Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Einar Bogsnes Hegge

Lattice-Based Zero Knowledge

Master's thesis in Physics and Mathematics
Supervisor: Kristian Gjøsteen
July 2020

NTNU
Norwegian University of
Science and Technology

Einar Bogsnes Hegge

Lattice-Based Zero Knowledge

Master’s thesis in Physics and Mathematics
Supervisor: Kristian Gjøsteen
July 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Norwegian University of
Science and Technology

Abstract

We present 3 schemes using lattice-based zero-knowledge, using the difficulty of the Approximate Shortest Vector Problem (SVP_γ), and the Short Integer Solution Problem (SIS). The first scheme is an identification scheme, proving knowledge of a secret key $\hat{\boldsymbol{s}}$. The second scheme is a signature scheme obtained by applying the Fiat-Shamir transform on the identification scheme to create a non-interactive zero-knowledge scheme. The last scheme is an argument of knowledge for the satisfiability of an arithmetic circuit. We also present various theory necessary for these algorithms to work, as well as examples illustrating what we are trying to accomplish in some of the more complicated parts of the constructions.

Table of Contents

Abstract **1**

Table of Contents **3**

1 Introduction **5**

2 Algebraic and Cryptographic Theory **7**
2.1 Notation . . . 7
2.2 Provable Security . . . 7
2.3 Lattices . . . 8
2.4 Lattice-Based Collision-Resistant Hash Function . . . 9
2.5 Lattice-Based Commitment Schemes . . . 10
2.6 Interactive zero-knowledge . . . 10
2.7 Fiat-Shamir and non-interactive zero-knowledge . . . 11
2.8 Argument of Knowledge for polynomials . . . 12
2.9 Aborting . . . 13
2.10 Arithmetic Circuits . . . 15

3 Lattice-based Identification and Signature Schemes **17**
3.1 Identification . . . 17
3.2 Signature scheme . . . 21

4 Reducing an arithmetic circuit into equivalent polynomials **23**
4.1 From arithmetic circuit to an equivalent polynomial . . . 23
4.2 Using field extensions to improve efficiency . . . 28

5 Argument of knowledge for the arithmetic circuit represented by polynomials **37**

6 Concluding remarks **47**

Bibliography **48**

Chapter 1

Introduction

Zero-knowledge has been a hot topic in cryptography in recent years. It's a very important part of many cryptographic protocols, to ensure that the protocol doesn't leak information other than the information that the sender intended to give to the receiver. A zero-knowledge argument can be used to convince a verifier that the prover possesses some specific information without giving any information to the verifier except what they already knew. For example, given a hash function $h(m) = \boldsymbol{M}$, we could construct a zero-knowledge argument for the knowledge of m. That is, the prover could convince the verifier that he possesses m such that $h(m) = \boldsymbol{M}$, but without revealing any information on m except for the fact that $h(m) = \boldsymbol{M}$ (which the verifier already knew).

Lattice-based cryptography is an important part of cryptography due to its resistance to attacks from quantum computers. In this thesis we will present 3 lattice-based zero-knowledge schemes, 2 of which are fairly similar and simple, and one that is more complicated. The construction of the identification scheme and signature scheme follow the work done in [4]. The more complicated scheme for creating a zero-knowledge argument for the satisfiability of an arithmetic circuit follows the work done in [2].

The identification scheme and signature scheme both use the approximate shortest vector problem SVP_γ as their underlying hard problem. For these algorithms, we use a public hash function $h(s) = \boldsymbol{S}$ for some matrix which will be detailed later. Given only $\boldsymbol{S}$ it will be difficult to find s. The scheme then uses a random y s. t. $h(y) = \boldsymbol{Y}$ and a random c. The verifier will be provided $z = sc + y$, the public value $\boldsymbol{S}$, and $\boldsymbol{Y}$ transmitted by the prover. By the homomorphic properties of the hash function, the verifier can then check that $h(z) = h(sc + y) = \boldsymbol{S}c + \boldsymbol{Y}$ which works given that we choose the random c, y appropriately.

The arithmetic circuit satisfiability problem uses the properties of a lattice-based commitment scheme to compress commitments to n elements of $\mathbb{Z}_p$ into r elements of $\mathbb{Z}_q$, where $n >> r$ and $p << q$. As a result of a property of the SIS-problem, we can let n get very large compared to r, resulting in the commitments having sublinear communication costs.

We first reduce the satisfiability of the arithmetic circuit to a set of multiplication con-

straints and a set of linear constraints in 3 sets of variables. Then we encode these constraints into polynomials. Then we finally create an argument of knowledge using these polynomials, and the compressing commitment scheme. This results in $O(\sqrt{N \log N})$ communication cost for an arithmetic circuit with N multiplication gates.

We first present theory necessary for these schemes in Chapter 2. Then in chapter 3 we detail the identification scheme and signature scheme. Chapter 4 will explain how we can transform an arithmetic circuit into multiplication constraints and linear constraints in 3 sets of variables, and then use those to create equivalent polynomials. In chapter 5 we detail the argument of knowledge for the arithmetic circuit. Finally, we end the thesis with a few concluding remarks in chapter 6.

Chapter 2

Algebraic and Cryptographic Theory

2.1 Notation

This thesis contains a large amount of variables, and this makes it hard to create a consistent notation scheme without constantly running out of reasonable letters to use. However, in general we will be using bold lowercase letters to denote vectors, bold lowercase letters with a hat to denote vectors of vectors, uppercase letters (both with and without a hat) to denote matrices, and bold uppercase letters to denote the output of hash function or commitment scheme in use. Uppercase letters will also be used to denote indeterminates in equations. Additionally, greek letters will be used in situations where they make sense (σ to denote variance, etc.). They will also be used in conjunction with similar letters in cases where we need to perform a function on two variables. So given that we have a matrix A that we wish to send into a commitment scheme taking two matrices as parameters, we might use α to denote the other matrix.

2.2 Provable Security

Security is the most important performance metric of any cryptographic system. As such, we would like to know with reasonable confidence that the system is not easily broken. The most obvious way to do this is to perform cryptanalysis on the system, and if we are unable to break it then it is more likely to be secure. The downside of this approach is that we need to spend a lot of time analysing each system, and with the large amount of new systems that are being created we can only spend so much time on each of them before declaring them to be ”secure”. On the other hand, an adversary can focus all his effort into a single system. This means that this adversary can spend more time trying to break the system than the time spent cryptanalysing to ensure it is secure.

The provable security approach is somewhat different. Here most of the time is spent

cryptanalysing only a few problems. Then for each new system we prove that a successful attack on the system can be used to solve one of these problems. Given that the number of these problems remains relatively low means that we can spend quite a bit of time analysing each of them. And since we've spent a lot of time on each problem, the probability of an exploitable flaw existing is much smaller.

The downside of this approach is that we need to prove that a successful attack on the system can be used to create a successful attack on the underlying problem. This can be quite difficult, even for simple systems. Furthermore, it is very easy to make mistakes during these proofs, so the proofs themselves need to be thoroughly checked, which can be time-consuming.

2.3 Lattices

An integer lattice Λ is a subgroup of $\mathbb{Z}^n$. Given any set $\boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_n$ of vectors that form a basis in $\mathbb{Z}^n$, the lattice generated by this basis is all linear combinations of the vectors with integer coefficients. That is:

$$\Lambda = \left[\sum_{i=1}^{n} a_i \boldsymbol{v}_i | a_i \in \mathbb{Z} \right]$$

For the first two schemes, we will primarily be working with lattices that correspond to ideals in the ring $\mathbb{Z}_p[x]/\langle x^n + 1 \rangle$, where p is some odd integer, and n typically will be a power of 2. In the last scheme, the lattices are in either the ring $\mathbb{Z}_q$ or in the ring $\mathbb{Z}_q[x]/\langle x^n + 1 \rangle$ where q is some odd integer and n is even. In the case of the lattices being $\mathbb{Z}_p[x]/\langle x^n + 1 \rangle$ we can represent the polynomials in the ring as vectors, where the vector $(\boldsymbol{v}_0,\ \boldsymbol{v}_1,\ ...,\ \boldsymbol{v}_{n-1})$ corresponds to the polynomial $\boldsymbol{v}_0 + \boldsymbol{v}_1 x + ... + \boldsymbol{v}_{n-1} x^{n-1}$. Furthermore, when we refer to multiplication of vectors in the first two schemes, what we actually mean is multiplication of the polynomials that those vectors represent. We can then get an element in a lattice of dimension m by choosing m such polynomials, and having the lattice element be the vector of vectors representing these polynomials.

In the schemes described later one might wonder where these lattices appear, as it isn't immediately obvious. In fact, lattices aren't necessary to do any of the calculations in the schemes. The reason why they are important is that there are certain lattice-based problems that are quite difficult to solve. Furthermore, the schemes are constructed in such a way that breaking them are equivalent to solving the difficult lattice-based problems. The lattice-based problems we will be using in this thesis is the approximate Shortest Vector Problem SVP$_\gamma$) for the first two schemes and the Short Integer Solution (SIS) for the last scheme.

Approximate Shortest Vector Problem

The Shortest Vector Problem (SVP) asks us to find the vector in Λ with the lowest infinity norm. The SVP$_\gamma$ is a variation of this problem where we attempt to find a vector that is at most γ times as large as the shortest vector. That is, if we have that $||\boldsymbol{w}_0||_\infty \leq ||\boldsymbol{w}||_\infty \forall \boldsymbol{w} \in \Lambda$ then $\boldsymbol{v}$ is a solution to the SVP$_\gamma$ if $\gamma ||\boldsymbol{v}||_\infty \leq ||\boldsymbol{w}_0||_\infty$. It is important to

note that while a lot is known about SVP for lattices in general, much less is known for SVP when using ideal lattices . Still, the best available algorithms for solving SVP do not appear to be able to take advantage of the usage of ideal lattices instead of general lattices. Thus we still expect this problem to be difficult to solve.

Short Integer Solution

The Short Integer Solution problem (SIS) asks us, given some matrix $A \in \mathbb{Z}_q^{r \times n}$ to find a short vector $\boldsymbol{x} \in \mathbb{Z}^r$ such that $Ax = 0 \pmod q$. The difficulty of this problem depends on what we define as a "short" vector. For x to be short it has to satisfy $||x|| < \beta$ for some β. If $\beta \geq q$ we can use $x = (q, 0, ..., 0)$ which is a trivial solution to the problem. Thus the problem is only interesting if $\beta < q$. This problem has been shown to be secure in the average case if SVP$_\gamma$ is hard in a worst case scenario by Ajtai [1] in 1996. Furthermore, if n is large, it was shown in [7] that one should solve SIS for a submatrix in $\mathbb{Z}^{r \times n'}$, where $n' = \sqrt{r \log q / \log \delta}$ for a constant δ. The best available algorithms are then able to find a solution s of length approximately $\min(q, 2^{\sqrt{r \log q \log \delta}})$.

2.4 Lattice-Based Collision-Resistant Hash Function

We will be using the constructions from [4] for our collision-resistant hash function, which we will use in the schemes presented in chapter 3. Let R be the ring $\mathbb{Z}_p[x]/\langle x^n + 1 \rangle$. We are interested in mapping values from a subset of R^m to R. To create a hash function h we choose a set of m elements in R, denoted by $\hat{\boldsymbol{a}} = (\boldsymbol{a}_1, \boldsymbol{a}_2, ..., \boldsymbol{a}_m)$, $\boldsymbol{a}_i \in R$. The hash value can then be calculated as $h(\hat{\boldsymbol{x}}) = \hat{\boldsymbol{a}} \cdot \hat{\boldsymbol{x}} = \boldsymbol{a}_1\boldsymbol{x}_1 + \boldsymbol{a}_2\boldsymbol{x}_2 + ... + \boldsymbol{a}_m\boldsymbol{x}_m$ (where the addition and multiplication is perform in R). It can be shown that $h(\hat{\boldsymbol{s}}\boldsymbol{c} + \hat{\boldsymbol{y}}) = h(\hat{\boldsymbol{s}})\boldsymbol{c} + h(\hat{\boldsymbol{y}})$ holds for this hash function, which we will utilize in the schemes in chapter 3.

If we then restrict the valid input values to the hash function to D, where $D \subseteq R$, the collision problem is then to find two distinct elements in D, $\hat{\boldsymbol{z}}$ and $\hat{\boldsymbol{z}}'$, such that $h(\boldsymbol{z}) = h(\boldsymbol{z}')$.

An important property of this hash function is that it is collision-resistant, which we define as follows:

Definition 1. *Let h be a hash function as defined above. Given $D \subseteq R$, the collision problem $Col(h, D)$ asks us to find two distinct elements $\hat{\boldsymbol{z}}$ and $\hat{\boldsymbol{z}}'$, such that $h(\hat{\boldsymbol{z}}) = h(\hat{\boldsymbol{z}}')$*

In [6] it was shown that solving this problem, given restrictions on D, m and R, is as hard as solving SVP$_\gamma$ for R, given by the following theorem:

Theorem 1. *Let $R = \mathbb{Z}_p[x]/\langle x^n + 1 \rangle$ be a ring where n is any power of 2, and define $D = \boldsymbol{y} \in R : ||\boldsymbol{y}||_\infty \leq d$ for some integer d. Let h be a hash function as defined above such that $m > \frac{\log(p)}{\log(2d)}$ and $p \geq 4dmn^{1.5}\log(n)$. If there is a polynomial-time algorithm that solves the $Col(h, D)$ problem for a random h with some non-negligible probability, then there is a polynomial-time algorithm that can solve SVP$_\gamma(\Gamma)$ for every (x^n+1)-cyclic lattice Γ, where $\gamma = 16dmn\log^2(n)$.*

2.5 Lattice-Based Commitment Schemes

A commitment scheme is a scheme where one party commits to specific secret values, and gains the option to later reveal those values. The other party would then be able to verify that the revealed value was indeed the value that was commited to. The two main properties of a commitment scheme are hiding and binding. Hiding means that the receiving party cannot find the secret values given only the commitment, and binding means that the sender cannot later change the value they commited to.

Typically this is done by using some collision-resistant one-way function f, that takes the message $\boldsymbol{m}$ and something random $\boldsymbol{r}$ as input, and outputs a commitment $\boldsymbol{M}$. Since f is a one-way function it is difficult to find $\boldsymbol{m}$ given only $\boldsymbol{M}$, thus satisfying the hiding property. Furthermore, since f is collision resistant it is difficult to find $\boldsymbol{m}', \boldsymbol{r}'$ such that $f(\boldsymbol{m}, \boldsymbol{r}) = f(\boldsymbol{m}', \boldsymbol{r}') = \boldsymbol{M}$. Then, given ck the receiver knows that a sender that can change the message can also find a collision. Since the function is collision-resistant we have satisfied the binding property.

This means that we can make a commitment scheme given a collision-resistant one way function. In this thesis we will be utilizing the short integer solution problem to create this function. Defining $f(\boldsymbol{s}) = A\boldsymbol{s} = \boldsymbol{S}$ for some random matrix A, we note that a $\boldsymbol{s}$ such that $||\boldsymbol{s}||_\infty$ is small would be a solution to the SIS-problem. Since this problem is very difficult, we have that the function is both collision resistant and one-way. As explained earlier in this chapter, given $A \in \mathbb{Z}_q^{r \times n}$ the best currently available algorithms can find $\boldsymbol{S}$ with $||\boldsymbol{s}||_\infty \approx \min(q, 2^{r \log q \log \delta})$.

We can then use this function to create a commitment scheme. Let A_1 be a matrix in $\mathbb{Z}_q^{r \times n}$ and A_2 be a matrix in $\mathbb{Z}_q^{r \times 2rlog_p q}$. Given that we have a message $\boldsymbol{m}$ that is a vector in $\mathbb{Z}_p^n$, and generate a random vector $\boldsymbol{r} \in \mathbb{Z}_p^{2r \log_p q}$. Given that $p << q$, we have that $||\boldsymbol{m}||_\infty \leq p$ and $||\boldsymbol{r}||_\infty \leq p$ are "small", and so we can send a commitment $\boldsymbol{S} = A_1\boldsymbol{m} + A_2\boldsymbol{r}$ to some receiver. If we later were to reveal $(\boldsymbol{m}, \boldsymbol{r})$ to the receiver, the receiver would be able to verify that $\boldsymbol{S} = A_1\boldsymbol{m} + A_2\boldsymbol{r}$. If the sender is then somehow able to change his message that would mean that the sender was able to find $[\boldsymbol{m}', \boldsymbol{r}']$ such that $f([\boldsymbol{m}', \boldsymbol{r}']) = \boldsymbol{S}$. In that case we have that $f([\boldsymbol{m} - \boldsymbol{m}', \boldsymbol{r} - \boldsymbol{r}']) = 0$, so $[\boldsymbol{m} - \boldsymbol{m}', \boldsymbol{r} - \boldsymbol{r}']$ would be a short integer solution for the matrix $A = [A_1, A_2]$. Since this is a difficult problem it is very unlikely that the sender can find such a pair $[\boldsymbol{m}', \boldsymbol{r}']$.

2.6 Interactive zero-knowledge

In cryptographic schemes it can often be necessary for one party to prove that they possess specific knowledge, such as knowing the value of a secret key. However, if you have to tell other parties the value of the secret key then it wouldn't remain a secret. So instead, the goal has to be to somehow convince the other party that you possess the secret key without giving them any information that would allow them to figure out what the secret key is. To do this we typically depend on a function where the secret key is an input and the output of the function is the public key. Then we can structure an interactive proof of knowledge from a prover to a verifier by using the following steps:

1. The secret key is s and the public key is $\boldsymbol{S} = f(s)$. The prover wants to prove that

they know the value s.

2. The prover picks a random value y, computes $Y = f(y)$ and sends $\boldsymbol{Y}$ to the verifier.

3. The verifier picks a random value c, also referred to as a challenge and sends it back to the prover.

4. The prover, knowing s and y now calculates $z = sc + y$ and sends z to the verifier.

5. The verifier then verifies that $f(z) = \boldsymbol{S}c + \boldsymbol{Y}$.

Obviously, there are some requirements to the process above. If the verifier is able to invert the function f, then they could find the secret key by computing $s = f^{-1}(\boldsymbol{S})$. This means that the function f has to be a one-way function for this to work. Additionally, the way the verifier checks that we indeed have the secret key is checking whether or not $f(z) = \boldsymbol{S}c + \boldsymbol{Y}$. Since $\boldsymbol{S} = f(s)$ and $\boldsymbol{Y} = f(y)$, for this to be true, we need that our function satisfies $f(sc + y) = f(s)c + f(y)$. Note though that plus and multiplication on the right side of the equtaion do not necessarily have to be plus and multiplication, just some mathematical operations that make sense in the group in which the image of the function resides. For example, if we were to construct this scheme using discrete logarithms, we could have $f(s) = g^s = \boldsymbol{S}$, in which case the operations on the right hand side would be multiplication and exponentiation instead of addition and multiplication (so we get $g^z = \boldsymbol{S}^c\boldsymbol{Y}$).

2.7 Fiat-Shamir and non-interactive zero-knowledge

In zero-knowledge schemes we often desire for the schemes to be non-interactive. While proving knowledge by interacting with some verifier is useful, it would be much more convenient to be able to upload some proof and have anyone be able to verify this proof. Fiat-Shamir is a technique used to transform an interactive scheme into a non-interactive one by using a random oracle (in practice this is done with a cryptographic hash function). The difference compared to the interactive proof is that instead of having the verifier send some random c, we use a random oracle to give us a random c. In practice we use the hash function to calculate some c that depends on the random value Y in the proof. This allows the verifier to check that the generated c is correct, and also makes it more difficult for a dishonest prover to generate a c that allows him to pretend to possess s. The steps in the proof of knowledge with Fiat-Shamir is very similar to the interactive version, except now the prover also has to calculate c himself as part of the proof:

1. The secret key is s and the public key is $\boldsymbol{S} = f(s)$. The prover wants to prove that they know the value s.

2. The prover picks a random value y, computes $\boldsymbol{Y} = f(y)$.

3. Now instead of having the verifier send a random c the prover instead calculates $c = H(\boldsymbol{S}, \boldsymbol{Y})$, where H is a cryptographic hash function.

4. The prover, knowing s and y now calculates $z = sc + y$ and sends $(z, \boldsymbol{Y})$ to the verifier.

5. The verifier then calculates $c = H(S, Y)$ and then verifies that $f(z) = \boldsymbol{S}c + \boldsymbol{Y}$.

In this version there is no interaction between the prover and the verifier - the prover simply sends the pair $(z, \boldsymbol{Y})$ (with the value $\boldsymbol{S}$ already being known since it is the public key of the prover), and then the verifier can check the proof. As a result anyone could verify this proof if it were to be uploaded, while in the interactive version only the person who sent the value c to the prover could verify the proof. The downside is that this proof also relies on the hash function being known and trusted.

2.8 Argument of Knowledge for polynomials

In this section we will explain how to create an argument of knowledge for a polynomial given a commitment scheme with certain homomorphic properties. A polynomial is of the form $f = c_0 + c_1x + c_2x^2 + ...c_nx^n$. Thus knowledge of the polynomial simply means knowing the coefficients $c_0, c_1, ...c_n$ of the polynomial. As a result of this, proving knowledge of the coefficients is sufficient to convince a verifier that you do indeed know the polynomial itself.

The typical process done to convince the verifier consists of the following steps:

1. The prover uses a commitment scheme to calculate commits for the coefficients, and sends these to the verifier. Let $\boldsymbol{F} = (\boldsymbol{F}_0, \boldsymbol{F}_1, ..., \boldsymbol{F}_n) = (CK(c_0), CK(c_1), ..., CK(c_n))$ denote these commitments.

2. The receiver picks a random evaluation point x for the polynomial and sends it to the prover.

3. The prover calculates the output y of the polynomial at the chosen evaluation point and sends it to the verifier.

4. The verifier checks that $CK(y) = \boldsymbol{F}(x)$ (where $\boldsymbol{F}(x)$ is calculated by using $\boldsymbol{F}_i$ as the coefficients in the polynomial).

Obviously, for this scheme to work there are a few requirements, as it is for example not immediately obvious that $CK(y) = F(x)$. For this to work, we require the commitment scheme to be homomorphic. If we chose CK such that $CK(ax + b) = CK(a)CK(x) + CK(b)$ is true, then the following equation also holds:

$$\begin{aligned}\boldsymbol{F}(x) &= \boldsymbol{F}_0 + \boldsymbol{F}_1CK(x) + ... + \boldsymbol{F}_nCK(x^n) \\ &= CK(\boldsymbol{F}_0 + \boldsymbol{F}_1x + ... + \boldsymbol{F}_nx^n) = CK(y)\end{aligned}$$

Additionally, since CK is hard to invert it would be difficult to find y only knowing $CK(y)$, so the prover shouldn't be able to calculate a y that satisfies $CK(y) = \boldsymbol{F}(x)$ without knowing the polynomial. This gives us a scheme that allows the prover to convince

the verifier that he possesses a polynomial. There are two other important properties we would like to scheme to have.

The first is seemingly obvious - we have been able to convince the verifier that we know some polynomial, but we haven't convinced them of any properties of the polynomial. The second issue is that the current scheme has no element of randomness, so the verifier can potentially figure out what the polynomial is by playing the role of the prover for various polynomials and see if he gets a match.

We will address the second issue first. The typical way to address this is to add an element of randomness to the commitment scheme. If we generate a random polynomial $g = d_0 + d_1x + ...d_nx^n$ of degree n and choose a modified commitment scheme where we can have two input values we get the following modified scheme:

1. The prover calculates $\boldsymbol{F} = (CK(c_0, d_0), ..., CK(c_n, d_n))$ and sends it to the verifier.

2. The receiver sends a random evaluation point x to the prover.

3. The prover calculates y, y' as the output of the two polynomials at the chosen evaluation point.

4. The verifier checks that $CK(y, y') = \boldsymbol{F}(x)$

The issue with f against a brute force attack was that if it is something simple like $f = x^n + x + 1$ then guessing it is far from impossible. On the other hand, guessing a random polynomial of degree n is much more difficult, so as a result the security of f is now guaranteed as long as g is difficult to guess (which should be the case for a randomly generated g given a reasonably large field and/or a reasonably large n).

The last property we need from our scheme is the ability to show that our polynomial has some specific properties. Since we have chosen a homomorphic commitment scheme we can continue to perform operations on the commited values. We can then use this to show that the polynomial in question has the desired properties. For example, if we wish to prove that our polynomial f has the root r_1 we could calculate $h = f/(x - r_1)$, calculate $y_1 = h(x)$, and providing $CK(y_1)$ as well as commitments to the coefficients of $h(x)$ to the verifier. The verifier could then check that $CK(y) = CK(y_1)CK(x - r_1)$ and that $\boldsymbol{H}(x) = CK(y_1)$. It also doesn't necessarily have to be a root we are trying to prove - we could demonstrate any property of the polynomial as long as the commitment scheme preserves that property.

2.9 Aborting

In the basic schemes presented in chapter 2.5 and 2.6, we send the value $z = sc + y$ to the verifier. The verifier has access to the value c, so given that the underlying algebraic structure is one where the verifier is able to invert c he would be able to access s if we didn't add y. This means that we need the value y to somehow protect us from leaking any information on sc. Imagine now as an arbitrary example that $0 \leq sc \leq 100$ and $0 \leq y \leq 100$. Then if the value z is greater than 100, say for example 150, the verifier

would know that since $y \leq 100$ then $sc \geq 50$, which leaks information on sc. One way to combat this is to have y be much larger than sc, since then the probability of this happening becomes much smaller. Another option is to only have y be a slightly larger than sc, but only send z if it doesn't reveal any information about sc. For example, if we let $0 \leq sc \ \leq 100$ and $0 \leq y \leq 200$, this would give us $0 \leq z \leq 300$. Now, we can choose to abort the scheme unless $100 \leq z \leq 200$. This way, for any value of sc, there is exactly 1 value of y that gives the transmitted value z, which means that z doesn't leak any information on y. Of course, this example is very simplified, but it still illustrates the concept behind why we will sometimes have to abort the scheme in the later constructions to avoid leaking information on the secret key.

Non-binary aborting

In the basic example described above, for a given value we either abort or proceed, depending on the value. This makes sense in the schemes that we presented in chapter 3, as in that case we can clearly separate between values that don't leak information values that do. For example, taking the above example of $0 \leq sc \leq 100$ and $0 \leq y \leq 200$, we note that for a random sc, any value $z = sc + y$ between 100 and 200 are leaks no information on sc, but values less than 100 and greater than 200 do.

However, we wish to also be able to account for cases where the equivalent to y and z can have the same values, but the probability distribution over these values are different for y and z. Imagine for example that

$$y = \begin{cases} 1 & \text{with probability } 0.7 \\ 2 & \text{with probability } 0.3 \end{cases} \qquad z = \begin{cases} 1 & \text{with probability } 0.5 \\ 2 & \text{with probability } 0.5 \end{cases}$$

This means that the distribution of z is different from that of y. To counteract this, we can force their distribution to be the same by aborting some percentage of the time in each case. If never abort when z is 1 and abort with probability $\frac{4}{7}$ when z is 2, then we end up with the distribution

$$z' = \begin{cases} 1 & \text{with probability } 0.5 \\ 2 & \text{with probability } \frac{3}{14} \\ \text{abort} & \text{with probability } \frac{4}{14} \end{cases}$$

This means that when we send z, it has the same distribution as y. Similarly, if we had 20 different values, each with one probability for one of the distributions and another for the other distributions, by aborting with various probabilities, we can make the probability distributions appear to be the same one in the cases where we do not abort. The more similar we want the probability distributions to be, the more often we will need to abort.

In the scheme presented in chapter 5 we will be using methods similar to what is described above. To accomplish this, we will present a probability distribution and a rejection sampling algorithm. Both are described in greater detail in [2]. Specifically, we will get matrices S, depending on the secret and Y, which has a Gaussian distribution. Then we wish to use aborting to make the distribution of $Z = S + Y$ be similar to the one of Y.

Probability Distribution for Rejection Sampling

We will be using a Gaussian distribution to create random matrices. However, the Gaussian distribution is continuous, and our values, being in $\mathbb{Z}_p$ and $\mathbb{Z}_q$, are discrete. Let $\rho_\sigma(x) = \exp(\frac{-x^2}{2\sigma^2})$ represent the continuous Gaussian distribution. Then we get the discrete Gaussian distribution over $\mathbb{Z}_q$ by calculating

$$G_{\sigma,q}(x) = \frac{\rho_\sigma(x)}{\sum_{i=1}^{q} \rho_\sigma(i)} \tag{2.1}$$

Additionally, given that $q >> \sigma$, we have that $G_{\sigma,q}(x) \approx G_\sigma$, where G_σ represents the distribution we get if we use $\mathbb{Z}$ instead of $\mathbb{Z}_q$.

Now, we define $G_\sigma^{m\times n}(x)$ as a $m \times n$ matrix where each element has probability distribution $G_\sigma(x)$. We also define $G_{\boldsymbol{v},\sigma}^{m}(x)$ as the discrete normal distribution of a vector in $\mathbb{Z}_q$, centered at the vector $\boldsymbol{v}$, and similar for matrices with $G_{\boldsymbol{M},\sigma}^{m\times n}(x)$.

Rejection Sampling Algorithm

Given matrices Z, S, variance σ and success probability ρ the rejection sampling algorithm proceeds with probability

$$Rej(Z, S, \sigma, \rho) = \frac{1}{\rho} \exp\left(\frac{-2\langle Z, S\rangle + ||S||^2}{2\sigma^2}\right) \tag{2.2}$$

and rejects otherwise. It was proven in [5] that for appropriately sized values σ and ρ, this algorithm proceeds with roughly probability $1/\rho$, and the statistical distance between Z and a Z' chosen randomly from G_σ is small if Z is not rejected.

2.10 Arithmetic Circuits

An arithmetic circuit C is a directed acyclic graph over some field F. In the graph, every source vertex contains an input value or a constant, and each other vertex contains either a "+" or a "$\times$". To calculate the output of the circuit, for each internal vertex that has only heads from vertices with numbers, we can calculate the value in this vertex by either adding or multiplying these numbers together (depending on whether the vertex itself contains "+" or "$\times$"). By repeating this process we will eventually have calculated the values in all the vertices in the graph, giving us as output the values in the sinks in the graph.

Given an arithmetic circuit and a specific output, it is typically very difficult to find inputs that outputs the desired value. This is known as the arithmetic circuit satisfiability problem. In this thesis we will later show how to provide a zero knowledge argument of knowledge for knowing a set of inputs that outputs some public value.

For the arithmetic circuits we will be looking at later in this thesis, we will assume that each vertex has indegree exactly 2. We can make this assumption because both the $+$ and $\times$ operations are associative, so instead of having a vertex of the form $a_1 + a_2 + a_3 = a_4$, we could have $a_1 + a_2 = a_5$ and $a_5 + a_3 = a_4$. Doing this makes allows us to create 2 (or more depending on the indegree) gates, each with indegree exactly 2 instead of a single gate with a higher indegree. The same can be done for multiplication. Thus, it is possible

to make an equivalent circuit where every vertex has exactly 2 inputs (but more gates in total).

Chapter 3

Lattice-based Identification and Signature Schemes

3.1 Identification

We will now present an identification scheme utilizing an interactive zero-knowledge proof, and the cryptographic hash function described in chapter 2.4.

First we create the instance of the hash function we will be utilizing. To do this, we pick m elements, $\boldsymbol{a}_1, \boldsymbol{a}_2, ..., \boldsymbol{a}_m$ at random from the ring R, given by $\mathbb{Z}_p[x]/\langle x^n + 1\rangle$. Now we set $\hat{\boldsymbol{a}} = (\boldsymbol{a}_1, \boldsymbol{a}_2, ..., \boldsymbol{a}_m)$ and $h(\hat{\boldsymbol{x}}) = \hat{a} \cdot \hat{\boldsymbol{x}} = \boldsymbol{a}_1\boldsymbol{x}_1 + \boldsymbol{a}_2\boldsymbol{a}_2 + ... + \boldsymbol{a}_m\boldsymbol{a}_m$. Then we can generate the secret key $\hat{\boldsymbol{s}}$ by similarly choosing m polynomials, $\boldsymbol{s}_1, \boldsymbol{s}_2, ..., \boldsymbol{s}_m$, at random, except now we also wish these polynomials to have an additional property, namely that $||\boldsymbol{s}_i||_\infty$ is small. For this scheme $\hat{\boldsymbol{s}}$ will be the secret key and $S = h(\hat{\boldsymbol{s}})$ will be the public key. The scheme works as follows:

1. The prover chooses m random elements in R, $\hat{\boldsymbol{y}} = (\boldsymbol{y}_1, \boldsymbol{y}_2, ..., \boldsymbol{y}_m)$, computes $Y = h(\hat{\boldsymbol{y}})$ and sends Y to the verifier.

2. The verifier then chooses a random polynomial $\boldsymbol{c}$ and sends $\boldsymbol{c}$ to the prover.

3. The prover computes $\hat{\boldsymbol{z}} = \hat{\boldsymbol{s}}\boldsymbol{c} + \hat{\boldsymbol{y}}$ and sends it to the verifier.

4. The verifier verifies that $h(\hat{\boldsymbol{z}}) = \boldsymbol{S}\boldsymbol{c} + \boldsymbol{Y}$.

For this to be a proper zero-knowledge scheme we require completeness, soundness and that it is zero-knowledge. The basic structure described above doesn't satisfy these properties, but it is possible to adjust it such that it does satisfy all three properties.

Completeness

First we note that it currently doesn't satisfy the completeness property. Since the hash function just takes the dot product of the value to be hashed and a set $\hat{\boldsymbol{a}} = \boldsymbol{a}_1, ..., \boldsymbol{a}_m$

publically known elements of R, we can easily find a solution to $h(\hat{\boldsymbol{z}}') = \boldsymbol{Sc} + \boldsymbol{Y}$. First we let $\hat{\boldsymbol{z}'} = (\boldsymbol{z}_1, \boldsymbol{z}_2, ..., \boldsymbol{z}_m)$, where $(\boldsymbol{z}_1, \boldsymbol{z}_2, ..., \boldsymbol{z}_{m-1}) = \boldsymbol{z}'$ are random polynomials. Then we denote $\hat{\boldsymbol{a}}' = \boldsymbol{a}_1, \boldsymbol{a}_2, ..., \boldsymbol{a}_{m-1}$. Now let $\boldsymbol{v} = \boldsymbol{Sc} + \boldsymbol{Y} - \hat{\boldsymbol{a}}' \cdot \hat{\boldsymbol{z}}'$. Then if we choose $\boldsymbol{z}_m = \boldsymbol{a}_m^{-1}\boldsymbol{v}$ we get $h(\hat{\boldsymbol{z}}') = \boldsymbol{Sc} + \boldsymbol{Y}$. The problem is that while we have required that $||\boldsymbol{s}_i||_\infty$ is small, we need to have a similar requirement on $\boldsymbol{z}_i$. Thankfully it is fairly simple to modify the scheme such that this will be the case.

It is not difficult to see that the following equation holds:

$$||\hat{\boldsymbol{s}}\boldsymbol{c} + \hat{\boldsymbol{y}}||_\infty \leq ||\hat{\boldsymbol{s}}||_\infty ||\boldsymbol{c}||_1 + ||\hat{\boldsymbol{y}}||_\infty \tag{3.1}$$

As a result, by also requiring $||\hat{\boldsymbol{s}}||_\infty$, $||\boldsymbol{c}||_1$ and $||\hat{\boldsymbol{y}}||_\infty$ to be small, the resulting $||\hat{\boldsymbol{z}}||_\infty$ will also be small. We define the following subsets of R.

$$\begin{aligned} D_s &= \boldsymbol{s} \in R : ||s||_\infty \leq \sigma \\ D_c &= \boldsymbol{c} \in R : ||c||_1 \leq \kappa \\ D_y &= \boldsymbol{y} \in R : ||y||_\infty \leq mn\sigma\kappa \\ G &= \boldsymbol{g} \in R : ||g||_\infty \leq mn\sigma\kappa - \sigma\kappa \end{aligned}$$

Now, by choosing $\hat{\boldsymbol{s}}$ from D_s^m, $\boldsymbol{c}$ from D_c and $\hat{\boldsymbol{y}}$ from D_y^m we get that $||\hat{\boldsymbol{z}}||_\infty$ is relatively small. From equation 3.1 we can already see that $||\hat{\boldsymbol{z}}||_\infty \leq mn\sigma\kappa + \sigma\kappa$. However, if $||\hat{\boldsymbol{z}}||_\infty > mn\sigma\kappa$ this would mean that $||\hat{\boldsymbol{s}}\boldsymbol{c}||_\infty > ||\hat{\boldsymbol{z}}||_\infty - mn\sigma\kappa$, which means that we are leaking on the secret. This leakage would cause it to fail to satisfy the zero-knowledge property, so it has to be avoided. This can be done by aborting the scheme every time it would leak on the secret. Furthermore, if $\hat{\boldsymbol{z}} \in D_y^m$ but $\hat{\boldsymbol{z}} \notin G^m$ there exists $\hat{\boldsymbol{s}}\boldsymbol{c}$ such that $\hat{\boldsymbol{z}} - \hat{\boldsymbol{s}}\boldsymbol{c} \notin D_y^m$, which means that this particular $\hat{\boldsymbol{s}}\boldsymbol{c}$ would not be possible since we know that $\hat{\boldsymbol{y}} = \hat{\boldsymbol{z}} - \hat{\boldsymbol{s}}\boldsymbol{c}$ and that $\hat{\boldsymbol{y}} \in D_y^m$. As a result, to avoid leaking on the secret, we also have to abort if $\hat{\boldsymbol{z}} \notin G^m$.

If $\hat{\boldsymbol{z}} \in G^m$ we have that any $\hat{\boldsymbol{z}} - \hat{\boldsymbol{s}}\boldsymbol{c}$ is in D_y^m, so there is no restrction on $\hat{\boldsymbol{s}}\boldsymbol{c}$ given that $\hat{\boldsymbol{z}}$. This means that no information is leaked, so the operations can proceed. The probability of $\hat{\boldsymbol{z}} \in G^m$ given that the other parameters are chosen randomly like above is the same as the probability that $\hat{\boldsymbol{y}}$ is in G^m. This is true because given any pair $(\hat{\boldsymbol{s}}\boldsymbol{c}, \hat{\boldsymbol{y}})$ such that $\hat{\boldsymbol{y}} \in G^m$ and $\hat{\boldsymbol{s}}\boldsymbol{c} + \hat{\boldsymbol{y}} \notin G^m$, there is also exactly 1 pair $(\hat{\boldsymbol{s}}'\boldsymbol{c}', \boldsymbol{y}')$ such that $\hat{\boldsymbol{y}}' \notin G^m$ but $\hat{\boldsymbol{s}}'\boldsymbol{c}' + \hat{\boldsymbol{y}}'$ is in G^m, given by $\hat{\boldsymbol{s}}'\boldsymbol{c}' = -\hat{\boldsymbol{s}}\boldsymbol{c}$ and $\hat{\boldsymbol{y}}' = \hat{\boldsymbol{s}}\boldsymbol{c} + \hat{\boldsymbol{y}}$.

This means that the probability of scheme proceeding is the probability of a random $\hat{\boldsymbol{y}} \in D_y^m$ being in G^m. Let p_G denote the probability of a random coefficient from a polynomial in D_y also being in G. $\hat{\boldsymbol{y}}$ is a set of m ring elements, each with n coefficients, so the probability of a random $\hat{\boldsymbol{y}} \in G^m$ is equal to p_G^{mn}. Since each of the coefficients have a random value between 0 and $mn\sigma\kappa$, we get that $p_G = \frac{mn\sigma\kappa - \sigma\kappa}{mn\sigma\kappa} = 1 - 1/mn$. Then the probability of $\hat{\boldsymbol{y}}$ being in G^m is $(1 - 1/mn)^{mn} \approx 1/e$. So we conclude that the scheme proceeds in roughly $1/e$ of cases, and aborts the rest of the time. As a result, the completeness of the scheme is $1/e$.

Obviously, completeness of $1/e$ by itself isn't satisfying. However, there are some simple measures that can be taken to increase this success rate dramatically. For example, we could just repeat the scheme every time it fails, and since the success rate is $1/e$ we'd only expect to have to repeat it e times before succeeding. Alternatively, we could have

the prover choose multiple $\hat{\boldsymbol{y}}_1, \hat{\boldsymbol{y}}_2, ..., \hat{\boldsymbol{y}}_k$. The verifier would then send back challenges for these $\hat{\boldsymbol{y}}_i$, and the prover could work his way down the list and respond to the first one where the scheme doesn't abort. That way, we can increase the chance of success by just adding more $\hat{\boldsymbol{y}}$'s. Additionally, instead of using different challenges for each $\hat{\boldsymbol{y}}$, we can use the same one each time, which saves us from having to generate and transmit so many challenges.

Soundness

In the previous chapter we explained how an algorithm that solves the $col(h, D)$ problem can be used to solve the SVP$_\gamma$ problem, which we believe to be a difficult problem. As a result, the goal will be to show that any attacker that breaks the described scheme can be used to solve the $col(h, D)$ problem and as a result also being able to solve the SVP$_\gamma$ problem.

Theorem 2. *For any given secret key $\hat{\boldsymbol{s}}$, with a very high probability there exists $\hat{\boldsymbol{s}}'$ such that $\hat{\boldsymbol{s}}' \in D_s^m$ and $h(\hat{\boldsymbol{s}}) = h(\hat{\boldsymbol{s}}')$.*

We note that from the restrictions on D_s, there are n coefficients in each element in D_s, and for each coefficient we have $2\sigma + 1$ possible choices $[-\sigma, ..., -1, 0, 1, ..., \sigma]$. This gives us a total of $(2\sigma + 1)^n$ elements in D_s, and $(2\sigma + 1)^{mn}$ elements in D_s^m. The hash function h maps elements from R^m to R. Since R is modulo p, and has n coefficients, the number of elements in the image of the hash is p^n. By setting p at some reasonable size, we can thus make it such that there are many more possible secret keys than elements in the image of the hash function which makes collisions very likely to occur. In [4] they suggested $p \approx (2\sigma + 1)^m 2^{\frac{-128}{n}}$. For this value of p, the number of elements in the image of the hash is roughly $(2\sigma + 1)^{mn} 2^{-128}$, which means that for every element of the image of the hash we have 2^{128} elements in D_s^m. Since the image has so many fewer elements than the preimage, any random element in the image has a very high probability of having a collision in the preimage.

Theorem 3. *Given 2 possible secret keys $(\hat{\boldsymbol{s}}, \hat{\boldsymbol{s}}')$ satisfying $h(\hat{\boldsymbol{s}}) = h(\hat{\boldsymbol{s}}')$ it is impossible for an adversary to tell which one was used for the scheme.*

We will here use the results from theorem 2. In this case the verifier will see the values $\boldsymbol{S} = h(\hat{\boldsymbol{s}})$, $\boldsymbol{Y} = h(\hat{\boldsymbol{y}})$ and $\hat{\boldsymbol{z}} = \hat{\boldsymbol{s}}\boldsymbol{c} + \hat{\boldsymbol{y}}$. We define

$$\hat{\boldsymbol{y}}' = \hat{\boldsymbol{y}} + \hat{\boldsymbol{s}}\boldsymbol{c} - \hat{\boldsymbol{s}}'\boldsymbol{c} = \hat{\boldsymbol{y}} + (\hat{\boldsymbol{s}} - \hat{\boldsymbol{s}}')\hat{\boldsymbol{c}}$$

We now note that $\hat{\boldsymbol{y}}' \in D_y^m$ because $\hat{\boldsymbol{s}}\boldsymbol{c} + \hat{\boldsymbol{y}} \in G^m$, and G^m and D_y^m are defined such that $\hat{\boldsymbol{z}} - \hat{\boldsymbol{s}}\boldsymbol{c} \in D_y^m$ for $\hat{\boldsymbol{z}} \in G^m, \hat{\boldsymbol{s}} \in D_s^m, \boldsymbol{c} \in D_c$. Furthermore, we have that

$$h(\hat{\boldsymbol{y}}') = h(\hat{\boldsymbol{y}}) + \boldsymbol{c}h(\hat{\boldsymbol{s}} - \hat{\boldsymbol{s}}')$$

As a result of $h(\hat{\boldsymbol{s}}) = h(\hat{\boldsymbol{s}}')$, we get $h(\hat{\boldsymbol{s}}) - h(\hat{\boldsymbol{s}}') = 0$ which means that $h(\hat{\boldsymbol{y}}') = h(\hat{\boldsymbol{y}}) = \boldsymbol{Y}$. Lastly, we have that

$$\hat{\boldsymbol{s}}'\boldsymbol{c} + \hat{\boldsymbol{y}}' = \hat{\boldsymbol{s}}'\boldsymbol{c} + \hat{\boldsymbol{y}} + \hat{\boldsymbol{s}}\boldsymbol{c} - \hat{\boldsymbol{s}}'\boldsymbol{c} = \hat{\boldsymbol{y}} + \hat{\boldsymbol{s}}\boldsymbol{c} = \hat{\boldsymbol{z}}$$

This means that whether we have secret key $\hat{\boldsymbol{s}}$ and random variable $\hat{\boldsymbol{y}}$ or secret key $\hat{\boldsymbol{s}}'$ and random variable $\hat{\boldsymbol{y}}'$, the information sent to the verifier will be the same. Thus, the verifier can't possibly know which of the 2 possible secret keys were used for the scheme.

Theorem 4. *Given that an adversary breaks the scheme and the above statements are true, it is possible to use this to break the $col(h, D)$ problem with a non-negligible probability.*

We assume that we have some adversary that first acts as the verifier of the scheme, and then acts as the prover for the same secret key. We will also be using the results from theorem 2 and theorem 3. We generate some secret $\hat{\boldsymbol{s}}$ and public key $\boldsymbol{S} = h(\hat{\boldsymbol{s}})$. First we will play the role of the prover and the adversary will play the role of the verifier. This is simple to perform since we know the secret key, so we can just perform each step of the scheme as normal. Then we have the adversary act as the prover. As the prover, the adversary will generate some $\hat{\boldsymbol{y}}$ and send $\boldsymbol{Y} = h(\hat{\boldsymbol{y}})$. We generate random challenges $\boldsymbol{c}_i$ until we get the response $\hat{\boldsymbol{z}_i}$ from the adversary (where $h(\hat{\boldsymbol{z}_i}) = \boldsymbol{S}\boldsymbol{c}_i + \boldsymbol{Y}$). Then we rewind the scheme and generate new random challenges $\boldsymbol{c}_j$ until we get a second response $\hat{\boldsymbol{z}_j}$ from the adversary. We have that

$$h(\hat{\boldsymbol{z}_i} - \hat{\boldsymbol{s}}\boldsymbol{c}_i) = h(\hat{\boldsymbol{z}_i}) - h(\hat{\boldsymbol{s}}\boldsymbol{c}_i) = \boldsymbol{S}\boldsymbol{c}_i + \boldsymbol{Y} - \boldsymbol{S}\boldsymbol{c}_i = \boldsymbol{Y}$$

Similarly

$$h(\hat{\boldsymbol{z}_j} - \hat{\boldsymbol{s}}\boldsymbol{c}_j) = \boldsymbol{S}\boldsymbol{c}_j + \boldsymbol{Y} - \boldsymbol{S}\boldsymbol{c}_j = \boldsymbol{Y}$$

Thus we have that $h(\hat{\boldsymbol{z}_i} - \hat{\boldsymbol{s}}\boldsymbol{c}_i) = h(\hat{\boldsymbol{z}_j} - \hat{\boldsymbol{s}}\boldsymbol{c}_j)$. From theorem 2 and theorem 3 we know that even if the adversary can break the scheme and finds a $\hat{\boldsymbol{s}}'$ such that $h(\hat{\boldsymbol{s}}') = \boldsymbol{S}$ it is still impossible for him to know whether or not $\hat{\boldsymbol{s}} = \hat{\boldsymbol{s}}'$. Since the probability of there being 2 or more values satisfying $h(\hat{\boldsymbol{s}}' = \boldsymbol{S})$ is very high from the theorem 2, the adversary will not know the exact secret key with probability at least $1/2$. This means that with probability at least $1/2$ we will have that $\hat{\boldsymbol{z}_i} - \hat{\boldsymbol{s}}\boldsymbol{c}_i$ and $\hat{\boldsymbol{z}_j} - \hat{\boldsymbol{s}}\boldsymbol{c}_j$ will be different (they will only be the same with non-negligible probability if both $\hat{\boldsymbol{z}_i}$ and $\hat{\boldsymbol{z}_j}$ are generated using $\hat{\boldsymbol{s}}$, which has a probability lower than $1/2$ to occur). Since they are different but both hash to $\boldsymbol{Y}$ we have used this adversary to break the $col(h, D)$ problem with probability at least $1/2$.

Theorem 5. *An successful attack on the described identification scheme can be used to construct an attack with a non-negligible success rate against the SVP$_\gamma$ problem.*

This follows directly from theorem 1 and theorem 4.

Zero-Knowledge

We want to show that the verifier learns nothing about $\hat{\boldsymbol{s}}$ during the scheme. By running the scheme, the verifier learns the values $\boldsymbol{Y}$, which does not depend on $\hat{\boldsymbol{s}}$, and $\hat{\boldsymbol{z}}$, which does depend on $\hat{\boldsymbol{s}}$. Furthermore, if the adversary would be able to find $\hat{\boldsymbol{y}}$ given $\boldsymbol{Y}$ he would also be able to find $\hat{\boldsymbol{s}}$ given $\boldsymbol{S}$, so learning $\boldsymbol{Y}$ doesn't help the adversary. To satisfy the zero-knowledge property we then just need $\hat{\boldsymbol{z}}$ to not leak any information on $\hat{\boldsymbol{s}}$. As described earlier in the proof of completeness we will abort the scheme if $\hat{\boldsymbol{z}} \notin G^m$. Given $\hat{\boldsymbol{z}} \in G^m$ we have that $\hat{\boldsymbol{z}} - \hat{\boldsymbol{s}}\boldsymbol{c} \in D_y^m$, which means that for any $(\hat{\boldsymbol{s}}, \boldsymbol{c})$ there exists exactly 1 value $\hat{\boldsymbol{y}}$ such that $\hat{\boldsymbol{y}} \in D_y^m$ and $\hat{\boldsymbol{z}} = \hat{\boldsymbol{s}}\boldsymbol{c} + \hat{\boldsymbol{y}}$. Then, for any $(\hat{\boldsymbol{z}}, \boldsymbol{c})$ known by the verifier all secret keys $\hat{\boldsymbol{s}}$ are possible (and equally likely), meaning that no information was leaked. This means that the scheme also satisfies the zero-knowledge property.

3.2 Signature scheme

Using Fiat-Shamir, we can transform the identification scheme from the previous section into a non-interactive signature scheme. Instead of having the verifier send a challenge $\boldsymbol{c}$ we will be using a publically known hash function H to generate $\boldsymbol{c}$. This hash function H can be any secure hash function that given input $\boldsymbol{S}, \boldsymbol{Y}$ gives us the output $H(\boldsymbol{S}, \boldsymbol{Y}) = \boldsymbol{c} \in D_c$. Now we can perform the scheme from the previous section, except we no longer need the interaction from the verifier to obtain $\boldsymbol{c}$. This gives us the following scheme:

1. The prover chooses m random elements, $\hat{\boldsymbol{y}} = (y_1, y_2, ..., y_m) \in D_y^m$ and computes $Y = h(\hat{\boldsymbol{y}})$.
2. The prover computes $c = H(\boldsymbol{S}, \boldsymbol{Y}) \in D_c$
3. The prover computes $\hat{\boldsymbol{z}} = \hat{\boldsymbol{s}}\boldsymbol{c} + \hat{\boldsymbol{y}}$.
4. The prover checks that $\hat{\boldsymbol{z}} \in G^m$, and either aborts if it is not, or sends $\hat{\boldsymbol{z}}$ to the verifier if it is.
5. The verifier verifies that $h(\hat{\boldsymbol{z}}) = \boldsymbol{S}\boldsymbol{c} + \boldsymbol{Y}$ and that $\boldsymbol{c} = H(\boldsymbol{S}, \boldsymbol{Y})$.

Since the only action the verifier does is to verify that the proof is indeed correct, we could even upload the proof on the internet, allowing anyone to take upon them the role of the verifier of the scheme. Although this scheme still only succeeds with probability $1/e$, we also no longer need new data from the verifier each time it fails. This means that we can generate new $\hat{\boldsymbol{y}}$ and repeat the scheme with the new $\hat{\boldsymbol{y}}$ until it succeeds. As a result, the verifier only participates in the scheme in the case of success, reducing the amount of necessary communication.

The completeness and zero-knowledge properties of this scheme follow from the identification scheme in the previous section. Furthermore, using similar ideas as the ones in the previous section, we can prove that a cheating adversary that manages to deceive the verifier can be used to find a collision in either h or H.

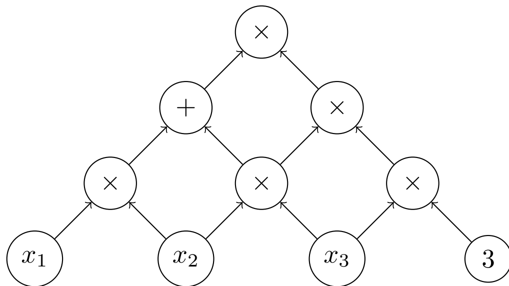
Chapter 4

Reducing an arithmetic circuit into equivalent polynomials

4.1 From arithmetic circuit to an equivalent polynomial

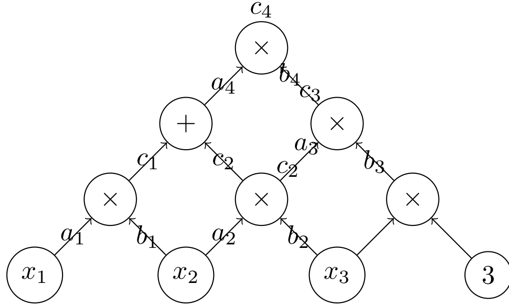
We want to provide an efficient method for creating arguments for the satisfiability of an arithmetic circuit. To do this, we will first reduce the arithmetic circuit to two sets of constraints. Then, once that is accomplished, we will create polynomials using these constraints, such that the constraints being satisfied implies equality on a specific coefficient in the polynomial equation . The material in this chapter follows the material presented in [2] and [3].

The first step is as mentioned to reduce an arithmetic circuit into two sets of constraints. To make the rest of the section easier to follow, we will start with an example that will illustrate the process, such that the reader can more easily understand what the two sets of constraints are, and how to obtain them. In our example we will be looking at the following circuit:

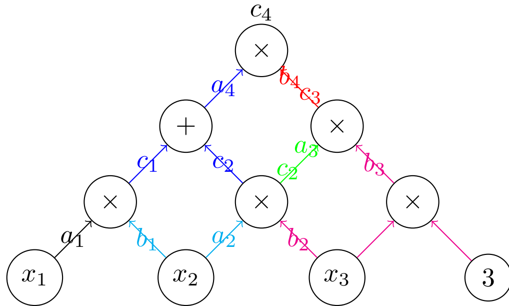


The goal is to transform the above circuit into equations with variables a_i, b_i, c_i for $i \in [1, 2, ..., N]$ of two different types. The first type of constraint is of the form $a_i b_i = c_i$. These will arise from the multiplication gates in which both inputs depend on the input values x_j. We will give each of these multiplication gates a number (in this case from 1

to 4, as one of the multiplication gates has 3 as an input which doesn't depend on the x_j). Then, for multiplication gate i, we define the left input as a_i, the right input as b_i and the output as c_i. Adding these to the circuit in the above example, we get the following:



The reader might observe that for example c_2 and a_3 share an edge. To ensure consistency among the variables in the multiplication constraints we will be using the second set of constraints - the linear constraints. These constraints will ensure that the variables in the multiplication constraints are consistent with the circuit. To create these, we break the circuit into subcircuits defined as the sections between the multiplication gates containing 2 or more of the values a_i, b_i, c_i used to form the multiplication constraints. To illustrate this we have given each edge in the circuit that is part of a linear relation a colour. We then get a subcircuit from the edges of each colour.



Now, for each subcircuit (represented by edges of a single colour), we can create linear constraints on the values a_i, b_i, c_i present in this subcircuit. This subcircuit will have some amount of edges going into the circuit (c_1 and c_2 in the case of the blue subcircuit) and some amount of edges going out of the circuit (a_4 in the case of the blue subcircuit). Because of the way the subcircuits are defined, each edge going into the subcircuit will be either a constant, one of the input values x_i of the arithmetic circuit, or one of the c_i. The output edges will always be one of the a_i or b_i, as the subcircuit ends at multiplication gates, and the input to those gates are always either a_i or b_i for some i.

Now, for each output edge of such a subcircuit we can create a linear constraint on it as a function of the input edges. For example, in the blue subcircuit we note that we have $a_4 = c_1 + c_2$. Let N define the number of multiplication constraints $a_i b_i = c_i$ and U denote the number of linear constraints. We can write all the linear constraints as:

$$\sum_{i=1}^{N} a_i w_{u,a,i} + \sum_{i=1}^{N} b_i w_{u,b,i} + \sum_{i=1}^{N} c_i w_{u,c,i} = K_u$$

for $u \in [1, 2, ..., U]$, where $w_{u,a,i}$ are the coefficients for the a_i in equation u, and similar for $w_{u,b,i}$ and $w_{u,c,i}$. Now we wish to transform our coloured example circuit into linear constraints.

1. The blue subcircuit gives us the equation $a_4 = c_1 + c_2$ or equivalently $a_4 - c_1 - c_2 = 0$.
2. The teal subcircuit gives us the equation $b_1 - a_2 = 0$
3. The magenta subcircuit gives us the equation $b_3 - 3b_2 = 0$
4. The green subcircuit gives us the equation $a_3 - c_2 = 0$
5. The red subcircuit gives us the equation $b_4 - c_3 = 0$
6. Additionally, we wish the arithmetic circuit in its entirety to be satisfied, which means that we wish c_4 to have some specific value. This adds the constraint $c_4 = K$, where K is the desired output of the arithmetic circuit.

We can now describe the circuit as the equivalent 4+6 constraints:

$$\begin{aligned} a_1 b_1 &= c_1 & a_2 b_2 &= c_2 \\ a_3 b_3 &= c_3 & a_4 b_4 &= c_4 \\ b_1 - a_2 &= 0 & b_3 - 3b_2 &= 0 \\ a_3 - c_2 &= 0 & b_4 - c_3 &= 0 \\ a_4 - c_1 - c_2 &= 0 & c_4 &= K \end{aligned}$$

The idea presented in [3] is to first represent the entire circuit as two different sets of constraints as illustrated in the example above. Knowing a_i, b_i, c_i satisfying the constraints then implies knowing inputs satisfying the arithmetic circuit. The prover can then convince a verifier of this knowledge by sending an opening to a_i, b_i, c_i to the verifier, then proving knowledge of the polynomial matching these a_i, b_i, c_i. The verifier would then use the homomorphic properties of the commitment scheme to check that the proven polynomial is a match with the commited values.

Now, having transformed the circuit into the two sets of constraints, let $N = nm$ denote the number of multiplication constraints of the form $a_i b_i = c_i$ (potentially padding N with some trivial constraints if necessary). We use these to create $m \times n$ matrices from these constraints, where A contains all the a_i, B contains all the b_i and C contains all the c_i. Let $\circ$ denote the Hadamard (entry-wise) product of matrices. Then we note that the multiplication constraints are all satisfied if the Hadamard product $A \circ B = C$ is true. We now define $\boldsymbol{a}_i, \boldsymbol{b}_i, \boldsymbol{c}_i$ as the rows of the matrices. Thus, $\boldsymbol{a}_i = (a_{i,1}, a_{i,2}, ..., a_{i,n})$ for $i \in [1, ..., m]$ and similarly for $\boldsymbol{b}_i$ and $\boldsymbol{c}_i$.

Now let U denote the number of linear constraints. We have that $U < 2N + O$, where O is the number of output values that the arithmetic circuit needs to satisfy. As mentioned in the example, the constraints arise from edges going "out" from the subcircuit, meaning that they are the inputs of a multiplication gate. These edges are always either a_j or b_j for some $j \in [1, 2, ..., N]$. Given that there are at most $2N$ edges a_i, b_i in total, we get that the total number of linear constraints is at most $2N + O$.

After we mapped all of the constraints into the matrices, we get that the linear constraints are of the form

$$\sum_{i=1}^{m} \boldsymbol{a}_i \boldsymbol{w}_{u,a,i} + \sum_{i=1}^{m} \boldsymbol{b}_i \boldsymbol{w}_{u,b,i} + \sum_{i=1}^{m} \boldsymbol{c}_i \boldsymbol{w}_{u,c,i} = K_u$$

for $u \in [1, 2, ..., U]$, where $\boldsymbol{w}_{u,a,i}$ denotes a vector representing the coefficients of $\boldsymbol{a}_i$ for equation u and K_u is a constant. Reusing the earlier example, we first set

$$(a_1, a_2, a_3, a_4) = (a_{1,1}, a_{1,2}, a_{2,1}, a_{2,2})$$

giving us

$$\boldsymbol{a}_1 = (a_{1,1}, a_{1,2}) = (a_1, a_2)$$
$$\boldsymbol{a}_2 = (a_{2,1}, a_{2,2}) = (a_3, a_4)$$

and do the same for b_i and c_i. This gives us

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \qquad B = \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix} \qquad C = \begin{pmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{pmatrix}$$

and the linear constraints now become

$$\sum_{i=1}^{2} \boldsymbol{a}_i \boldsymbol{w}_{u,a,i} + \sum_{i=1}^{2} \boldsymbol{b}_i \boldsymbol{w}_{u,b,i} + \sum_{i=1}^{2} \boldsymbol{c}_i \boldsymbol{w}_{u,c,i} = K_u \tag{4.1}$$

for $u \in [1, 2, ..., 6]$

Then, if we for example let $u = 1$ represent the linear constraint $a_4 - c_1 - c_2 = 0$, with the new indexes we get $a_{2,2} - c_{1,1} - c_{1,2} = 0$. Then, transforming this to the form of equation 4.1 we get

$$\begin{aligned} \boldsymbol{w}_{1,a,1} &= (0, 0) & \boldsymbol{w}_{1,a,2} &= (0, 1) \\ \boldsymbol{w}_{1,b,1} &= (0, 0) & \boldsymbol{w}_{1,b,2} &= (0, 0) \\ \boldsymbol{w}_{1,c,1} &= (-1, -1) & \boldsymbol{w}_{1,c,2} &= (0, 0) \\ K_1 &= 0 \end{aligned}$$

We can now create a polynomial by embedding the constraints into different powers of the indeterminate. This will allow us to make an argument of knowledge for the

polynomial, which we can use to convince the verifier that we know values satisfying the arithmetic circuit. Let Y be the indeterminate for our polynomial. Then we first note that $\sum_{i=1}^{m} \boldsymbol{a}_i \boldsymbol{b}_i Y^i = \sum_{i=1}^{m} \boldsymbol{c}_i Y^i$. Now let $\boldsymbol{Y}'$ denote $(1, Y^m, ..., Y^{nm-m})$. Then since $(\boldsymbol{a} \circ \boldsymbol{b}) \cdot \boldsymbol{Y}' = (\boldsymbol{a} \circ \boldsymbol{Y}') \cdot \boldsymbol{b}$ we can multiply in $\boldsymbol{Y}'$ on both sides to get

$$\sum_{i=1}^{m} \boldsymbol{a}_i \cdot (\boldsymbol{b}_i \circ \boldsymbol{Y}') Y^i = \sum_{i=1}^{m} \boldsymbol{c}_i Y^i \boldsymbol{Y}' \tag{4.2}$$

We now note that in this polynomial we have $nm = N$ different powers of Y, each corresponding to a single equation. Since we also have N equations, each power of Y must correspond to exactly 1 equation. As a result, if the polynomial on the LHS and the polynomial on the RHS are equal then it must also be the case that all the equations are satisfied. And if all the equations are satisfied it must then mean that we have satisfied all the multiplication constraints in the circuit.

We also want our polynomial to include all the linear constraints. We note that the above polynomials uses powers of Y starting from Y^1 and ending at Y^{nm}. Then, given U linear constraints, we could embed these to the powers from Y^{N+1} to Y^{N+U}.

Multiplying the relations by the powers of Y gives us:

$$\sum_{u=1}^{U} (\sum_{i=1}^{m} \boldsymbol{a}_i \boldsymbol{w}_{u,a,i} + \sum_{i=1}^{m} \boldsymbol{b}_i \boldsymbol{w}_{u,b,i} + \sum_{i=1}^{m} \boldsymbol{c}_i \boldsymbol{w}_{u,c,i}) Y^{N+u} = \sum_{u=1}^{U} K_u Y^{N+u}$$

Then, to make the equation look somewhat cleaner we define

$$\boldsymbol{w}_{a,i} = \sum_{u=1}^{U} \boldsymbol{w}_{u,a,i} Y^{N+u} \qquad \boldsymbol{w}_{b,i} = \sum_{u=1}^{U} \boldsymbol{w}_{u,b,i} Y^{N+u}$$

$$\boldsymbol{w}_{c,i} = \sum_{u=1}^{U} \boldsymbol{w}_{u,c,i} Y^{N+u} \qquad K(Y) = \sum_{u=1}^{U} K_u Y^{N+u}$$

Which gives us

$$\sum_{i=1}^{m} \boldsymbol{a}_i \boldsymbol{w}_{a,i} + \sum_{i=1}^{m} \boldsymbol{b}_i \boldsymbol{w}_{b,i} + \sum_{i=1}^{m} \boldsymbol{c}_i \boldsymbol{w}_{c,i} - K(Y) = 0 \tag{4.3}$$

Now, knowing matrices A, B, C satisfying 4.2 and 4.3 is equivalent to knowing inputs satisfying the arithmetic circuit. This means that if we can prove knowledge of these polynomials the verifier will be convinced that we know A, B, C.

Instead of proving the entire polynomial directly, we will create polynomials with a second indeterminate, X, such that the multiplication constraints or linear constraints are condensed into a single coefficient (specifically, the X^{m+1}-term).

$$\boldsymbol{a}(X) = \boldsymbol{a}_0 + \sum_{i=1}^{m} \boldsymbol{a}_i Y^i X^i$$
$$\boldsymbol{b}(X) = \boldsymbol{b}_{m+1} + \sum_{i=1}^{m} \boldsymbol{b}_i X^{m+1-i} \quad (4.4)$$
$$\boldsymbol{c} = \sum_{i=1}^{m} \boldsymbol{c}_i Y^i$$

We then note that

$$\boldsymbol{a}(X) \circ \boldsymbol{b}(X) = \boldsymbol{c} X^{m+1} + \sum_{i=0, i \neq m+1}^{2m} \boldsymbol{h}_i X^i \quad (4.5)$$

This means that the X^{m+1} term of equation 4.5 is equal if $\boldsymbol{a}_i \circ \boldsymbol{b}_i = \boldsymbol{c}_i$. This allows us to prove knowledge of the multiplication relations by proving knowledge of these polynomials $\boldsymbol{a}(X), \boldsymbol{b}(X), \boldsymbol{c}$, by proving that the X^{m+1} term is equal. The $\boldsymbol{h}_i$ are here vectors in $\mathbb{Z}_p^n$ that depend on $\boldsymbol{a}(X)$ and $\boldsymbol{b}(X)$. The prover will calculate these during the scheme and send openings for these values. Similarly, for the linear constraints we can define

$$\boldsymbol{a}(X) = \boldsymbol{a}_0 + \sum_{i=1}^{m} \boldsymbol{a}_i X^i \qquad \boldsymbol{w}_a = \sum_{i=1}^{m} X^{m+1-i} \boldsymbol{w}_{a,i}$$
$$\boldsymbol{b}(X) = \boldsymbol{b}_0 + \sum_{i=1}^{m} \boldsymbol{b}_i X^i \qquad \boldsymbol{w}_b = \sum_{i=1}^{m} X^{m+1-i} \boldsymbol{w}_{b,i}$$
$$\boldsymbol{c}(X) = \boldsymbol{c}_0 + \sum_{i=1}^{m} \boldsymbol{c}_i X^i \qquad \boldsymbol{w}_c = \sum_{i=1}^{m} X^{m+1-i} \boldsymbol{w}_{c,i}$$

Given these, we can then calculate (where $\cdot$ represents the dot product of the vectors).

$$\boldsymbol{a}(X) \cdot \boldsymbol{w}_a + \boldsymbol{b}(X) \cdot \boldsymbol{w}_b + \boldsymbol{c}(X) \cdot \boldsymbol{w}_c = \sum_{u=1}^{U} X^{m+1} Y^{N+u} K_u + \sum_{i=0, i \neq m+1}^{2m} h_i X^i \quad (4.6)$$

And similarly to equation 4.5, equation 4.6 is equal in the X^{m+1}-term if the linear constraints are satisfied. From here it would be possible to make a zero knowledge argument, but this argument will not be detailed in this thesis. Instead we will first look into how we can improve the efficiency of the scheme by using field extensions to reduce the amount of necessary calculations.

4.2 Using field extensions to improve efficiency

The previous section explained how to transform $N = nm$ multiplication constraints into matrices such that the constraints are equivalent to a Hadamard product of the matrices.

Extending this to $GF(p^{2k})$ we can have each element of the matrix be a field element represented by a vector. In this case, the $N = nmk$ relations (with new values n, m) can be represented in a smaller matrix, which speeds up computations. Instead of having 1 relation be represented by the multiplication $ab = c$ for $a, b, c \in \mathbb{Z}_p$, we want to represent k relations at the same time as the product $\boldsymbol{ab} = \boldsymbol{c}$ for $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in GF(p^{2k})$, where $GF(p^{2k}) \cong \mathbb{Z}[\phi]/\langle f(\phi)\rangle$. We will use $*$ to denote multiplication of field elements represented by vectors.

To illustrate the purpose of using the field elements, we will be using a simple example. Let $p = 3$ and $k = 2$. Now we have that our field is $GF(3^4)$. Let the field be represented by $\mathbb{Z}_3[x]/\langle x^4 + x + 2\rangle$. Then, we choose as our basis

$$\beta = u_1, u_2, u_3, u_4$$

where

$$u_1 = x + 2 \qquad u_2 = 2x + 2$$
$$u_3 = x^2 + 2 \qquad u_4 = x^3 + 2x$$

We note that (u_1, u_2, u_3, u_4) are linearly independent, and as a result form a basis in our field. Now we wish to map two multiplication relations to a single field element. Let

$$\begin{aligned}\boldsymbol{a} &= (a_1, a_2, 0, 0)\\ \boldsymbol{b} &= (b_1, b_2, 0, 0)\\ \boldsymbol{c} &= (c_1, c_2, c_1', c_2')\end{aligned}$$

for some $c_1', c_2' \in \mathbb{Z}_3$. If we now calculate

$$\boldsymbol{a} * \boldsymbol{b} = (a_1u_1 + a_2u_2)(b_1u_1 + b_2u_2) = a_1b_1u_1^2 + (a_1b_2 + a_2b_1)u_1u_2 + a_2b_2u_2^2$$

We also have that

$$\begin{aligned}u_1^2 &= (x+2)^2 \equiv x^2 + x + 4 = (x+2) + (x^2+2) = u_1 + u_3 \pmod 3\\ u_2^2 &= (2x+2)^2 \equiv x^2 + 2x + 4 = (2x+2) + (x^2+2) = u_2 + u_3 \pmod 3\\ u_1u_2 &= (x+2)(2x+2) \equiv 2x^2 + 4 = 2(x^2+2) = 2u_3 \pmod 3\end{aligned}$$

This means that we have

$$\begin{aligned}\boldsymbol{a} * \boldsymbol{b} &= a_1b_1(u_1 + u_3) + (a_1b_2 + a_2b_1)2u_3 + a_2b_2(u_2 + u_3)\\ &= a_1b_1u_1 + a_2b_2u_2 + (a_1b_1 + a_2b_2 + 2a_1b_2 + 2a_2b_1)u_3\end{aligned}$$

Now let

$$c_1' = (a_1b_1 + a_2b_2 + 2a_1b_2 + 2a_2b_1)$$

As a result

$$\boldsymbol{a} * \boldsymbol{b} = c_1u_1 + c_2u_2 + c_1'u_3$$

This means that with our current basis we can multiply field elements of the form $\boldsymbol{a}, \boldsymbol{b}$ and get a field element where the multiplication constraints are preserved for the first two coefficients of the vector.

Given a field extension $GF(p^{2k})$, we wish to find a basis with similar properties to the one in the above example. Let $\boldsymbol{a} = (a_1, a_2, ..., a_k, 0, ..., 0)$ and $\boldsymbol{b} = (b_1, b_2, ..., b_k, 0, ..., 0)$, where a_i, b_i are k multiplication constraints. Similarly, let $(c_1, c_2, ..., c_k)$ be the k c_i corresponding to the a_i, b_i in $\boldsymbol{a}, \boldsymbol{b}$. Then we want a field representation where $\hat{a} * \hat{b} = (c_1, c_2, ..., c_k, c'_1, c'_2, ..., c'_k)$, where c'_i are arbitrary coefficients depending on $\boldsymbol{a}, \boldsymbol{b}$. Now note that if we could find polynomials $(f_0, f_1, ..., f_k)$ satisfying the following equation

$$(\sum_{i=1}^{k} a_i f_i)(\sum_{i=1}^{k} b_i f_i) \equiv (\sum_{i=1}^{k} c_i f_i) \pmod{f_0} \tag{4.7}$$

such that that the degree of f_0 is k, the degree of $f_1, f_2, ..., f_k$ are $k-1$, and that all the polynomials are independent, then we get

$$(\sum_{i=1}^{k} a_i f_i)(\sum_{i=1}^{k} b_i f_i) = (\sum_{i=1}^{k} c_i f_i) + \sum_{i=0}^{k-2} f_0 c'_i \phi^i \tag{4.8}$$

In this case, if we use $\beta = f_1, f_2, ..., f_k, f_0, \phi f_0, ..., \phi^{k-1} f_0$ as our basis, we observe that this field representation satisfies $\boldsymbol{a} * \boldsymbol{a} = (c_1, c_2, ..., c_k, c'_1, ..., c'_k)$. This means that finding polynomials $f_1, f_2, ..., f_k, f_0$ satisfying these constraints will give us our desired basis.

Let $e_1, e_2, ..., e_k$ be distinct points in $\mathbb{Z}_p$. Let f_i be lagrange polynomials of degree $k-1$ associated with these points. Thus for all f_i we have $f_i(e_i) = 1$ and $f_i(e_j) = 0, i \neq j$. Furthermore, let $f_0 = \prod_{i=1}^{k}(\phi - e_i)$. We wish to prove that these polynomials satisfy equation 4.7.

We note that the left hand side of equation 4.7 can be split into k^2 equations of the form $a_i f_i b_j f_j$. We wish to prove that

1. $a_i f_i b_j f_j \equiv 0 \pmod{f_0}$ if $i \neq j$
2. $a_i f_i b_j f_j \equiv c_i f_i \pmod{f_0}$ if $i = j$

If these two statements hold, then equation 4.7 also holds for the same polynomials.

First we look at $a_i f_i b_j f_j \equiv 0 \pmod{f_0}$ for $i \neq j$. Since $f_i(e_j) = 0$ for $i \neq j$ it follows that we can write f_i as

$$f_i = g_i \prod_{l=1, i \neq j}^{k} (\phi - e_l)$$

Then

$$f_i f_j = g_i g_j \prod_{l=1, l \neq i,j}^{k} (\phi - e_l) \prod_{l=1}^{k} (\phi - e_l) = g_i g_j f_0 \prod_{l=1, l \neq (i,j)}^{k} (\phi - e_l)$$

If we define $g_{ij} = g_i g_j \prod_{l=1, l\neq i,j}^{k}$, we then get

$$a_i f_i b_j f_j = (a_i b_i g_{ij}) f_0 \equiv 0 \pmod{f_0}$$

which obviously holds.

Then we look at

$$a_i f_i b_i f_i \equiv f_i c_i \pmod{f_0}$$

Since $a_i b_i = c_i$ this reduces to

$$f_i^2 \equiv f_i \pmod{f_0}$$

Moving f_i to the left hand side, we get

$$f_i(f_i - 1) \equiv 0 \pmod{f_0}$$

Now,

$$f_0 = \prod_{i=1}^{k} (\phi - e_i)$$

and

$$f_i = g_i \prod_{j=1, i\neq j}^{k} (\phi - e_j)$$

Additionally, by our definition of f_i, $f_i(e_i) = 1$, so

$$f_i - 1 = g_i'(\phi - e_i)$$

for some polynomial g_i'. Then we have that

$$f_i(f_i - 1) = g_i g_i' f_0$$

and so

$$f_i^2 - f_i \equiv 0 \pmod{f_0}$$

and

$$f_i^2 \equiv f_i \pmod{f_0}$$

This means that $\beta = f_1, f_2, ..., f_k, f_0, \phi f_0, ..., \phi^{k-1} f_0$ is a basis for $GF(p^{2k})$ that preserves k multiplication relations, which was what we were looking for.

Similarly, we want to create a basis for the linear constraints such that we can have k linear relations represented by a single equation with field elements, in the same way that we represented the multiplication relations above. To do this we need to find a basis where this is doable.

Using a similar process as the one we used to create equation 4.3, we can create the equation

$$\sum_{i=1}^{k} a_i w_{a,i} + \sum_{i=1}^{k} b_i w_{b,i} + \sum_{i=1}^{k} c_i w_{c,i} = K \pmod{p}$$

Now adding an indeterminate ϕ to the equation, we can send all the linear constraints to the ϕ^{k-1}-term of the new equation.

$$\sum_{i=1}^{k} a_i\phi^{i-1}\sum_{i=1}^{k} w_{a,i}\phi^{k-i} + \sum_{i=1}^{k} b_i\phi^{i-1}\sum_{i=1}^{k} w_{b,i}\phi^{k-i} \\ + \sum_{i=1}^{k} c_i\phi^{i-1}\sum_{i=1}^{k} w_{c,i}\phi^{k-i} = K\phi^{k-1} + \sum_{i=0,i\neq k-1}^{2k-2} K_i\phi^i \pmod p \tag{4.9}$$

This new equation now contains all the linear constraints in the ϕ^{k-1}-term, with the coefficients of the other powers of ϕ being arbitrary, depending on the a, b, c, w-values. That means that if we choose a basis $\beta' = (1, \phi, ..., \phi^{2k-1})$, then we can represent $a_1, a_2, ..., a_k$ as a single field element in our linear constraints. Essentially, calculating the left hand side of the equation 4.9 will give a field element where the k-th component of the field element will be the sum of the linear constraints.

By using these new equations, instead of having N relations represented by a $m' \times n'$ matrix where all the elements of the matrix are in $\mathbb{Z}_p$, we now have the N relations represented by a $m \times n$ matrix where all the elements of the matrix are in $GF(p^{2k})$, and each element in $GF(p^{2k})$ is used to represent k such relations. This allows n, m to be smaller than n', m'.

We will now describe the structure for the matrix A representing $\boldsymbol{a}_1, \boldsymbol{a}_2, ..., \boldsymbol{a}_N$ in detail (the matrix B has the same structure, so only A will be described in detail). Let $A = (A_1, A_2, ..., A_m)$ be a vector with m elements. Then, we let each A_i be a $2k \times n$ matrix on this form

$$A_i = \begin{pmatrix} a_{i,1,1} & a_{i,1,2} & \cdots & a_{i,1,n} \\ a_{i,2,1} & a_{i,2,2} & \cdots & a_{i,2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{i,k,1} & a_{i,k,2} & \cdots & a_{i,k,n} \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}$$

We also define $\boldsymbol{a}_{i,j}$ as the j-th row in matrix A_i, consisting of the elements $(a_{i,j,1}, a_{i,j,2}, ..., a_{i,j,n})$, similar to how we used $\boldsymbol{a}_i$ to denote the rows of the matrix in $\mathbb{Z}_p$ in the previous section.

The matrix C_i has the same structure as above in its upper half. The lower half of the matrix also has the same structure as A_i, B_i in the case of linear constraints, but in the case of multiplication constraints the lower half contains arbitrary coefficients $c_{i,j,l}$ depending on A, B instead of 0's.

$$C_i = \begin{pmatrix} c_{i,1,1} & c_{i,1,2} & \cdots & c_{i,1,n} \\ c_{i,2,1} & c_{i,2,2} & \cdots & c_{i,2,n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{i,k,1} & c_{i,k,2} & \cdots & c_{i,k,n} \\ c'_{i,1,1} & c'_{i,1,2} & \cdots & c'_{i,1,n} \\ c'_{i,2,1} & c'_{i,2,2} & \cdots & c'_{i,2,n} \\ \vdots & \vdots & \ddots & \vdots \\ c'_{i,k,1} & c'_{i,k,2} & \cdots & c'_{i,k,n} \end{pmatrix}$$

In this matrix, each column vector represents a single element of $GF(p^{2k})$, where the last half of the coefficients are all 0. If we denote $\hat{a}_{i,j}$ as the j-th column of A_i, then let $\circledast$ be the multiplication of the field elements represented by the columns of a matrix. That is,

$$A_i \circledast B_i = (\hat{a}_{i,1} * \hat{b}_{i,1}, \hat{a}_{i,2} * \hat{b}_{i,2}, ..., \hat{a}_{i,n} * \hat{b}_{i,n})$$

Furthermore, given a field element x represented by a vector in $\mathbb{Z}_p^{2k}$, there exists a matrix M_x such that for field elements x, y then $x * y = M_x y$. This means that we can represent all the field multiplication as matrix multiplication. Then

$$A_i \circledast B_i = (M_{\hat{a}_{i,1}} \hat{b}_{i,1}, M_{\hat{a}_{i,2}} \hat{b}_{i,2}, ..., M_{\hat{a}_{i,n}} \hat{b}_{i,n})$$

This can be used to create polynomials similar to the ones found in 4.4, but now using elements in $GF(p^{2k})$ instead of $\mathbb{Z}_p$.

$$\begin{aligned} A(X) &= A_0 + \sum_{i=1}^{m} (M_y)^i (M_x)^i A_i \\ B(X) &= B_{m+1} + \sum_{i=1}^{m} (M_x)^{m+1-i} B_i X^{m+1-i} \\ C &= \sum_{i=1}^{m} (M_y)^i C_i \end{aligned} \tag{4.10}$$

Which gives us the equation

$$A(X) \circledast B(X) \pmod p = (M_x)^{m+1} C + \sum_{i=0, i \neq m+1}^{2m} (M_x)^i H_i \pmod p \tag{4.11}$$

Here we get that H_i again are arbitrary matrices depending on the $A(X)$ and $B(X)$ functions. The prover will calculate these during the scheme and send openings for these coefficients to the verifier.

Similarly to the definition of $\circledast$, for scalar products we define $\odot$ as the scalar product of the field elements represented by the columns, so

$$A_i \odot B_i = (\hat{a}_{i,1} * \hat{b}_{i,1} + \hat{a}_{i,2} * \hat{b}_{i,2} + ... + \hat{a}_{i,n} * \hat{b}_{i,n})$$

and

$$A_i \odot B_i = (M_{\hat{a}_{i,1}} \hat{b}_{i,1} + M_{\hat{a}_{i,2}} \hat{b}_{i,2} + ... + M_{\hat{a}_{i,n}} \hat{b}_{i,n})$$

We now wish to write equation 4.9 as a scalar product of columns. Noting that the powers of $\boldsymbol{w}_{a,i}$ are the same as the powers of a_i, but in reverse order, we can then define

$$W_{u,a,i} = \begin{pmatrix} w_{u,a,i,k,1} & w_{u,a,i,k,2} & \cdots & w_{u,a,i,k,n} \\ w_{u,a,i,k-1,1} & w_{u,a,i,k-1,2} & \cdots & w_{u,a,i,k-1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{u,a,i,1,1} & w_{u,a,i,1,2} & \cdots & w_{u,a,i,1,n} \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}$$

While we initially had the powers of Y be from 1 to $N + U$ such that each constraint had a different power, since we will prove the multiplication constraints and the linear constraints separately it makes more sense to instead let the powers be from 1 to N and from 1 to U respectively. Thus we now define

$$W_{a,i} = \sum_{u=1}^{U} M_Y^u W_{u,a,i}$$

Now let the relations represented by $A_i, W_{a,i}, B_i, W_{b,i}, C_i, W_{c,i}$ be denoted as $K_{i,j}$ for $i \in [1, 2, ..., n]$ and $j \in [1, 2, ..., k]$. As a result of our basis $\beta = 1, \phi, ..., \phi^{2k-1}$, we get

$$\sum_{i=1}^{m} (A_i \odot W_{a,i} + B_i \odot W_{b,i} + C_i \odot W_{c,i}) = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \sum_{u=1}^{U} K_u \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \begin{pmatrix} t_1 \\ t_2 \\ \vdots \\ t_{k-1} \\ t_k \\ t_{k+1} \\ \vdots \\ t_{2k} \end{pmatrix}$$

where t_i are arbitrary coefficients depending on $A_i, W_{a,i}, B_i, W_{b,i}, C_i, W_{c,i}$. Notice that all the linear constraints are sent to the k-th term of the output vector. We will now use an indeterminate X to separate these from the rest of the arbitrary coefficients in that term by sending the constraints to the X^{m+1}-term, and the other coefficients to other terms. In that case, we will have the constraints separated from each other as a result of them being different powers of Y, and separated from the rest of the arbitrary coefficients as a result of the other coefficients either not being the k-th term in the field representation, or having a different power of X. Additionally, we add A_0, B_0, C_0 similar to how we added A_0, B_{m+1} to the multiplication constraint scheme to avoid getting the same output every time if the verifier chooses the same values every time, and to avoid leaking on the secret by making its distribution similar to that of the discrete. Keeping all this in mind, we define

$$A(X) = A_0 + \sum_{i=1}^{m} M_X^i A_i \qquad W_a(X) = \sum_{i=1}^{m} M_X^{m+1-i} W_{a,i}$$

$$B(X) = B_0 + \sum_{i=1}^{m} M_X^i B_i \qquad W_b(X) = \sum_{i=1}^{m} M_X^{m+1-i} W_{b,i}$$

$$C(X) = A_0 + \sum_{i=1}^{m} M_X^i C_i \qquad W_c(X) = \sum_{i=1}^{m} M_X^{m+1-i} W_{c,i}$$

Then we get

$$\begin{aligned} &A(X) \odot W_a(X) + B(X) \odot W_b(X) + C(X) \odot W_c(X) \\ &= M_X^{m+1} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \sum_{u=1}^{U} M_Y^u K_u \\ 0 \\ \vdots \\ 0 \end{pmatrix} + M_X^{m+1} \begin{pmatrix} v_1 \\ \vdots \\ v_{k-1} \\ 0 \\ v_{k+1} \\ \vdots \\ v_{2k} \end{pmatrix} + \sum_{i=0, i \neq m+1}^{2m} M_X^i H_i \end{aligned} \tag{4.12}$$

Where v_i are arbitrary vectors and H_i are arbitrary matrices with columns representing elements in $GF(p^{2k})$ depending on the above functions.

Using equation 4.11 and equation 4.12 we can create a argument of knowledge for the arithmetic circuit.

Chapter 5

Argument of knowledge for the arithmetic circuit represented by polynomials

A note on the commitment scheme

In chapter 4 assumed that all calculations take place in $\mathbb{Z}_p$ (where field operations have been calculated using vectors with elements in $\mathbb{Z}_p$). The commitment scheme we will be using will map elements from $\mathbb{Z}_p$ to elements in $\mathbb{Z}_q$, where $q >> p$. The construction is very similar to the one presented in chapter 3, and the security is a result of the difficulty of finding short integer solutions in a lattice. As a result of the commitment scheme mapping elements from $\mathbb{Z}_p$ to $\mathbb{Z}_q$, we have to be careful such that the prover and the verifier ends up with the same values in their calculations. The prover and verifier will reduce terms $\bmod p$ multiple times during the computations, but not at the same times. This means that we end up with values that are congruent $\bmod p$, but not necessarily equal in $\mathbb{Z}_q$. Since the prover has access to all the information that the prover has, he can do the calculations that the verifier will do to calculate this discrepancy D, and send it to the verifier.

In the schemes, we need commitments to matrices. These will be computed by creating commitments to each row of the matrix individually, such that a $2k \times n$ matrix has $2k$ commitments to the rows of the matrix. We define

$$CK(M, R) = \begin{bmatrix} CK(m_1, r_1) \\ CK(m_2, r_2) \\ \vdots \\ CK(m_{2k}, r_{2k}) \end{bmatrix}$$

as the commitment to the matrix $M \in \mathbb{Z}_p^{2k \times n}$ with randomness $R \in \mathbb{Z}_p^{2k \times 2r \log_p(q)}$. Note that the verifier can access a commitment to a subset of the rows of the matrix by choosing a subset of the rows in the commitment matrix. This will be useful because the

matrix C is equal in the first k rows both for the multiplication and linear constraints, but the lower half is all 0's in the linear constraints.

Now let $\boldsymbol{S}_i$ denote the columns of $CK(M,R)$. By the homomorphic properties of the commitment scheme, we have that for an element $x \in GF(p^{2k})$

$$M_x CK(M,R) = [M_x\boldsymbol{S}_1||M_x\boldsymbol{S}_2||...||M_x\boldsymbol{S}_n] = CK(x*M, x*R) = CK(M_xM, M_xR) \tag{5.1}$$

where M_x is the matrix describing field multiplication by x as mentioned in chapter 4. This is true because the commitment scheme applies a linear transformation on the rows of its input. Importantly, it applies the same linear transformation on each row of the input. Similarly, $M_{\boldsymbol{x}}$ applies the same linear transformation on each column of the matrix we multiply it by. Since the same operations are performed on each row by the commitment scheme, and on each column by $M_{\boldsymbol{x}}$, we can do them in either order, and as a result we have that equation 5.1 holds.

Argument of knowledge for multiplication constraints

The following section follows the material presented in [2]. The prover knows A, B, C such that $A = (A_1, A_2, ..., A_m)$, and A_i are the matrices described in chapter 4.2, and the same holds for B, C. Let $\underline{A}_i, \underline{B}_i, \underline{C}_i$ denote the first k of these matrices. Then we have that

$$\underline{A_i} \circ \underline{B_i} = \underline{C}_i$$

and

$$\begin{bmatrix} \underline{A}_i \\ \mathbf{0}^{k\times n} \end{bmatrix} \circledast \begin{bmatrix} \underline{B}_i \\ \mathbf{0}^{k\times n} \end{bmatrix} = A_i \circledast B_i = \begin{bmatrix} \underline{C}_i \\ C'_i \end{bmatrix} = C_i$$

Initially, we need to create some randomness that we will later use in the rejection sampling, to ensure the outputs are indistinguishable from ones from a discrete Gaussian. In [2] they use the following standard deviations, and success rates of the rejection sampling:

$$\sigma_1 = 48\sqrt{kn}kmp^2 \qquad \sigma_2 = 72\sqrt{2kn}kmp$$
$$\sigma_3 = 24\sqrt{2kn}kp(1+6kmp) \qquad \sigma_4 = 24\sqrt{2}k^2pn\sigma_2$$
$$\rho = e$$

Now we generate A_0 and B_{m+1} at random from $G_{\sigma_1}^{2k\times n}$. Our commitment scheme uses a message in $\mathbb{Z}_p^n$ and randomness in $\mathbb{Z}_p^{2r\log_p(q)}$. Let $n' = 2r\log_p(q)$. Then, the prover generates random matrices $\alpha'_i, \beta'_i \in \mathbb{Z}_p^{k\times n}$ for $i \in [1, 2, ..., m]$. Then, let

$$\alpha_i = \begin{bmatrix} \alpha'_i \\ \mathbf{0}^{k\times n} \end{bmatrix} \qquad \beta_i = \begin{bmatrix} \beta'_i \\ \mathbf{0}^{k\times n} \end{bmatrix}$$

We also generate random matrices $\gamma_i \in \mathbb{Z}_p^{2k\times n}$, as well as $\alpha_0, \beta_{m+1} \in G_{\sigma_1}^{2k\times n'}$.

The prover then calculates

$$\begin{aligned} \boldsymbol{A}_i &= CK(A_i, \alpha_i) & \boldsymbol{A}_0 &= CK(A_0, \alpha_0) \\ \boldsymbol{B}_i &= CK(B_i, \beta_i) & \boldsymbol{B}_{m+1} &= CK(B_{m+1}, \beta_{m+1}) \\ \boldsymbol{C}_i &= CK(C_i, \gamma_i) \end{aligned}$$

for $i \in [0, 1, ..., k]$. The prover then sends $\boldsymbol{A}_i, \boldsymbol{B}_i, \boldsymbol{C}_i, \boldsymbol{A}_0, \boldsymbol{B}_{m+1}$ to the verifier.

We will be using $\boldsymbol{A}_i, \boldsymbol{B}_i$ for $i \in [1, 2, ..., 2k]$ and $\boldsymbol{C}_i$ for $i \in [1, 2, ..., k]$ for both the linear constraints and the multiplication constraints, but we will do each set of constraints as a separate argument. For the linear constraints we will also generate new $\boldsymbol{A}_0, \boldsymbol{B}_0, \boldsymbol{C}_0$ and trivial commitments for C_i when $i \in [k+1, k+2, ..., 2k]$. This will be explained in the section detailing the argument for linear constraints.

The verifier now chooses $\boldsymbol{y} \in GF(p^{2k})$ and sends $\boldsymbol{y}$ to the prover.

The prover then calculates $A(X), B(X), \hat{C}$ by using the $\boldsymbol{y}$ that the prover received from the verifier as follows

$$\begin{aligned} A(X) &= A_0 + \sum_{i=1}^{m} M_X^i (M_{\boldsymbol{y}}^i \bmod p) A_i \\ B(X) &= B_{m+1} + \sum_{i=1}^{m} M_X^{m+1-i} B_i \\ \hat{C} &= \sum_{i=1}^{m} M_{\boldsymbol{y}}^i C_i \pmod p \end{aligned}$$

The prover then computes

$$A(X) \circledast B(X) \pmod p = M_X^{m+1}\hat{C} + \sum_{i=0, i\neq m+1}^{2m} (M_X)^i H_i \pmod p \tag{5.2}$$

Having calculated the H_i, he chooses random $\eta_i \in \mathbb{Z}_p^{2k\times n'}$ and computes

$$\boldsymbol{H}_i = CK(H_i, \eta_i)$$

for $i \in [1, 2, ..., 2m], i \neq m+1$ and sends these to the verifier.

The verifier now chooses $\boldsymbol{x} \in GF(p^{2k})$ and sends $\boldsymbol{x}$ to the prover.

The prover then computes

$$\begin{aligned} \hat{A} &= A_0 + \sum_{i=1}^{m} (M_{\boldsymbol{x}}^i M_{\boldsymbol{y}}^i \bmod p) A_i & \hat{B} &= B_{m+1} + \sum_{i=1}^{m} (M_{\boldsymbol{x}}^{m+1-i} \bmod p) B_i \\ \hat{\alpha} &= \alpha_0 + \sum_{i=1}^{m} (M_{\boldsymbol{x}}^i M_{\boldsymbol{y}}^i \bmod p) \alpha_i & \hat{\beta} &= \beta_{m+1} + \sum_{i=1}^{m} (M_{\boldsymbol{x}}^{m+1-i} \bmod p) \beta_i \end{aligned}$$

Having now calculated these values, the verifier can calculate the discrepancy between the numbers that are equal $\pmod p$, but not equal in $\mathbb{Z}_q$ by calculating

$$D = (\hat{A} \circledast \hat{B} \bmod p) - \sum_{i=1}^{m} (M_{\boldsymbol{y}}^{i} M_{\boldsymbol{x}}^{m+1} \bmod p) C_i - \sum_{i=1, i \neq m+1}^{2m} (M_{\boldsymbol{x}}^{i} \bmod p) H_i \quad (5.3)$$

We have now calculated the discrepancy D. Note that we have $D \equiv 0 \bmod p$, as it was calculated using values that were equal modulo p, but not equal in $\mathbb{Z}_q$. However, we do not wish to give the verifier this value directly. Instead we will create an argument of knowledge for this value that the verifier can use when verifying that our argument holds.

To do this, the prover generates $\delta \in G_{\sigma_2}^{2k \times n'}$, $E \in p \cdot G_{\sigma_3}^{2k \times n}$ and $\epsilon \in G_{\sigma_4}^{2k \times n'}$. Note by the definition of E that it is $0 \bmod p$. The prover now computes $\boldsymbol{D} = CK(D, \delta)$ and $\boldsymbol{E} = CK(E, \epsilon)$ and sends $\boldsymbol{D}, \boldsymbol{E}$ to the verifier.

The verifier now picks $\boldsymbol{z} \in GF(p^{2k})$ and sends it to the prover.

Let $Z = (\hat{A}||\hat{\alpha}||\hat{B}||\hat{\beta})$ and $Y = (A_0||\alpha_0||B_{m+1}||\beta_{m+1})$. Using the rejection sampling algorithm described in equation 2.2, the prover runs $Rej(Z, Z - Y, \sigma_1, e)$, and aborts or proceeds depending on the result of the rejection sampling.

When using the commitment scheme we need a message and some "randomness" as the second part of the vector which is multiplied by the matrix defining the specific instance of our commitment scheme. We generated randomness which we used in our commitments to A, B, C. Since we have calculated the coefficients in the right hand side in equation 5.2, we can now use these to calculate the secondary input to the commitment scheme, using the initial values γ, η, δ we used to commit to C, H, D.

$$\tau = \sum_{i=1}^{m} (M_{\boldsymbol{x}}^{m+1} M_{\boldsymbol{y}}^{i} \bmod p)\gamma_i + \sum_{i=0, i \neq m+1}^{2m} (M_{\boldsymbol{x}}^{i} \bmod p)\eta_i + \delta$$

The prover then performs the following calculations and rejection sampling:

$$\begin{aligned} Rej(\tau, \tau - \delta, \sigma_2, e) \\ \hat{D} = (M_{\boldsymbol{z}} \bmod p) D + E \\ \hat{\delta} = (M_{\boldsymbol{z}} \bmod p)\delta + \epsilon \\ Rej(\hat{D}/p, D/p, \sigma_3, e) Rej(\hat{\delta}, \delta, \sigma_4, e) \end{aligned}$$

Then, if none of the rejection sampling causes the scheme to abort, the prover sends $\hat{A}, \hat{\alpha}, \hat{B}, \hat{\beta}, \tau, \hat{D}, \hat{\delta}$ to the verifier.

Finally, the verifier checks that

$$CK(\hat{A}, \hat{\alpha}) = \sum_{i=0}^{m} (M_{\boldsymbol{x}}^{i} M_{\boldsymbol{y}}^{i} \bmod p) \boldsymbol{A}_i$$

$$CK(\hat{B}, \hat{\beta}) = \sum_{i=1}^{m+1} (M_{\boldsymbol{x}}^{m+1-i} \bmod p) \boldsymbol{B}_i$$

$$CK(\hat{A} \circledast \hat{B} \bmod p, \tau) = \sum_{i=1}^{m} (M_{\boldsymbol{x}}^{m+1} M_{\boldsymbol{y}}^{i} \bmod p)\boldsymbol{C}_i + \sum_{i=0, i \neq m+1}^{2m} (M_{\boldsymbol{x}}^{i} \bmod p)\boldsymbol{H}_i + \boldsymbol{D}$$

$$CK(\hat{D}, \hat{\delta}) = (M_z \bmod p)\boldsymbol{D} + \boldsymbol{E}$$

$$\hat{D} = (M_z \bmod p)D + E \equiv 0 \pmod{p}$$

hold. Careful inspection would reveal that these all hold for an honest prover, using equation 5.1 and the fact that both D and E are $0 \bmod p$. In [2] they also bounded the values in G_σ as a function of σ. The verifier will then also check that the norms of various matrices satisfy the bounds for the elements in G_σ, which is written in more detail in the cited paper.

Now, having proved the multiplication relations, we will use the initial commits to A, B, C to also prove the linear relations. By using the same commits we ensure that both types of relations have to hold at the same time to convince the verifier (as it is obviously trivial to construct only the multiplication relations with no linear relations or vice versa).

Linear Constraints

We wish to prove that the A, B, C we initially commited to satisfy the constraints

$$\sum_{i=1,j=1}^{m,k} \boldsymbol{a}_{i,j}\boldsymbol{w}_{u,a,i,j} + \sum_{i=1,j=1}^{m,k} \boldsymbol{b}_{i,j}\boldsymbol{w}_{u,b,i,j} + \sum_{i=1,j=1}^{m,k} \boldsymbol{c}_{i,j}\boldsymbol{w}_{u,c,i,j} - K_u = 0 \tag{5.4}$$

for $u \in [1, 2, ..., U]$, very similarly to equation 4.3. The prover has already sent the openings for A, B, C, but since we don't use the C'_i in this scheme, these will be replaced by trivial commitments to 0, such that the matrix C has the same structure as A and B. So we replace

$$\boldsymbol{C}_i = \begin{bmatrix} CK(\boldsymbol{c}_{i,1}, \gamma_{i,1}) \\ CK(\boldsymbol{c}_{i,2}, \gamma_{i,2}) \\ \vdots \\ CK(\boldsymbol{c}_{i,k}, \gamma_{i,k}) \\ CK(\boldsymbol{c}'_{i,1}, \gamma_{i,k+1}) \\ \vdots \\ CK(\boldsymbol{c}'_{i,k}, \gamma_{i,2k}) \end{bmatrix} \quad \text{with} \quad \boldsymbol{C}_i = \begin{bmatrix} CK(\boldsymbol{c}_{i,1}, \gamma_{i,1}) \\ CK(\boldsymbol{c}_{i,2}, \gamma_{i,2}) \\ \vdots \\ CK(\boldsymbol{c}_{i,k}, \gamma_{i,k}) \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

This is easily accomplished by keeping the commitments to the first k rows of the matrix $\boldsymbol{C}_i$, and replacing the other commitments with trivial commitments to 0.

Additionally, we will generate new randomness $A_0, B_0, C_0 \in G_{\sigma_1}^{2k \times n}$ and $\alpha_0, \beta_0, \gamma_0 \in G_{\sigma_1}^{2k \times n'}$. This ensures that each argument will be different, even if the verifier were to choose the same input values to our polynomials.

The verifier then chooses $\boldsymbol{y} \in GF(p^{2k})$ and sends it to the prover.

Using the constructions from equation 4.9, we calculate the following matrices.

$$W_{a,i} = \sum_{u=1}^{U} (M_{\boldsymbol{y}}^u \bmod p) W_{u,a,i}$$
$$W_{b,i} = \sum_{u=1}^{U} (M_{\boldsymbol{y}}^u \bmod p) W_{u,b,i}$$
$$W_{c,i} = \sum_{u=1}^{U} (M_{\boldsymbol{y}}^u \bmod p) W_{u,c,i}$$

Then we calculate the following polynomials in the indeterminate X

$$A(X) = A_0 + \sum_{i=1}^{m} M_X^i A_i \qquad W_a(X) = \sum_{i=1}^{m} M_X^{m+1-i} W_{a,i}$$
$$B(X) = B_0 + \sum_{i=1}^{m} M_X^i B_i \qquad W_b(X) = \sum_{i=1}^{m} M_X^{m+1-i} W_{b,i}$$
$$C(X) = A_0 + \sum_{i=1}^{m} M_X^i C_i \qquad W_c(X) = \sum_{i=1}^{m} M_X^{m+1-i} W_{c,i}$$

The prover now calculates the arbitrary coefficients t_i and the arbitrary vectors $\boldsymbol{h}_i$, as we described in equation 4.12.

$$A(X) \odot W_a(X) + B(X) \odot W_b(X) + C(X) \odot W_c(X)$$
$$= M_X^{m+1} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \sum_{u=1}^{U} M_{\boldsymbol{y}}^u K_u \\ 0 \\ \vdots \\ 0 \end{pmatrix} + M_X^{m+1} \begin{pmatrix} \boldsymbol{v}_1 \\ \vdots \\ \boldsymbol{v}_{k-1} \\ 0 \\ \boldsymbol{v}_{k+1} \\ \vdots \\ \boldsymbol{v}_{2k} \end{pmatrix} + \sum_{i=0, i \neq m+1}^{2m} M_X^i H_i$$

We will now create commitments to v_i and H_i. The prover chooses η_i at random from $\mathbb{Z}_p^{2k \times n'}$ for $i \in [1, 2, ..., 2m], i \neq m+1$ and ν_i at random from $\mathbb{Z}_p^n$ for $i \in [1, 2, ..., 2k], i \neq k$. The prover then creates the matrices

$$\nu = \begin{bmatrix} \nu_1 \\ \vdots \\ \nu_{k-1} \\ 0 \\ \nu_{k+1} \\ \vdots \\ \nu_{2k} \end{bmatrix} \qquad V = \begin{bmatrix} \boldsymbol{v}_1 \\ \vdots \\ \boldsymbol{v}_{k-1} \\ 0 \\ \boldsymbol{v}_{k+1} \\ \vdots \\ \boldsymbol{v}_{2k} \end{bmatrix}$$

Now the prover calculates $\boldsymbol{H}_i = CK(H_i, \eta_i)$ for $i \in [0, 1, ..., 2m], i \neq m+1$ and $\boldsymbol{V} = CK(V, \nu)$. The prover then sends the $\boldsymbol{H}_i$ and $\boldsymbol{V}$ to the verifier.

The verifier then chooses $\boldsymbol{x} \in GF(p^{2k})$ and sends it to the prover.

The prover then calculates the following values, using $\boldsymbol{x}$

$$\hat{A} = A_0 + \sum_{i=0}^{m} (M_{\boldsymbol{x}}^i \bmod p) A_i \qquad \hat{\alpha} = \alpha_0 + \sum_{i=0}^{m} (M_{\boldsymbol{x}}^i \bmod p) \alpha_i$$

$$\hat{B} = B_0 + \sum_{i=0}^{m} (M_{\boldsymbol{x}}^i \bmod p) B_i \qquad \hat{\beta} = \beta_0 + \sum_{i=0}^{m} (M_{\boldsymbol{x}}^i \bmod p) \beta_i$$

$$\hat{C} = C_0 + \sum_{i=0}^{m} (M_{\boldsymbol{x}}^i \bmod p) C_i \qquad \hat{\gamma} = \gamma_0 + \sum_{i=0}^{m} (M_{\boldsymbol{x}}^i \bmod p) \gamma_i$$

$$W_a = \sum_{i=1}^{m} (M_{\boldsymbol{x}}^{m+1-i} \bmod p) W_{a,i} \qquad W_b = \sum_{i=1}^{m} (M_{\boldsymbol{x}}^{m+1-i} \bmod p) W_{b,i}$$

$$W_c = \sum_{i=1}^{m} (M_{\boldsymbol{x}}^{m+1-i} \bmod p) W_{c,i}$$

Now the prover can calculate the discrepancy D between the prover and verifier as a multiple of p.

$$D = (\hat{A} \odot W_a + \hat{B} \odot W_b + \hat{C} \odot W_c \bmod p) - M_x^{m+1} \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \sum_{u=1}^{U} M_{\boldsymbol{y}}^u K_u \\ 0 \\ \vdots \\ 0 \end{pmatrix} \bmod p - (M_{\boldsymbol{x}}^{m+1} \bmod p) \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_{k-1} \\ v_k \\ v_{k+1} \\ \vdots \\ v_{2k} \end{pmatrix} - \sum_{i=0, i \neq m+1}^{2m} (M_{\boldsymbol{x}}^i \bmod p) \boldsymbol{h}_i$$

The prover generates $\delta \in G_{\sigma_2}^{2k \times n'}$, $E \in p \cdot G_{\sigma_3}^{2k \times n}$ and $\epsilon \in G_{\sigma_4}^{2k \times n'}$. The prover now computes $\boldsymbol{D} = CK(D, \delta)$ and $\boldsymbol{E} = CK(E, \epsilon)$ and sends $\boldsymbol{D}, \boldsymbol{E}$ to the verifier.

The prover also calculates the value

$$\tau = (M_{\boldsymbol{x}}^{m+1} \bmod p)\nu + \sum_{i=0, i \neq m+1}^{2m} (M_{\boldsymbol{x}}^i \bmod p)\eta_i + \delta$$

The verifier chooses $\boldsymbol{z} \in GF(p^{2k})$ and sends it to the prover.

The prover now calculates

$$\hat{D} = (M_{\boldsymbol{z}} \bmod p) D + E \qquad \hat{\delta} = (M_{\boldsymbol{z}} \bmod p)\delta + \epsilon$$

Defining $Z = (\hat{A}||\hat{\alpha}||\hat{B}||\hat{\beta}||\hat{C}||\hat{\gamma})$ and $(\hat{A}_0||\hat{\alpha}_0||\hat{B}_0||\hat{\beta}_0||\hat{C}_0||\hat{\gamma}_0)$, the prover then runs

$$\begin{aligned} Rej(Z, Z - Y, \sigma_1, e) \\ Rej(\hat{\delta}, \delta, \sigma_2, e) \\ Rej(\hat{D}/p, D/p, \sigma_3, e) \end{aligned}$$

and aborts or proceeds according to the results of these rejection samplings. If the prover does not abort, he finally sends $\hat{A}, \hat{\alpha}, \hat{B}, \hat{\beta}, \hat{C}, \hat{\gamma}, \tau, \hat{D}, \hat{\delta}$ to the verifier.

The verifier now checks that

$$\begin{aligned} CK(\hat{A}) &= \sum_{i=0}^{m} (M_{\boldsymbol{x}}^i \bmod p)\boldsymbol{A}_i \\ CK(\hat{B}) &= \sum_{i=0}^{m} (M_{\boldsymbol{x}}^i \bmod p)\boldsymbol{B}_i \\ CK(\hat{C}) &= \sum_{i=0}^{m} (M_{\boldsymbol{x}}^i \bmod p)\boldsymbol{C}_i \end{aligned}$$

$$CK((\hat{A} \cdot W_a + \hat{B} \cdot W_b + \hat{C} \cdot W_c) \bmod p, \tau)$$
$$= \sum_{u=1}^{U} M_{\boldsymbol{x}}^{m+1} M_{\boldsymbol{y}}^{u} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ CK(K_u, \boldsymbol{0}) \\ 0 \\ \vdots \\ 0 \end{pmatrix} \bmod p + (M_{\boldsymbol{x}}^{m+1} \bmod p)\boldsymbol{V} + \sum_{i=0, i \neq m+1}^{2m} (M_{\boldsymbol{x}}^i \bmod p)\boldsymbol{H}_i + \boldsymbol{D}$$

$$CK(\hat{D}, \hat{\delta}) = T_1\hat{D} + T_2\hat{\delta}] = (M_{\boldsymbol{z}} \bmod p)\boldsymbol{D} + \boldsymbol{E}$$

$$\hat{D} = (M_{\boldsymbol{x}} \bmod p)D + E \equiv 0 \pmod{p}$$

And again, the verifier should also check that the norms of the matrices satisfy the bounds imposed on them by σ_i, simiarly to in the multiplication constraint scheme. If all of this holds, the verifier will be convinced, and the argument of knowledge is complete.

Summary

Having performed both of these schemes, we have successfully created an argument of knowledge for the satisfaction of an arithmetic circuit. We note that the amount of communication is similar in both schemes, so the total asymptotic complexity of both schemes should be the same as the complexity of the multiplication scheme alone. The commitment scheme takes a message in $\mathbb{Z}_p^n$ and outputs an element in $\mathbb{Z}_q^r$. In [2] they used

$r = O(\log n)$. By using carefully chosen values of n, m, k, they were able to obtain communication costs $O(\sqrt{N \log N})$ elements in $\mathbb{Z}_q$, with computational cost $O(N \log N)$ and $O(N)$ for respectively the prover and verifier.

We also note that we use the rejection sampling algorithm 4 times for the multiplication constraints and 3 times for the linear constraints. Since we can choose new random values A_0, B_{m+1} or A_0, B_0, C_0, a rejection in the linear constraints doesn't mean we need to do the multiplication constraints again. It does however mean that we expect roughly $e^4 + e^3 \approx 75$ aborts before the scheme finally succeeds. By using techniques similar to the ones we did in chapter 3.2, where we created a non-interactive signature scheme from a signature scheme by using the Fiat-Shamir transform, we could also transform the above scheme into a non-interactive one. We note that the only interaction from the verifier is that the verifier provides the values $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \in GF(p^{2k})$. By instead computing these using a hash function H, we could perform the same scheme without any interaction, and only transmit values in the case where we succeed.

Chapter 6

Concluding remarks

We started by studying how lattice-based cryptography works in general, and then used this in zero-knowledge signature and identification schemes. These turned out to be quite simple to understand. Then we wanted to study how to create an argument of knowledge for an arithmetic circuit. This turned out to be quite a bit more complicated, requiring a lot more work to detail what exactly is going on. The argument of knowledge for arithmetic circuits manages to convince a verifier of knowledge of inputs to N multiplication gates with communication costs $\sqrt{N \log N}$. The reason this is possible is that we can create a SIS problem instance where it is difficult to find a small $\boldsymbol{s} \in \mathbb{Z}_p^n$ satisfying $A\boldsymbol{s} = \boldsymbol{S}$ for $\boldsymbol{S} \in \mathbb{Z}_q^r$. According to [7], to solve a SIS problem instance when n is very large, the best algorithm will solve only submatrix of A, by setting some of the elements in $\boldsymbol{s}$ to 0. This means that we can choose $n >> r$, allowing us to transmit commitments to n elements in $\mathbb{Z}_p$ in only $r \log q$ bits. Thus the usage of lattice-based cryptography grants us access to this compression that we wouldn't have otherwise, and allows the communication cost to be sub-linear in the number of multiplication gates in the circuit.

Future work

We have managed to learn a lot about the workings of lattice-based cryptography in this thesis, but there is still more that could have be done. There was a lot of material to cover, and in the end there wasn't enough time to cover it all. The most obvious next step is to include detailed proofs of completeness and soundness of the arithmetic circuit argument. Additionally, there are multiple places where we have referred to proofs from earlier works, where given more time we would want to include these proofs in this thesis.

Bibliography

[1] M. Ajtai. Generating hard instances of lattice problems. *Electronic Colloquium on Computational Complexity (ECCC)*, 3, 1996.

[2] C. Baum, J. Bootle, A. Cerulli, R. Pino, J. Groth, and V. Lyubashevsky. Sub-linear lattice-based zero-knowledge arguments for arithmetic circuits. 01 2018.

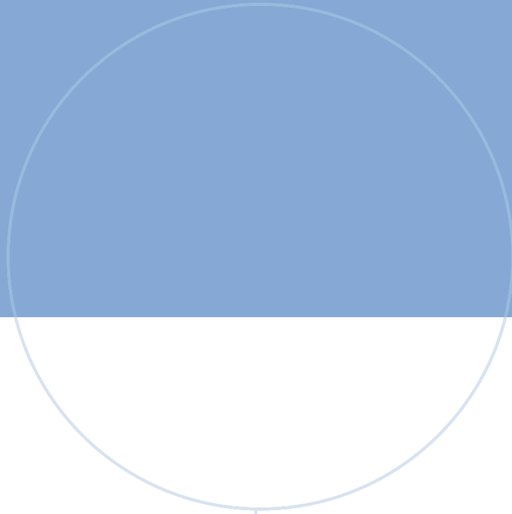
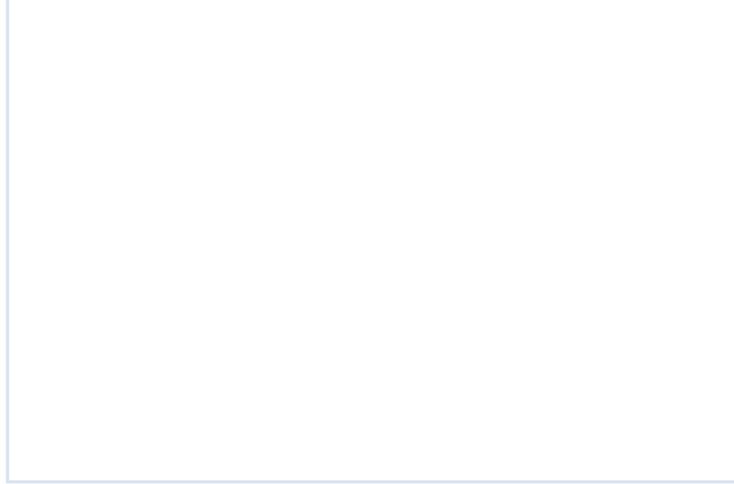
[3] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. pages 327–357, 05 2016.

[4] V. Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures, 2009.

[5] V. Lyubashevsky. Lattice signatures without trapdoors. In *Proceedings of the 31st Annual International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'12, page 738–755, Berlin, Heidelberg, 2012. Springer-Verlag.

[6] V. Lyubashevsky and D. Micciancio. Generalized compact knapsacks are collision resistant, 2006.

[7] D. Micciancio and O. Regev. *Lattice-based cryptography*. Springer, 2009.

NTNU
Norwegian University of
Science and Technology