialEvalPaillier;

            @Override
            protected void buildCircuit() {
                paillierModulus = createLongElementInput(paillierModulusSize);
                //Initialize the list of our coefficients in our R1CS
                coefficientsInR1CS = new ArrayList<>();
                for(int j = 0; j < coeff_in_paillier.size(); j++){
                    Wire[] coefficientWire = createProverWitnessWireArray(coeff_in_paillier.get(j).toByteArray().length);
                    for(int i = 0; i < coeff_in_paillier.get(j).toByteArray().length; i++){
                        coefficientWire[i].restrictBitLength(8);
                    }
                    coefficientsInR1CS.add(coefficientWire);
                }

                //Call of our object that will generate the R1CS with the Horner's method
                hornerPolynomialEvalPaillier = new HornerPolynomialEvalPaillier(paillierModulus, coefficientsInR1CS, x, paillierModulusSize);

                Wire[] cipherTextInBytes = hornerPolynomialEvalPaillier.getOutputWires(); // in bytes

                // group every 8 bytes together
                cipherText = new WireArray(cipherTextInBytes).packWordsIntoLargerWords(8, 8);
                makeOutputArray(cipherText,
                        "Output cipher text");
            }

            @Override
            public void generateSampleInput(CircuitEvaluator evaluator) {
                //Set the value of our coefficients in our R1CS
                for(int j = 0; j < coeff_in_paillier.size(); j++){
                    byte[] array = coeff_in_paillier.get(j).toByteArray();
                    for(int i = 0; i < array.length; i++){
                        long num = array[array.length - i - 1] & 0xff;
                        evaluator.setWireValue(coefficientsInR1CS.get(j)[i], num);
                    }

                }
                //Set the value of our Paillier's modulus
                evaluator.setWireValue(this.paillierModulus, paillierModulusValue,
                            LongElement.CHUNK_BITWIDTH);
            }
        };
        //Generate our R1CS circuit
        generator.generateCircuit();
        //Evaluation of the circuit to compute the witness and the output
        generator.evalCircuit();
        //Generate file for libsnark (use to compute the proof of the the result)
        generator.prepFiles();
        ...
    }
}
```

Articles

[ako] akosba. *jsnark*. https://github.com/akosba/jsnark.

[Ano] 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.

[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.

[lab] scipr lab. *libsnark*. https://github.com/scipr-lab/libsnark.

[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.

[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.

[rus] ruscot. *snark-protocol-experiment*. https://github.com/ruscot/snark-protocol-experiment.