Cybersecurity MASTER

UNIVERSITÉ Grenoble Alpes

Grenoble INP ensimag

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

Dynamic and Structured Secure Multi-party Computations

Anthony Martinez

<Defense Date>

Research project performed at Laboratoire Jean Kuntzmann

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

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

[June/September] 2022

Abstract

To do later

Contents

Abstract **i**

1 Introduction **1**

2 Groth16 **3**
2.0.1 Overview Groth 16 . . . 4
2.1 Protocol . . . 4
2.1.1 Snark protocol in clear . . . 5
Setup function algorithm . . . 5
Prove function algorithm . . . 6
Verify function algorithm . . . 6
Proof of the equation . . . 6
Problem . . . 7
2.1.2 Snark protocol in ZK . . . 8
2.1.3 Evaluation of the polynomial . . . 8
Setup function algorithm . . . 8
Prove function algorithm . . . 9
Verify function algorithm . . . 9
Summarize of the Groth16 algorithm in a Client/Server architecture . . . 10
2.1.4 Implementation in libsnark . . . 10
2.1.5 Results with libsnark . . . 11
Time of calculation . . . 11
Data to save . . . 12

3 FHE **13**
3.1 Efficiently Verifiable Computation on Encrypted Data . . . 13
3.1.1 LWE definition . . . 13
3.1.2 RLWE definition . . . 13
3.1.3 Homomorphic Encryption (HE) functions . . . 14
HE.ParamGen function . . . 14
HE.KeyGen function . . . 14
HE.Enc function . . . 15
HE.Dec function . . . 15
3.1.4 Homomorphic hash function . . . 15

H.KeyGen function 15
H function 15
H.Eval function 16
3.1.5 Protocol 16
KeyGen function 16
ProbGen function 16
Compute function 16
Verify function 17
Proof of the equality 17
Protocol diagram 17

4 Conclusion **19**

Articles **21**

— 1 —
Introduction

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

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

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

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

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

First we will give an overview of the existing protocols and techniques and compare them with the protocol created by one of the LJK (Laboratoire Jean Kuntzmann) team, implemented with **Relic**. For the rest of the report we'll refer to it by **LJKP** (LJK Protocol) [Dum+].

We'll focus ourself on two tools to create the proof :
- **Zero-Knowledge Succinct Non-Interactive Argument of Knowledge** (zkSNARK) proof
- **Fully Homomorphic Encryption** (FHE)

Each part of the report will describe an algorithm with a protocol, and for each of them we'll give some results and compare them to other protocol.
We consider three type of server :

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

For our case we consider an untrusted server on which we want to know if it has cheat or not. And we have two different scenario :

- **Clear data** : the client don't care if the server see his data and just want the good result
- **Hide data** : the client want the good result and the server don't learn anything on his data

For thoose scenario we will try to perform a polynomial evaluation on a server. So we want a proof of the correctness of the result and for the second we want our data to be ciphered.

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

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

- Computation time : how long it tooks to perform the evaluation and compute the proof
- Extra storage : the data we need to store to compute the proof
- Communication volume : the amount of communication required
- Number of communication : how many communication we need to perform in order to achieve the protocol and the quantity of this data

As a reminder **Zero-Knowledge Proof** enable a prover to convince a verifier that a statement is true without revealing anything else. It have 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 does not reveal anything but the truth of the statement, in particular it doesn't reveal the prover's witness.

Here we give a table with all the protocol we have tried :

Protocol	Polynomial coefficient clear/ciphered	Input clear/ciphered	Verification Public/private	Method	Library	Dynamic
Groth16	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
LJKP	ciphered	clear	private	Paillier and Pairing		Yes

— 2 —
Groth16

This chapter will present how the Groth16 algorithm works, it's based on zkSNARK . Then we'll specifie a protocol with our adaptation to our problem, and our contribution.

We'll describe briefly what are zkSNARK and how they work. It's a type of cryptographic proof protocol that reveal no information about the knowledge we try to prove. Here is the meaning of each letters.

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

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

In order to construct the proof, zkSNARK use polynomials evaluation, the idea is to convert our problem into a "Rank-1 Constraint System" **R1CS**. We defined a **R1CS** as a sequence of groups of three vectors (a, b, c) of size m. And the solution to an **R1CS** is a vector s, where s must satisfy the equation "s.a * s.b - s.c = 0" where "." is the dot product.

Then we convert the R1CS in a "Quadratic Arithmetic Circuit" (**QAP**), defined below, with Lagrange interpolation. After that we can evaluate our equation system on a point and check the equality.

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

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

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

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

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

Now we can find $H \in F^n$ such that 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 then H = t / Z.
See [Pan] for more explanations.
This two websites give a good example of how each transformations are done [But] and [Ano].

2.0.1 Overview Groth 16

[Gro] The aim of the algorithm is to prove to a verifier that we know a secret without revealing it. The algorithm will use a QAP in order to create a "verification key" **vk** and a "proving key" **pk**. With theese two keys we can generate and check a proof that correspond to an equality of polynomials. Anyone who has the key **vk** can check the proof generated by **pk**. This pair of keys need to be generated by yourself or a **trust party**. We can see the protocol as follow :

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

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

2.1 Protocol

Now we have an idea of what are SNARK and how they work, our aim is to use them to create a remote polynomial evaluation. Where the server give a proof of the correctness of the computation. For this we'll use the groth16 algorithm implement in libsnark. All our results

will be on this github repository. First of all we'll detail the steps of the Groth16 algorithm to have a better understanding of it.

Our initial step is to convert our polynomial into a **rank 1 constraint system** (R1CS). Secondly we'll convert it in a QAP and then we'll create our keys and start our protocol.

2.1.1 Snark protocol in clear

Let's says that we have a polynomial P that we wnat to evaluate at the point **w**. The client generate an **R1CS** corresponding to P. And then generate the corresponding **QAP** with the polynomials $\{A_i[X],\ B_i[X],\ C_i[X]\}_{i=0}^{m}$ where he knows the value $a_{1...l,l+1...m} \in F$ to solve it. Let's recall that $a_{1..l}$ correspond to the input and ouput, so our **w** will be in it, and $a_{l+1..m}$ correspond to our intermediate value of our **R1CS**.

Our polynomials A_i, B_i, C_i and Z are of degree n. So we have $l, m \in F$ and $A_i, B_i, C_i \in F^n$. Let's call the QAP **R** such that $R = \{F, m, l, \{A_i, B_i, C_i\}_{i=0}^{m}, \{Z_i\}_{i=0}^{n}\}$
We define three methods Setup, Prove and Verify such that :

$Setup(R) \rightarrow (\sigma, \tau)$: this function will be call by the client to generate the proving key and te verification key which are contain in σ

$Prove(R, \sigma, a_{i...l}, a_{l+1...m}) \rightarrow \pi$: this one is called by the server or someone who want to prove that he knows how to compute the polynomial. More precisely it proves that he knows all the a_i.

$Verify(R, \sigma, a_{i...l}, \pi) \rightarrow 0/1$: anyone who want to check the computation and the proof generated by the function **Prove** can call this function.

Setup function algorithm

Setup(R) :

$\alpha, \beta, \gamma, \delta, s \xleftarrow{\$} F^*$
$\tau = (\alpha, \beta, \gamma, \delta, s)$
$\sigma = (\alpha, \beta, \gamma, \delta, \{s^i\}_{i=0}^{n-1})$
return (τ, σ)

Prove function algorithm

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

$\underline{Prove(R, \sigma, a_{1...l}, a_{l+1...m}) :}$

r, k$\xleftarrow{\$}$F
U = $\alpha + \sum_{i=0}^{m} a_i A_i(s) + r\delta$
V = $\beta + \sum_{i=0}^{m} a_i B_i(s) + k\delta$
We compute t = $(\sum_{i=0}^{m} a_i A_i(s))(\sum_{i=0}^{m} a_i B_i(s)) - \sum_{i=0}^{m} a_i C_i(s)$
And then H = t/Z(s)
$\mathbf{Aa} = \sum_{i=l+1}^{m} a_i A_i$
$\mathbf{Ba} = \sum_{i=l+1}^{m} a_i B_i$
$\mathbf{Ca} = \sum_{i=l+1}^{m} a_i C_i$
$S = \frac{\beta \mathbf{Aa}(s) + \alpha \mathbf{Ba}(s) + \mathbf{Ca}(s)}{\delta}$
W = $S + \frac{H * Z(s)}{\delta} + Uk + rV - rk\delta$
$\pi = (U, V, W)$
return π

Verify function algorithm

$\underline{Verify(R, \sigma, a_{i...l}, \pi) :}$

$\mathbf{Aa2} = \sum_{i=0}^{l} a_i A_i$
$\mathbf{Ba2} = \sum_{i=0}^{l} a_i B_i$
$\mathbf{Ca2} = \sum_{i=0}^{l} a_i C_i$
$Y = \frac{\beta \mathbf{Aa2}(s) + \alpha \mathbf{Ba2}(s) + \mathbf{Ca2}(s)}{\gamma}$
if $UV == \alpha\beta + Y\gamma + W\delta$
    return 1
else
    return 0

You can the paper of [Gro] for more explanations.

Proof of the equation

Now let's prove that the if statement in the function verify is correct. So what we want is to check the following equality :

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

Now let's detail the calculation for the if statement :

For the first term UV we have :

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

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

And for the second term of our equality $\alpha\beta + Y\gamma + W\delta$ we have :

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

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

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

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

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

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

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

Now if we check UV = $\alpha\beta + Y\gamma + W\delta$ we have :
$\alpha\beta + \alpha(\sum_{i=0}^{m} a_i B_i(s)) + k\alpha\delta + \beta(\sum_{i=0}^{m} a_i A_i(s)) + (\sum_{i=0}^{m} a_i A_i(s))(\sum_{i=0}^{m} a_i B_i(s)) + k\delta(\sum_{i=0}^{m} a_i A_i(s)) + r\delta\beta + r\delta(\sum_{i=0}^{m} a_i B_i(s) + rk\delta\delta = \alpha\beta + \beta\sum_{i=0}^{m} a_i A_i(s) + \alpha\sum_{i=0}^{m} a_i B_i(s) + \sum_{i=0}^{m} a_i C_i(s) + k\delta\alpha + k\delta\sum_{i=0}^{m} a_i A_i(s) + kr\delta\delta + r\delta\beta + r\delta\sum_{i=0}^{m} a_i B_i(s) + H(s)Z(s)$
(For more simplicity I have highlighted the terms in the equation that vanished)

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

So our set of function (Setup, Verify, Prove) check the equality and gives the correct answer.

Problem

But with thoose function we have a problem. Since the server knows α, β, δ, x. He can cheat and make any verifier think that he knows the witness even if it's not the case. By this way :
Pick U,V over F at random
Compute W=$\frac{UV - \alpha\beta - \sum_{i=0}^{l}(a_i(\beta A_i(s) + \alpha B_i(s) + C_i(s)))}{\delta}$
Now our equality is :
$UV = \alpha\beta + \frac{\beta\sum_{i=0}^{l} a_i A_i(s) + \alpha\sum_{i=0}^{l} a_i B_i(s) + \sum_{i=0}^{l} a_i C_i(s)}{\gamma}\gamma + \frac{UV - \alpha\beta - \sum_{i=0}^{l}(a_i(\beta A_i(x) + \alpha B_i(x) + C_i(x)))}{\delta}\delta$

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

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

$0 = 0$
So the if statement in the verify function will return true and any verifier will think that the server knows $a_{1..m}$. The next part will give a way to avoid this problem.

2.1.2 Snark protocol in ZK

In order to avoid the problem identify above we'll use pairing with elliptic curves, with the following definition :
Bilinear map is defined by seven elements $(p, G_1, G_2, G_T, e, g, h)$ such that :
- $e : G_1 * G_2 \rightarrow G_T$
- G_1, G_2, G_T are groups of prime order p
- g is a generator of G_1
- h is a generator of G_2
- e(g,h) is a generator of G_T
- $e(g^a, h^b) = e(g,h)^{ab}$

As above let's says that we have a polynomial P that we want to evaluate at the point **w**. The client generate an **R1CS** corresponding to P. And then generate the corresponding **QAP** with the polynomials $\{A_i[X], B_i[X], C_i[X]\}_{i=0}^{m}$ where he knows the value $a_{1...l,l+1...m} \in F$ to solve it. Let's recall that $a_{1..l}$ correspond to the input and ouput, so our **w** will be in it, and $a_{l+1..m}$ correspond to our intermediate value of our **R1CS**.

Our polynomials A_i, B_i, C_i and Z are of degree n. So we have $l, m \in F$ and $A_i, B_i, C_i \in F^n$. Let's call the QAP **R** such that $R = \{F, m, l, \{A_i, B_i, C_i\}_{i=0}^{m}, \{Z_i\}_{i=0}^{n}\}$. And we add in **R** our bilinear map so **R**$= \{p, G1, G2, Gt, e, g, h, l, \{A_i, B_i, C_i\}_{i=0}^{m}, \{Z_i\}_{i=0}^{m}\}$

For the same method Setup, Prove and Verify we just change their algorithm and their output.

2.1.3 Evaluation of the polynomial

Setup function algorithm

With p prime, g the generator of G1, h the generator of G2 and e such that e(g,h) is the generator of Gt.
Setup(R) :

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

Send **pk** to the prover and **vk** to the verifier.

Prove function algorithm

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

$\underline{Prove(pk, a_{1...l}, a_{l+1...m}) :}$

r, k$\xleftarrow{\$}$F

$\mathbf{Aa} = \sum_{i=0}^{m} a_i A_i$

$\mathbf{Ba} = \sum_{i=0}^{m} a_i B_i$

$\mathbf{Ca} = \sum_{i=0}^{m} a_i C_i$

U = $\alpha_1(\Pi_{i=0}^{m} \mathbf{S1}_i^{\mathbf{Aa}_i})(\delta_1)^r$

V1= $\beta_1(\Pi_{i=0}^{m} \mathbf{S1}_i^{\mathbf{Ba}_i})(\delta_1)^k$

V2= $\beta_2(\Pi_{i=0}^{m} \mathbf{S2}_i^{\mathbf{Ba}_i})(\delta_2)^k$

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

And then $H_i = t_i / Z_i$

$S = \Pi_{i=l+1}^{m} (\mathbf{Sa1}_i)^{a_i}$

W = $\frac{S(\Pi_{i=0}^{n} (\mathbf{Sb1}_i)^{H_i}) U^k \mathbf{V1}^k}{{\delta_1}^{rk}}$

$\pi = (U, \mathbf{V2}, W)$

return π

Verify function algorithm

$\underline{Verify(vk, a_{1...l}, \pi) :}$

$Y = \Pi_{i=0}^{l} (\mathbf{Sc1}_i)^{a_i}$

if $e(U, \mathbf{V2}) == e(\alpha_1, \beta_2) e(Y, \gamma_2) e(W, \delta_2)$

    return 1

else

    return 0

Now that we have solved our previous problem let's verify that the if statement still check the same equality : $(\sum_{i=0}^{m} a_i A_i(s))(\sum_{i=0}^{m} a_i B_i(s)) = \sum_{i=0}^{m} a_i C_i(s) + H(s)Z(s)$
Here is the detailled calcul for the if statement in verify :
$e(U, \mathbf{V2}) = e((\alpha_1 \Pi_{i=0}^{m} \mathbf{S1}_i^{\mathbf{Aa}_i} (\delta_1)^r), (\beta_2 \Pi_{i=0}^{m} \mathbf{S2}_i^{\mathbf{Ba}_i} (\delta_2)^k))$

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

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

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

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

For our both side we have the same equation as the proof in 4.4 just in the g_T exponent. So our equality is still verified.

Summarize of the Groth16 algorithm in a Client/Server architecture

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

Now what we want is to perform a remote polynomial evaluation on a given point with a proof of correctness. So using the **Groth16** algorithm as describe above we just want that the server gave with the proof the a_i corresponding to the output of our **R1CS**. Which correspond to the result of our polynomial evaluation on our point.

From all we said before we have implemented a remote polynomial evaluation based on libsnark which implement the **Groth16** algorithm. The next chapter give some explanation of it.

2.1.4 Implementation in libsnark

All the code that we'll explain in this section is given in this github repository. Libsnark already implement the following properties "r1cs_ppzksnark_generator" (corresponding to the Setup function), "r1cs_ppzksnark_prover" (Prove function) and "r1cs_ppzksnark_verifier_strong_IC" (Verify function). I added a function "create_constraint_horner_method", that generate an **R1CS** from a polynomial P with the horner's method. It's based on the Horner's rules which

is the following : With a_i the coefficients of our polynomial $a_0 + a_1x + a_2x^2 + ... + a_nx^n = a_0 + x(a_1 + x(a_2 + ... + x(a_{n-1} + xa_n)...))$ After that we can give our **R1CS** to the "r1cs_ppzksnark_generator" function to generate the keys.

Then I gave a method to compute the a_i, corresponding to the witness and the output, from the server. After that we can give them to the "r1cs_ppzksnark_prover" function which give a proof of the computation.

And finally we have to run the "r1cs_ppzksnark_verifier_strong_IC" function on the given proof to see if the computation is correct or not.

Here is our main algorithm given in the "libsnark-experiment/src/snark_polynomial_in_clear.cpp" file.

main() :

1 : P ← Generate a random polynomial
2 : R ← create_constraint_horner_method(P)
3 : x ← choose random point of evaluation
4 : (vk, pk) ← r1cs_ppzksnark_generator(R)
5 : a_i ← Calculate the witness and output of R with x as input
6 : π ← r1cs_ppzksnark_prover(a_i)
7 : res ← r1cs_ppzksnark_verifier_strong_IC(π)
8 : if res == True : the a_i outputs are good
9 : else not the good result don't trust the server

In an architecture Client/Server the steps 1 to 4 are executed by the Client or a trust party in the setup phase. The steps 5 to 6 are executed by the server or the person who want to calcultate P(x) and give a proof of his result. And the last steps are executed by the Client/Trust party.

Now that we have a function that simulate this interaction we have made some tests. To see, depending of the size of the polynomial, how the computation time and the size of data we need to store evolves. Our results are in the next chapter.

2.1.5 Results with libsnark

First we have measure the computational time in seconds during the different phases of the protocol. We compare our results with the protocol created by one of the LJK team, implemented with **Relic**. This protocol use Paillier's encryption and we make the comparaison depending of the Paillier's key size.

Time of calculation

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

And now we compare with Relic the amount of data we need to store or send through commu-

nication during our protocol. The size of data is given in bits.

Data to save

	Libsnark				
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	914940	914940	254	2548
512	3629	1828860	1828860	254	2548
1024	3629	3656700	3656700	254	2548
2048	3629	7312380	7312380	254	2548
4096	3629	14623740	14623740	254	2548
8192	3629	29246460	29246460	254	2548
16384	3629	58491900	58491900	254	2548
32768	3629	116982780	116982780	254	2548
65536	3629	233964540	233964540	254	2548
131072	3629	467928060	467928060	254	2548

	Relic				
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	722944	722944	256	768
512	5376	1443840	1443840	256	768
1024	5376	2885632	2885632	256	768
2048	5376	5769216	5769216	256	768
4096	5376	11536384	11536384	256	768
8192	5376	23070720	23070720	256	768
16384	5376	46139392	46139392	256	768
32768	5376	92276736	92276736	256	768
65536	5376	184551424	184551424	256	768
131072	5376	369100800	369100800	256	768

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

— 3 —
FHE

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

3.1 Efficiently Verifiable Computation on Encrypted Data

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

3.1.1 LWE definition

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

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

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

3.1.2 RLWE definition

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

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

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

3.1.3 Homomorphic Encryption (HE) functions

We define the following functions for our RLWE.

HE.ParamGen function

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

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

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

$D_{\mathbb{Z}^n,\sigma}$
ZO_n

HE.KeyGen function

HE.KeyGen(1^λ) :

$a \xleftarrow{\$} R_q$
$s, e \xleftarrow{\$} D_{\mathbb{Z}^n,\sigma}$
$b = as + pe$
$dk = s$
$pk = (a, b)$
return dk, pk

HE.Enc function

m is the message we want to cipher with $m \in R_q$
$HE.Enc$(pk, m) :

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

HE.Dec function

$c \in \mathbb{Z}_q[X][Y]$ is a ciphertext with $c = c_0 + c_1 Y + c_2 Y^2$, $c_i \in \mathbb{Z}_q[X]$
$HE.Dec$(dk, c) :

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

3.1.4 Homomorphic hash function

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

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

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

H.KeyGen function

H.KeyGen() :

$\alpha, \beta \xleftarrow{\$} \mathbb{Z}_q$
$k = (\alpha, \beta)$
return k

H function

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

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

H.Eval function

On input two values $v_1, v_2 \in \mathbb{Z}_q$ and an operation f_g which is addition or multiplication
H.Eval(f_g, v_1, v_2) :
$res = f_g(v_1, v_2)$
return *res*

3.1.5 Protocol

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

KeyGen function

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

ProbGen function

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

Compute function

Compute(PK, σ_x) :
$\gamma = \sum_{i=0}^{t} x^i \gamma_i$

$G_t = \prod_{i=0}^{t} (G_{T,i})^{x^i}$
$\sigma_y = (\gamma, G_t)$
return σ_y

Verify function

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

Proof of the equality

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

Protocol diagram

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

— 4 —
Conclusion

Some text...

Articles

[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.

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

[Dum+] Jean-Guillaume Dumas et al. “VESPo: Verified Evaluation of Secret Polynomials”. In: (). URL: https://hal.archives-ouvertes.fr/hal-03365854.

[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.

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

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

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