CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

TRABAJO FIN DE GRADO

Cryptographic methods for secure delegation of computation in electronic voting applications

Autor: Anaïs Querol Cruz

Director: Manuel Carro Liñares

Cotutor: Dario Fiore (IMDEA Software)

MADRID, JUNIO DE 2017

Few persons can be made to believe that it is not quite an easy thing to invent a method of secret writing which shall baffle investigation. Yet it may be roundly asserted that human ingenuity cannot concoct a cipher which human ingenuity cannot resolve... It may be observed, generally, that in such investigations the analytic ability is very forcibly called into action; and for this reason, cryptographical solutions might with great propriety be introduced into academies as the means of giving tone to the most important of the powers of the mind.

Edgar Allan Poe; *A Few Words On Secret Writing*, 1841

To those few persons

ACKNOWLEDGEMENTS

Four years ago, hesitant about my future as a mathematician or an artist, I decided to study a degree in Computer Engineering. I would like to show my most sincere gratitude to the Technical University of Madrid for having provided access to this flamboyant area and in particular to some of the most vocational professors I have ever had the pleasure to meet: Damiano Zanardini, Pablo Nogueira Iglesias, Mª Luisa Córdoba Cabeza and Antonio Tabernero Galán. I am grateful to IMDEA Software Foundation for offering my first work experience, concretely to Manuel Carro Liñares who has been guiding me despite his great responsibilities as Director. Little did I know this opportunity would fulfil my aspirations for cryptography, working together with my advisor Dario Fiore; whose wisdom and patience is the reason why this project is a reality. Kind recognition to my school, institute and EPM, for having seeded, watered and harvested my passion for numbers. As ever, to my family, friends and partner; for being who you are, for helping me unconditionally and for encouraging me into academia.

ABSTRACT

Este Trabajo de Fin de Grado comprende un estudio teórico de los esquemas de voto electrónico, desde un punto de vista criptográfico. Concretamente, trata de encontrar en las pruebas SNARK una alternativa más barata (en cuanto a comunicación y computación) a los protocolos actuales de verificación de barajado de votos, los cuales resultan inviables en la práctica.

This Present Final Term Project comprises a theoretically oriented study on electronic voting schemes, from the point of view of cryptography. Concretely, it aims to find in SNARK proofs a cheaper alternative (both communication and computation) over current verifiable shuffling protocols for ballots, which turn out unfeasible in practice.

Palabras clave: criptografía, seguridad, voto electrónico, delegación de computación, barajado verificable, conocimiento cero, pruebas SNARK, QAPs, circuitos aritméticos, teoría de grupos, curvas elípticas

Keywords: cryptography, security, electronic voting, secure delegation of computation, verifiable shuffle, zero knowledge, SNARK proofs, QAPs, arithmetic circuits, group theory, elliptic curves

INDEX

List of Tables

Table **Page**

List of Figures

Figure **Page**

GLOSSARY

$\mathcal{L}$ Language. 11

$\mathcal{P}$ Prover. 6, 9, 10

$\mathcal{R}$ Relation. 11

$\mathcal{V}$ Verifier. 6, 9, 10

CV Confirmation Value. 15, 17, 18

ABC Attribute-Based Credentials. 13

BB Ballot Box. 14, 16–18, 37, 39

BG Bayer-Groth Minimal Shuffle Protocol. 19, 20, 35, 36, 44–46

CG Code Generator. 14, 17, 19

CM Confirmation Message. 17, 18

CRS Common Reference String. 10

CSAT Circuit Satisfiability Problem. 11

DoS Denial of Service. 8

EA Election Authority. 14, 15, 19

FC Finalization Code. 15, 18

FDH Full Domain Hash. 15

FFT Fast Fourier Transform. 44

KEA Knowledge of Exponent Assumption. 9

NIZK Non-Interactive Zero-Knowledge. 10, 16

NP Nondeterministic Polynomial Time. 5, 10, 19

PBB Public Bulletin Board. 14, 15, 18

PCP Probabilistic Checkable Proof. 10

QAP Quadratic Arithmetic Program. 11, 39, 41–44

QSP Quadratic Span Program. 10, 11

RC Return Code. 15, 17

RF Reference Value. 15, 17

ROM Random Oracle Model. 45

RSA Rivest Shamir Adleman signature scheme. 15

SHVZK Special Honest Verifier Zero Knowledge. 23–27, 29–33

SNARG Succinct Non-interactive Argument. 8

SNARK Succinct Non-interactive Argument of Knowledge. 8–10, 36, 38, 42, 43, 45

Tor The Onion Router. 13

VC Verification Card. 14, 15, 17

VD Voting Device. 16, 17, 19

VS Voting Server. 16, 17, 19

ZK Zero Knowledge. 9, 20

zk-SNARK Zero Knowledge SNARK. 9, 10, 35, 36, 43, 44

CHAPTER 1

INTRODUCTION

If there is anything everyone will agree, that is bureaucratic procedures are very time consuming. That may be the reason why technology is fighting for its own place in everyday objects. Some examples are e-mail, which have virtually substituted handwritten letters, DNI-electrónico, an identity for both real and virtual, and Bitcoin, as opposed to traditional currency. The aim of DataMantium project is to take a step forwards on electronic voting. In the near future, democratic decisions will be made anywhere, with very little else than a few taps on our devices.

1.1 Electoral Modernization

Ancient Greece, 500 BC. Democracy is first introduced by Athenians as a system of government where citizens were able to exercise power. Back from the times of city states, democratic states have not changed much in terms of procedure. The people in smaller villages could make decisions directly by discussing altogether. As this became harder to organize, citizens elected representatives from among themselves. As we all know, it was not until the Age of Enlightenment that democracy was reinstated in Europe.

Ever since the late 18th century, democratic procedures have remained unaltered. Law will determine eligible voters, who can submit their choice via one ballot at their corresponding electoral colleges. This piece of paper is deposited into an urn and will only be revealed, along with all the other ballots, when election day is over.

At this moment, tally phase is conducted to obtain the final result. Whoever gets the majority of votes in favour becomes the winner of the election.

Despite this method has a familiar ring to it, it is about time to adapt traditional voting to the Internet. Not only would it be cheaper and more comfortable for the users, since it would require no kind of displacement, but could also offer many additional properties. Perhaps, the most obvious question anyone wonders when results become public, is whether their vote was counted or got lost at some point. The good news is this can be achieved thanks to cryptography and information security, as well as these following desirable features.

Authorization System makes sure only eligible voters will actually be able to vote. Using *reductio ad absurdum*, a non eligible voter could steal some eligible voter's identity card and try to trick election authorities that he is such person. Therefore, a non authorized person could have submitted a ballot.

Unicity Each election settling has its own rules. One of them determines the maximum amount of ballots that each voter can submit. This is meant to prevent malicious entities from double voting. In real life this is performed using such list of eligible voters.

Authentication System makes sure entities who want to vote hold their actual identities. However, a malicious citizen could steal someone's identity card and try to trick election authorities that he was in fact who claims to be. Therefore, this malicious entity could have voted twice.

Confidentiality Voter's choice is kept secret throughout the whole process, thanks to the use of encryption schemes. Guessing someone's options means either such voter revealed his vote intentionally or encryption scheme was broken, which could only happen with negligible probability.

Uncoercity A voter cannot prove he voted in a particular way. This prevents citizens to sell their vote.

Integrity Every ballot value remains unaltered throughout the whole electoral process. Nowadays, we must trust administrative assistants who are supposed to count votes correctly under supervision of representatives of different political parties.

Untraceability Shuffling and randomizing ballots make it impossible to trace choices back to voters identities. In real world, urns are opened to start tally phase. The order in which ballots will show up is supposed to be random, so untraceability depends on the assumption that no camera has recorded the movement of any ballot.

Anonymity Voters privacy is to be maintained all the time. As a consequence, no one could even guess who voted or who did not. In real life, there exist lists of eligible voters identities whose entries are crossed out when they vote. This means, there is real evidence of actual voters.

Verifiability This property is solely achievable in computer settings. When delegating computations to a third party, there exists methods for the client to verify calculations were conducted properly. A voting scheme is universally verifiable if any entity can verify the totality of ballots were correctly counted, whereas it allows designated verification if each individual voter can check their own ballot was tallied.

1.2 DataMantium Project

IMDEA Software Institute (*Instituto Madrileño de Estudios Avanzados*) participates in a new project coordinated by Scytl, funded by MINECO (*Ministerio de Economía*). The goal of this project, entitled DataMantium, is to develop security mechanisms to protect integrity and privacy of users' data and processes in untrusted Cloud scenarios.

The work presented in this Final Term Project was developed as part of DataMantium research phase, addressing cryptographic aspects of online voting. I was supervised by Manuel Carro Liñares [1] and coadvised by Dario Fiore [2].

[1] `http://software.imdea.org/es/people/manuel.carro/`
[2] `http://www.dariofiore.it/`

CHAPTER 2

STATE OF THE ART

This chapter will introduce some terminology and notation which will be used throughout this document. The first section provides insight into electronic voting schemes which are currently in use. The second section introduces verification techniques for Cloud environments. The third section shows today's situation on mix-net verification protocols. The fourth section explains the most recent construction of short proofs of knowledge. The fifth and sixth sections give some insight on how these proofs can be merged with a new characterization of class NP.

2.1 Online Voting

Electronic voting became a reality back in 1960s, when punched cards debuted. From then on, different machines have been used to record voters choices. However, these settings seem quite outdated in comparison to the problem issued in this document. For the sake of clarity, the term e-voting will be used to refer to online voting, which was first used in early 2000s. Some of the first countries to introduce this kind of systems were USA and Switzerland, in 2004. Online voting became popular within the latter, given that they carry out around 3 or 4 referendums a year.

There are some legal aspects why online voting is not yet implemented throughout the globe. Depending on specific verification characteristics of a voting scheme, there is a maximum percentage of electorate who will be allowed to use an online system.

There exist three levels, which determine the amount of online voters: cast-as-intended, to check the ballot to be emitted corresponds to voter's options; recorded-as-intended, to check ballots have been stored correctly; and counted-as-recorded, to audit final result is in fact the sum of confirmed ballots.

The biggest electoral technology company in the world is called Scytl. It was born within Cryptography Department at Universidad Autónoma de Barcelona, with the aim of making secure online voting a reality and a real alternative to paper-based elections. Nowadays, 21 countries have introduced e-voting systems, among which 19 use Scytl technology. The starting point of this document will be Scytl proposal from 2015 for the canton of Neuchâtel [1], which presents a protocol that provides cast-as-intended verification to be used by up to 50% of the electorate. This scheme is an evolution of the voting protocol used in Norway in 2011 and 2013 [2]. However, there are many other e-voting companies out there. For instance, Council of Madrid recently performed a local online voting procedure to engage citizens in civic decisions, using so called nVotes platform.

2.2 Verifiable Computation

There exist many problems with respect to migrating electoral systems to Cloud environments. When voters submit their ballots, it is assumed that the process of counting votes is delegated to some powerful server, which has access to all the inputs. Then election authorities publish the final result and citizens and auditors would like to have a tool to make sure this is a correct result. Needless to say, this check should be less costly for the client than evaluating the whole computation by himself. This property is essential for e-voting schemes, where devices would be simple smartphones or limited computers. This issue is solved using techniques on verifiable security for delegation of computation [3].

Two entities define these schemes. There is a client, the verifier $\mathcal{V}$, who wants the result of a calculation $y = f(x)$ for which he does not have enough computational power. He decides to hire Cloud services, the prover $\mathcal{P}$, who could perform this computation and send y to the verifier. Verifiable computation constructions provide the verifier with a proof to check the result, with no need of actually computing $f(x)$, therefore running in shorter time.

Despite the fact that this concept is easy to understand, it is trickier to figure out how these schemes might work. Let me show a real world setting which will surely make up your mind [4]. Imagine two friends, Merlin and Arthur, discussing about the colour of two balls. The first one claims there is one green and another red. However, colourblind Arthur, argues they are both the same colour. Then, they come up with a brilliant idea.

When Arthur is holding the spheres with both hands, Merlin turns back and asks Arthur to either swap the balls or keep the same position. Then Merlin looks at him and answers whether Arthur swapped the spheres. The first time, Arthur may think he was just lucky to guess with 0.5 probability. As they repeat this experiment k times, chances of Merlin to correctly guess Arthur's decisions descend to $2-k$, which equals the probability of tossing k coins and getting k consecutive heads. As we know, this has negligible chance of happening for large k, unless Merlin is right and is in fact able to distinguish both balls perfectly. As a consequence, Merlin, the prover, convinces Arthur, the verifier, that two balls have different colour.

Depending on specific calculations, different constructions would be implemented. As an example, last semester I did a research internship at IMDEA Software where we studied improvements of the protocol described in the recent paper [5], which presented a proof system of the result of a multivariate polynomial. More specifically, given K vectors of n elements and a polynomial C of size s and degree d, verifying the batch evaluation of the circuit costs Arthur $\widetilde{\mathcal{O}}(K(n+d)+s)$. This is computationally cheaper for the verifier, since evaluating the polynomial would have cost $\mathcal{O}(Ks)$ time.

2.3 Shuffling Protocols

Mix-nets are protocols used in e-voting schemes to provide anonymity through shuffling ballots. To grant confidentiality, these do not show its plain content but they are ciphered. Mix-nets are the equivalent in real life to electoral urns, where ballots are deposited in random order inside a unique envelope to prevent third parties from reading them. However, if ballots were merely permuted, it would be possible to identify their final position due to the fact that every vote is different from each other. Therefore, shuffles are permutations of encrypted ballots, which in turn re-encrypts them to avoid traceability.

It is essential to grant confidentiality at all times that ballots are not decrypted until shuffling step is over, what brings us to the following question. Homomorphic schemes allow us to compute a function of a hidden value with no need to reveal such secret. More technically, the encryption of the multiplication of two messages equals the product of the encryption of both messages separately. That is, $\varepsilon_{pk}(M_1 M_2; \rho_1 + \rho_2) = \varepsilon_{pk}(M_1; \rho_1)\varepsilon_{pk}(M_2; \rho_2)$, which some schemes such as ElGamal satisfy. Some protocols also use homomorphic commitment schemes, such as Pedersen scheme [6], which satisfies $\mathrm{com}_{\mathrm{ck}}(a+b; r+s) = \mathrm{com}_{\mathrm{ck}}(a; r)\mathrm{com}_{\mathrm{ck}}(b; s)$.

The most efficient argument for correctness of a shuffle [7] is interactive and has sublinear communication complexity, efficient prover and lower computation cost for the verifier compared to previous work [8]. It assumes homomorphic encryptions for the ballots and provides zero knowledge. Their implementation proves and verifies 100,000 ElGamal ciphertexts (ballots) in around 2 minutes, using a proof of length 0.7 MB.

2.4 SNARK Proofs

Scalability is the main obstacle electronic voting is facing. The most common way to determine an algorithm's complexity, so called Big O notation, consists of expressing the number of operations as a function of its input length. Considering an electoral process at national level, the system could easily receive millions of ballots. Whereas network activity might also be a handicap, another cryptographic problem arises. Creating proofs of the shuffling becomes a hard task, even for a powerful computer. With that said, client computation is too demanding and proof length takes too much storage space. In order to protect from DoS attacks, interaction between voters and server must be diminished.

Some recent constructions offer efficient proofs using constant round interaction, called succinct non-interactive arguments (SNARGs) of knowledge (SNARKs) [9]. SNARK can be defined as a triple of algorithms $\Pi : \{G(1^\lambda, T) \rightarrow \mathrm{crs}, P(\mathrm{prs}, y, w) \rightarrow \pi, V(\mathrm{vst}, y, \pi) \rightarrow b\}$ which satisfy following three properties. Completeness refers to the fact that it should not be possible to create proofs of false statements. Succinctness gives bounds to running times and proof length. Concretely, generation algorithm runs in polynomial time $p(\lambda + T)$, prove algorithm in time $p(\lambda + |M| + |x| + T)$, verification algorithm in time $p(\lambda + |M| + |x| + \log T)$ and proof has size $p(\lambda + \log T)$. Finally, adaptive proof of knowledge means no prover can extract a valid proof for a false statement.

The basic idea behind SNARKs relies on homomorphic encryption schemes [10]. Let additive homomorphic encryption $E(x+y) = g^{x+y} = g^x g^y = E(x)E(y)$ for a group generator g with pairing function $e(g^x, g^y) = e(g,g)^{xy}$, if $\mathcal{V}$ sets secret value s and d, maximum polynomial degree, then $\mathcal{P}$ can compute encryption $E(f(s))$ for any polynomial f if he has values $E(s^0), \ldots, E(s^d)$. $\mathcal{V}$ will also publish values $E(\alpha s^0), \ldots, E(\alpha s^d)$, useful to check $\mathcal{P}$ made computations correctly. Then $\mathcal{P}$ would publish $A := E(f(s))$, $B := E(\alpha f(s))$ so that $\mathcal{V}$ can check whether they match using pairing function. That is, $e(A, g^\alpha = E(\alpha s^0)) = e(g^{f(s)}, g^\alpha) = e(g,g)^{\alpha f(s)} = e(g^{\alpha f(s),g} = e(B,g)$. The reason why $\mathcal{V}$ may trust $\mathcal{P}$, is called d-power knowledge of exponent assumption (KEA). This is, he assumes $\mathcal{P}$ is not that powerful to come up with different values A, B which also satisfy pairing function equality.

Moreover, zk-SNARKs are SNARKs which also satisfy zero knowledge property, what means they give a proof for a secret statement such that the only information that can be learned is the prover knows such true statement. This property is highly desirable in a e-voting scheme, since clients would be able to verify certain calculations over the ballots without revealing their values.

So as to clarify the term zero knowledge, a simple example will follow. Suppose two friends, Merlin and Arthur, are outside a cave. This cave has two entrances; one on the left and one on the right. Merlin claims to know a winding path inside the cave which communicates both entrances. However, Arthur does not believe him and Merlin does not want to show such path to his friend. Then, they come up with a brilliant idea. Merlin will enter the cave through whichever entrance Arthur chooses (e.g. left), whereas Arthur will be waiting for him in the outside. If Merlin gets out through the other entrance (e.g. right), that means not only there exists a way from left to right, but also Merlin knows such secret path. Therefore, Merlin gives Arthur a proof of knowledge of this secret, revealing him nothing else but the truth in one round of interaction.

In order to include ZK property to SNARKs, we need to randomize those published elements. The prover will pick a random δ in the group to compute $A' := E(\delta + f(s)) = E(\delta)A$, $B' := E(\alpha(\delta + f(s))) = E(\alpha)^\delta B$ which he will publish instead of A, B. Then $\mathcal{V}$ will proceed as before, checking equality of pairing function and he will learn nothing about secret point s on which $\mathcal{P}$ encrypted the result of a polynomial.

2.5 QSP Characterization

Now the question is whether it is possible to find such proofs of any statement. Probabilistic checkable proofs theorem [11] provided a characterization that shows NP statements have PCPs that can be verified in polylogarithmic time in the size of a classical proof. The class NP is the set of problems that have a polynomial time verification algorithm and are not thought to be solvable in polynomial time. It is assumed there are some problems for which is easy to check a solution but it is hard to find one.

More recent results show it is possible to construct very efficient succinct NIZK proofs without PCPs [12]. They give a new characterization of class NP called quadratic span programs, what means any NP statement can be transformed into a QSP. Q_f consists of a set of polynomials V, W and target polynomial t, so that a QSP accepts an input if a multiple of target polynomial can be expressed as a linear combination of both sets of polynomials. Concretely, if $\mathcal{P}$ finds coefficients restricted to input u such that $th = (a_0 + a_1 v_1 + \cdots + a_m v_m)(b_0 + b_1 w_1 + \cdots + b_m w_m)$ then boolean function $f(u) = 1$.

As in the case of standard SNARKs, given an encoding scheme E with quadratic root detection and image verification, V publishes values $\{E(s_i), E(\alpha s^i)\}_{i\in[0,d]}$. However, now there is not one single polynomial but $2m+1$ of them. Therefore, values $E(t(s))$, $E(\alpha t(s))$, $\{E(v_i(s)), E(\alpha v_i(s))\}_0^m$, $\{E(w_i(s)), E(\alpha w_i(s))\}_0^m$ are published as well. CRS is set up to $E(\gamma), E(\beta_v\gamma), E(\beta_w\gamma)$, $E(\beta_v t(s)), E(\beta_w t(s))$, $\{E(\beta_v v_i(s))\}_1^m$, $\{E(\beta_w w_i(s))\}_1^m$ with secret numbers γ, β_v, β_w. We define $v_{free}(x) = \sum_{k\in I_{free}} a_k v_k(x)$ and $w(x) = \sum_1^m b_i w_i(x)$, where I_{free} is a set of independent indices. Then $\mathcal{V}$ will check the proof, consisting of 7 group elements $V_{free}, V'_{free}, W, W', H, H'$, using pairing function. Concretely, pairings $(E(v_{free}(s)), E(\alpha v_{free}(s)))$, $(E(w(s)), E(\alpha w(s)))$ and $(E(h(s)), E(\alpha h(s)))$ to verify $\mathcal{P}$ evaluated some polynomial built up from CRS. Then paring $Y := E(\beta_v v_{free}(s) + \beta_w w(s))$ to check V, W were actually the polynomials used. Finally, $\mathcal{V}$ checks QSP linear combination holds by $e(E(v_0(s))E(v_{in}(s))V_{free}, E(w_0(s))W) = e(H, E(t(s)))$.

In order to get zk-SNARK proof, we must make some values indistinguishable from random. Concretely, $v_{free}(s)$ is incremented by $\delta_{free}t(s)$ and $w(s)$ is incremented by $\delta_w t(s)$. Note $h(s)$ should be shifted as well, so it will be replaced by $h(s) + \delta_{free}(w_0(s) + w(s)) + \delta_w(v_0(s) + v_{in}(s) + v_{free}(s)) + \delta_{free}\delta_w t(s)$. Now QSP linear combination holds, and its verification reveals nothing else about the secret values but computations were made correctly, with overwhelming probability.

2.6 QAP Transformation

The main inconvenient about QSPs is they only work with boolean circuits. This means to compute any arithmetic function, it first needs to be converted into boolean expressions, which has inherent high cost. The other option is to work with different constructions which can compute arithmetic circuits directly - i.e., circuits composed of additions and multiplications modulo q, where q is the group order.

Analogously to QSPs, strong quadratic arithmetic programs [12] use three sets of m polynomials each V, W, Y and target polynomial $t(x)$, which therefore lead to longer arguments. The problem now is to prove there exist m coefficients $\{a_i\}_1^m$ so that target polynomial divides a linear combination of V polynomials times a linear combination of W polynomials, minus a linear combination of Y polynomials. This is, $t(x)$ divides $(v_0(x) + \sum_{k=1}^{m} a_k v_k(x))(w_0(x) + \sum_{k=1}^{m} a_k w_k(x)) - (y_0(x) + \sum_{k=1}^{m} a_k y_k(x))$. If we do so, we can prove a certain function was developed correctly.

Firstly, it might be a good idea to understand CSAT definition. Let n, h, l be input, witness and output size respectively, circuit satisfiability problem of a circuit $C : \mathbb{F}^n \times \mathbb{F}^h \rightarrow \mathbb{F}^l$ is defined by a relation $\mathcal{R}_C$ whose language is $\mathcal{L}_C$. This language is composed by inputs $\vec{x} \in \mathbb{F}^n$ for which there exist witnesses $\vec{w} \in \mathbb{F}^h$ such that the outputs of circuit C are all 0. In other words, $\mathcal{R}_C = \{(\vec{x}, \vec{w}) \in \mathbb{F}^n \times \mathbb{F}^h : C(\vec{x}, \vec{w}) = 0^l\}$.

Unfortunately, creating a QAP for any arithmetic circuit is not a simple task [13]. To do so, we assume the utilization of existing polynomial time algorithms, as described in previous work [14]. Let C be an arithmetic circuit with bilinear gates as defined above, with a wires and b gates, QAPinst will produce sets of polynomials V, W, Y and target polynomial $t(x)$, which combined create Q of size m and degree d over $\mathbb{F}$. On input the circuit, its input and witness, QAPwit will generate an extended witness $\vec{s}$ for the Q. Some interesting relations derive from building QAPs in this way. The size of the resulting QAP is precisely a, the number of wires of the original circuit C. Moreover, Q's degree equals $d + n + l + 1$, which is essentially the sum of the original degree of C, the length of its input $\vec{x}$ and the length of its output. When discussing optimizations, the choice of less output gates over more wires may gain importance.

CHAPTER 3

RESEARCH

Far from widespread assumption that a research project can be narrowed to just googling, Nobel Prize in Physiology winner Albert Szent-Györgyi said "*Research is to see what everybody else has seen, and to think what nobody else has thought*". Ironically, I did google this quote to illustrate our work throughout these pages. This chapter shows our analysis of state of the art, which has served to create a new verifiable shuffling scheme for online voting.

3.1 Study on Scytl's e-voting Scheme

DataMantium project aims to improve current Scytl's voting scheme [1]. Such improvements range from Tor network with traffic analysis attacks resistance to ABCs for proving voters' eligibility. In order to include better cryptographic primitives, it is a must to study the existing system thoroughly. This section presents an analysis on this voting protocol from a security standpoint, as well as some basic notation.

As in any social event, it is recommended to first know which participants will be involved. In order to there be any electoral procedure, *electoral authorities* must configure it through an open call. Just like in traditional systems, *voters*, who do have voice and vote in the final result, will be authorized to submit their ballot thanks to some private identification codes *registrars* provide to them. After votes are counted, *auditors* may ask for a proof of correctness so as to validate the whole procedure.

Voters will send their ballot to the *voting server* using a *voting device*. Votes are stored in the *ballot box*, BB, with which *code generator* creates some *return codes*, and some public information is published on the *public bulletin board*, PBB, to which algorithms have read-only access.

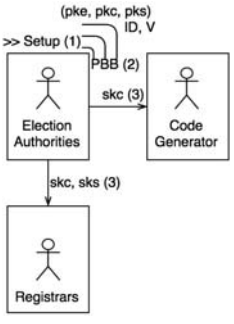
The whole procedure can be split along four phases. *Configuration* is the first step, where the election begins and some needed codes are published on PBB. Then *registration* phase follows, where voters receive their identification information in a *verification card*, VC. Third, *voting* step is carried out, which essentially consists on voters casting their ballot and receiving a confirmation that it was received correctly. Finally, the *counting* phase occurs, where election authorities shuffle the BB, decipher the ballots and publish a verified result. Concretely, this work is focused on the latter stage, where our new shuffling scheme will be included.

3.1.1 Protocol

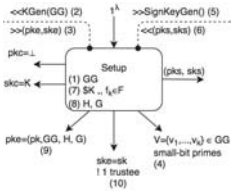
This subsection illustrates the whole voting scheme, including its workflow, algorithms and building blocks. Entities are represented with a stick man within a rectangle, and communication among them (any interaction to share information) is done through open arrows (→). Whenever an entity executes an algorithm, a prompt (>>) appears along with the name of such algorithm, on top of the entity who runs it. Public bulletin board is represented as three curved lines radiating, whereas ballot box is a cube. Storage is represented as a classical pile of disks. Algorithms are shown as round rectangles and their inputs/outputs are represented by triangle arrows (—►). These algorithms will use cryptography blocks, which are shown as rectangles. Each of these calls are represented as dashed lines (•––•). Decisions are shown as simple diamond gates (◇). Note the order in which steps are developed can be deduced from numbers in parenthesis.

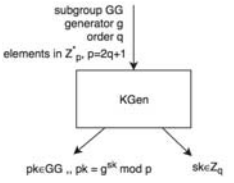
3.1.1.1 Configuration

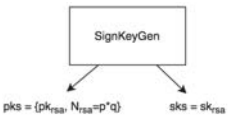
In this phase of the procedure, EAs configure the election using Setup algorithm. Then election, code generation and signing public keys (pk$e$, pk$c$, pk$s$, respectively) are published on PBB along with the list of eligible voters identities ID and the list of possible votes V. EAs send code generation secret key sk$c$ to both CG and registrars, who also receive secret signing key sk$s$.



Upon reception of security parameter λ, Setup(1^λ) algorithm will choose a message group $\mathbb{G}$ of order q with generator g on which ElGamal block KGen($\mathbb{G}$) will generate a pair of keys for the election. The secret key is chosen randomly from $\mathbb{Z}_q$ and pk $\in \mathbb{Z}*_p$ is calculated as $g^{\text{sk}} \mod (p)$, where $p = 2q + 1$. A total number of k different options V are chosen among small primes in this group $\mathbb{G}$ (we will see why in tally phase). Then it calls SignKeyGen() block to generate a pair of signing keys using RSA-FDH. It essentially assigns the sk_{rsa} to the sk$s$, and pk$s$ consists of two elements: pk_{rsa} and the product of two distinct primes. Going back to Setup algorithm, a random K is used to choose pseudorandom function $f_K \in F$, where F is a family of functions, in order to get hash functions H, G. Election public key is configured as the union of ElGamal pk, group $\mathbb{G}$, and hash functions. Election secret key corresponds to ElGamal sk, or a set of shares for different EAs. Finally, pk$c$ is set to void and sk$c$ is that random value K.

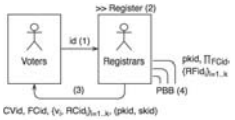




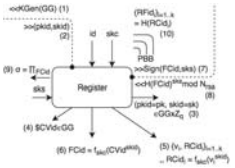


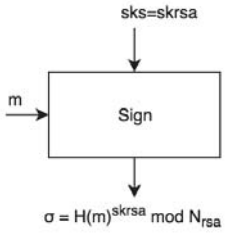
3.1.1.2 Registration

Each voter show his id to registrars, who run Register algorithm to generate his private VC, which consists of a confirmation value, finalization code and return codes for each voting option. Voters are also given their secret key and their public key is published on PBB along with reference values and signature.



Register(id, sk$c$, sk$s$) algorithm calls KGen($\mathbb{G}$) block to get a pair of keys for the voter (pkid, skid). A random confirmation value $\text{CV}_i d \in \mathbb{G}$ is chosen, return codes are computed as $\{v_i, \text{RC}_i = f_{\text{skc}}(v_i^{id})\}_{i=1\ldots k}$, finalization code is $\text{FC}id = f_{\text{skc}}(\text{CV}id^{\text{sk}id})$ and reference values linked to return codes $\{\text{RF}_i\}_{i=1\ldots k}$ are computed hashing return codes of voting options. Then block Sign(FCid, sk$s$) is run to get the signature $\sigma = H(\text{FC}id)^{\text{sk}s} \mod (N_{rsa})$, which will later be referenced as validity proof for finalization code $\Pi_{\text{FC}id}$.

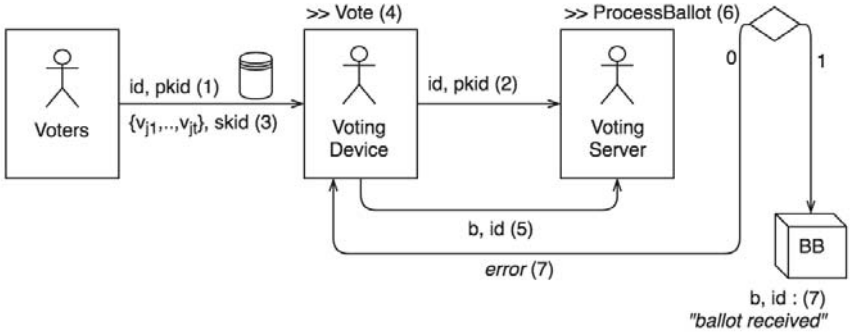




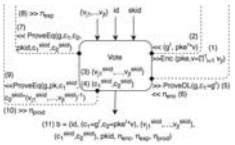
3.1.1.3 Voting

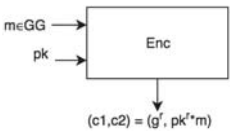
This is the part which requires more interaction from the voters. As such, it needs longer explanation so we will split this phase in five steps.

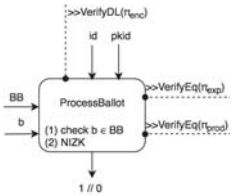
Ballot casting Each voter must insert his id and public key into the VD, which will send them to the server. Then among all voting options, he will just select his intended choices, along with his private key. Then the voting device executes Vote algorithm to generate a ballot for such options and sends it to the server, which VS runs ProcessBallot. If the voter behaved properly, the ballot is introduced into the BB and VS updates its state to *received*. However, if any problem occurs, an error will be displayed on the device.



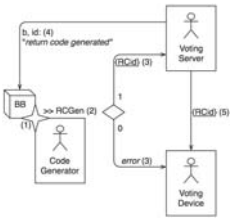
Suppose voting device runs Vote$(id, \text{sk}id, \{v_{jl}\}_{l=1...t})$ algorithm for voter j. The first step will be to encrypt his choices using Enc(pke, v) block, where v is the product of such choices. In return, it gets two ciphertexts $(c_1 = g^r, c_2 = \text{pk}e \cdot v)$ that will be raised to skid, as well as the voting options. Then, three NIZK proofs are computed to prove correct computation of these values: ProveDL(g, c_1) to prove knowledge of randomness r, ProveEq$(g, c_1, c_2, \text{pk}id, c_1^{\text{sk}id}, c_2^{\text{sk}id})$ to prove these are in fact computed by raising ciphertexts to voter's private key, and ProveEq$(g, pk, c_1^{\text{sk}id}, c_2^{\text{sk}id}, (\{v_{jl}^{\text{sk}id}\}_{l=1...t})^{-1})$ to prove raised (c_1, c_2) are the encryption of the product of raised voter's options, due to homomorphism. Finally, the algorithm outputs a ballot $b = (id, (c_1, c_2), \{v_{jl}^{\text{sk}id}\}_{l=1...t}, (c_1^{\text{sk}id}, c_2^{\text{sk}id}), \text{pk}id, \pi_{enc}, \pi_{exp}, \pi_{prod})$. Voting server will run ProcessBallot(BB,$id, b,$ pkid), to confirm voter id is eligible to submit a ballot. It does so by checking inside BB there is no ballot from such voter. Then, VS verifies NIZK proofs running VerifyDL(π_{enc}), VerifyEq(π_{exp}) and VerifyEq(π_{prod}). If everything is fine, server accepts his ballot.



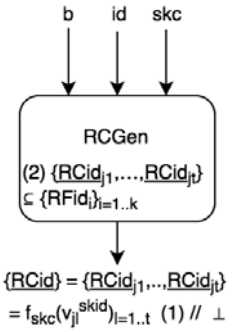




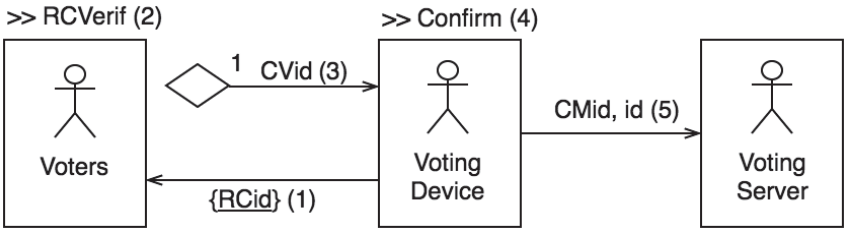
Return code generation CG is notified if there is an update in BB. RCGen algorithm is run to generate return codes for the voting options in the new ballot, which are sent back to the server. If everything goes well, VD shows these values and VS updates its state to *return code generated*.



The server will notify the code generator whenever a new ballot b is introduced into BB, so it can execute algorithm RCGen$(b, id,$ sk$c)$. CG reads the ballot and computes function $f_{\text{sk}c}$ over voting options raised to that voter's secret key. Note if these are correct values, t resulting return codes $\overline{RCid}$ will be a subset of $\{\text{RF}id_i\}_{i=1\ldots k}$, which were calculated in registration phase using the same function f_K.

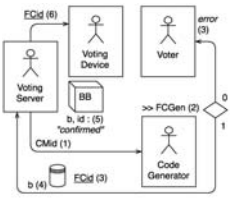


Confirmation message upload Voting device will show return codes to the voter, who will perform RCVerif algorithm to check they match to those in his verification card. If so, he writes his private confirmation value from his VC on the device. Then VD runs Confirm algorithm to generate a confirmation message for this voter and uploads it to the server.

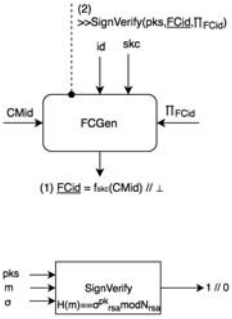


Algorithm RCVerif$(\{v_{jl}\}_{l=1\ldots t}, \{\overline{RCid_l}\}_{l=1\ldots t}$,VC) - to give it a name - simply requires the voter to check if the return codes which appear on the device coincide with those in his card, $\{v_i$,RC$_i\}_{i=1\ldots k}$. If that is the case, Confirm(CV$id, id,$ skid) algorithm computes confirmation message CMid raising voter's confirmation value to his secret key skid, which was stored in the device.

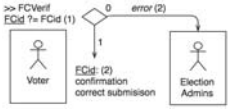
Ballot confirmation Voting server sends confirmation message to code generator that will compute a finalization code. This code is stored in the server along with its ballot and it is forwarded to the device if no problems emerged. Ballot box is updated with the status *confirmed* for that ballot. Otherwise, an error is displayed.



The aim of algorithm FCGen(CMid, id, sk$c$, $\Pi_{\text{FC}id}$) is to compute a finalization code for the voting procedure of the voter id. Given that in registration phase FCid was computed as $f_{\text{sk}c}(\text{CV}id^{\text{sk}id})$, a correct finalization code $\overline{FCid}$ shall be computed as $f_{\text{sk}c}(\text{CM}id)$. Apart from that, signature verification SignVerify(pk$s$, $\overline{FCid}$, $\Pi_{\text{FC}id}$) should accept the signature since $H(\overline{FCid})$ shall equal $\Pi_{\text{FC}id}^{\text{pk}s} \mod N_{rsa}$. Note we defined $\Pi_{\text{FC}id}$ as $H(\text{FC}id)^{\text{sk}s} \mod N_{rsa}$.

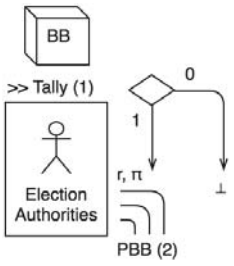


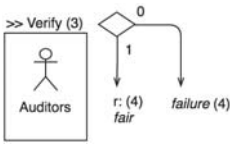
Submission verification At this point, the voter simply checks that finalization code on the screen is the same as the one in his verification card (FCVerif). If so, voting phase is concluded and the voter was able to verify he cast his intended options.



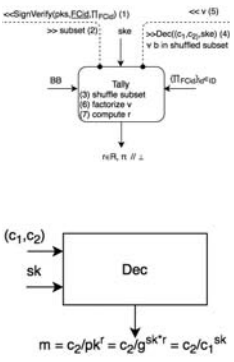
3.1.1.4 Counting

Last but not least, counting phase is performed after voters had submitted their ballots. It is somewhat the most critical part of the protocol, since the whole result depends on the correctness of this step. First, election authorities run Tally algorithm on the ballot box to come up with a result for the election, as well as a proof of correctness, which are made public through PBB. Verification is a very important feature in e-voting schemes. In this scheme, auditors can run Verify algorithm on the public bulletin board to announce fairness of the election.





Tally This step can be seen as the equivalent for emptying electoral urns and counting the result. Unfortunately, in an electronic voting scenario it is a bit more complicated than that. First of all, the algorithm Tally(BB,sk$e$, $\{\Pi_{\text{FC}id}\}_{id\in ID}$) runs SignVerify block over all pairs of finalization codes and their corresponding signature. That is, the algorithm verifies which submissions are properly signed, obtaining the subset of correct indexes among $\{\overline{FCid}\}_{id\in ID}$. For each of these correct values, there exist one corresponding ballot in BB.



Now all valid ballots must be shuffled, which means not only shall they be permuted but also rerandomized - this to prevent traceability to voters. Assuming there are many election authorities, they agree on decrypting shuffled ballots according to the protocol. They run Dec over every pair of ElGamal ciphertexts $(c_1 = g^r, c_2 = \text{pk}e^r \cdot v)$, using the secret key of the election. Note original message v can be simply obtained as c_2/c_1^{sk}, and it was the product of the options chosen by voter id, who remains secret. It is trivial to see now why voting options were small bit primes: in order to get the values of the votes, v must be factorized. Unless factors are small, this is in general a hard task, since there is no polynomial time algorithm to do so. In fact, it is thought to be in class NP - be thankful for that. Once this algorithm is run for every ballot, the only thing that remains is a simple recount of the votes to obtain a final result for the election.

Audit This electronic voting scheme uses BG protocol for verification of the shuffle. Our work consists of finding a better algorithm to do this. Moreover, not only do we want Verify algorithm to be run by official auditors, but also by any voter.

3.1.2 Security Analysis

One of the most relevant consequences of a modularized scheme, apart from the ability to distribute costly computations to third parties, relies on security. It is feasible to split responsibilities between different components. This means in order to attack the system, many supposedly independent entities would have to collude. Otherwise, we would confront the question of "*who watches the watchmen?*".

Therefore, it is essential that we point out undesirable combinations of malicious parties. On the one hand, to keep cast-as-intended verifiability, there are four dangerous pairings. If voting device colludes with code generator, VD could create two ballots using voter's secret key and CG could generate a return code for the wrong ballot and trick the voter. Given that registrars know all return codes voters should get back, registrars could collude with VD so as to send wrong ballots and voters will still confirm them. If VD colludes with the server, two ballots could be generated to show CG the right one though VS will store the wrong ballot. On the other hand, for privacy, we assume none of these scenarios will take place. Despite having distributed keys along election authorities, in case they agree altogether on decrypting ballots prior to shuffling, EAs would discover all voters' choices. Obviously, if contents in verification cards became public, anyone could impersonate voters.

3.2 Analysis of Bayer-Groth Minimal Shuffle

Our starting point is the most efficient protocol in literature for ZK verification of a shuffle [7], BG protocol thereafter. It is, indeed, the protocol currently used in Scytl's voting system. This section presents computational and communication cost of BG protocol, as well as some important notation to be used in forthcoming subsections.

Let $\mathbb{G}$ be a cyclic group of large prime order q with generator G, both Pedersen commitment scheme and ElGamal encryption scheme will be defined over this group. This means commits belong to the group $\mathbb{G}$ and ciphertexts belong to $\mathbb{H} = \mathbb{G} \times \mathbb{G}$. We define $\mathbb{Z}_q^*$ as the set of integers modulo q. A bilinear map is defined as $\mathbb{H}^n \times \mathbb{Z}_q^n \to \mathbb{H}$ by $\vec{C}^{\vec{a}} = (C_1, \ldots, C_n)^{(a_1,\ldots,a_n)^T} = \prod_{i=1}^n C_i^{a_i}$. This operation requires n ciphertext (elements in $\mathbb{H}$ exponentiations, which are $2n$ exponentiations in $\mathbb{G}$.

We will be using ElGamal homomorphic encryption scheme, which satisfies $\varepsilon_{\text{pk}}(M_1 M_2; \rho_1 + \rho_2) = \varepsilon_{\text{pk}}(M_1; \rho_1)\varepsilon_{\text{pk}}(M_2; \rho_2)$. ElGamal scheme consists of three algorithms. Key generation is run to get a random secret key $\text{sk} = x \in \mathbb{Z}_q^*$ and the public key $\text{pk} = G^{\text{sk}}$. To encrypt a message $M \in \mathbb{G}$ we choose a random $\rho \in \mathbb{Z}_q$ and compute the ciphertext in $\mathbb{H} = \mathbb{G} \times \mathbb{G}$ as $\varepsilon_{\text{pk}}(M; \rho) := (G^\rho, \text{pk}^\rho M)$. To decrypt a ciphertext $(C_1, C_2) \in \mathbb{H}$ the decryption algorithm computes $M = C_2 C_1^{-\text{sk}}$. Note that each encryption requires 2 exponentiations of elements in $\mathbb{G}$.

We will be using generalization of the Pedersen commitment scheme, which satisfies $\text{com}_{ck}(a+b; r+s) = \text{com}_{ck}(a; r)\text{com}_{ck}(b; s)$. The key generation algorithm chooses $n+1$ generators of the cyclic group $\mathbb{G}$ and sets the commitment key $ck = (\mathbb{G}, G_1, \ldots, G_n, H)$. To commit to n elements $(a_1, \ldots, a_n) \in \mathbb{Z}_q^n$, we pick randomness $r \in \mathbb{Z}_q$ and compute $\text{com}_{ck}(a_1, \ldots, a_n; r) = H^r \prod_{i=1}^n G_i^{a_i}$. This operation requires (n+1) exponentiations in $\mathbb{G}$. One can also commit to a matrix $A \in \mathbb{Z}_q^{n \times m}$ with columns $\vec{a}_1 \ldots \vec{a}_m$ as $\text{com}_{ck}(A; \vec{r}) = (\text{com}_{ck}(\vec{a}_1; r_1) \ldots, \text{com}_{ck}(\vec{a}_m; r_m))$, which will require $N + m$ exponentiations in $\mathbb{G}$. We also define $\vec{c}^{\vec{b}} = \prod_{j=1}^m c_j^{b_j}$, taking m exponentiations in $\mathbb{G}$, and $\vec{c}^B = (\vec{c}^{\vec{b}_1}, \ldots, \vec{c}^{\vec{b}_m})$, taking m^2 exponentiations in $\mathbb{G}$.

3.2.1 Shuffle Argument

In this section we present both computational and communication cost of an argument of knowledge of a permutation $\pi \in \Sigma_N$ and randomness $\{\rho_i\}_{i=1}^N$ such that for given ciphertexts $\vec{C} = \{C_i\}_{i=1}^N$, $\vec{C}' = \{C_i'\}_{i=1}^N$ we have $C_i' = C_{\pi(i)}\varepsilon_{\text{pk}}(1; \rho_i)$.

3.2.1.1 Communication Cost

The beginning of the protocol has a communication cost of $2m\mathbb{G} + 3\mathbb{Z}_q$ with 4 rounds of interaction. Then a product and multi-exponentiation argument will run in parallel.

$\mathcal{P}$		$\mathcal{V}$	Size
$\vec{c}_A \in \mathbb{G}^m$	$\longrightarrow$		$m\mathbb{G}$
	$\longleftarrow$	$x \in \mathbb{Z}_q^*$	$\mathbb{Z}_q$
$\vec{c}_B \in \mathbb{G}^m$	$\longrightarrow$		$m\mathbb{G}$
	$\longleftarrow$	$y, z \in \mathbb{Z}_q^*$	$2\mathbb{Z}_q$

The first step for the prover is to commit to the permutation. This is done by committing to $\vec{c_A} = \text{com}_{ck}(\vec{a}; \vec{r})$, where $\vec{a} = \pi(i)_{i=1}^N$ and $\vec{r} \longleftarrow \mathbb{Z}_q^m$ is the randomness used. Then the verifier challenges the prover by sending $x \in \mathbb{Z}_q^*$ and the prover answers by sending $\vec{c}_B = \text{com}_{ck}(\vec{b}; \vec{s})$, where $\vec{b} = \{x^{\pi(i)}\}_{i=1}^N$. Next, the verifier sends challenges $y, z \in \mathbb{Z}_q^*$.

3.2.1.2 Verifier Computation

This part of the protocol takes the verifier $(2N + m + n)$ exponentiations:

			Exponentiations
$\vec{c}_{-z}$	:=	$\text{com}_{ck}(-z, \ldots, -z; \vec{0})$	n
$\vec{c}_D$	=	$\vec{c}_A^{\,y} \vec{c}_B$	m
$\vec{C}^{\vec{x}}$	=	$\prod_{i=1}^N C_i^{x^i}$	$2N$

Some non-costly steps of verification is to be done, which consists on computing $\prod_{i=1}^N (yi + x^i - z)$ and checking $\vec{c}_A, \vec{c}_B \in \mathbb{G}^m$. The verifier computes a commit to the challenge z as defined above, $\text{com}_{ck}(-\vec{z} \in \mathbb{Z}_q^N; \{0\}_1^m) = (H^0 \prod_{i=1}^n G_i^{-z}, \ldots, H^0 \prod_{i=1}^n G_i^{-z})$, which requires n different exponentiations. He computes commits to $\vec{d} = y\vec{a} + \vec{b}$ thanks to homomorphic commitment properties as $\vec{c}_D = \vec{c}_A^{\,y} \vec{c}_B$. Given that $\vec{c}_A \in \mathbb{G}^m$, this costs m exponentiations. Finally, he will need to compute $2N$ exponentiations to get $\vec{C}^{\vec{x}}$.

3.2.1.3 Prover Computation

This part of the protocol takes the prover $(2N+2m)$ exponentiations.

			Exponentiations
$\vec{c}_A$	:=	$\text{com}_{ck}(\vec{a};\vec{r})$	$N+m$
$\vec{c}_B$	:=	$\text{com}_{ck}(\vec{b};\vec{s})$	$N+m$

First, the prover commits to the permutation π by committing to the columns of $\vec{a} = \{\pi(i)\}_{i=1}^{N}$ as $\text{com}_{ck}(\vec{a};\vec{r}) = (\text{com}_{ck}(;)H^{r_1}\prod_{i=1}^{n} G_i^{a_{i1}}, \ldots, H^{r_m}\prod_{i=1}^{n} G_i^{a_{im}})$. Similarly, after seeing the challenge he commits to $\vec{b} = \{x^{\pi(i)}\}_{i=1}^{N}$, so that he will demonstrate the same permutation has been used. Note both cases require $m(n+1) = N+m$ exponentiations.

3.2.2 Product Argument

In this section we present both computational and communication cost of an argument that a set of committed values, $\vec{c}_A$ to $A = \{a_{ij}\}_{i,j=1}^{n,m}$, have a particular product $b = \prod_{i=1}^{n}\prod_{j=1}^{m} a_{ij}$ by committing to it.

3.2.2.1 Communication Cost

The total communication cost for the product argument is $(3m+7)\mathbb{G} + (4n+9)\mathbb{Z}_q$. Note that the first move the prover makes may be used to transmit elements for three arguments altogether, resulting in 5 rounds of interaction when run in parallel.

$\mathcal{P}$		$\mathcal{V}$	Size	Section
$c_b \in \mathbb{G}$	$\longrightarrow$		$\mathbb{G}$	Product
$\vec{c}_B \in \mathbb{G}^m$	$\longrightarrow$		$m\mathbb{G}$	Hadamard
	$\longleftarrow$	$x, y \in \mathbb{Z}_q^*$	$2\mathbb{Z}_q$	Hadamard
$c_{A_0}, c_{B_m} \in \mathbb{G}, \vec{c}_D \in \mathbb{G}^{2m+1}$	$\longrightarrow$		$(2m+3)\mathbb{G}$	Zero
	$\longleftarrow$	$x \in \mathbb{Z}_q^*$	$\mathbb{Z}_q$	Zero
$\vec{a}, \vec{b} \in \mathbb{Z}^n, r, s, t \in \mathbb{Z}$	$\longrightarrow$		$(2n+3)\mathbb{Z}_q$	Zero
$c_d, c_\delta, c_\Delta \in \mathbb{G}$	$\longrightarrow$		$3\mathbb{G}$	Single
	$\longleftarrow$	$x \in \mathbb{Z}_q$	$\mathbb{Z}_q$	Single
$\tilde{a}_1, \tilde{b}_1, \ldots, \tilde{a}_n, \tilde{b}_n, \tilde{r}, \tilde{s} \in \mathbb{Z}_q$	$\longrightarrow$		$(2n+2)\mathbb{Z}_q$	Single

Firstly, the prover gives the verifier an argument of knowledge that $c_b = \text{com}_{ck}(b_1 = \prod_{j=1}^{m} a_{1j}, \ldots, b_n = \prod_{j=1}^{m} a_{nj}; s)$ is correct. Then they engage (in parallel) in two SHVZK arguments of knowledge.

In Hadamard product argument, the prover calculates a matrix $B \in \mathbb{Z}_q^{n \times m}$ and sends its commit $\vec{c_B} = (c_{B_1} = \text{com}_{ck}(\vec{b_1}; s_i), \ldots, c_{B_m} = \text{com}_{ck}(\vec{b_m}; s_m))$. The verifier sends back two challenges and they engage in an SHVZK zero argument for the committed values satisfying $0 = \sum_{i=1}^{m-1} \vec{a}_{i+1} * \vec{d}_i - \vec{1} * \vec{d}$. Now the prover wants to show that $0 = \sum_{i=1}^{m} \vec{a}_i * \vec{b}_{i-1}$, for which the prover sends $c_{A_0} = \text{com}_{ck}(\vec{a}_0; r_0)$, $c_{B_m} = \text{com}_{ck}(\vec{b}_m; s_m)$ and $\vec{c_D} = \text{com}_{ck}(\vec{d}; \vec{t} \leftarrow \mathbb{Z}_q^{2m+1})$. The verifier will challenge him with x and the prover will send $\vec{a}, \vec{b}, r, s, t$, which are calculated using such challenge.

In parallel, they will run single value product argument of knowledge of committed single values having a particular product. That is, $c_a = \text{com}_c(\vec{a}; r)$ and $b = \prod_{i=1}^{n} a_i$ is prover's witness. The prover starts by sending commits c_d, c_δ, c_Δ and receives a challenge x. Then he sends values $\tilde{a}_1, \tilde{b}_1, \ldots, \tilde{a}_n, \tilde{b}_n, \tilde{r}, \tilde{s}$ so that the verifier can use them to accept the argument.

3.2.2.2 Verifier Computation

This part of the protocol takes the verifier $(6m + 5n + 5)$ exponentiations.

			Exp
$\{c_{D_i}\}_{i=1}^{m}$	:=	$c_{B_i}^{x^i}$	m
c_D	:=	$\prod_{i=1}^{m-1} c_{B_{i+1}}^{x^i}$	$m-1$
$\prod_{i=0}^{m} c_{A_i}^{x^i}$	=	$c_{A_0} \prod_{i=1}^{m} c_{A_i}^{x^i}$	m
$\prod_{j=0}^{m} c_{B_j}^{x^{m-j}}$	=	$c_{B_m} \prod_{j=0}^{m-1} c_{B_j}^{x^{m-j}}$	m
$\prod_{k=0}^{2m} c_{D_k}^{x^k}$	=	$c_{D_0} \prod_{k=1}^{2m} c_{D_k}^{x^k}$	$2m$
$\text{com}_{ck}(\vec{a}; r)$	=	$H^r \prod_{i=1}^{n} G_i^{a_i}$	$n+1$
$\text{com}_{ck}(\vec{b}; s)$	=	$H^s \prod_{i=1}^{n} G_i^{b_i}$	$n+1$

			Exp
$\text{com}_{ck}(\vec{a} * \vec{b}; t)$	=	$H^t \prod_{i=1}^{n} G_i^{(\vec{a}*\vec{b})_i}$	$n+1$
$c_a^x c_d$	=	$\text{com}_{ck}(x\vec{a} + \vec{d}; xr + r_d)$	1
$c_\Delta^x c_\delta$	=	$\text{com}_{ck}(\{x(\delta_{i+1} - a_{i+1}\delta_i - b_i d_{i+1})$ $-\delta_i d_{i+1}\}_{i=1}^{n-1}; xs_x + s_1)$	1
$\text{com}_{ck}(\tilde{a}_1, \ldots, \tilde{a}_n; \tilde{r})$	=	$\text{com}_{ck}(xa_1 + d_1, \ldots, xa_n + d_n; xr + r_d)$	$n+1$
$\text{com}_{ck}(\{x\tilde{b}_{i+1} - \tilde{b}_i\tilde{a}_{i+1}\}_{i=1}^{n-1}; \tilde{s})$	=	$H^{\tilde{s}} \prod_{i=1}^{n-1} G_i^{x\tilde{b}_{i+1} - \tilde{b}_i\tilde{a}_{i+1}}$	n

The verifier accepts if $c_b \in \mathbb{G}$ and both Hadamard product and single value product SHVZK arguments are convincing. Despite the fact that these are run in parallel, we will analyse Hadamard argument in the first place.

Hadamard product argument as such is not time-consuming for the verifier. He first has to check $c_{B_2}, \ldots, c_{B_{m-1}} \in \mathbb{G}, c_{B_1} = c_{A_1}, c_{B_m} = c_b$ and define $c_{-1} = \text{com}_{ck} = (-\vec{1}; 0) = \prod_{i=1}^{n} \frac{1}{G_i}$ using no exponentiations. Then he computes $c_{D_i} = \{c_{B_i}^{x^i}\}_{i=1}^{m}$, which requires m exponentiations, and $c_D = \prod_{i=1}^{m-1} c_{B_{i+1}}^{x^i}$ with $m-1$ exponentiations. Unfortunately, one cannot avoid this computation reusing c_{D_i}.

They engage in a zero argument where the verifier will accept if $\vec{a}, \vec{b} \in \mathbb{Z}_q^n, r, s, t \in \mathbb{Z}_q$, $\vec{c}_D \in \mathbb{G}^{2m+1}$, $c_{D_{m+1}} = \text{com}_{ck}(0; 0) = 1, c_{A_0}, c_{B_m} \in \mathbb{G}$, and the three following equalities hold. First, $\prod_{j=0}^{m} c_{B_j}^{x^{m-j}} = c_{B_m} \prod_{j=0}^{m-1} c_{B_j}^{x^{m-j}}$, with m exponentiations, must equal $\text{com}_{ck}(\vec{b}; s) = H^s \prod_{i=1}^{n} G_i^{b_i}$, with $n+1$ exponentiations. Similarly, $\prod_{i=0}^{m} c_{A_i}^{x^i} = c_{A_0} \prod_{i=1}^{m} c_{A_i}^{x^i}$, calculated with m exponentiations, must equal $\text{com}_{ck}(\vec{a}; r) = H^r \prod_{i=1}^{n} G_i^{a_i}$, with $n+1$ exponentiations. Lastly, $\prod_{k=0}^{2m} c_{D_k}^{x^k} = c_{D_0} \prod_{k=1}^{2m} c_{D_k}^{x^k}$, with $2m$ exponentiations, must equal $\text{com}_{ck}(\vec{a} * \vec{b}; t) = H^t \prod_{i=1}^{n} G_i^{(\vec{a}*\vec{b})_i}$, taking $n+1$ exponentiations.

The remaining SHVZK gives an argument of knowledge of committed single values having a particular product. The verification phase consists on checking $c_d, c_\delta, c_\Delta \in \mathbb{G}$, $\tilde{a}_1, \tilde{b}_1, \ldots, \tilde{a}_n, \tilde{b}_n, \tilde{r}, \tilde{s} \in \mathbb{Z}_q, \tilde{b}_1 = \tilde{a}_1, \tilde{b}_n = xb$. Moreover, the verifier checks $c_a^x c_d = \text{com}_{ck}(\tilde{a}_1, \ldots, \tilde{a}_n; \tilde{r})$, which takes $(n+2)$ exponentiations, and $c_\Delta^x c_\delta = \text{com}_{ck}(x\tilde{b}_2 - \tilde{b}_1\tilde{a}_2, \ldots, x\tilde{b}_n - \tilde{b}_{n-1}\tilde{a}_n; \tilde{s}) = H^{\tilde{s}} \prod_{i=1}^{n-1} G_i^{x\tilde{b}_{i+1} - \tilde{b}_i\tilde{a}_{i+1}}$, with $(n+1)$ exponentiations.

3.2.2.3 Prover Computation

This part of the protocol takes the prover $(N+7m+4n+1)$ exponentiations.

			Exponentiations
c_b	:=	$\text{com}_{ck}(\prod_{j=1}^{m} a_{1j}, \ldots, \prod_{j=1}^{m} a_{nj}; s)$	$n+1$
$\vec{c}_B$	:=	$[c_{A_1}, \{\text{com}_{ck}(\vec{b}_j; s_j)\}_{j=2}^{m-1}, c_b]$	$N+m-2n-2$
$\{c_{D_i}\}_{i=1}^{m}$	:=	$c_{B_i}^{x^i}$	m
c_D	:=	$\prod_{i=1}^{m-1} c_{B_{i+1}}^{x^i}$	$m-1$
c_{A_0}	:=	$\text{com}_{ck}(\vec{a}_0; r_0)$	$n+1$
c_{B_m}	:=	$\text{com}_{ck}(\vec{b}_m; s_m)$	$n+1$
$\vec{c}_D$	:=	$\{\text{com}_{ck}(d_i; t_i)\}_{i=\{0,\ldots,m+1,\ldots,2m\}}$	$4m$
c_d	:=	$\text{com}_{ck}(\vec{d}; r_d)$	$n+1$
c_δ	:=	$\text{com}_{ck}(\{-\delta_i d_{i+1}\}_{i=1}^{n-1}; s_1)$	n
c_Δ	:=	$\text{com}_{ck}(\{\delta_{i+1} - a_{i+1}\delta_i - b_i d_{i+1}\}_{i=1}^{n-1}; s_x)$	n

The prover gives an argument for a set of committed values have a particular product. That is, having commitments $\vec{c}_A$ to $A = \{a_{ij}\}_{i,j=1}^{n,m}$ he gives an argument of knowledge that b is the product of elements in A. For this he will compute a commitment c_b to n rows of the matrix, using $1+n$ exponentiations and they will engage in parallel in two SHVZK arguments of knowledge.

Hadamard product argument is used to proof the products of rows in matrix $A \in \mathbb{Z}_q^{n\times m}$ having particular values $\{b_i = \prod_{j=1}^{m} a_{ij}\}_{i=1}^{n}$. Now the prover's strategy is to define $\{\vec{b}_j = \prod_{k=1}^{j} \vec{a}_k\}_{j=1}^{m}$ and commit to them, getting $\vec{c}_B = [c_{A_1}, \{\text{com}_{ck}(\vec{b}_j; s_j)\}_{j=2}^{m-1}, c_b]$ with $(m-2)(n+1)$ exponentiations. When he receives challenge y, he defines the bilinear map $* : \mathbb{Z}_q^n \times \mathbb{Z}_q^n \to \mathbb{Z}_q$ by $\sum_{j=1}^{n} a_j d_j y^j$. Assuming a vector $\vec{y} = (y, y^2, \ldots, y^n)$ was previously calculated, this mapping has no exponentiation cost. He defines $c_{-1} = \text{com}_{ck} = (-\vec{1}; 0) = \prod_{i=1}^{n} \frac{1}{G_i}$ using no exponentiations. Then he computes $c_{D_i} = \{c_{B_i}^{x^i}\}_{i=1}^{m}$, which requires m exponentiations, and $c_D = \prod_{i=1}^{m-1} c_{B_{i+1}}^{x^i}$ with $m-1$ exponentiations.

Given a bilinear map and commitments to $\vec{a}_1, \vec{b}_0, \ldots, \vec{a}_m, \vec{b}_{m-1}$, they engage in a zero argument to show that $0 = \sum_{i=1}^{m} \vec{a}_i * \vec{b}_{i-1}$. He computes commits c_{A_0}, c_{B_m} in $(n+1)$ exponentiations and $\vec{c}_D = \text{com}_{ck}(\vec{d}; \vec{t}) = (\text{com}_{ck}(d_0; t_0), \ldots, \text{com}_{ck}(d_{2m}; t_{2m}))$, in $2(2m)$ exponentiations. Note $c_{D_{m+1}} = \text{com}_{ck}(0;0)$ because t_{m+1} is set to 0 and $\{d_k\}_0^{2m} = \sum_{\substack{0 \le i,j \le m, \\ j=(m-k)+i}} \vec{a}_i * \vec{b}_j$.

In parallel, a single value argument will be run where the prover gives an argument of knowledge of committed single values having a particular product. Prover's computation in this part consists of calculating three commits c_d, c_δ, c_Δ, in $(n+1), n, n$ exponentiations each.

3.2.3 Multi-Exponentiation Argument

In this section we present an argument that a ciphertext C has the same plaintext as the product of the main diagonal of a square matrix where elements are of the form $\{\vec{C}_i^{\vec{a}_j}\}, \forall i, j = 1 \ldots m$. In particular, $\vec{C} = (\vec{C}_1 = (C_1, \ldots, C_m), \ldots, \vec{C}_m = (C_{N-m+1}, \ldots, C_N))$ and $A = (\vec{a}_1, \ldots, \vec{a}_m) = \vec{x} = (x, \ldots, x^N)^T$ from shuffle argument. That is, we show the cost of an argument which proves $C = \varepsilon(1;\rho) \prod_{i=1}^{m} \vec{C}_i^{\vec{a}_i}$.

Eventually, they engage in an SHVZK argument for a smaller statement $C' = \varepsilon_{\text{pk}}(1;\rho')$ $\prod_{l=1}^{m'} \vec{C}_l'^{\vec{a}'_l}$. Two different ways of doing so will be presented. A naive method, which has a prohibitive cost for large m', can be used to get constant round interaction and a loglinear computational cost for the prover. Another option is to run recursively an optimized method, which leads to logarithmic round interaction and linear computational cost for the prover.

3.2.3.1 Communication Cost

The communication cost for the common part of multi-exponentiation argument is $(6\mu - 3)\mathbb{G} + 3\mathbb{Z}_q$. One may get constant, $9 = \mathcal{O}(1)$, or logarithmic, $\mathcal{O}(\log m)$, round complexity. This depends on the choice of the underlying SHVZK argument. When using the recursive method, the exact number of rounds depends on the choice of the variable μ such that $m = \mu m'$, where the matrix of $m \times m$ elements can be divided into m' blocks of μ^2 elements each. No matter which SHVZK argument is chosen, both share the same beginning using the non-naive method for the main multi-exponentiation argument. Note this is a 3-step communication, so the third message can be combined with the first message of SHVZK argument.

$\mathcal{P}$		$\mathcal{V}$	Size
$\vec{c}_b \in \mathbb{G}^{2\mu-1}, \vec{E} \in (\mathbb{G}\times\mathbb{G})^{2\mu-1}$	$\longrightarrow$		$(6\mu-3)\mathbb{G}$
	$\longleftarrow$	$x \in \mathbb{Z}_q^*$	$\mathbb{Z}_q$
$b, s \in \mathbb{Z}_q$	$\longrightarrow$		$2\mathbb{Z}_q$

The strategy consists on computing the products of the elements in the diagonals which are inside the blocks along the main diagonal. That is, to get $(2\mu-1)$ elements $E_k \in \mathbb{H}$. Those elements are randomized and the prover sends them to the verifier along with the commits to the encryption used, $\vec{c}_b$. Afterwards, the prover receives challenge x and calculates b, s using it.Eventually they engage in a SHVZK argument of openings $\vec{a}'_1, \ldots, \vec{a}'_m, \vec{r}', \rho'$ such that $C' = \varepsilon_{\text{pk}}(1;\rho')\prod_{l=1}^{m'} \vec{C}_l'^{\vec{a}'_l}$.

Recursive In case they run the recursive method, the previously explained strategy is repeated, with m' being the new m_2, until m'_k is no longer divisible. To put it in another way, they would end up with a square matrix of $m_2 = {m'_1}^2$ elements, divided into m'_2 blocks of ${\mu_2}^2$ elements each. This is, the non-naive protocol is repeated $\log m$ times, where each time adds one round (verifier's answer) and the last message from the prover overlaps with his first message in the next recursion. The following diagram illustrates the way rounds can be combined in order to get logarithmic rounds in m.

$\mathcal{P}$		$\mathcal{V}$	$\mathcal{P}$		$\mathcal{V}$	$\mathcal{P}$		$\mathcal{V}$
	$\cdots$							
$b, s \in \mathbb{Z}_q$	$\longrightarrow$		$\vec{c'}_b \in \mathbb{G}^{2\mu_k-1}$, $\vec{E}' \in (\mathbb{G}\times\mathbb{G})^{2\mu_k-1}$	$\longrightarrow$				
				$\longleftarrow$	$x' \in \mathbb{Z}_q^*$			
			$b', s' \in \mathbb{Z}_q$	$\longrightarrow$		$\vec{c''}_b \in \mathbb{G}^{2\mu_{k+1}-1}$, $\vec{E}'' \in (\mathbb{G}\times\mathbb{G})^{2\mu_{k+1}-1}$	$\longrightarrow$	
							$\cdots$	

This protocol will be repeated $\log m$ times, resulting in $2\log m + 1$ rounds. Every time they run this SHVZK argument, they use a corresponding communication cost of $(6\mu_k - 3)\mathbb{G} + 3\mathbb{Z}_q$.

Rounds	Size
$2\log m + 1$	$\sum_{k=1}^{\log m}((6\mu_k - 3)\mathbb{G} + 3\mathbb{Z}_q)$

Naive In case they run the naive method for the smaller statement, 3 steps of interaction are added to the common part. As before, the first message from the prover in the naive method can be overlapped with his last message in the common part. This means a 2 steps increase in round complexity and additional communication cost of $(6m+1)\mathbb{G}+(n+5)\mathbb{Z}_q$.

$\mathcal{P}$		$\mathcal{V}$	$\mathcal{P}$		$\mathcal{V}$	Size
	$\cdots$					
$b, s \in \mathbb{Z}_q$	$\longrightarrow$		$c_{A_0} \in \mathbb{G}$	$\longrightarrow$		$\mathbb{G}$
			$\{c_{B_k}\}_{k=0}^{2m-1} \in \mathbb{G}$			$2m\mathbb{G}$
			$\{E_k\}_{k=0}^{2m-1} \in \mathbb{H}$			$4m\mathbb{G}$
				$\longleftarrow$	$x \in \mathbb{Z}_q^*$	$\mathbb{Z}_q$
			$\vec{a} \in \mathbb{Z}_q^n, r, b, s, \tau \in \mathbb{Z}_q$	$\longrightarrow$		$(n+4)\mathbb{Z}_q$

The prover sends the encryption of the $2m$ products of the diagonals of a square matrix of size $m \times m$, containing elements $\vec{C}_i^{\vec{a}_j}$. Note there are $2m-1$ diagonals, and the product in the main diagonal E_m equals C, the fact he wants to prove. He will also send commits to the encryption used in each case, and commit to $\vec{a}_0$. When he receives a challenge, he sends new values the verifier will need to accept or reject.

3.2.3.2 Verifier Computation

This phase of the protocol requires the verifier calculate $2N+m+6\mu-1$ exponentiations.

			Exponentiations
$\{\vec{C}'_l\}_{l=1}^{m'}$	:=	$\prod_{i=1}^{\mu} \vec{C}_{\mu(l-1)+i}^{x^{\mu-i}}$	$2N$
$\{c_{A'_l}\}_{l=1}^{m'}$	:=	$\prod_{j=1}^{\mu} c_{A_{\mu(l-1)+j}}^{x^{j-1}}$	m
C'	:=	$\varepsilon_{\mathrm{pk}}(G^{-b};0)\vec{E}^{\vec{x}}$	$4\mu-1$

After challenging the prover, the verifier will compute elements $\vec{C}'_l = \prod_{i=1}^{\mu} \vec{C}_{\mu(l-1)+i}^{x^{\mu-i}}$, as well in $m'\mu n2$ exponentiations, $c_{A'_l} = \prod_{j=1}^{\mu} c_{A_{\mu(l-1)+j}}^{x^{j-1}}$ in $m'\mu$ exponentiations and $C' = \varepsilon_{\text{pk}}(G^{-b};0)\vec{E}^{\vec{x}}$ in $2(2\mu-1)+1$ exponentiations. Verification phase follows the SHVZK argument of knowledge that $C' = \varepsilon_{\text{pk}}(1;\rho)\prod_{l=1}^{m'} \vec{C}_l^{\prime\vec{a}'_l}$. He will check $\vec{c}_b \in \mathbb{G}^{2\mu-1}$, $\vec{E} \in \mathbb{H}$, $b, x \in \mathbb{Z}_q$. The verifier will accept if the argument was valid and $c_{b_{\mu-1}} = \text{com}_{ck}(0;0)$, $E_{\mu-1} = C$ and $\vec{c}_b^{\vec{x}} = \text{com}_{ck}(b;s)$, where the latter costs $(2\mu-1)$ and 2 exponentiations.

Recursive In case they run recursive SHVZK argument, the verifier will accept if such argument is recursively valid. As we saw before, it will be repeated $\log m$ times. Therefore, the total verification cost for non-naive multi-exponentiation argument is $\sum_{k=1}^{\log m}(2nm_k + m_k + 6\mu_k - 1)$ exponentiations.

Naive In case they run naive SHVZK argument, the verifier will check that the elements he received from the prover are in their corresponding group and $c_{B_m} = \text{com}_{ck}(0;0)$, $E_m = C$. Apart from that, he will compute certain values and check they are equal, by pairs:

			Exponentiations
$c_{A_0}\vec{c}_A^{\vec{x}}$	=	$c_{A_0}\prod_{i=1}^{m'} c_{A_i}^{x^i}$	m'
$\text{com}_{ck}(\vec{a} = \vec{a}_0 + A\vec{x}; r)$	=	$H^r\prod_{i=1}^{n} G_i^{a_i}$	$n+1$
$c_{B_0}\vec{c}_B^{\vec{x}}$	=	$c_{B_0}\prod_{k=1}^{2m'-1} c_{B_k}^{x^k}$	$2m'-1$
$\text{com}_{ck}(b;s)$	=	$H^s G_1^b$	2
$E_0\vec{E}_k^{\vec{x}}$	=	$E_0\prod_{k=1}^{2m'-1} E_k^{x^k}$	$4m'-2$
$\varepsilon_{\text{pk}}(G^b;\tau)\prod_{i=1}^{m'} \vec{C}_i^{x^{m'-i}\vec{a}}$	=	$\varepsilon_{\text{pk}}(G^b;\tau)\prod_{i=1}^{m'}\prod_{j=1}^{n} C_{ij}^{x^{m'-i}a_j}$	$2nm'+2$

Resulting verification time for this naive step is $2nm' + 7m' + n + 2$ exponentiations, which would be added to the common step cost, with a total verification cost for multi-exponentiation argument of $2N + 2nm' + m + n + 6\mu + 7m' + 1$ exponentiations.

3.2.3.3 Verifier Computation

This common part of the protocol takes the prover $4N\mu - 2N + m + 12\mu - 9$ exponentiations.

			Exponentiations
$\{E_k\}_{k=0}^{2\mu-2}$	:=	$\varepsilon_{\text{pk}}(G^{b_k};\tau_k)\prod_{l=0}^{m'-1}\prod_{\substack{i=1,j=1\\ j=(k+1-\mu)+i}}^{\mu,\mu}\vec{C}_{\mu l+i}^{\vec{a}_{\mu l+j}}$	$4(N\mu-N+\mu-1)$
$\vec{c}_b$	:=	$\{\text{com}_{ck}(b_k;s_k)\}_{k=0}^{2\mu-2}$	$4\mu-4$
$\{\vec{C}'_l\}_{l=1}^{m'}$	:=	$\prod_{i=1}^{\mu}\vec{C}_{\mu(l-1)+i}^{x^{\mu-i}}$	$2N$
$\{c_{A'_l}\}_{l=1}^{m'}$	:=	$\prod_{j=1}^{\mu}c_{A_{\mu(l-1)+j}}^{x^{j-1}}$	m
C'	:=	$\varepsilon_{\text{pk}}(G^{-b};0)\vec{E}^{\vec{x}}$	$4\mu-1$

The prover computes the products of the elements in the diagonals of the blocks along the main diagonal and rerandomizes them, resulting in $(2\mu-1)$ elements E_k. Note since $E_{\mu-1}=C$, the product of the main diagonal, there is no need for the prover to recompute this value. Therefore, getting E_k values costs $(2\mu-2)(m'\mu 2n+2)=(2\mu-2)(2N+2)=2N(2\mu-2)+4\mu-4$ exponentiations. The prover computes commits to the b_k values used for encryption. Setting $b_{\mu-1}=0, s_{\mu-1}=0$, commits cost $2(2\mu-2)$ exponentiations.

Upon reception of challenge x, the prover will compute m' values $\vec{a}'_l, r'_l, \rho'$, which correspond to each block, in no costly time. He will define values $\vec{C}'_l=\prod_{i=1}^{\mu}\vec{C}_{\mu(l-1)+i}^{x^{\mu-i}}$, taking $m'\mu n2=2N$ exponentiations, $c_{A'_l}=\prod_{j=1}^{\mu}c_{A_{\mu(l-1)+j}}^{x^{j-1}}$ in $m'\mu=m$ exponentiations and $C'=\varepsilon_{\text{pk}}(G^{-b};0)\vec{E}^{\vec{x}}$ in $2(2\mu-1)+1=4\mu-1$ exponentiations. Eventually, they engage in an SHVZK argument of openings $\{\vec{a}'_l\}_{l=1}^{m'}, \vec{r}'$ and $\rho'=\vec{\tau}\vec{x}$ such that $C'=\varepsilon_{\text{pk}}(1;\rho)\prod_{l=1}^{m'}\vec{C}_l^{\vec{a}'_l}$.

Recursive If they run recursive SHVZK argument, the previously explained strategy is repeated, with m' being the new m, until m'_k is no longer divisible. This is, it will be repeated $\log m$ times. Therefore, the total cost for multi-exponentiation argument for the prover, using non-naive method recursively, is $\sum_{k=1}^{\log m}(4nm_k\mu_k-2nm_k+m_k+12\mu_k-9)$ exponentiations.

Naive In case they run the naive method for the smaller statement, the prover aligns m' new ciphertexts in a square matrix. He encrypts the product of diagonals of this matrix, commitments to the randomization used and a commit to $\vec{a}_0$. He will compute $\vec{a}, r, b, s, \tau$, which is not costly for the prover.

			Exponentiations
c_{A_0}	:=	$\text{com}_{ck}(\vec{a}_0; r_0)$	$n+1$
$\{c_{B_k}\}_{k=0}^{2m'-1}$	:=	$\{\text{com}_{ck}(b_k; s_k)\}_{k=0}^{2m'-1}$	$4m'$
$\{E_k\}_{k=0}^{2m'-1}$	:=	$\varepsilon_{\text{pk}}(G^{b_k}; \tau_k) \prod_{\substack{i=1,j=0 \\ j=(k-m')+i}}^{m',m'} \vec{C}_i^{\vec{a}_j}$	$4nm'^2$

This gives us $4nm'^2 + 4m' + n + 1$ exponentiations for the naive SHVZK argument. It results in a total number of $4N\mu + 4nm'^2 - 2N + m + n + 12\mu + 4m' - 8$ exponentiations for the multi-exponentiation argument for the prover.

3.2.4 Paper Comparison

We will now compare our analysis on complexity and communication to the theoretical cost as shown in the paper. Note two different approaches will appear depending on the choice of SHVZK argument in multi-exponentiation argument; naive and recursive.

		Communication	Verifier	Prover
Shuffle		$2m\mathbb{G} + 3\mathbb{Z}_q$	$2N + m + n$	$2N + 2m$
Product		$(3m+7)\mathbb{G} + (4n+9)\mathbb{Z}_q$	$6m + 5n + 5$	$N + 7m + 4n + 1$
Multiexpo	Common	$(6\mu - 3)\mathbb{G} + 3\mathbb{Z}_q$	$2N + m + 6\mu - 1$	$4N\mu - 2N + m$ $+12\mu - 9$
Multiexpo	Naive	$(6m+1)\mathbb{G} + (n+5)\mathbb{Z}_q$	$2nm' + 7m' + n + 2$	$4nm'^2 + 4m' + n + 1$
Multiexpo	Recursive	$\sum_{k=2}^{\log m}((6\mu_k - 3)\mathbb{G} +$ $3\mathbb{Z}_q)$	$\sum_{k=2}^{\log m}(2nm_k +$ $m_k + 6\mu_k - 1)$	$\sum_{k=2}^{\log m}(4nm_k\mu_k -$ $2nm_k + m_k + 12\mu_k - 9)$

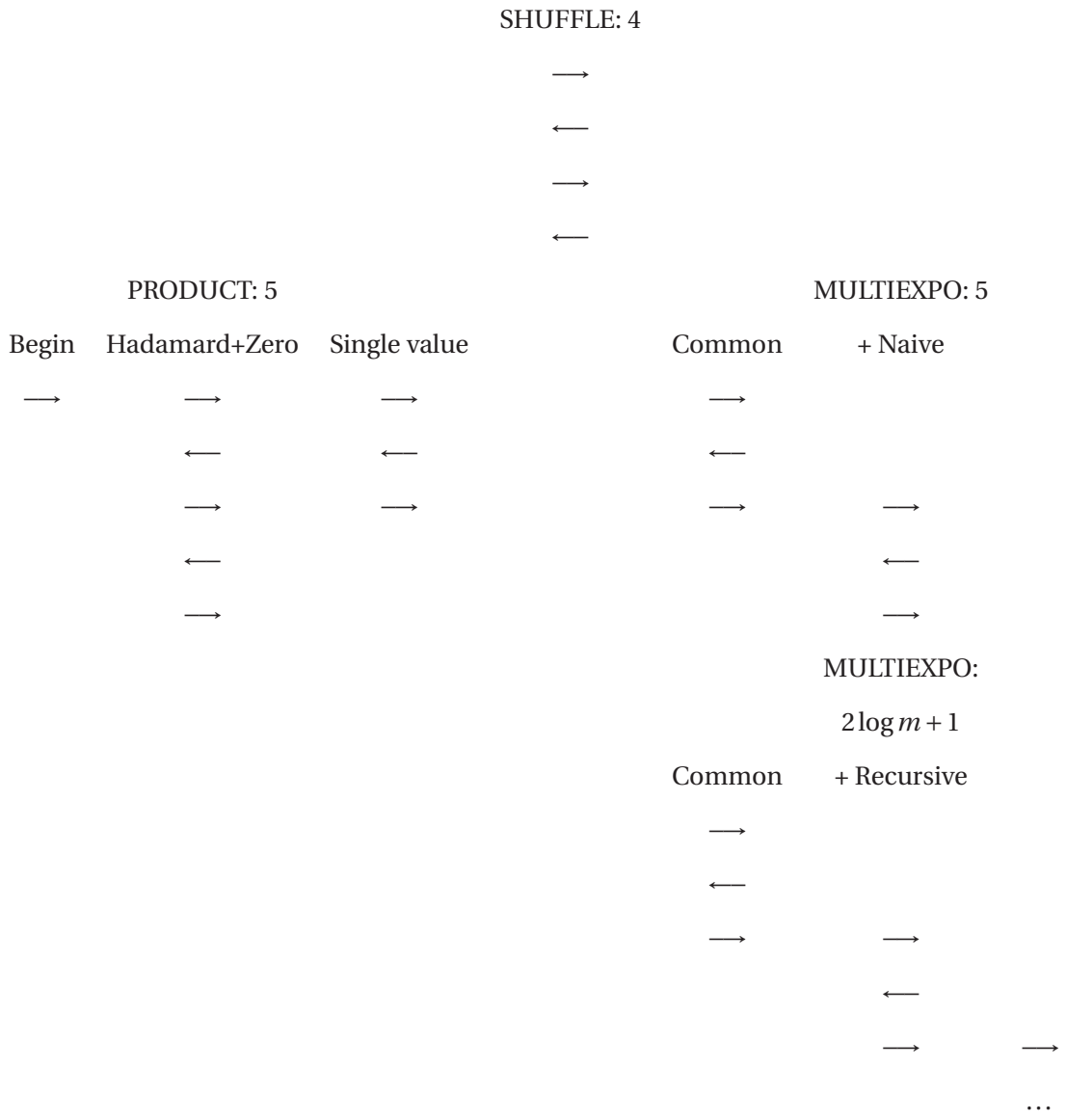
		Communication	Verifier	Prover
Total	Naive	$(11m+6\mu+5)\mathbb{G}$	$4N+2nm'+8m+$	$4N\mu+4nm'^2+N+5n$
		$+(5n+20)\mathbb{Z}_q$	$7n+6\mu+7m'+6$	$+10m+12\mu+4m'-7$
Total	Recursive	$(5m+7-3\log m$	$2N+7m+6n+5$	$3N+9m+4n+1-$
		$+6\sum_{k=1}^{\log m}\mu_k)\mathbb{G}+(4n$	$-\log m+\sum_{k=1}^{log\,m}($	$9\log m+\sum_{k=1}^{\log m}($
		$+12+3\log m)\mathbb{Z}_q$	$2nm_k+m_k+6\mu_k)$	$4nm_k\mu_k-2nm_k$
				$+m_k+12\mu_k)$
BigO	Naive	$11m\mathbb{G}+5n\mathbb{Z}_q$	$4N$	$4N\mu+4nm'^2$
BigO	Recursive	$5m\mathbb{G}+4n\mathbb{Z}_q$	$4N$	$\mathcal{O}(N)$
Paper	Naive	$11m\mathbb{G}+5n\mathbb{Z}_q$	$4N$	$2N\log m$
Paper	Recursive		$4N$	$\mathcal{O}(N)$

Note if we fix μ to be a constant, we get linear computation cost for the prover running recursive SHVZK argument. This is, to fix the block size instead of the number of blocks in the main diagonal. Thanks to this notation for naive SHVZK argument, which has one term on both μ and m', one can see how cost changes when fixing a certain value, due to the dependency $m = m'\mu$.

3.2.4.1 Interaction

First, let us define one round of interaction as the action of sending some message from the one side of the protocol to the other. This means sending a message and receiving back a challenge corresponds to 2 rounds. Taking this into account, one can overlap rounds in the same direction, whenever these were independent of one another to reduce rounds complexity. The paper shows one can get constant round of 9 when prover's computation is $2N\log m$, whereas rounds may be logarithmic in m to get linear prover computation.

The following diagram summarizes both cases, depending on the choice of SHVZK argument. It demonstrates round interaction claimed on the paper is aligned with our analysis. Note rounds are parallelized and overlapped whenever possible. In conclusion, using naive SHVZK we get 9 rounds of interaction and using recursive SHVZK we get 9 or $2\log m + 5$ rounds, whatever is larger.



CHAPTER 4

CONTRIBUTION

I was told once the so-called sphere of knowledge theory as it follows: Imagine anything so far known is confined within a sphere. When you finish elementary school, you learn the very centre of it. Studying high school lengthens your radius a bit. Then you go to University and study a bachelor degree, which leaves a little bump in your personal area of wisdom. If you take a master to gain speciality, that bump will stretch. If you read papers on that area, you end up reaching the edge of the sphere. Now a PhD consists on pushing at that boundary until a dent is made on the global sphere.

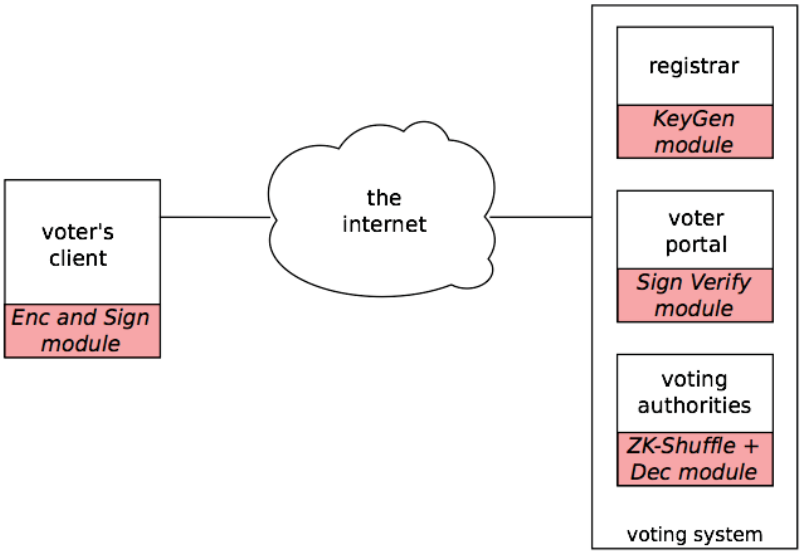
4.1 Vanilla Protocol

The goal of our investigation is to find other solutions for verifiable shuffling that could be valid alternatives to the BG protocol. To this end, we will apply recent advances in the context of zk-SNARKs.In particular, we will pursue this preliminary design idea following two approaches based on zk-SNARKs.

The first approach is to use generically the zk-SNARK technologies to prove correctness of the shuffle, and to tune the other cryptographic primitives required for the voting system (for example, the homomorphic encryption scheme) in order to get the best efficiency. zk-SNARKs allow to define a rich class of relationships, in particular, it is possible to define the relationship between the ballots before and after the shuffling, thus proving in zero-knowledge the consistency of the shuffling procedure.

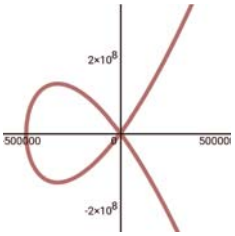
The prover efficiency and the verifier complexity for this method depend on the complexity of the relationship and the number of ballots N. Interestingly, the number of rounds in a zk-SNARK-based solution is 1, and the size of the proof is constant. However, it is not trivial to define the relationship in the language of zk-SNARKs, and to figure out their concrete efficiency. Such formalization needs to be carefully crafted to get the best efficiency.

The second approach is to "open the box" of the elegant BG protocol and substitute or modify partially the various pieces that compose the protocol with ad hoc solutions from the zk-SNARK literature. This should generate different hybrid solution which, we hope, would inherit the best of both worlds. The complexity of such approach needs to be carefully studied, the goals are to reach better prover and verifier efficiency (at least as efficient as the BG protocol) in comparison with the first approach, while shortening the size of the proof and the number of rounds.



4.1.1 Elliptic Curves

Elliptic curves are not ellipses, but rather curves with a general equation of $y^2 = x^3 + ax + b$. The reason why these are relevant for our work is there exist some constructions [15] that build SNARKs upon them (e.g. Curve25519: $y^2 = x^3 + 486662x^2 + x$). These curves have two main characteristics: if a line intersects two points, it intersects a third; and if a line is tangent to one point, it intersects another point [16]. Points in these curves are pairs of elements in $\mathbb{F}$.



An elliptic curve E under point addition is a group. Let us check if the properties of a group hold for E. First, any point has its identity at infinity $P \oplus I = P$. Points do have an inverse, which is their reflection in X axis. One can check it is associative, since $(A \oplus B) \oplus C = A(\oplus B \oplus C)$, no matter the points. More important, it is closed on the curve. This means operating elements in the curve always result in another element in the curve. And what is more, the fact that $A \oplus B = B \oplus A$ makes E commutative. In other words, not only the curve under $\oplus$ operation is a group but it is abelian (and therefore, suitable for cryptography).

Arithmetic operations in elliptic curves differ from regular operations. To understand point addition $\oplus$, we must imagine a line intersecting two points P_1 and P_2. According to elliptic curves' properties, there exist a third point P_3 that the line will intersect. Point addition results in the coordinates of the symmetric point of P_3 with respect to the abscissa axis. This can be solved as a simple geometry problem. Verification of point addition costs 4 multiplications when the curve is expressed in the Montgomery form (as in the case of Curve25519 [17]).

$$
\begin{aligned}
P_1 \oplus P_2 &= P_3' \\
y &= \lambda x + \beta \\
\lambda &= \frac{y_2 - y_1}{x_2 - x_1} \\
x_3 &= \lambda^2 - x_1 - x2 \\
y_3 &= \lambda(x_1 - x_3) - y_1 \\
P_3' &= (x_3, -y_3)
\end{aligned}
$$

The second operation in elliptic curves is point multiplication, $\otimes$, and the straightforward way to compute it is called *double-and-add*. Suppose g is a point in the curve, then to raise this element, we can decompose the exponent binarily. This is, $g^x = \prod_{i=0}^{\text{bits}} g^{2^i x_i}$. Writing this in additive notation results in $\sum_{i=0}^{\text{bits}} 2^i \otimes g \otimes x_i$, where $2^i \otimes x_i$ is a field operation. Now values $g_i = 2^i \otimes g$ can be precomputed, getting the expression $\sum_{i=0}^{\text{bits}} g_i \otimes x_i$, which will require $4\log_2(x)$ times the cost of $\otimes$. Note for this particular case, values x_i will be either 1 or 0. This means in case bit=1 the value g_i results, whereas it will be the neutral element in case bit=0. Let $g_i \otimes x_i = (x, y), x_i = b$, we can represent the result with the general form $(x, y) = (1 - b)(g_{i,1}, g_{i,2})$. We end up with two equations, $x = g_{i,1} - bg_{i,1}, y = g_{i,2} - bg_{i,2}$, which consist of 2 multiplications. Consequently, verification of point multiplication costs $6\log_2(x)$ multiplications, i.e., 6 products per bit in x.

Worth recalling are the definitions of our ballots, since there exists one handicap. Before counting process, voters have submitted their ciphered ballots. Whenever election authorities open the BB, they will find nothing else than ElGamal ciphertexts. Each ballot is therefore composed of 2 elements in $\mathbb{G}$, an additive cyclic group of large prime order q which has its origin in an elliptic curve. We define two operations over $\mathbb{G}$:

A simple point addition, $\oplus$; and concatenation of several additions, point multiplication $\otimes$. However, arithmetic circuits are defined over finite fields $\mathbb{F}$, and regular operations must be expressed in this different notation. In comparison to common arithmetic, multiplications convert into $\oplus$ and exponentiations are represented as $\otimes$. Note an element in $\mathbb{G}$ is a subset of elements in $\mathbb{F}^2$.

With that in mind, we proceed to illustrate ElGamal scheme in $\mathbb{F}$. Key generation will create a pair composed by secret and public keys ($\text{sk} = x \in \mathbb{Z}_q^*, \text{pk} = x \otimes G$). Each input ciphertext C_i is computed in the additive group $\mathbb{G}$, where M is a plain ballot, as $\varepsilon_{\text{pk}}(M_i; r_i) = (C_{i,1} = r \otimes G, C_{i,2} = M \oplus r \otimes \text{pk})$. Prior to getting shuffled ciphertexts $\vec{C}'$, ElGamal ciphertexts must be rerandomized by doing $\vec{C} \oplus \varepsilon_{\text{pk}}(\vec{1}; \vec{\rho})$, which results in pairs $(c_{i,1} = C_{i,1} \oplus \rho_i \otimes G, c_{i,2} = C_{i,2} \oplus \rho_i \otimes \text{pk})$. Now, these values should be the same as input $\vec{C}'$, in some permuted order. Since G, pk are public parameters we can treat them as constants, reducing the number of effective point multiplications.

4.1.2 Arithmetic Circuit

Before we build our circuit, we should consider some useful aspects of SNARKs. A very important property is additions and constant scalar multiplications are free in SNARK. For this reason, the main optimization technique will aim to reduce multiplication gates. Bit level operations are quite expensive ($n+1$ multiplication gates). However if an element is already in binary representation, rotation and shift are free as well.

We define our permutation Π as a permutation of identity matrix of size the number of ballots, N. This is, a binary square matrix of size $N \times N$ where columns contain $N-1$ zeros and 1 one. Another condition this matrix must meet for it to be a permutation, is the sum of its rows equals the column vector of N ones. Note the inverse of this matrix, which coincides with its transpose matrix, gets the original order when applied to shuffled ciphertexts. In our notation, π_i refers to the i-th row of matrix Π. Since the value of shuffled ciphertext C'_i has two elements, $(C'_{i,1}, C'_{i,2})$, this permutation must be applied twice for each ciphertext. Let k be an index varying between 1 and 2, then $C'_{i,k}$ is calculated as $\sum_{j=1}^{N} \pi_{i,j} \otimes c_{j,k}$ over $\mathbb{G}$, where $c_{i,k}$ is the k-th ciphertext of rerandomized ciphertext c_i, what is equivalent to $\prod_{j=1}^{N} (c_{j,k})^{\pi_{i,j}}$ in multiplicative notation. Since this matrix is sparse, it will be interesting to analyse the case where its size is reduced to $3N$ elements. For every nonzero element in Π, we would only store its row and column position along with its value (note sparse matrixes are less costly as long as $N > 3$).

Now we shall define the function we would like our QAP to compute. This function should only accept its inputs if the shuffle was done correctly. That is, $f \to \{0,1\}$ outputs 1 if and only if shuffled ciphertexts $\vec{C}'$ correspond to re-encrypted ballots from the BB using random values $\vec{\rho}$, permuted according to Π.

This function will receive ciphertexts $\vec{C}, \vec{C}'$ which will act as input u, and secret values $\Pi, \vec{\rho}$ which will be the witness w. Input size is $4N$ elements in $\mathbb{G}$ ($8N$ elements in $\mathbb{F}$), whereas witness size is N^2 elements in $\{0,1\}$ and N elements in $\mathbb{Z}_q$. Randomizers should be represented in binary notation, what results in $N\log_2(q)$ elements in $\mathbb{F}$. For the sake of verifiability, one parameter λ will be included in the witness for each group addition.

At this point, we can describe the circuit for our function. Remember we must check four kind of equations hold, some are related to the permutation itself and others are related to correct shuffling. The first kind will check permutation matrix only contains 0s and 1s, whereas the second checks it is in fact a permutation of identity matrix by checking the sum of their rows. Now, for every ciphertext in BB, we check if both components were used properly to compute shuffled ciphertexts using the secret permutation and randomizers in the witness.

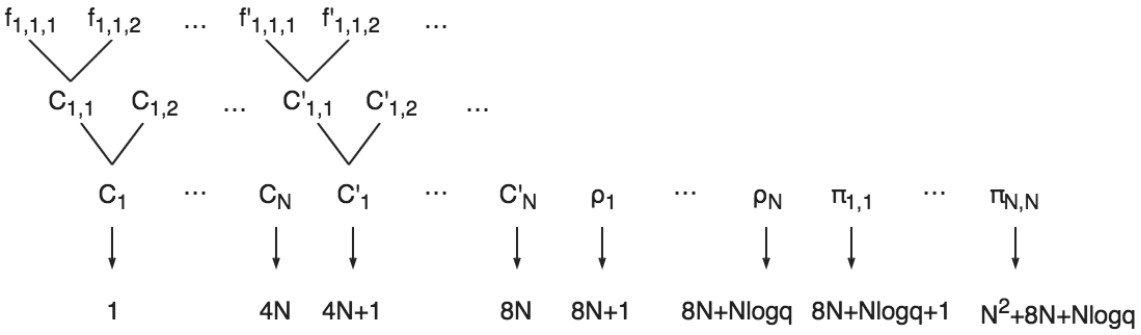
$$\text{Over } \mathbb{G} \qquad \forall i \in \{1 \dots N\} : \sum_{j=1}^{N} (\pi_{i,j} \otimes (C_{j,1} \oplus \rho_j \otimes G)) - C'_{i,1} = (0,0) \tag{4.1}$$

$$\text{Over } \mathbb{G} \qquad \forall i \in \{1 \dots N\} : \sum_{j=1}^{N} (\pi_{i,j} \otimes (C_{j,2} \oplus \rho_j \otimes \mathrm{pk})) - C'_{i,2} = (0,0) \tag{4.2}$$

$$\text{Over } \mathbb{F} \qquad \forall \pi_{i,j} \in \Pi : (\pi_{i,j})(\pi_{i,j} - 1) = 0 \tag{4.3}$$

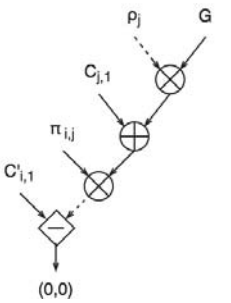
$$\text{Over } \mathbb{F} \qquad \forall \pi_i \in \Pi : \sum_{j=1}^{N} \pi_{i,j} = 1 \tag{4.4}$$

Note the circuit will need values G, group generator, and pk, the public key that election authorities used to randomize ciphertexts. Since these are public parameters, we can treat them as constants to reduce multiplicative gates in the circuit. Remember as well, elements in $\mathbb{G}$ are represented as pairs of elements in $\mathbb{F}$. This means input length $n = 8N$ elements in $\mathbb{F}$. All these values, both in the input and witness, correspond to the $N^2 + 8N + N\log_2(q)$ input wires for our circuit.

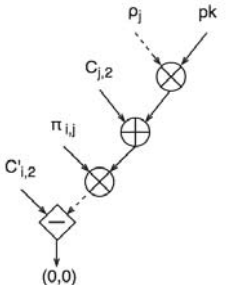


Remember as well, randomizers are given in binary representation, which takes $\log_2 q$ bits to represent an element in $\mathbb{Z}_q$. This implies each point multiplication involving randomizers would convert to $\log_2 q$ group additions. This makes the whole computation much simpler. Another point of discussion would rely on the choice of the base of $\vec{\rho}$. If they were represented in base-ten, logarithmic expressions could be faster.

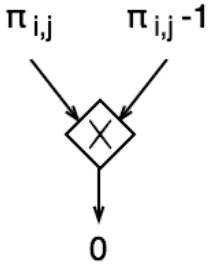
This piece of circuitry is a high-level illustration of equation 4.1 in the arithmetic circuit. Note the right input to subtraction gate is a sum in $j = 1 \ldots N$ but the right input to $\pi_{i,j}\otimes$ can be computed only once for all N ciphertexts. Like we said, randomizer point multiplication is in fact $\log_2(q)$ group additions, to recompose each ρ_i out of its binary representation. Then, multiplication by constant is not costly. This part has $N(N\log_2 q + 6N + 4)$ wires (ciphertexts are 2 $\mathbb{F}$ elements), $N(\log_2 q + 1) + N - 1$ $\oplus$ gates and N^2 $\otimes$ gates for all ciphertexts.



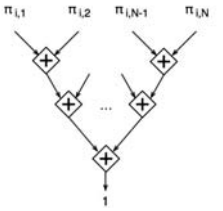
The second piece of circuitry represents equation 4.2. Note it is so similar to the previous equation, that everything already explained also applies here.



Unfortunately, this native field multiplication must be applied N^2 times (one per element in the permutation square matrix), with 3 wires per $\pi_{i,j}$. An important source of optimization might be to use a different operation to check Π only contains 1s and 0s.



The remaining branch of the circuit is absolutely free. It solely contains field additions to check the sum of all rows in permutation matrix results in the column vector of 1s. This piece has $N - 1$ addition gates per row in Π, what results in $2N - 1$ wires.



	C	
Input	n	$8N$
Witness	l	$N^2 + N\log_2 q$
Output	h	$N^2 + 5N$
Wires	a	$2N^2\log_2 q + 11N^2 + 3N$
Size	b	$12N^2\log_2 q + N^2 + 8N\log_2 q + 16N - 8$

To sum up, our circuit C will have $(2N^2)\otimes$ gates, $(2N\log_2 q + 4N - 2)\oplus$ gates and $(N^2)\times$ gates with total quadratic size $b = (12N^2 + 8N)\log_2 q + N^2 + 16N - 8$ multiplications and $2N(N\log_2 q + 6N + 4) + 4N + 3N^2 + N(2N-1) = 2N^2\log_2 q + 11N^2 + 3N = a$ wires; including input, intermediate and output wires.

4.1.2.1 Resulting QAP

The QAP Q is output upon execution of QAPinst and QAPwit algorithms. According to some efficiency results, it will have size $m = a$ multiplicative gates with degree $d = n + b + l + 1$ over $\mathbb{F}$.

	Q	
Input	n	$8N$
Size	m	$2N^2\log_2 q + 11N^2 + 3N$
Degree	d	$12N^2\log_2 q + 2N^2 + 8N\log_2 q + 29N - 7$

4.1.3 Optimizations

Throughout this document, we have mentioned various types of improvements we would like to include in a future version of the scheme. However, we found it more handy to collect them all into a specific subsection.

Remember as well additions and multiplication by constant are free in SNARKs. It is multiplication gates what can make a SNARK protocol inefficient. Our original circuit uses N^2 field multiplications to verify such a simple property: that a matrix only contains ones and zeros. This distances itself from the objective of a subquadratic complexity for the prover. Therefore, we should either think of a different operation to check the same property, or outsource its verification to a different procedure. Note the latest does not imply witness size would be reduced by N^2 elements, since $\pi_{i,j}$ would still be needed for the remaining computations.

Please remember the degree of the QAP depends on the number of output wires in C. Right now, the permutation check returns N^2 zeros, one per element in the matrix. It might be appealing to diminish the size of l in turns of more gates (as long as they are additions or multiplications by scalar). However, this would augment the number of total wires of the circuit.

Another not so notorious source of optimization might be the base of randomizers. This might be useful because many expressions are defined in function of logarithms, which represent the number of effective digits needed to represent each randomizer. In case these are binary, $\log_2 q$ is used, where $\vec{\rho} \in \mathbb{Z}_q$. Note if they were given in denary, $\log_{10} q$ would be used and it would be much more natural to the human eye.

We should not forget a correct permutation matrix only contains N ones whereas the rest of elements are zeros. This kind of matrixes are called sparse and some numerical tools use specific algorithms to reduce the cost of computations. Concretely, only row, column and value are stored in these structures, what makes a difference from N^2 to $3N$. If we knew there are only unitary components, the cost could be reduced to $2N$. However, the piece of circuit which verifies the shuffling itself might result more complicated, since not every $\pi_{i,j}$ would be available.

Even if we did not use sparse treatment, from a low-level perspective a multiplication by zero is trivial. For this reason, in practice, real computation will not be that costly. This does not only apply to the check of the matrix, but also to multiplication gates in the shuffling equations, where there would not be a quadratic number of point multiplications, but N of them. It would be interesting to have a software implementation in order to check the real time it takes, because this theoretical upper bound we calculated is too broad.

Assuming the aforementioned simplification holds, let us give some lower bounds of the circuit in practice. The number of wires would be unavoidable identical. However, this does not mean they are all nonzero wires. Moreover, the number of effective products would be dramatically reduced. Concretely, instead of $(N^2)\times$ gates for the permutation check we would have N nonzero gates, and instead of $(2N^2)\otimes$ gates for the shuffling we would have $2N$ nonzero. In other words, instead of size in the order of $12N^2\log_2 q$ multiplications, the lower bound for the number of effective multiplicative gates would be $b' = 20N\log_2 q + 17N - 8$: the real complexity becomes quasilinear.

4.1.4 New zk-SNARK Shuffling Scheme

In order to have some insight into the cost of adding zk-SNARKs to our QAP, we will refer to the annexed table in [14]. For further reading on building SNARK proofs out of QAPs one may consult Pinocchio protocol [18].

Some public parameters will be available to the entities. These are two cyclic groups $\mathbb{G}_1, \mathbb{G}_2$ of order a prime r with generators g_1, g_2, forming the domain of the pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, where $\mathbb{G}_T$ is another cyclic group of the same order. Please remember this pairing function satisfies $e(g_1^x, g_2^x) = e(g_1, g_2)^{xy}$. Using different groups not only is more generic, but also avoids some attacks. Note this table assumes additive notation, so steps and naming will differ from the notation in [12]. For instance, homomorphic encoding function $E(x+y) = g^{x+y} = g^x g^y = E(x)E(y)$ is not represented as an exponentiation anymore, so it is therefore more efficient.

4.1.4.1 Key Generation

On input our circuit $C : \mathbb{F}_r^n \times \mathbb{F}_r^h \to \mathbb{F}_r^l$, key generator $\mathcal{G}$ outputs proving key pk with $(6a+b+n+l+26)\mathbb{G}_1$ elements and $(a+4)\mathbb{G}_2$ elements and verification key vk with $(n+3)\mathbb{G}_1$ elements and $5\mathbb{G}_2$ elements. The main computational cost consists of the polynomial running time to compute polynomials $(\vec{A}, \vec{B}, \vec{C}, Z) := \mathsf{QAPinst}(C)$ and $(d+1)$ exponentiations to compute part of pk.

	$\mathbb{G}_1$	$\mathbb{G}_2$
pk	$14N^2\log_2 q + 13N^2 + 9N\log_2 q + 27N + 18$	$2N^2\log_2 q + 11N^2 + 3N + 4$
vk	$8N+3$	5

4.1.4.2 Prover Computation

On input the pk, $\vec{x} \in \mathbb{F}_r^n$ and witness $\vec{w} \in \mathbb{F}_r^h$, the prover $\mathcal{P}$ outputs a proof π of constant size $7\mathbb{G}_1 + 1\mathbb{G}_2$ elements. $\mathcal{P}$ runs in polynomial time algorithms $\mathsf{QAPinst}(C)$ and $\mathsf{QAPwit}(C, \vec{x}, \vec{w})$ to get sets of polynomials for the QAP as well as the expanded witness $\vec{s}$. Then he computes the $d+1$ coefficients of polynomial H such that $Z(z)H(z) = A(z)B(z) - C(z)$, by interpolation. The most efficient implementation for this step [18] uses a FFT-based polynomial multiplication library along with an efficient polynomial interpolation algorithm, giving a total time of $\mathcal{O}(d \log^2 d)$, much better than Lagrangian $\mathcal{O}(d^2)$. This is by far, the most expensive step for the prover.

	$\mathbb{G}_1$	$\mathbb{G}_2$
π	7	1

4.1.4.3 Verifier Computation

On input the vk, $\vec{x} \in \mathbb{F}_r^n$ and proof π, the verifier $\mathcal{V}$ outputs a bit with value $b = 1$ if and only if all the checks succeed. Concretely, V accepts the proof if the check of validity of knowledge commitments for the sets of polynomials and the check of same coefficients used and the check of QAP divisibility are correct. This step only requires the verifier calculate a value based on the input and computing 12 pairings using function e.

4.1.5 Comparison

The most notorious advantage of this scheme over the minimal shuffle is the fact that both communication and proof size are constant (which is much more voter friendly). Moreover, verifier computation is much smaller than $4N$, very suitable for mobile computing devices. Nonetheless, the prover computation will be at least $\mathcal{O}(d \log^2 d)$, where degree d is unfortunately in the order of $12N^2 \log_2 q$. This result is still very far from BG outcome of $2N \log m$ exponentiations for the prover, despite all optimizations on the sparseness of the permutation matrix. However, if we managed to reduce the input size to linear in the number of ballots, the prover's computation will be reduced dramatically using this same technique: find a function that verifies the shuffling and its corresponding arithmetic circuit, transform it to a QAP and finally apply zk-SNARK. The good news is all remaining aspects would not worsen, but rather improved as well.

4.2 Future Work

What any research project share in common is probably the fact that time never suffices. DataMantium is a 2 year length project which just started its investigation phase. Once the initial proposal is delivered, prototype phase will commence. Despite this MINECO funded project will eventually pursue a final product, my focus relies on theoretical analysis with the intention of obtaining better solutions to the current state-of-the-art. When we develop all the optimizations we have in mind, we will like to publish some paper on our results.

4.2.1 SNARKs for the Minimal Shuffle

Thanks to our meticulous analysis of BG protocol, we could consider the possibility of building a SNARK proof for a particular block. In particular, it would be interesting to substitute multi-exponentiation argument, since it is the most costly component. Hopefully, that could make the prover's computation linear in the number of votes, even for the naive approach of constant round interaction. Nonetheless, there are a bunch of other interesting research approaches once we "*open the box*" of BG protocol.

So far, we have observed that the reason why we get quadratic complexity is that permutation itself is included in the witness. This makes us recall the strategy addressed in the minimal shuffle. Instead of considering the permutation matrix as part of the input, they will commit to the permutation. Remember as well, this is an interactive protocol, ranging from constant to logarithmic round interaction.

Our goal will be to build a completely non-interactive protocol based on SNARK using linear input size in the number of ballots. We think this approach will get closer to BG results. As in the case of the minimal shuffle, we would commit to the permutation by doing $\text{com}_{snark}(\vec{a} = \{\pi(i)\}_{i=1}^{N}; \vec{r}), \text{com}_{snark}(\vec{b} = \{x^{\pi(i)}\}_{i=1}^{N}; \vec{s})$, which results in $2m$ elements in $\mathbb{G}$, $4m\mathbb{F}$ (much better than N^2, where m is in the order of $\sqrt{N}$). Unlike BG, we would use the Fiat-Shamir heuristic [19] to make it non-interactive. This property is extremely desirable in the setting of electronic voting verification, since there might be millions of clients who wanted to check the proof. The reason why this holds is public-coin challenges x, y, z are not sent by $\mathcal{V}$ but computed by $\mathcal{P}$ hashing the commit end instead. This heuristic is known to be secure in ROM, where the cryptographic hash function is modelled as a random oracle that returns a uniformly random answer.

4.2.2 Implementation

Prototype phase of this project is not expected to be completed before 1 year time. Scytl will be in charge of the proper implementation of the whole scheme, given their previous experience in this field. Besides, IMDEA Software will give assistance on theoretical aspects for the exact same reason.

In parallel to prototype stage and for the sake of science, we would like to have an implementation of this scheme once we get closer to the cost of BG protocol. On the base of previous works [20], and its powerful numerical libraries, we think it would be a good idea to implement the protocol in Python. SageMath is an example of a mathematical based programming language based on Python that has the ability to treat finite fields and groups. However, if we wished the code to run concurrently, we would use C instead.

REFERENCES

[1] D. Galindo, S. Guasch, and J. Puiggalí, "2015 Neuchâtel's Cast-as-Intended Verification Mechanism," in *VoteID 2015* (R. H. et al., ed.), vol. 9269 of *Lecture Notes in Computer Science*, (Barcelona, Spain), pp. 3–18, Scytl Secure Electronic Voting, Springer International Publsihing Switzerland, 2015.

[2] K. Gjøsteen and A. S. Lund, "The Norwegian Internet Voting Protocol: A new Instantiation." Cryptology ePrint Archive, Report 2015/503, 2015. `http://eprint.iacr.org/2015/503`.

[3] R. Gennaro, C. Gentry, and B. Parno, "Non-Interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers." Cryptology ePrint Archive, Report 2009/547, 2009. `http://eprint.iacr.org/2009/547`.

[4] S. Arora and B. Barak, *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

[5] R. R. Williams, "Strong ETH Breaks With Merlin and Arthur: Short Non-Interactive Proofs of Batch Evaluation," in *31st Conference on Computational Complexity (CCC 2016)* (R. Raz, ed.), vol. 50 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 2:1–2:17, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.

[6] T. P. Pedersen, "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing," in *Advances in Cryptology – CRYPTO'91* (J. Feigenbaum, ed.), vol. 576 of *Lecture Notes in Computer Science*, (Santa Barbara, CA, USA), pp. 129–140, Springer, Heidelberg, Germany, Aug. 11–15, 1992.

[7] S. Bayer and J. Groth, "Efficient Zero-Knowledge Argument for Correctness of a Shuffle," in *Advances in Cryptology – EUROCRYPT 2012* (D. Pointcheval and T. Johansson, eds.), vol. 7237 of *Lecture Notes in Computer Science*, (Cambridge, UK), pp. 263–280, Springer, Heidelberg, Germany, Apr. 15–19, 2012.

[8] J. Groth, "A Verifiable Secret Shuffle of Homomorphic Encryptions," *Journal of Cryptology*, vol. 23, pp. 546–579, Oct. 2010.

[9] N. Bitansky, R. Canetti, A. Chiesa, S. Goldwasser, H. Lin, A. Rubinstein, and E. Tromer, "The Hunting of the SNARK." Cryptology ePrint Archive, Report 2014/580, 2014. http://eprint.iacr.org/2014/580.

[10] C. Reitweißer, "zkSNARKs in a Nutshell." Ethereum, December 2016.

[11] S. Arora and S. Safra, "Probabilistic Checking of Proofs; A New Characterization of NP," in *33rd Annual Symposium on Foundations of Computer Science*, (Pittsburgh, Pennsylvania), pp. 2–13, IEEE Computer Society Press, Oct. 24–27, 1992.

[12] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic Span Programs and Succinct NIZKs without PCPs," in *Advances in Cryptology – EUROCRYPT 2013* (T. Johansson and P. Q. Nguyen, eds.), vol. 7881 of *Lecture Notes in Computer Science*, (Athens, Greece), pp. 626–645, Springer, Heidelberg, Germany, May 26–30, 2013.

[13] V. Buterin, "Quadratic Arithmetic Programs: from Zero to Hero," December 2016. https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649.

[14] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct Non-Interactive Arguments for a von Neumann Architecture." Cryptology ePrint Archive, Report 2013/879, 2013. http://eprint.iacr.org/2013/879.

[15] A. Kosba, Z. Zhao, A. Miller, Y. Qian, H. Chan, C. Papamanthou, R. Pass, abhi shelat, and E. Shi, "CøCø: A Framework for Building Composable Zero-Knowledge Proofs," in *CCAF*, 2016. https://eprint.iacr.org/2015/1093.pdf.

[16] Riverninj4, "Elliptic Curve Point Addition." MAO Math Presentation Competition, Feb. 2011. https://www.youtube.com/watch?v=XmygBPb7DPM&t=131s.

[17] D. J. Bernstein, "Curve25519: new Diffie-Hellman speed records," in *In Public Key Cryptography (PKC), Springer-Verlag LNCS 3958*, p. 2006, 2006.

[18] B. Parno, C. Gentry, J. Howell, and M. Raykova, "Pinocchio: Nearly practical verifiable computation." Cryptology ePrint Archive, Report 2013/279, 2013. http://eprint.iacr.org/2013/279.

[19] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, vol. 263 of *Lecture Notes in Computer Science*, pp. 186–194, Springer, 1986.

[20] V. Buterin, "zkSNARK Compiler for Educational Purposes." https://github.com/ethereum/research/tree/master/zksnark, December 2016. Ethereum GitHub.

Este documento esta firmado por

Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
Fecha/Hora	Thu Jun 08 14:16:53 CEST 2017
Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
Numero de Serie	630
Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)