Threshold Wallets

Martin Runne Due Jensen, 201709208
Michael Duy Nguyen, 201707968

Master Thesis (30 ECTS) in Computer Science

Advisor: Claudio Orlandi & Peter Scholl

Department of Computer Science, Aarhus University

December 12, 2023

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

A threshold wallet is the idea of taking a signature scheme used in blockchain transactions and distributing the signing rights between machines. In this thesis we study the Monero whitepaper called Cryptonote and try to implement it in the context of threshold wallets.

In detail we start off by describing the results in the whitepaper in detail whereas after we build up theory to describe a theoretical tool called RingCT which Monero uses to hide the amount in the transactions. After this we go through some relevant theory of SPDZ, specifically we study the paper ”Breaking the SPDZ Limits” where after we present a MPC version of the ring signature scheme from the whitepaper and our implementation of it using the MP-SPDZ library.

Martin Runne Due Jensen, and Michael Duy Nguyen
Aarhus, December 12, 2023.

Contents

1 Acknowledgement

First and foremost, we would like to thank our advisor Peter Scholl for his guidance and support throughout or thesis. Both in terms of direction of what to include in the project as well as his help in understanding theoretical parts during our weekly scheduled meetings.

We would also like to thank Marcel Keller for taking his time and answering our questions concerning the MPC library that we used for our implementation.

2 Introduction

These days the concept of cryptocurrencies has become really popular. Currently there is an idea called threshold wallets which is an application of MPC. A threshold wallet is two- or multiparty computation where there is a secret key for a signature scheme, that is secret shared among parties in the protocol. One key requirement of the secret sharing is that even if the adversary can steal one share of your key, then they won't be able to steal any of your money.

Throughout this project we dive into the Cryptonote paper, which the cryptocurrency Monero is based off. Cryptonote uses Ed25519 as the base signature algorithm and is based on the elliptic curve discrete logarithm problem.

We start off by going in depth on how a transaction is made in the system and how the One-Time Ring signature scheme works. By combining these two we get a standard cryptoNote transaction which is both untraceable and unlinkable, meaning that for each incoming transaction all possible senders are equiprobable and that for any two outgoing transactions it is impossible to prove they were sent to the same person, thus achieving a new level of privacy.

In order to do MPC over the ring signature scheme, there was one key challenge which was to hide the senders index, as each party have to do computation with it. To solve this we use secure comparison protocols, to create shares of the sender's index and slightly modify the equations where the senders index is used.

We will also be looking into the BulletProof framework which are short non-interactive zero-knowledge proofs which require no trusted setup. This framework is used when constructing Ring confidential transactions (RingCT), which is what Monero uses to hide the amount of a transaction. We will briefly describe how bulletproofs and the RingCT in Monero works, and compare it with newer proposed RingCT called Omniring.

For the implementation we have implemented the threshold wallet in c++ version 11+ using the library called "MP-SPDZ: Versatile framework for multiparty computation". The library supports various kind of MPC, and we have implemented the threshold wallets for some of them and done benchmark to see how they perform.

For limitations when looking at all the material we had available there really is not anything there. However, since we just wanted to realise the ideas in the implementation we performed limited testing on it, therefore there is a risk of errors that might arise. When doing the benchmarks we only managed to do them locally except one protocol, since at the time we managed to implement the majority of them, it was rather close to the due date, and we couldn't access the only network that allowed us to do online benchmarking due to logistics.

3 Background

Monero is one the cryptocurrency that has emerged due to the techniques provided in the cryptonote paper. Monero is a fully anonymous cryptpcurrency which relies on ring confidential transactions (RingCT) and ring signature schemes to hide the transactions which will be described in the later sections. Monero is often compared with zcash which came out later, and also offers privacy, however unlike Monero, not all transactions are actually private (shielded). In the zcash network users are allowed to either send a normal or shielded transaction. Zcash uses zero knowledge scheme called zk-SNARKs, which theoretically is much stronger and secure than the protocols that Monero uses, as it doesn't revel any information about the transaction at all. However, in reality the majority of transcations in the Zcash network are normal transactions, since it is computationally expensive to create zk-SNARKs, thus most users do not enable them.

4 Notation

We note that the Cryptonote paper uses additive notation instead of multiplicative notation. When describing the Cryptonote section we will also refer to additive notation, whereas in bulletproofs and RingCT we will refer to multiplicative notation.

Bulletproof: When describing the bulletproof protocol we will describe vectors using bold text, e.g. $\mathbf{a} \in \mathbb{Z}_p^n$. We use $\circ$ to represent entry wise multiplication e.g. $\mathbf{a} \circ \mathbf{b} = (a_1 \cdot b_1, ..., a_n \cdot b_n)$, another operation is, let $\mathbf{a}, \mathbf{b}$ be vectors then $\mathbf{a}^{\mathbf{b}} = \prod_{i=1}^{n} \mathbf{a}_i^{\mathbf{b}_i}$. We use $\mathbf{a}^c$ to denote the first c powers of a. We will use the following notation to represent halving of vectors, e.g. let $f \in [1, .., n]$, then $\mathbf{a}_{[:f]} = (a_1, .., a_f)$ and $\mathbf{a}_{[f:]} = (a_{f+1}, .., a_n)$ for some vector $\mathbf{a}$ of n-dimensions. We use the notation $\langle \cdot, \cdot \rangle$ when computing inner-product.

MPC: In the MPC section we describe shares as $[a]$ for passive security (meaning no MAC keys) which should be thought of as a vector, $(a_0, .., a_{n-1})$ containing additive shares of a which sums to a i.e. $a = \sum_{i=0}^{n-1} a_i$. The number of parties participating in the protocol will always be denoted as n. When we write a set like $\{[a]\}^m$ we mean that we have a set of m shares of a values i.e. $\{[a]\}^m = \{[a_0], .., [a_{m-1}]\}$. These m, a values can be different. When referencing protocols from [3] and when writing the ring signature as a MPC protocol we will make use of the notation $\langle a \rangle$ which just means that the share has an associated MAC key sharing. The operation **Open()** is a public opening of a secret shared value, more specifically in the context of SPDZ it references partially openings as described in [3].

5 CryptoNote Paper

5.1 Definitions

Elliptic curve parameters Based on the Ed25519 signature scheme the curve parameters are listed on page 5 in [10] and we mention few:

1. The curve equation: $-x^2 + y^2 = 1 + dx^2y^2$, where $d = -\frac{121665}{121666} \in \mathbb{F}_q$ and $q = 2^{255} - 19$ is a prime number.

2. The base point $G = (x, -\frac{4}{5})$.

3. The security parameter l, which is a prime order of the base point:

$$l = 2^{252} + 27742317777372353535851937790883648493$$

4. Then we have two hash functions. One being deterministic, $\mathcal{H}_p$, and one being cryptographic $\mathcal{H}_s$:

$$\mathcal{H}_p : E(\mathbb{F}_q) \to E(\mathbb{F}_q)$$
$$\mathcal{H}_s : \{0,1\}^* \to \mathbb{F}_q$$

The CryptoNote paper uses some terms where we can chosen to include some of it:

1. Secret keys are standard elliptic curve secret keys: $a \in_R [1, l-1]$, where l is the security parameter from above.

2. Public keys are standard elliptic curve public keys: $A = aG$, where G is the base point of the elliptic curve.

3. Each party in the system has a its own **secret key pair** (a, b) of elliptic curve secret keys and a corresponding **public key pair** (A, B) and a **tracking key** (a, B) where $B = bG \wedge a \neq b$.

4. The **standard address** is a representation of a party's public key pair in a "human friendly format" with some error correction that each party will make available to all other parties.

5. The **destination key** is a one-time key that the sender generates. It is constructed such that only the intended receiver can recognise that the transaction is meant for him.

6. The **one-time secret key** is a key that the receiver of a transaction can compute from the destination key using his secret key pair (a, b)

5.2 Standalone transaction

We consider a standalone transaction from both the sender's and receiver's perspective. What we mean by standalone transaction is that it is only the transaction and doesn't incorporate the signature scheme. The transaction itself can be described as two part. The first part called the output contains the amount and the destination key. The second part contains is the transaction's public key. Let's name the sender Alice and the receiver Bob. Now the construction of a transaction works as follows:

Setup: Alice wants to send a transaction to Bob. Alice and Bob both has public and secret key pairs. Alice controls most of what is going on and Bob acts only as a listener who does some local computations on the transactions that passes by him. Alice's goal is to construct a transaction such that only the intended receiver, in this case Bob, can verify that it is meant for him.

Alice $\rightarrow$ Bob:

1. Alice unpacks Bob's public key (A, B) from Bob's standard address.
2. Choose $r \in_R [1, l-1]$ and compute a destination key $D = \mathcal{H}_s(rA)G + B$
3. Now Alice constructs her transaction. She computes the public key for the transaction $R = rG$. Combining the transaction's output (destination key, amount) and the transactions public key gives her the final transaction.
4. Alice sends her transaction.

Bob's point of view: Bob will examine each transaction that passes by him in the following way:

5. Bob retrieves the transaction's public key R and the transaction's destination key D. Now using his private key pair (a, b) he computes:

$$D' = \mathcal{H}_s(aR)G + B$$

6. To check if the transaction is meant for him Bob checks for equality:

$$D' \overset{?}{=} D$$

   If this equality holds then Bob knows that he is the intended receiver of the transaction and he would add it to his wallet where it would exist as what we call an unspent transaction.

When Bob has received a transaction he has the option of spending it. For this purpose he would compute a one-time secret key using the unspent transaction's destination key:

7. Bob computes the one-time secret key, x, by:

$$x = \mathcal{H}_s(aR) + b, \text{ where } (a, b) \text{ is Bob's secret key pair}$$

Later when this standard transaction is combined with the one-time ring signature scheme then Bob will create the signature under this one-time secret key x.

If $D' = D$ why is the transaction meant for Bob? This is because $\mathcal{H}_s(rA) = \mathcal{H}_s(aR)$ only in the case that the transaction's destination key is made from Bob's public key pair as otherwise there will be no relation between the Bob's secret key a and the public key A used to compute the destination key D in 2. This can be seen by noting $aR = arG = rA$ iff. $A = aG$.

5.3 One-Time Ring Signature Scheme

When sending a transaction it is required that the sender signs it. The protocol is based on one-time ring signatures. The one-time ring signature scheme consist of four algorithms $(\mathsf{GEN}, \mathsf{SIG}, \mathsf{VER}, \mathsf{LNK})$. For a one-time ring signature to be secure four properties need to established [10], which allows users to achieve unconditional unlinkability. . The properties are the following:

- **Linkability:** Given the secret keys secret keys $\{x_i\}_{i=0}^{m-1}$ for a set S of m public keys where $\forall i \; P_i = x_i G$. Using these two sets it is impossible to produce $m + 1$ valid signatures $\sigma_0 ... \sigma_m$ such that they all pass the LNK phase.

- **Exculpability:** Given a set of public keys S of size m, at most $m - 1$ corresponding private keys x_i as well as the key image I_j produced from the secret key x_j that is different from all known $m - 1$ secret keys. Then it is impossible to produce a valid signature using the key image I_j.

- **Unforgeability:** Given only the set of public keys S it is impossible to produce a valid signature

- **Anonymity:** Given a signature σ and the set of public keys $\mathcal{S}$ used to construct it. It is impossible to learn the index of the sender with probability greater than $\frac{1}{m}$.

which we will prove holds later in section 5.7.

GEN: The algorithm takes l which is the prime order of the base point G as input and outputs a key pair (P, x) and I. P is the public key and $x \in_R [1, l-1]$ is the one-secret key such that $P = xG$, and $I = x\mathcal{H}_p(P)$ is a key image.

SIG: The algorithm takes message msg, a set S' of one-time public keys $S' = \{P_i\}_{i \neq s}$, a pair (P_s, x_s) where s denotes the senders secret index, and the key image I_s and outputs a signature σ and set $S = S' \cup \{P_s\}$ so $|S| = m$ for some m. Specifically the algorithm generates a one-time ring signature by using a non-interactive zero-knowledge OR proof using the relation for equality of discrete logs in elliptic curves.

The idea is that sender wants to prove to others that he knows a witness w for the relation

$$R = \{(\alpha, w) | \alpha = (G, \mathcal{H}_p(P_i), P_i, I) \wedge P_i = wG, I = w\mathcal{H}_p(P_i)\} \tag{1}$$

over the set of one-time public keys S. The witness is the sender's one-time secret key x. Usually the common input is denoted as x but to reduce confusion we denote it as α.

Instead of doing interaction with a verifier the sender makes a non-interactive zero-knowledge proof (NIZK). The NIZK will take commitments from the sender return a challenge, $c = \mathcal{H}_s(msg, commitments)$ to the sender.

The way SIG implements the above NIZK proof is by: First two random sets $\{q_i | i = 0..m-1\}$ and $\{w_i | i = 0, ...s-1, s+1, .., m-1\}$ are sampled where $q_i, w_i \in [1, l]$ and applies the following transformation:

$$L_i = \begin{cases} q_i G & \text{if } i = s \\ q_i G + w_i P_i & \text{else} \end{cases}$$

$$R_i = \begin{cases} q_i \mathcal{H}(P_i) & \text{if } i = s \\ q_i \mathcal{H}(P_i) + w_i I & \text{else} \end{cases}$$

Which constructs m commitments where $m-1$ of them acts as the simulated commitments in an OR protocol. [4]. The sender then computes the non-interactive challenge:

$$c = \mathcal{H}_s(msg, L_0, .., L_{m-1}, R_0, .., R_{m-1})$$

As we use the OR-protocol [4] we require that the number of challenges corresponds to the number of commitments. So we split the challenge c into m challenges $c_0, .., c_{m-1}$. To do so we compute c_i as:

$$c_i = \begin{cases} c - \sum_{s \neq j, j=0}^{m-1} w_j \mod l & \text{if } i = s \\ w_i & \text{else} \end{cases}$$

Now lastly the sender computes responses to the challenges:

$$r_i = \begin{cases} q_s - c_s x \mod l & \text{if } i = s \\ q_i & \text{else} \end{cases}$$

Combining all these intermediate steps we get the final signature:

$$\sigma = (I, c_0, ..., c_{m-1}, r_0, ..., r_{m-1})$$

VER: This algorithm takes a message msg, set S and a signature σ and either accepts or rejects the signature. For $i = 0...m-1$ the verifier applies the inverse transformations:

$$L'_i = r_iG + c_iP_i$$
$$R'_i = r_i\mathcal{H}_p(P_i) + c_iI$$

and then checks if:

$$\sum_{i=0}^{m-1} c_i \overset{?}{=} \mathcal{H}_s(m, L'_0, ..., L'_{m-1}, R'_0, .., R'_{m-1}) \bmod l \tag{2}$$

If above check returns true then the verifier proceeds to run the LNK algorithm. If the check returns false then the verifier rejects the signature and outputs $\perp$.

LNK: The algorithm checks if I has been used in past signatures by checking if $I \in \mathcal{I}$, where $\mathcal{I}$ is a set that any party can build by retrieving the key images used in the previous transactions. Each key image is stored on the blockchain. If $i \in \mathcal{I}$ it rejects the signature otherwise it accepts.

5.4 Combination of Transaction and Signature

Now we will describe how to combine the one-time ring signature with the standard transaction. As a note we have included a visual representation of it on figure 1.

Setup: There is some amount of transaction outputs on the blockchain. Each consisting of an amount value and a destination key D. Now, say we have a party Alice who wants to spend an unspent transaction of hers. From the unspent transaction's destination key she can compute a one-time secret key x_s as described in 7. Since Alice wants to spend her transaction we modify GEN(l) into GEN(l, x) such that it can take a one-time secret key as a parameter instead of sampling it at random.

Sender preprocessing: As Alice wants to send a transaction to a third party she goes through the steps 1, 2, 3 in 5.2. By this, she has computed the destination key for the receiver, let us call it D_r, and the transaction's public key R. Furthermore, using her one-time secret key, x_s, she will execute the $Gen(l, x_s) \rightarrow I_s$, (P_s, x_s). Since x_s is the one-time secret key for D_r it follows that $P_s = D_r$ as

$$D_r = \mathcal{H}_s(rA)G + B$$
$$x = \mathcal{H}_s(aR) + b$$
$$P_s = xG = (\mathcal{H}_s(aR) + b)G = \mathcal{H}_s(rA)G + B = D_r$$

Then Alice randomly samples $m-1$ destination keys from the blockchain for some number m. These $m-1$ destination keys will be the set S' for SIG.

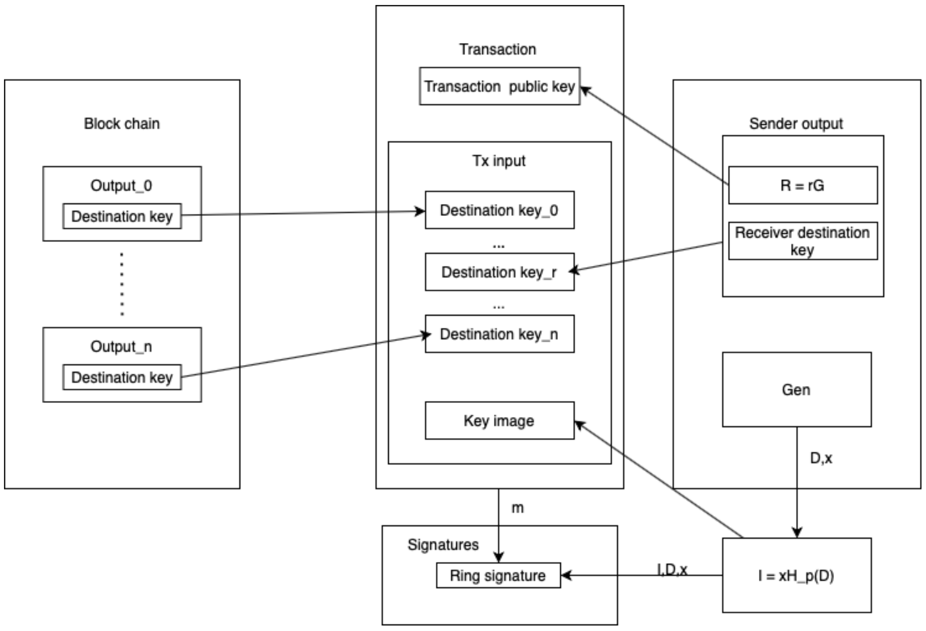

Figure 1: Standard transaction combined with ring signature

Transaction construction and signing: Now for Alice to construct the final transaction she combines the set of sampled destination keys S', the receiver's destination key D_r, the amount value from the transaction she is spending, the new transactions public key R and lastly the key image I_s. We will denote her constructed transaction as TX_{new}. To sign TX_{new} Alice will run the signature algorithm: $\mathsf{SIG}(TX_{new}, S', (P_s, x_s), I_s) \rightarrow (\sigma, S)$. Alice will then send off this transaction.

Receiver's perspective: Each potential receiver will now additionally run VER and LNK if they find that a transaction is intended for them by using the logic in 6

5.5 Showing that the signature algorithm is a NIZK proof

For this we have to show that the steps in SIG fulfils completeness, soundness and zero knowledge. We show for the case where the indexes matches the sender, $i = s$. I.e. we show that the system is NIZK in the case that the sender knows the secret key related to the public key. We show the system in the abstract case i.e:

1. The sender computes commitments L_s, R_s and sends them to the verifier.

2. The verifier computes a random challenge $c_s \in \mathbb{F}_q$ and sends it to the sender.

3. The sender computes a response $r_s = q_s - c_s x \mod l$

As the scheme non interactive zero knowledge the challenge the verifier sends is replaced with the sender computing $c = \mathcal{H}_s(msg, L_s, R_s)$.

Completeness A verifier that is checking the statement has a signature $\sigma = (I, c_s, r_s)$. To check the statement the verifier computes the inverse of the L, R transformation and checks (2). Therefore, for completeness it is enough to show that the inverse transformations hold.

$$\begin{aligned}
L'_s &= r_s G + c_s P_s \\
&= (q_s - c_s x)G + c_s x G \\
&= q_s G - c_s x G + c_s x G \\
&= q_s G = L_s \\
R'_s &= r_s \mathcal{H}_p(P_s) + c_s I \\
&= (q_s - c_s x)\mathcal{H}_p(P_s) + c_s x \mathcal{H}_p(P_s) \\
&= q_s \mathcal{H}_p(P_s) - c_s x \mathcal{H}_p(P_s) + c_s x \mathcal{H}_p(P_s) \\
&= q_s \mathcal{H}_p(P_s) = R_s
\end{aligned}$$

Soundness We assume that for any valid common input α in 1, that we have two valid transcripts of $(L_s, R_s, c_s, r_s), (L_s, R_s, \bar{c_s}, \bar{r_s})$, where $c_s \neq \bar{c_s}$ and the responses $r_s, \bar{r_s}$ can be different. We have to show that given these two transcripts it is possible to efficiently extract the witness $w = x$. As it will be shown for the L_s transformation in the exculpability proof 5.7 we only show for the R_s transformation here.

For the verifier to convince himself of the sender's statement it computes $R'_s = r\mathcal{H}_p(P_s) + c_s I$ and checks if this equals the transformation R_s. The verifier will compute one R'_s for each transcripts. Let's denote them as $R'_s, \bar{R'_s}$ which would be computed using $(L_s, R_s, c_s, r_s), (L_s, R_S, \bar{c_s}, \bar{r_s})$ respectively. Now by the soundness property we have that the values L_s, R_s are the same in both transcripts. In addition to this and the fact that both transcripts are accepting then it implies that $R'_s = \bar{R}'_s$. From this we are able to efficiently extract the witness $w = x$:

$$r_s\mathcal{H}_p(P_s) + c_sI = \bar{r_s}\mathcal{H}_p(P_s) + \bar{c_s}I$$

$$r_s\mathcal{H}_p(P_s) + c_sx\mathcal{H}_p(P_s) = \bar{r_s}\mathcal{H}_p(P_s) + \bar{c_s}x\mathcal{H}_p(P_s)$$

By taking log to the base $\mathcal{H}_p(P_s)$ we get:

$$r_s + c_sx = \bar{r_s} + \bar{c_s}x$$

Isolating x:

$$x = \frac{r_s - \bar{r_s}}{\bar{c_s} - c_s}$$

Zero knowledge We have to show that it is possible to construct a simulator M which on inputs α, c_s outputs a valid transcript with the same probability distribution as conversations between an honest verifier and sender on common input α. We construct the simulator as follows:

1. The simulator samples $r_s \in_R [1, .., l]$.
2. Using the sampled value r_s it computes:

$$L'_s = r_sG + c_sP$$
$$R'_s = r_s\mathcal{H}_p(P_s) + c_sI$$

   which it can because it knows, r, c, I and the hash function $\mathcal{H}_p()$.
3. The simulator outputs (L'_s, R'_s, c_s, r_s)

We have that $L'_s = L_s, R'_s = R_s$ by completeness. We also have that the simulated conversation has the same probability distribution as a real conversation between honest an honest sender and verifier. The simulator randomly samples the response r_s and computes the R'_s, L'_s from it. This makes it so that these two values are also uniformly random, which in turn implies that L_s, R_s are uniformly random. This is the same distribution as in the real conversations as L_s, R_s are computed from a uniformly random value q_s. This gets us that the simulated conversation is perfectly indistinguishable of a real conversation.

5.6 OR

Completeness: Follows from completeness of the underlying protocol used in the OR-proof, i.e completeness of equality of discrete log which is described in 5.5

Special soundness: To prove that the protocol is sound we assume we have two accepting transcripts

$$(L_0, ..., L_{m-1}, R_0, ..., R_{m-1}, c, c_0, ..., c_{m-1}, r_0, ..., r_{m-1})$$

and

$$(L_0, ..., L_{m-1}, R_0, ..., R_{m-1}, \bar{c}, \bar{c_0}, ..., \bar{c_{m-1}}, \bar{r_0}, ..., \bar{r_{m-1}})$$

with $c \neq \bar{c}$. Then it must follow that for some index i with $c_i \neq \bar{c_i}$ we can retrieve (L_i, R_i, c_i, r_i) and $(L_i, R_i, \bar{c_i}, \bar{r_i})$. Using these two it is possible to extract a witness $w = x$ such that $(\alpha_i, w) \in R$ by special soundness of the underlying protocol.

Honest Verifier Zero-knowledge: If we are given a challenge c we sample the m challenges, $c_0, .., c_{m-1}$, at random under the constraint that $c = \sum_{i=0}^{m-1} c_i$. We would then run the simulator M, m times on inputs α_0, c_0 up to α_{m-1}, c_{m-1}.

Assume that we are given an arbitrary verifier V^*. The distribution of conversations between P, V^* would then be

$$(L_0, .., L_{m-1}, R_1, ..., R_{m-1}, c_0, ..., c_{m-1}, r_0, ..., r_{m-1})$$

where $L_i's, R_is'$ would be distributed honestly as the sender is honest and c has the distribution of whatever V^* outputs. All the c_i's are random under the constraint that $c = \sum_{i=0}^{m-1} c_i$. Finally the responses $r_i's$ are distributed honestly by the sender. From HVZK of the underlying protocol we get that all instances are perfectly indistinguishable and therefore independent of the sender's index.

5.7 Proof of properties

Linkability: Given the secret keys secret keys $\{x_i\}_{i=0}^{m-1}$ for a set S of m public keys where $\forall i \; P_i = x_i G$. Using these two sets it is impossible to produce $m+1$ valid signatures $\sigma_0 ... \sigma_m$ such that they all pass the LNK phase.

Theorem 5.1. *The one-time ring signature scheme is linkable in the random oracle model*

Proof. Assume an adversary is able to produce valid $m+1$ signatures. The key image for the first m signatures are constructed using the m public, private key pairs such that for any $i, j \in [0.., m-1]$ $I_i \neq I_j$. The last key image is assumed to exist. Since the cardinality of S is m and not $m+1$ there is at least one key image $I_j \neq x_i \mathcal{H}_p(P_i)$ for every $i \in [0, .., m-1]$.

Let $\sigma = (I_j, c_0, ..., c_{m-1}, r_0, ..., r_{m-1})$ be a signature such that $\mathsf{VER}(\sigma) = True$. This implies that:

$$L'_i = r_i G + c_i P_i \tag{3}$$

$$R'_i = r_i \mathcal{H}_p(P_i) + c_i I_j \tag{4}$$

$$\sum_{i=0}^{m-1} c_i = \mathcal{H}_s(m, L'_0, ..., L'_{m-1}, R'_0, ..., R'_{m-1}) \tag{5}$$

Taking the *log* in the above equations 3 and 4 we get:

$$log_G(L_i') = r_i + c_i x_i \tag{6}$$

$$log_{\mathcal{H}_p(P_i)}(R_i') = r + c_i log_{\mathcal{H}_p(P_i)}(I_j) \tag{7}$$

Since the adversary doesn't know the secret and public key corresponding to the $m + 1$'th key image then he cannot reconstruct the challenge from the NIZK verifier. Furthermore, every challenge c_i for $i \in [0, .., m-1]$ is uniquely determined from the other factors in 7.

This is seen because for the adversary to compute challenge for the j'th index he would either have to know P_j or invert the hash function. As he doesn't know the public key P_j then to compute $c_j = c - \sum_{j \neq i, i=0}^{m-1} c_i$ he would have to invert the hash function to learn c. Which is assumed to be hard and therefore we have the linkability property. □

Exculpability: Given a set of public keys S of size m, at most $m - 1$ corresponding private keys x_i as well as the key image I_j produced from the secret key x_j that is different from all known $m - 1$ secret keys. Then it is impossible to produce a valid signature using the key image I_j.

Theorem 5.2. *The one time ring signature scheme is exculpable under the discrete logarithm assumption in the random oracle model*

Proof. Suppose an adversary can produce a valid signature σ with key image $I_j = x_j \mathcal{H}_p(P_j)$. Then it is possible to construct an algorithm A that solves the discrete logarithm problem in the elliptic curve. Let (G, P) be an instance of the DLP where the goal is to find x_j such that $P = x_j G$. The algorithm A simulates the random oracle to get a random message that it feeds to the signing oracle which replies with a signature using I_j: $\sigma = (I_j, c_0, ..., c_{m-1}, r_0, ..., r_{m-1})$ and $\bar{\sigma} = (I_j, \bar{c_0}, ..., \bar{c_{m-1}}, \bar{r_0}, ..., \bar{r'_{m-1}})$ with $P_j = P$. Since we are using the oracles we have that the commitments, $L_i, R_i, \bar{L_i}, \bar{R_i}$ will be the same. This in turn implies that $L_i' = \bar{L_i'}$. From this we can write:

$$L_j' = r_j G + c_j P_j = \bar{r_j} G + \bar{c_j} P_j = \bar{L_j'}$$

Taking the logarithm of L_j' and $\bar{L_j'}$ we get:

$$log_G(L_j') = r_j + c_j x_j$$
$$log_G(\bar{L_j'}) = \bar{r_j} + \bar{c_j} x_j$$

Since $L_j' = \bar{L_j'}$ then it also follows that $log_G(L_j') = log_G(\bar{L_j'})$ and therefore we can compute the following:

$$\begin{gathered} r_j + c_j x_j = \bar{r_j} + \bar{c_j} x_j \\ \Updownarrow \\ r_j - \bar{r_j} = \bar{c_j} x_j - c_j x_j \\ \Updownarrow \\ r_j - \bar{r_j} = (\bar{c_j} - c_j) x_j \\ \Updownarrow \\ x_j = \frac{r_j - \bar{r_j}}{\bar{c_j} - c_j} \end{gathered}$$

Since $I_j = x_j \mathcal{H}_p(P_j)$ in both signatures then we have $x_j = log_{\mathcal{H}_p(P_j)}(I_j)$ and we get that:

$$x_j = log_{\mathcal{H}_p(P_j)}(I) = \frac{r_j - \bar{r_j}}{\bar{c_j} - c_j}$$

So if such an adversary exists we can construct an algorithm which breaks the DLP in the elliptic curve. □

Unforgeability: Given only the set of public keys S it is impossible to produce a valid signature

Theorem 5.3. *If a one-time ring signature scheme is linkable and exculpable then it has the unforgeablility property.*

Proof. This follows from the linkable and exculpable properties. Assume an adversary can forge a signature, $\sigma = (I, c_0 ... c_{m-1}, r_0 ... r_{m-1})$ for a set of public keys S of size m. If we then consider the set of valid signatures then there is also a set of valid key images $\mathcal{I}$ constructed using the set S. Then we have two case either $I \in \mathcal{I}$ or $I \notin \mathcal{I}$ contradicting exculpability and linkability respectively. □

Anonymity: Given a signature σ and the set of public keys $\mathcal{S}$ used to construct it. It is impossible to learn the index of the sender with probability greater than $\frac{1}{m}$.

Theorem 5.4. *The one-time ring signature scheme has the anonymous property under the DDH assumption in the random oracle model.*

Proof. Assume that there exists an adversary A that can extract the secret index j of a sender from a signature with probability $p = \frac{1}{m} + \epsilon$. Then it is possible to create an algorithm using A as a subroute which breaks the DDH problem in $\mathbb{F}_q$ with probability $p = \frac{1}{2} + \frac{\epsilon}{2}$.

Say we have some instance of the DDHP (G, aG, bG, cG). With $P_j = bG = x_jG, I = cG, \mathcal{H}_p(P_j) = aG$. The goal is then to determine if $log_G(bG) = log_{aG}(cG)$

The algorithm takes a valid signature with P_j, I and will then simulate a random oracle that returns $\mathcal{H}_p(P_j)$ on P_j. Now using this information the algorithm will make the adversary return a guess k for the index $i : I = x_i\mathcal{H}_p(P_i)$. If $k = j$ then the algorithm outputs 1 else it outputs a random bit r. The probability for the right choice is:

$$\begin{aligned}
Pr[success] &= \frac{1}{2} + Pr[1|DDHP] - Pr[1|\neg DDHP] \\
&= \frac{1}{2} + Pr[k = j|DDHP] + Pr[k \neq j|DDHP]Pr[r = 1] \\
&\quad - Pr[k = j|\neg DDHP]Pr[k \neq j|\neg DDHP]Pr[r = 0] \\
&= \frac{1}{2} + \frac{1}{n} + \epsilon + (\frac{n-1}{n} - \epsilon) \cdot \frac{1}{2} - \frac{1}{2} - \frac{n-1}{n} \cdot \frac{1}{2} \\
&= \frac{1}{2} + \frac{\epsilon}{2}
\end{aligned}$$

The reason why it corresponds to the DDHP can be seen if we consider the case where the adversary guesses the correct index. Then we have:

$$\begin{aligned}
I &= x_j\mathcal{H}_p(P_j) \\
&= x_jaG \\
&= baG
\end{aligned}$$

Since the specified DDHP states that $I = cG$ then we have that if the adversary guesses the correct index then he solves the DDHP. □

6 Bulletproof Framework

Range proofs are proofs that shows that a committed values lies in a specific range. There are many different types of range proofs which differs in complexity. The one used in Monero is Bulletproofs [1] which is a non-interactive zero-knowledge proofs which require no trusted setup, where the proof size is only logarithmic in the size of the witness. With bulletproofs one party can prove that multiple committed amounts lies in a given range. Before we go into detail of the bulletproof framework, we will briefly describe commitment schemes.

6.1 Commitment schemes

In general a commitment scheme is a technique where a prover P chooses a random value r from $\{0, 1\}^l$, where l is a security parameter. The prover commits to a bit / message b, to produce a commitment $C = com(r, b)$. This commitment is sent to the verifier V. When the prover wants the verifier to open the commitment it would send C, r, b to the verifier. Now the verifier checks that the received commitment C is indeed what the prover committed to i.e. the verifier computes $C' = com(r, b)$ and checks whether $C = C'$. The commitment functions is chosen so the verifier will have a hard time computing the committed value without the 'key' (r, b). The verifier can't figure out what the commitments value is, without getting the opening values from the prover. The important properties of commitment schemes is the binding and the hiding property [5].

Hiding and binding: Hiding in itself is trying to make it difficult for the verifier to compute the value that was committed to in the comitment without receiving the opening values from the prover.

Binding means that the prover when to prover has comitted to a value x and has send the commitment to the verifier. Then it will be hard for the prover to change his mind and convice the verifier that he committed to some value y different from x. The hiding and binding properties both comes in two flavors: Unconditional hiding and binding, and computational hiding and binding [5].

Pederson commitment: The commitment scheme that we will use and refer to in this thesis is the Pederson Commitment. For the setup this commitment scheme uses two random generators g, h which are chosen from a cyclic group of prime order p. Let x be the message the prover wants to commit to, and let r be the randomness used in the commitment. The way a Pedersen commitment is computed is by $C = com(x, r) = g^x h^r$. The prover then later reveals x and r so the verifier can check if indeed $C = com(x, r)$ [9].

We can also use vectors with the Pederson commitment to construct a Pederson vector commitment. Let the two generators $\mathbf{g} \in \mathbb{G}^n$ and $h \in \mathbb{G}$ and $\mathbf{x}$ be the vector that is to be committed. Then the commitment function is defined as $C = com(\mathbf{x}, r) = h^r \mathbf{g}^x = h^r \prod_i g_i^{x_i}$ [8].

Pederson commitments are unconditionally hiding since for any committed value x as there exists unique randomness r that will make the commitment uniformly random such that the verifier gets no information on x. Pederson commitments are also computationally binding. This can be seen given that if the prover can find two different values x, x' which both are valid openings for the commitment C such that $g^x h^r = g^{x'} h^{r'}$, where $r' \neq r$. Then the prover can solve the discrete log problem. As the discrete log problem is hard, the prover cannot open the commitment with another value x', and therefore it is computationally binding.

6.2 Improved Inner-product Argument

The inputs for the inner-product argument are two generators $\mathbf{g}, \mathbf{h} \in \mathbb{G}^n$, the binding commitment $P \in \mathbb{G}$ of two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_p^n$ and a scalar $c \in \mathbb{Z}_p$.

The argument lets the prover convince the verifier that the it knows a Pederson commitment to vectors $\mathbf{a}, \mathbf{b}$ under the condition that $c = \langle a, b\rangle$. The inner product argument is an proof system for the following relation:

$$\{(\mathbf{g}, \mathbf{h} \in \mathbb{G}^n, P \in \mathbb{G}, c \in \mathbb{Z}_p; \mathbf{a}, \mathbf{b} \in \mathbb{Z}_p^n) : P = \mathbf{g}^{\mathbf{a}}\mathbf{h}^{\mathbf{b}} \wedge c = \langle \mathbf{a}, \mathbf{b}\rangle\} \tag{8}$$

To prove the relation the simplest method is to let the prover send $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_p^n$ to the verifier, which accepts if the vectors are valid witness for the relation. This method is sound, but it requires sending $2n$ elements, where it is possible to only send $2\log_2(n)$. Thus to prove the relation a new element $u \in \mathbb{G}$ is introduced, which has a discrete log relation to the $\mathbf{g}, \mathbf{h} \in \mathbb{G}^n$. The binding commitment then becomes $P = \mathbf{g}^{\mathbf{a}}\mathbf{h}^{\mathbf{b}}u^{\langle \mathbf{a}, \mathbf{b}\rangle}$, and the relation becomes:

$$\{(\mathbf{g}, \mathbf{h} \in \mathbb{G}^n, u, P \in \mathbb{G}, c \in \mathbb{Z}_p; \mathbf{a}, \mathbf{b} \in \mathbb{Z}_p^n) : P = \mathbf{g}^{\mathbf{a}}\mathbf{h}^{\mathbf{b}}u^{\langle \mathbf{a}, \mathbf{b}\rangle}\} \tag{9}$$

We are able to prove the first relation 8 by giving a proof of the second relation 9. To define how the proof system works a hash function $H : \mathbb{Z}_p^{2n+1} \leftarrow \mathbb{G}$ is introduced. Let $n' = \frac{n}{2}$ and fix generators $\mathbf{g}, \mathbf{h} \in \mathbb{G}^n$ and $u \in \mathbb{G}$. The hash function will then take as input $\mathbf{a}, \mathbf{a'}, \mathbf{b}, \mathbf{b'} \in \mathbb{Z}_p^n$ and $c \in \mathbb{Z}_p$ and will output:

$$H(\mathbf{a}, \mathbf{a'}, \mathbf{b}, \mathbf{b'}, c) = \mathbf{g}_{[:n']}^{\mathbf{a}} \cdot \mathbf{g}_{[n':]}^{\mathbf{a'}} \cdot \mathbf{h}_{[:n']}^{\mathbf{b}} \cdot \mathbf{g}_{[n':]}^{\mathbf{b'}} \cdot u^c$$

It is then possible to rewrite the commitment P from the relation 9 as $P = H(\mathbf{a}_{[:n']}, \mathbf{a'}_{[n':]}, \mathbf{b}_{[:n']}, \mathbf{b'}_{[n':]}, < a, b >)$, which works since the hash function H is chosen such that it is additively homomorphic in its inputs.

We now describe the protocol which can be seen in appendix A. First the prover computes $L, R \in \mathbb{G}$ as follow:

$$L = H(\mathbf{0}^{n'}, \mathbf{a}_{[:n']}, \mathbf{b}_{[n':]}.\mathbf{0}^{n'}, \langle \mathbf{a}_{[:n']}, \mathbf{b}_{[n':]}\rangle)$$
$$R = H(\mathbf{a}_{[n':]}, \mathbf{0}^{n'}, \mathbf{0}^{n'}, \mathbf{b}_{[:n']}, \langle \mathbf{a}_{[:n']}, \mathbf{b}_{[n':]}\rangle)$$

and sends those to the verifier. The verifier then chooses a random $x \in \mathbb{Z}_p$ and sends x to the prover who then computes $\mathbf{a'} = x\mathbf{a}_{[:n']} + x^{-1}\mathbf{a}_{[n':]} \in \mathbb{Z}_p^{n'}$ and $\mathbf{b'} = x^{-1}\mathbf{b}_{[:n']} + x\mathbf{a}_{[n':]} \in \mathbb{Z}_p^{n'}$ and sends them to the verifier. Now given $(L, R, \mathbf{a'}\mathbf{b'})$ the verifer computes $P' = H(x^{-1}\mathbf{a'}, x\mathbf{a'}, x\mathbf{b'}, x^{-1}\mathbf{b'}, \langle \mathbf{a'}, \mathbf{b'}\rangle)$. We note here that this is equivalent to test that

$$P' = (\mathbf{g}_{[:n']}^{x-1} \circ \mathbf{g}_{[n':]}^{x})^{\mathbf{a'}} \cdot (\mathbf{h}_{[:n']}^{x} \circ \mathbf{h}_{[n':]}^{x-1})^{\mathbf{b'}} \cdot u^{\langle \mathbf{a'}, \mathbf{b'}\rangle}$$

Thus the prover can recursively engage in an inner-product argument with P' with respect to generators $(\mathbf{g}_{[:n']}^{x-1} \circ \mathbf{g}_{[n':]}^{x}, \mathbf{h}_{[:n']}^{x} \circ \mathbf{h}_{[n':]}^{x-1}, u)$, to get dimension of $n' = \frac{n}{2}$, instead if sending $\mathbf{a'}$, $\mathbf{b'}$ to the verifier. This is continued until $n' = 1$

6.3 Inner-product Range Proof

The following protocol uses the improved inner-product argument to construct a range proof, where the prover convinces the verifier that a commitment V

contains a number v that lies in a certain range, without revealing the value v. The entire protocol is shown on appendix B, and we will refer to the numbers of the steps when explaining the protocol.

Let $v \in \mathbb{Z}_p$ and let $V \in \mathbb{G}$ be a Pederson commitmnt to v using randomness γ. The proof system proves the following relation:

$$\{(g, h \in \mathbb{G}, V, n; v, \gamma \in \mathbb{Z}_p) : V = h^\gamma g^v \land v \in [0, 2^n - 1]\} \tag{10}$$

In other words the proof system will convince the verifier that $v \in [0,^2 n - 1]$. Now let $\mathbf{a}_L = (a_1, ..., a_n) \in \{0, 1\}^n$ be the vector which contains the bits of v such that $\langle \mathbf{a}_L, \mathbf{2}^n \rangle = v$. To commit to $\mathbf{a}_L$ the prover will use a constant size vector commitment $A \in \mathbb{G}$. The prover convinces the verifier that v lies in $[0, 2^n - 1]$ by proving that it knows an opening $\mathbf{a}_L \in \mathbb{Z}_p^n$ of A and $v, \gamma \in \mathbb{Z}_p$ such that $V = h^y g^v$ and that the following conditions holds:

$$\langle \mathbf{a}_L, \mathbf{2}^n \rangle = v \text{ and } \mathbf{a}_L \circ \mathbf{a}_R = \mathbf{0}^n \text{ and } \mathbf{a}_R = \mathbf{a}_L - \mathbf{1}^n \tag{11}$$

The first constraint ensures that $\mathbf{a}_L$ is composed of the bits of v and the second and third constraints ensure that the entries is $\mathbf{a}_L$ are all 0 or 1. Now we have $2n + 1$ elements that we want to comrpess into a single inner-product element. To ensure this the verifier sends a random $y \in \mathbb{Z}_p$ to the prover which in turn will prove that $\langle \mathbf{b}, \mathbf{y}^n \rangle = 0$ where $\mathbf{b} \in \mathbb{Z}_p^n$ is the vector being committed to. It is made such that if $\langle \mathbf{b}, \mathbf{y}^n \rangle = 0$ then $\mathbf{b}$ is the zero vector and if $\langle \mathbf{b}, \mathbf{y}^n \rangle \neq 0$ then the equality holds with at most negligible probability $\frac{n}{p}$. So if $\langle \mathbf{b}, \mathbf{y}^n \rangle = 0$ then prover convinces the verifier that $\mathbf{b} = 0^n$.

Thus using $y \in \mathbb{Z}_p$ from the verifier the prover can prove 11 holds by showing that the following constraints hold true

$$\langle \mathbf{a}_L, \mathbf{2}^n \rangle = v \text{ and } \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle = 0 \text{ and } \langle \mathbf{a}_L - \mathbf{1}^n - \mathbf{a}_R, \mathbf{y}^n \rangle = 0 \tag{12}$$

These three equalities can be combined into a single equality by letting the verifer choose a random $z \in \mathbb{Z}_p$ so now the prover then proves that

$$z^2 \cdot \langle \mathbf{a}_L, \mathbf{2}^n \rangle + z \cdot \langle \mathbf{a}_L - \mathbf{1}^n - \mathbf{a}_R, \mathbf{y}^n \rangle + \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle = z^2 \cdot b$$

which can be re-written as:

$$\langle \mathbf{a}_L - z \cdot \mathbf{1}^n, \mathbf{y}^n \circ (\mathbf{a}_R + z \cdot \mathbf{1}^n) + z^2 \cdot \mathbf{2}^n \rangle = z^2 \cdot v + \delta(y, z) \tag{13}$$

where $\delta(y, z) = (z - z^2) \cdot -z^3 \langle \mathbf{1}^n, \mathbf{y}^n \rangle - z^3 \langle \mathbf{1}^n, \mathbf{2}^n \rangle \in \mathbb{Z}_p$.

From all this we get that the original constraints 12 have been reduced into a single inner product. Given that we now only have one single inner product the improved inner product argument can be applied.

There is one issue which is the prover cannot send the two vectors in the inner product in 13 as information about $\mathbf{a}_L$ and $\mathbf{a}_R$ will be leaked. The fix for this is to introduce two additional blinding terms $\mathbf{S}_L, \mathbf{S}_R \in \mathbb{Z}_P^n$. Using these two blinding terms the prover computes another commitment on line (3) in B, and sends it to the verifier along with the commitment A of $\mathbf{a}_L$ and $\mathbf{a}_R$, computed at step (2) in B.

Next two linear polynomials $l(X), r(X) \in \mathbb{Z}_p^n[X]$ are defined as follows:

$$l(X) = (\mathbf{a}_L - z \cdot \mathbf{1}^n) + \mathbf{S}_L \cdot X$$
$$r(x) = \mathbf{y}^n \circ (\mathbf{a}_R + z \cdot 1 + \mathbf{S}_R \cdot X) + z^2 \cdot \mathbf{2}^n$$

where the constant terms are the inner product vectors in 13. A quadratic polynomial $t(X) \in \mathbb{Z}_p[X]$ is also defined as follows:

$$t(X) = \langle l(X), r(X) \rangle = t_0 + t_1 \cdot X + t_2 \cdot X^2$$

Now the prover must convince the verifer that it knows the constant term of $t(X)$, denoted t_0, as this is the result of the inner product $t_0 = v \cdot z^2 + \delta(y, z)$. To prove this the prover will have to convince the verifier that it have committed to $t_1, t_2 \in \mathbb{Z}_p$, by checking the value of $t(X)$ at a random point $\in \mathbb{Z}_p^*$.

The commitment to t_0, t_1 are done on line (8) in B, and they are then sent to the verifier. The verifier then sample a challenge $x \in \mathbb{Z}_p$ and sends it to the prover. The prover then computes the polynomials $\mathbf{l}, \mathbf{r} \in \mathbb{Z}_p$ and their inner product t as described earlier. The prover also construct a blinding value for t on line (16) in B and a value μ on on lines (17) in B which is the blinding values of A and S, and send everything to the verifier. To verify the verifier checks that l, r corresponds to $l(x), r(x)$ on line, $t = \langle \mathbf{l}, \mathbf{r} \rangle$ and $t = t(x) = t_0 + t_1 x + t_2 x^2$ on lines (23 - 25) in B.

Logarithmic Range Proof In the range proof protocol the prover sends $\mathbf{l}$ and $\mathbf{r}$ to the verifer, whose size is linear in n, but the goal is to construct with is logarithmic in n. To do so we eliminate the transfer of $\mathbf{l}$ and $\mathbf{r}$ using the inner-product argument from section 6.2. We note that verifying (23) and (25) in B is the same as verifying that $\mathbf{l}$ and $\mathbf{r}$ satisfies the inner product relation on public input $(\mathbf{g}, \mathbf{h'}, Ph^{-\mu}, t)$, where $P \in \mathbb{G}$ is a commitment to vectors $\mathbf{l}, \mathbf{r} \in \mathbb{Z}_p^n$ with inner product t and $\mathbf{h'}$ comes from the computation on line (20) in B. On line (18) in B we can therefor only transfer (τ_x, μ, t) and an execution of an inner product argument. Thus instead of transmitting $\mathbf{l}$ and $\mathbf{r}$, which has communication cost of $2n$ elements, we transmits $2\lceil \log_2(n) \rceil + 2$ elements [1].

7 Monero Additions

As mentioned before the cryptocurrency that we take the most interest in is Monero. In all cryptocurrencies there is the notion of a transaction. This transaction has an output which is basically says that "person x has spending rights up to some amount of money". In most cases this amount is transmitted in cleartext. This motivation for cleartext is that it easily allows verifiers to verify that the amount spend equals the amount transferred [7].

Monero is based off the whitepaper called CryptoNote which is thorougly described in the first part of this thesis. In addition to the privacy means of the whitepaper Monero also wants to hide the output amounts from each party whom it is not intended for, while keeping the confidence that the amount spent equals the amount sent [7].

To do this Monero has used commitments to hide the output as well as incorporating range proofs that proves to others that the amount is within some range that fits the spending amount.

7.1 RingCT Intro

Even though that verifiers have no means of knowing how much money is contained in the transaction inputs and outputs they still need to be able to verify that the sum of inputs equal the sum of outputs [7]. For this purpose Monero implemented a techinque called RingCT back in 2017 which allows allows this type of checking [7].

The idea is that if we assume we have m inputs that has amounts $a_1, ..., a_m$ and k outputs that has amounts $b_1, .., b_k$. Then any verifier should be able to expect that

$$\sum_i^m a_i - \sum_j^k b_j = 0$$

Now since the commitments used are Pedersen commitments which are additively homomorphic as well as the fact that the randomness γ is unknown then the spendings could be proven to verifiers by summing the commitments and showing that the difference is 0. However, this approach does not avoid sender identifiability. Therefore, Monero instead uses the fact that the amounts spent is related to the previous transactions' outputs. These outputs would have had the following commitments:

$$c_j^a = g^{x_j} + h^{a_j}$$

where a_j would be the amount. A sender could from this compute another commitment to the same amount with different randomness / blinding factor such as:

$$c_j^{a\prime} = g^{x_j'} + h^{a_j}$$

The sender would be able to compute the private key for the difference of the commitments by computing:

$$c_j^{(a_j - a_j')} = g^{(x_j - x_j')} + 1$$

which follows by the homomorphic property of Pedersen commitments. Using this new private key $z_j = (x_j - x_j')$ the prover can make a commitment to 0, namely it would prove that

$$c_j^{(a_j - a_j')} = g^{z_j} + h^0$$

The extra commitment ${c_j^a}'$ is known as a pseudo output commitment [7]. The way a prover would choose the randomness / blinding factors for the pseudo, output commitments would be such that

$$\sum_i^m x_i' - \sum_j^k y_j = 0$$

where x_j would be the blinding factor for the pseudo output commitment and y_i would be for the output commitment. Under this constraint it is possible to prove that the input amount corresponds to the output amount by

$$\sum_i^m {c_i^a}' - \sum_j^k c_j^b = 0$$

The way the blinding factors are chosen in Monero is just by random sampling except for the m'th pseudo blinding factor which is chosen by:

$$x_m' = \sum_j^k y_j - \sum_i^{m-1} x_i'$$

There is an issue in this which speaks to the motivation of the application of range proof. The issue lies in the fact that the commited output amounts can be negative. In the basic construction when the sum of output commitments would be subtracted from the input commitments the sign of the negative commitments would flip and cause a nonvalid commitmnet to potentially be valid. Example from [7] would be $(6 + 5) - (21 + -10) = 0$. The solution Monero implemented is using bullet proofs to show that a value is within a given range.

This is the general idea of RingCT used in Monero and for the details of the specific implementation used in Monero currently we reference to chapter 6 of [7].

In the following subsections we formally present the RingCT security properties, concept of Omniring and the idea of RingCT with logarithmic sized proofs from [8]

7.2 RingCT Security Properties

The security properties of RingCT are known as balance, privacy and non-slanderability [8].

Balance means that the spender cannot double-spend, or spend more than they possess. More concretely a RingCT scheme is balanced if it satisfies the following two properties. The first property is that the algorithms "CheckTag" which is the amount checking algorithm and "CheckAmount" the tag checking algorithm are binding similar to a commitment scheme. Further details of the algorithms can be seen in section 2.2 in [8] This ensures that the tag is computationally bound to a source account, which makes it so that checking for duplicate tags is sufficient to prevent double-spending. Similarly, the amount is also computationally bound to an account, which ensures that one can't just randomly create/generate money, by changing the amount of coins in a given account. The second property just states that for any adversary $\mathcal{A}$ which produces a transaction with a proof, there exists an extractor $\mathcal{E}_{\mathcal{A}}$ such that if the proof is valid, then the extractor with high probability is able to extract the witness leading to the transaction [8].

Privacy means that an adversary cannot be able to distinguish two transactions with the same ring and their proofs. This should hold even if the majority of the ring is corrupted, and the adversary has prior knowledge about the identities of the spenders and receivers together with the amounts that is being transferred. An adversary is allowed to specify a ring which has arbitrarily many corrupt accounts but at least two honest accounts which are the potential spenders. The adversary also specifies two accounts of receivers and the amounts that they are supposed to receive. Using one of the two specifications a transaction is created, and it should hold that the adversary cannot tell which of the specifications were used to create the transaction [8].

Non-slanderability means than one cannot produce a valid proof on behalf of another user. A transaction has a set of tags that is bound to a set of source accounts, thus if the owner of one of the source accounts wants to spends from the account it will not be accepted since the tag corresponding to the amount has already been published. Formally, non-slanderability is modelled by defining a security experiment where the adversary produces a transaction-proof tuple, after several queries to an oracle on figure 1 in [8]. We say that the adversary is successful if the tuple is valid and not produced by the oracle, and some of the tags specified in the slandering transaction collide with those that are signed by the oracle [8].

7.3 Omniring

Omniring is a RingCT, whose proof size is $O(\log(|\mathcal{R}||\mathcal{S}| + \beta|\mathcal{T}|))$, where $|\mathcal{R}|$ is the size of the ring, $|\mathcal{S}|$ is the set of source accounts and $|\mathcal{T}|$ are the set of target accounts. The maximum amount allowed to be transferred to an account is 2^{β}. To do so Omniring combines a ring signature scheme with a range proof system. The range proof system that is used is the Bulletproofs framework described earlier in 6. However, as both proof system uses different techniques, if the two just gets combined naively, one would end up with a RingCT scheme with signature size asymptotically equal to that of Omniring but with worse concrete efficiency.

In order to combine them the bulletpoof framework gets extended with new techniques which it possible to prove knowledge of a discrete logarithm representation. This makes it so instead of using Bulletproofs for only range proofs, one can also prove ownership of ones coins, and their spendability. All of these proof statements can then be combined into a single zero-knowledge proof.

Now we describe how one can construct a ring membership proof using the Bulletproofs framework. Let $\mathbf{r}$ be the vector that consists of the ring members' public keys $r_1, ..., r_n$. To prove membership one would prove knowledge of a tuple (i, x_i) such that $r_i = h^{x_i}$ for public generator h. This is equivalent to proving knowledge of a unit vector $\mathbf{e}_i$ and an integer $-x_i$ such that $I = h^{-x_i}\mathbf{r}^{\mathbf{e}_i}$, where I is the identity element. This relation is called the "main equality" [8].

We want to embed the term $h^{-x_i}\mathbf{r}^{\mathbf{e}_i}$ into a part of the commitment A on line (2) in B to show that $\mathbf{e}$ is a unit vector, and show that the main equality holds. To do so we first take an arbitrary vector $\mathbf{p}$ of group elements that are chosen randomly and independently of $\mathbf{r}$, then for any $w \in \mathbb{Z}_q$ the discrete log problem base $\mathbf{g}_w = (h||\mathbf{r})^w \circ \mathbf{p}$ for generator h is equivalent to the standard discrete logarithm problem.

Now let $\mathbf{a} = (-x_i||\mathbf{e}_i)$ then it holds that $\mathbf{g}_w^{\mathbf{a}} = \mathbf{g}_{w'}^{\mathbf{a}}$ for any $w, w' \in \mathbb{Z}_q$ because of the main equality. Thus if $A = h^\alpha \mathbf{g}_w^{\mathbf{a}} \mathbf{h}^{\mathbf{b}}$ for some $w \in \mathbb{Z}_p$, then for any other $w' \in \mathbb{Z}_p$ then it also holds that $A = h^\alpha \mathbf{g}_{w'}^{\mathbf{a}} \mathbf{h}^{\mathbf{b}}$. Now the prover is able to run the a bulletproof-style protocol twice on A with two different bases, and if the prover is able to convince the verifier in both runs, then one can construct an extractor which extracts the exponents $(\alpha, \mathbf{a}, \mathbf{b})$ such that $A = h^\alpha \mathbf{g}_w^{\mathbf{a}} \mathbf{h}^{\mathbf{b}} = h^\alpha \mathbf{g}_{w'}^{\mathbf{a}} \mathbf{h}^{\mathbf{b}}$. Dividing both representations of A yields the main equality.

The only problem with this is that one would have to execute the Bulletproof protocol twice, which increases the proof size by a factor of 2. To get around this the prover computes two commitments A, S as in the normal Range Proof lines (2 - 3) in B, with $A = h^\alpha \mathbf{g}_0^{\mathbf{a}} \mathbf{h}^{\mathbf{b}}$ and $S = h^\alpha \mathbf{g}_w^{\mathbf{a}} \mathbf{h}^{\mathbf{b}}$, where w is obtained by hashing A. By sending these two commitments we have compressed two executions into a single execution of the range proof. The remaining steps of the Range Proof will also need to get slightly modified to be able to verify which can be seen on section 5 in[8].

For performance comparison RingCT have been compared with the scheme currently employed in Monero and two other schemes namely RingCT 2.0 [11] and RingCT 3.0, [12] which improves upon RingCT 2.0, in terms of proof time and running time which can be seen on figure 2 and 3

Scheme	**Spend proof size (in elements)**	**Pairing**	**Trusted setup**
Monero [27, 45] with Bulletproofs range proofs	$O(\lvert\mathcal{R}\rvert\lvert\mathcal{S}\rvert+\log(\beta\lvert\mathcal{T}\rvert))$	No	No
RingCT 2.0 [58] with Bulletproofs range proofs	$O(\lvert\mathcal{S}\rvert+\log(\beta\lvert\mathcal{T}\rvert))$	Yes	Yes
RingCT 3.0 [62] with Bulletproofs range proofs	$O(\lvert\mathcal{S}\rvert\log\lvert\mathcal{R}\rvert+\log(\beta\lvert\mathcal{T}\rvert))$	No	No
Omniring	$O(\log(\lvert\mathcal{R}\rvert\lvert\mathcal{S}\rvert+\beta\lvert\mathcal{T}\rvert))$	No	No

Figure 2: Proof comparison of RingCT schemes

Looking at 2 we see that the proof size Omniring is significantly smaller than that of Monero, and the difference in proof size only grows larger the bigger $|\mathcal{R}|$.

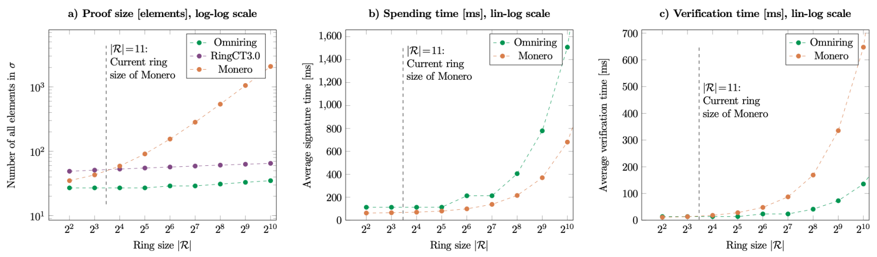

Figure 3: Performance comparison of RingCT schemes

As for performance we see that the time needed for generating proofs is faster in Monero, as Omniring need twice the amount of time, however Omniring has considerably faster verification time, for larger $|\mathcal{R}|$ values [8].

8 BGV

For the pre-pocessing phase in the offline phase the underlying encryption scheme is the BGV scheme [3]. For the underlying algebra in the BGV scheme we fix the ring $R_q = (\mathbb{Z}/q\mathbb{Z})[X]/\Phi_m(X)$ for some cyclotomic polynomial $\Phi_m(X)$. Let $R = \mathbb{Z}[X]/\Phi_m(X)$ and $\phi(m)$ be the degree of R over $\mathbb{Z}$, then the message space of the scheme is R_p for a prime p, and ciphertexts will lie in either $R^2_{q_0}$ or $R^2_{q_1}$ for one of two moduli q_0 and q_1.

Throughout the definitions we will also refer to a polynomial $a \in R$ as a vector of size n. To make the cryptosystem work, we need to sample from different distributions to generate the vector with length n and coefficients mod p or q. The following distributions are

- $\mathcal{ZO}_s(0,5,n)$ generates a vector of length n where each entry are in the set $\{-1,0,1\}$. Zero appears with probability $\frac{1}{2}$, whereas $\{-1,1\}$ appears each with probability $\frac{1}{4}$
- $\mathcal{DG}_s(\sigma^2,n)$ generates a vector of length n where each elements is chosen by the Gaussian distribution: $\mathcal{N}(\sigma^2)$
- $\mathcal{RC}_s(0,5,\sigma^2,n)$ generates a triple (v,e_0,e_1) where $v \in \mathcal{ZO}_s(0,5,n)$ and $e_0,e_1 \in \mathcal{DG}_s(\sigma^2,n)$
- $\mathcal{U}_s(q,n)$ generates a vector of length n with elements generated uniformly modulo q

8.1 Key Generation

To generate the public/private key pair sample $a \leftarrow \mathcal{U}(q_1, \phi(m))$. Compute $b = a \cdot s + p \cdot \epsilon$, where s is the secret key such that $s = s_1 + ...s_n$ and ϵ is a "small" error term. The public key is then $pk = (a, b)$

8.2 Encryption and Decryption

- $\mathsf{Enc}_{pk}(m)$: To encrypt a message $m \in R_q$ sample a small polynomial and two Gaussian polynomials $(v, e_0, e_1) \leftarrow \mathcal{RC}_s(0, 5, \sigma^2, \phi(m))$. Set the ciphertext to be a pair $c = (c_0, c_1, 1)$ where $c_0 = b \cdot v + p \cdot e_0 + m$, $c_1 = a \cdot v + p \cdot e_1$ and 1 is the level of the ciphertext.

- $\mathsf{Dec}_s(c)$: To decrypt a ciphertext $c = (c_0, c_1, l)$ party P_1 computes $v_1 = c_0 - s_i \cdot c_1$, and each other player P_i computes $v_i = -s_i \cdot c_1$. Each P_i then computes $t_i = v_i + p \cdot r_i$ for some random value $r_i \in R$. The message is then recovered as $t_1 + ... t_n$ modulo p. If one party knows the shared secret key s then decryption of a ciphertext could be done by setting $m' = [c_0 - s \cdot c_1]_{ql}$ and then output m' modulo p, but since no one knows the shared secret key, this can never actually be performed.

Proof for correctness can be seen in appendix C

9 MPC

9.1 SPDZ

SPDZ Setup

The way MPC is done is over a finite field $\mathbb{F}_p$ of characteristic $p > 2$. As mentioned we assume that we have n players which will compute values using additive secret sharing of data and MAC keys.

There is a MAC key α which each party has a share of such that P_i has α_i such that $\alpha = \sum_{i=0}^{n-1} \alpha_i$. A party P_i has a value a that is secret shared if it holds the tuple, i.e. a share, $(a_i, \gamma(a)_i)$, where a_i is an additive secret sharing of the value a and $\gamma(a)_i$ is an additive secret sharing of $\gamma(a) = a \cdot \alpha$. The value $\gamma(a)_i$ is used to check a for correctness [3].

There is the notion of a share being partially opened which means that only the value a is being revealed and the MAC key remains secret shared.

SPDZ is split into an offline phase and an online phase. In the offline phase, sometimes also called preprocessing, the idea is that we want to create data such as multiplication triples, squaring, bits and even protocol specific data that will be consumed in the online phase. During the offline phase when data is being produced a somewhat homomorphic encryption (SHE) scheme is leveraged. The SHE scheme used is BGV which is briefly described earlier.

The informal idea of how multiplication triples are produced is the following; Each party samples random values and encrypts them. Let's call them a_i, b_i for party P_i. Now they compute $c = (a_0 + ... + a_{n-1}) \cdot (b_0 + ... + b_{n-1})$ by using the multiplicative property of the SHE scheme. After this they run the distributed decryption to decrypt the value c and each get a share of c. By this $[a], [b]$ and

$[c]$ has been produced under the constraint that $ab = c$. However, to have it fulfill the $\langle\rangle$ notation MACs on the values should also be computed in somewhat the same way [3].

Sub Protocols Used

The protocols from [3] that we will go through concerns MAC checking, and the production / checking of data in the offline phase. However, these protocols use some sub protocols in their routines. We will briefly describe the protocol Π_{commit}, EncCommit of the ideal functionality $\mathcal{F}_{SHE}$, and the protocol ReShare [3].

Π_{commit}: The commit function $\mathsf{com}(m)$ generates a random value r, sets $o = m||r$ and computes $c = \mathcal{H}(o)$ for some hash function $\mathcal{H}$. To open the commitment c the comitter outputs the value o and the verifer checks if $c = \mathcal{H}(o)$ and accepts if it's the case otherwise reject [3].

$\mathcal{F}_{SHE}$: the feature EncCommit of this protocol has all parties sample a random seed (the adversary's seed might not be random). Parties setups a PRF using the random seed and samples a message m_i and in turn computes an encryption $c_i = Enc(m_i, \mathcal{RC}_{s_i}(0.5, \sigma^2, \phi(m)))$, where s_i is the random seed and Enc is the encryption from BGV. Lastly, parties share the encryption they computed. So after this protocol each party has a plaintext m_i (the one they sampled from the PRF) and an encryptions c_i of a_i from all parties [3].

$ReShare$: takes a public ciphertext, c_m, as input and a paramenter $enc = NewCiphertext \vee NoNewCiphertext$. The output of this protocol is a share m_i of m to each party P_i. If $enc = NewCiphertext$ then it also outputs a cipher c'_m [3].

The ciphertext that is input could be a product of two ciphertexts. ReShare will then convert it to a new ciphertext c'_m. Note that ReShare uses distributed decryption of BGV and it might not be that $c_m = c'_m$ however it holds that $\sum_i m_i$ is the plaintext of the new cipher c'_m. A full and formal description of the protocol can be found at [3].

SPDZ MAC Checking

When we want to check a MAC value we are in the scenario where a tuple $(a, \gamma(a)_i)$ has been partially opened. The goal is that we want to be able to check whether a is correctly computed without revealing any information on α. Since a is public then the computation $\gamma(a)_i - a\alpha$ is linear over shares. Therefore, parties can locally evaluate $\gamma(a)_i - a\alpha = 0$ without anything being leaked on α. The following protocol describes the procedure in detail:

Protocol 1 MACCheck

Input: party P_i has input $\alpha_i, \gamma(a_j)_i$ for $j \in [1, ..., t]$. Each party have the set $\{a_1, ..., a_t\}$ of opened values.

1. All parties sample a seed s_i. Then they run the commit function of $\mathcal{F}_{commit}$ with s_i as input to have $\tau_i^s \leftarrow Commit(s_i)$ broadcasted.
2. All parties run the open function of $\mathcal{F}_{commit}$ with τ_i^s as input. By this all players receive $s_j \forall j$, where j is over the number of parties.
3. All parties compute $s \leftarrow s_1 \oplus ... \oplus s_n$
4. All parties set s as the seed for the distribution $\mathcal{U}()$ from the BGV scheme and sample a vector $\mathbf{r} \leftarrow \mathcal{U}_s(p, t)$.
5. All parties compute $a \leftarrow \sum_{j1}^{t} r_j a_j$, where r_j is the j'th entry of the vector $\mathbf{r}$.
6. P_i computes: $\gamma_i \leftarrow \sum_{j=1}^{t} r_j \gamma(a_j)_i$ and $\sigma_i \leftarrow \gamma_i - \alpha_i a$.
7. Run $\mathcal{F}_{commit}$ as in steps 1,2 but for σ_i. By this all parties $\sigma_j \forall j$, where j is over the number of parties.
8. If $\sigma_1 + ... + \sigma_n = 0$ output accept else output $\perp$ and reject.

We have that the protocol is correct as in step 8 we are summing all the σ shares which is the same as revealing sigma. So the computation is essentially:

$$\sum_{j=1}^{t} r_j \gamma(a_j)_i - \alpha_i a$$

$$\Updownarrow$$

$$\sum_{j=1}^{t} r_j \gamma(a_j)_i - \sum_{j=1}^{t} r_j a_j \alpha_i$$

$$\Updownarrow$$

$$\sum_{j=1}^{t} r_j \gamma(a_j)_i - \sum_{j=1}^{t} r_j \gamma(a_j)_i = 0$$

Theorem 9.1. *The protocol MACCheck is correct and sound. Sound meaning it outputs reject except with probability $\frac{2}{p}$ if one or more values or MACs are incorrect.*

Proof. To show the probability bound a game is constructed:

1. A challenger generates a secret key $\alpha = \alpha_1 + ... + \alpha_n$ and MACs $\gamma(a_j)_i = \alpha a_j$. It then sends messages $a_1, ..., a_t$ to an adversary.

2. The adversary replies with $a'_1, ..., a'_t$.
3. The challenger generates random values $r_1, ..., r_t \in_R \mathbb{F}_p$ and sends them to the adversary.
4. The adversary chooses an error value Δ.
5. Define $a = \sum_{j=0}^{t} r_j a'_j, \gamma_i = \sum_{j=0}^{t} r_j \gamma(a_j)_i$ and $\sigma_i = \gamma_i - \alpha_i a$.
6. The challenger checks that $\sigma_1 + ... + \sigma_n = \Delta$

The adversary wins if the challenger accepts in the case where one or more $a_j \neq a'_j$. Step 2 simulates the fact that the adversary can lie about the shares he is opening and Δ simulates the fact that he can introduce errors in the mac values.

If the game passes then we have:

$$\begin{aligned}
\Delta = \sum_{i=1}^{n} \sigma_i &= \sum_{i=1}^{n} (\gamma_i \alpha_i a) \\
&= \sum_{i=1}^{n} (\sum_{j=1}^{t} r_j \gamma(a_j)_i - \alpha_i \sum_{j=1}^{t} r_j a'_j) \\
&= \sum_{i=1}^{n} (\sum_{j=1}^{t} (r_j \gamma(a_j)_i - \alpha_i r_j a'_j) \\
&= \sum_{j=1}^{t} (r_j \sum_{i=1}^{n} (\gamma(a_j)_i - \alpha_i a'_j)) \\
&= \sum_{j=1}^{t} r_j (\alpha a_j - \alpha a'_j) \\
&= \alpha \sum_{j=1}^{t} r_j (a_j - a'_j)
\end{aligned}$$

If we are considering the case where $\sum_{j=1}^{t} r_j(a_j - a'_j) \neq 0$ then it implies $\alpha = \Delta / \sum_{j=1}^{t} r_j(a_j - a'_j)$. So if the adversary can pass in this case then he can guess the secret key α, which has a probability of $1/|\mathbb{F}_p|$ given the adversary has no information about α. For the other case $\sum_{j=1}^{t} r_j(a_j - a'_j) = 0$ we define $\mu_j = (a'_j - a_j), \mu = (\mu_1, .., \mu_t), r = (r_1, ..., r_t)$. Note taht at least one μ_i is not 0. From this a linear mapping can be defined $f_\mu(r) = r\mu = \sum_{j=}^{t} r_j \mu_j$. From the **rank-nullity theorem** [3] we get $dim(ker(f_\mu) = t - 1$. Since the adversary chooses his values a'_j independent of r then the probability that $r \in ker(f_\mu)$ is $|\mathbb{F}_p^{t-1}| / |\mathbb{F}_p^t| = 1/|\mathbb{F}_p|$.

Now combining the probabilities from the two cases we get that the adversary wins with probability $2/|\mathbb{F}_p|$. [3] □

SPDZ Offline Phase

For the offline phase it is assumed that a BGV key is distributed to all parties, as well as shares of a secret MAC key and an encryption of the MAC key. We here go over the protocol for production and checking of multiplication triples and bits [3]. The reason that we omit the others is because we only use these two in our implementation. So the rest is irrelevant for the scope of this project.

Protocol 2 TripleProduction

Input: Takes an integer n_m as input which denotes number of tuples to produce.

Output: Outputs at least $2 \cdot n_m$ tuples. Overproduced because some will be sacrificed during checking.
Triples: for $k \in [1, .., 4 \cdot n_m]$, do:

1. All parties run EncCommit of $\mathcal{F}_{SHE}$ with R_p as input. Then party P_i get plaintext a_i and all players get $c_i \leftarrow enc(a_i)$.
2. All parties locally compute $c_a \leftarrow c_{a_1} + ... + c_{a_n}$. Let $a = \sum_{i=1}^{n} a_i$ without it being computed.
3. Same as step 1. and 2. however, it happens in the way that P_i gets plaintext b_i instead of a_i and other parties get $c_i \leftarrow enc(b_i)$.
4. All parties compute $c_{ab} \leftarrow c_a \cdot c_b$
5. All parties run ReShare($c_{ab}, NewCipherText$), which secret shares the plaintext of c_{ab} and they obtain the ciphertext encrypting ab. Denote the new cipher as c_c.
6. All parties compute $c_{\gamma(a)} \leftarrow c_a \cdot c_\alpha$, $c_{\gamma(b)} \leftarrow c_b \cdot c_\alpha$ and $c_{\gamma(c)} \leftarrow c_c \cdot c_\alpha$.
7. All parties run the ReShare procedure on inputs $(c_{\gamma(a)}, NoNewCiphertext)$, $(c_{\gamma(b)}, NoNewCiphertext)$, $(c_{\gamma(c)}, NoNewCiphertext)$.

Protocol 3 BitProduction

Input: Takes an integer n_b as input which denotes number of bits to produce.

Output: Outputs at least n_b bit sharings.

Bits: for $k \in [1, .., 2 \cdot n_b + 1]$, do:

1. All parties run $EncCommit$ of $\mathcal{F}_{SHE}$ with R_p as input. Then party P_i gets a plaintext a_i and all parties get $c_i \leftarrow enc(a_i)$.
2. All parties locally compute $c_a \leftarrow c_{a_1} + ... + c_{a_n}$. Let $a = \sum_{i=1}^{n} a_i$ without it being computed.
3. All parties compute $c_{a^2} \leftarrow c_a \cdot c_a$.
4. Parties run the distributed decryption described in BGV. All parties then get $s = a^2$.
5. Think of s as a vector. If any entry in s is 0 set it to 1.
6. A fixed square root t of s is taken (either the negative or positive one modulos p).
7. Compute $c_v \leftarrow t^{-1} \cdot c_a$ which is an encryption of $t^{-1} \cdot a = v$. We get that $v \in \{-1, 1\}^n$.
8. All parties compute $c_{\gamma(v)} \leftarrow c_v \cdot c_\alpha$.
9. All parties run the ReShare procedure on inputs $(c_{\gamma(v)}, NoNewCiphertext)$ and $(c_v, NoNewCiphertext)$.
10. For each entry i output $[b_i] \leftarrow \frac{1}{2} \cdot ([v_i] + 1)$

Protocol 4 TripleCheck

Setup: All parties have agreed on values t_m and t_{sb}.

Input: takes a number n_m which is the number of checked tuples that will be returned
Output: is at least n_m triples such that for a triple (a_i, b_i, c_i) it holds that $c_i = a_i \cdot b_i$.
TripleCheck: for $k \in [1, .., n_m]$, do:

1. Two unused multiplication triples are sampled. Let's note them $(\langle a\rangle, \langle b\rangle, \langle c\rangle), (\langle d\rangle, \langle e\rangle, \langle f\rangle)$.
2. The values $t_m \cdot \langle a\rangle - \langle d\rangle$ and $\langle b\rangle - \langle e\rangle$ are partially opened. Let the revealed results be noted ρ, σ respectively.
3. Compute and partially open the value $t_m \cdot \langle c\rangle - \langle f\rangle - \sigma \cdot \langle d\rangle - \rho\langle e\rangle - \sigma\rho$. Let the revealed result be τ.
4. If $\tau = 0$ output accept else output $\perp$ and reject.

To briefly show correctness of step 4 in TripleCheck we compute the following:

$$t_m \cdot \langle c\rangle - \langle f\rangle - \sigma \cdot \langle d\rangle - \rho\langle e\rangle - \sigma\rho$$
$$\Updownarrow$$
$$t_m\langle c\rangle - \langle f\rangle - (\langle b\rangle - \langle e\rangle)\langle d\rangle - (t_m\langle a\rangle - \langle d\rangle)\langle e\rangle - (t_m\langle a\rangle - \langle d\rangle)(\langle b\rangle - \langle e\rangle)$$
$$\Updownarrow$$
$$t_m\langle c\rangle - \langle f\rangle - \langle b\rangle\langle d\rangle + \langle e\rangle\langle d\rangle - t_m\langle a\rangle\langle e\rangle + \langle d\rangle\langle e\rangle - t_m\langle a\rangle\langle b\rangle + t_m\langle a\rangle\langle e\rangle + \langle d\rangle\langle b\rangle - \langle d\rangle\langle e\rangle$$

In the last equation everything cancels out and becomes 0.

Protocol 5 BitCheck

Setup: All parties have agreed on values t_m and t_{sb}.

Input: takes a number n_b which is the number of checked bits that will be returned.
Output: is at least n_b checked bits such that $b_i \in \{0, 1\}$.

BitCheck: for $k \in [1, .., n_b]$, do:

1. Take unused squaring tuple and a bit sharing. Let's note them $(\langle a\rangle, \langle b\rangle), \langle c\rangle$.
2. Partially open the value $t_{sb} \cdot \langle c\rangle - \langle a\rangle$. Denote the revealed result as ρ.
3. Compute and partially open $t_{sb}^2 \cdot \langle c\rangle - \langle b\rangle - \rho(t_{sb} \cdot \langle c\rangle + \langle a\rangle)$. Let the revealed result be τ.
4. If $\tau = 0$ output accept else output $\perp$ and reject.

To briefly show correctness of step 4 in BitCheckwe compute the following:

$$
\begin{array}{l}
t_{sb}^2\langle c\rangle - \langle b\rangle - \rho(t_{sb}\langle c\rangle + \langle a\rangle) \\
\Updownarrow \\
t_{sb}^2\langle c\rangle - \langle b\rangle - (t_{sb}\langle c\rangle - \langle a\rangle)(t_{sb}\langle c\rangle + \langle a\rangle) \\
\Updownarrow \\
t_{sb}^2\langle c\rangle - \langle b\rangle - t_{sb}t_{sb}\langle c\rangle\langle c\rangle - t_{sb}\langle c\rangle\langle a\rangle + t_{sb}\langle c\rangle\langle a\rangle + \langle a\rangle\langle a\rangle
\end{array}
$$

In the last equation everything cancels out and becomes 0.

9.2 Secure Comparison Protocols

In the ring signature scheme from [10] it can be seen that we need to compare with the signer's index. Given that it is an important part of the protocol to hide the sender's index we cannot let its index be a public value. Therefore, as we need to do computations with the signer's index we create additive shares of it. To actually do the check whether $i = s$ we will have the parties participating in the following protocols:

Protocol 6 PRandM(k, m)

Input: on input k, m each participates in the following computations:

```
1. $[r''] \in_R [0, ..., 2^{k+\kappa-m} - 1]$
2. $\forall i \in [0, .., m-1]$, do: $[b_i] \in_R \{0, 1\}$
3. $[r'] \leftarrow \sum_{i=0}^{m-1} 2^i [b_i]$
4. Return $[r''], [r'], [b_0], .., [b_{m-1}]$
```

Protocol 7 EqualityTesting([a], k)

Input: on input $[a], k$ each participates in the following

```
1. $[r''], [r'], [b_0], ..., [b_{k-1}] \leftarrow PRandM(k, k)$
2. $c \leftarrow Open([a] + 2^k [r''] + [r'])$
3. $(c_0, ..., c_{k-1}) \leftarrow bit\_repr(c, k)$
4. $\forall i \in [0, .., k-1]$, do: $[d_i] \leftarrow c_i + [b_i] - 2c_i[b_i]$
5. $[z] \leftarrow 1 - KOR([d_0], .., [d_{k-1}])$
```

These two protocols are from [2] where we have made some modifications to them. For Protocol 2 in the public opening at step 2 we have changed it from:

$$c \leftarrow Open(2^{k-1} + [a] + 2^k [r''] + [r'])$$

The reason for this modification is by including this 2^{k-1} the reduction didn't work as we would have that term leftover.

Now for parties to do the equality testing, Protocol 2, the first step is to call $PRandM(k, k)$ which returns a value $[r'']$, k bits $[b_0], ..., [b_{k-1}]$ and $[r']$ which is computed as $[r'] = \sum_{i=0}^{k-1} 2^{[b_i]}$. The next step is to compute the share $[c] = [a] + 2^k [r] \cdot [r']$ and open it.

Now let $c_0, ..., c_{k-1}$ be the k least significant bits. The next step for each party is to compute XOR for each bit c_i and $[b_i]$ and store the result in $[d_i]$. They then outputs $1 - KOR([d_0], ..., [d_{k-1}]$.

The way that we have done the KOR operation is that we have parties compute an OR of two shares by

$$OR([d_i], [d_{i+1}]) = [d_i] + [d_{i+1}] - [d_i][d_{i+1}] \tag{14}$$

We start by having $i = 0$ and then storing the result and calling OR with the result and the next value until we reach the end. This makes it so there will be $log(k)$ operations done.

Now if $a = 0$ then the first k bits of c are all equal to the bits of r'. Furthermore, since d_i is computed to be the XOR of c_i and b_i we get that all d_i are 0 and the output will be 1 in the case where $a = 0$ and otherwise 0.

9.3 Full MPC Protocol

Protocol 8 RingSignatureOverMPC Ideal Functionality

Input/params: a set of public keys S, equality testing parameters k, κ, secret shared signer index $\langle s \rangle$, secret shared secret key $\langle sk \rangle$ and the key image I

Signing step: let $m = |S|$ then on input $(msg, \langle s \rangle, \langle sk \rangle, S, I, \{\langle q \rangle\}^m, \{\langle w \rangle\}^m)$:
$\forall i \in [0, .., m-1]$, do:

1. $\langle a \rangle \leftarrow \langle s \rangle - i$
2. $\langle b_i \rangle \leftarrow EqualityTesting(\langle a \rangle, k, \kappa)$
3. Compute and store:
   $\langle L_i \rangle \leftarrow \langle q_i \rangle G + \langle w_i \rangle (1 - \langle b_i \rangle) P_i$
   $\langle R_i \rangle \leftarrow \langle q_i \rangle \mathcal{H}_p(P_i) + \langle w_i \rangle (1 - \langle b_i \rangle) I$

$\forall i \in [0, .., m-1]$, do: $Open(\langle L_i \rangle, \langle R_i \rangle)$
Compute the challenge $c \leftarrow \mathcal{H}_s(msg, L_0, .., L_{m-1}, R_0, .., R_{m-1})$

$\forall i \in \langle 0, ..., m-1]$, do:

4. Compute and store:
   $\langle c_i \rangle \leftarrow \langle b_i \rangle (c - (\sum_{j=0}^{m-1} w_j \text{mod } l) + \langle w_i \rangle) + \langle w_i \rangle (1 - \langle b_i \rangle)$
   $\langle r_i \rangle \leftarrow \langle q_i \rangle - \langle c_i \rangle \langle sk \rangle \langle b_i \rangle$

Lastly, we can construct the signature and open it by:
$\langle \sigma \rangle \leftarrow (I, \langle c_0 \rangle, .., \langle c_{m-1} \rangle, \langle r_0 \rangle, .., \langle r_{m-1} \rangle)$
$\sigma \leftarrow Open(\langle \sigma \rangle)$

Verification step: on input (σ, S, msg):
$\forall i \in [0, .., m-1]$, do:

5. Compute:
   $L'_i \leftarrow r_i G + c_i P_i$
   $R'_i \leftarrow r_i \mathcal{H}_p(P_i) + c_i I$

Return the evaluation of $\sum_{i=0}^{n} c_i == \mathcal{H}_s(msg, L'_0, ..., L'_{m-1}, R'_0, ..., R'_{m-1})$

The MPC protocol follows the structure of the ring signature scheme. The simpler changes is that the operations will now occur over shares instead of the values themselves. Though instead of sampling the q and w values we take them in as input as shares. This is done as in the implementation part we run an offline phase where we do some pre-computation.

This protocol is only considering the actual signing part of the signature scheme. Which makes it so that the key generation is excluded from it. Therefore, it is assume that in practice there will already be some existing public key / private key pairs.

The input that we are giving to the equality testing protocol is $\langle a \rangle = \langle s \rangle - i$ where $\langle s \rangle$ is a share of the senders secret index and i is the current public key's index in the set S. The protocol checks and compares if $\langle a \rangle = 0$, since if $\langle a \rangle = 0$ then $\langle s \rangle = i$, after opening $\langle s \rangle$ of course, and if we are in this case the equality testing returns $\langle 1 \rangle$ otherwise $\langle 0 \rangle$.

The bigger changes comes to the L and R transformations as well as when computing the responses. The changes include the bit sharings that are output from the equality testing. Instead of having two cases for the values that they are put together into one single computation. By this, we are still able to have the property of hiding the signer's identity.

Another thing to note is that we also excluded the LNK part of the scheme. This is because this is some local computation that any party can run anytime using the common input available.

10 Implementation

For the implementation we have based it off a library called "MP-SPDZ: Versatile framework for multi-party computation" [6]. This is a very large library with the aim to run the same computation over different protocols. It allows for many different MPC protocols to be run.

When looking at the docs the library seems mostly to cover the features of the high-level approach which is using python for the implementation. For this project we have used the low-level approach and used the c++ interface. Our primary source of documentation has been the github for MP-SPDZ and the documentation for MP-SPDZ. Throughout our implementation we experience that not all relevant low-level features / functions where listed in the documentation. Therefore, we had to spent quite a lot of time on reading the source code and trying to figure out a way to do what we wanted to accomplish. However, we didn't always succeed but in these cases we had the option to reach out and ask Marcel, the contributor of the library.

What we have accomplished is to implement the relevant parts of the signature scheme described in protocol 8 in section 9.3. We have the implementation split into an offline and online phase, however it should be noted that some multiplication triplets are consumed in the offline phase as this is where the equality testing is also implemented.

The generel top-level structure of the MP-SPDZ library, with the most relevant directories included - rest omitted, is the following:

```
MP-SPDZ
├── ECDSA
├── MATH
├── Networking
├── OT
├── Player-Data
├── Protocols
└── Processor
```

The folders that we have primarily been directly working with is OT, Protocols, Player-Data, Processor. One thing to note is that Player-Data is not a folder that is there by default however, it is necessary to be generated to run some protocols. The OT folder contains files that are related to supporting oblivious transfer.

The protocols folder contains the files that support the different share types that are in the library as well as the protocols that are used to handle multiplication of shares etc such as a Beaver implementation. Furthermore, this is also where the MAC checking part of the library is implemented with the primary implementation residing in MAC_Check.h and MAC_Check.hpp.

In those files are the check functions that are executed when the MCp.check() functions is called in the code. However, when we in the code call protocol.check then this is a function that is specific to the protocol in our case Beaver.

The Player-Data folder is the default set name for the folder which will store all preprocessing data that is generated for later use in the online phase. This is not required if you're running a protocol which computes the preprocessing data between parties in the execution. It is strictly required for implementations that only run the online phase of a protocol.

10.1 Implementation Workflow

From what we experienced the general implementation work flow, if you want to implement a protocol and have it benchmarked, is the following:

1. Decide which curve you want to use.
2. Create preprocessing file which implements the offline phase.
3. Create sign file which implements the online phase
4. Create entry point file with a main function that runs a setup function, usually called run().
5. Create file that implements the setup in a run() function.

This is the idea of the ECDSA folder in the library from our perspective and this is how we went about it. On further inspection ECDSA has the following 'core' files:

```
ECDSA
├── CurveElement.*
├── P256Element.*
├── preprocessing.hpp
├── sign.hpp
└── EcdsaOptions.h
```

Besides the preprocessing and sign described above there is the CurveElement and P256Element that both are implementations of curves. The P256Element file uses openssl's implementation of secp256k1 curve, which can be exchanged with any other curve supported by the openssl library, whereas CurveElement uses libsodium's implementation of ristretto255, which is build on top of the Curve25519 curve.

Risretto255 efficiently implements a prime-order group by dividing the curve's cofactor by 4 or 8, and by using Risretto255 it possible for systems using Ed25519 signatures, which is used in the ring signature scheme, to be safely extended with zero-knowledge protocols.

Then there is the EcdsaOptions file which is a command-line parser class, that is used to set different flags, when using the various protocols. Fake-ECDSA which uses Protocols/fake-stuff to generate preprocessing data into a local folder - by default Player-data.

Besides the files mentioned here there are some files which are omitted that act as entry points to running the protocols. We will go into depth with these when explaining our own implementation.

10.2 Our code

Our code is found in our thesis repository. In the repository we have included a simple bash script called make.sh which can be run where after you can run "mk" in the shell and have our project compiled into the MP-SPDZ directory. When doing this the folder structure should be the following:

```
top-level-dir
├── MP-SPDZ
└── Thesis
```

and the MP-SPDZ folder should have a folder called RSIG inside.

The main implementation is found in the folder called RSIG and the stuff in the top directory is some files that we have modified from the MP-SPDZ library to make our code work. The modified files are MascotPrep.h, MascotPrep.hpp, Data_files.h and fake-stuff.hpp.

For the first three files they are modified with the purpose of support generation of secret shared bits. This is strictly of OT which did not seem to have a buffer_bits method. Therefore using the modification the protocol OT would then have a method called buffer_bits in addition to its buffer_triples. The way the actual bit creation is implemented is taking the "MascotTriplePrep" class which is the one invoked from the library and adding buffer_bits to it by taking the buffer_bits code from the "MascotDabitOnlyPrep" class.

The last file fake-stuff is which is primarily used for the fake offline phase, i.e. the one where preprocessing data is stored on disk and used later. We have added two methods to it which are make_bit and make_sk_share. The make_bit function loads the secret sharing of the bits that is used in the equality testing prototol, whereas make_sk_share is used to secret share the senders secret key to the other parties. It is implemted with a side effect such that it also generates a random scalar. This is used to sample the random r value which is used in the equality testing protocol.

If we go into the RSIG folder this is where the largest part of our code resides. We have the CurveElement file which is the same as the one in the ECDSA folder from the library with some modifications.

Since we are working with a whitepaper we wanted to emulate the setup so we created a file called transaction.* with the purpose of emulating the blockchain setup as is in the whitepaper. Specifically we are calling the "genTransaction" function from it which creates a "blockchain" object, adds some fake transactions to it and outputs a "SignatureTransaction" which is the object that we will hash in the signature scheme.

Lastly we get to the files that implements the protocol. They are structured in the way such that some of the files act as an entry point to the program. These are the files which includes calls the run functions, an example is the file mascot-rsig-party.cpp.

The run function is implemented in different files depending on the use case. As an example mascot calls the run function in ot-rsig-party.hpp as it uses OT as the underlying protocol in our code. We also have hm-rsig-party.hpp which also has the run function but instead of OT it uses honest-majority and lastly we have fake-spdz-rsig-party which has the content of a run function.

As apparent in the RSIG folder compared to the ECDSA folder we have a lot of files. This is due to the fact that we could not have generic OT-rsig-party / hm-rsig-party which could take any generic share type. This is because not all share types has the same interface for accessing the share value and connected MAC value. The library is made such that if the share type is only a single value then you have to use assignment - the case of Shamir secret share and if the share type does have several values such as the replicated secret share they can be accessed using subscript / indexing. Lastly, the type called Share has getter and setter methods for accessing the share and MAC values.

For an overview of the structure we have the following table:

Protocol Run	Parties	Entry File	Preprocessing Setup File	Share/MAC access
Mascot	2	Mascot-rsig-party	ot-rsig-party	Getter/setter
Fake	2	nan	fake-spdz-rsig-party	Getter/setter
Semi-Honest OT	2	semi-rsig-party	ot-semi-rsig-party	Assignment
Malicious Shamir	3	mal-rsig-party	hm-rsig-shamir-party	Assignment
Semi-Honest Shamir	3	shamir-rsig-party	hm-rsig-shamir-party	Assignment
Malicious Replicated	3	mal-rep-rsig-party	hm-rsig-party	Indexing
Semi-Honest Replicated	3	rep-rsig-party	hm-rsig-party	Indexing

If we take a look at fake-spdz-rsig-party as an example there is a parameter that needs to be passed into the Names class, this is for each setup file. The parameter is a class you would have to make called "SigningSchemeName"Options. We have included a file called RSIGOptions that is passed to the Names class in the "preprocessing setup files". However, this RSIGOptions does not have any influence on the code run. As a note it has the same content as ECDSA/EcdsaOptions.h. In general this class is used along side ezOptionParser which is a command line parser class that is used to set different flags in various protocols using OT as the underlying protocol. As an example the library's ECDSA implementation has a flag that decides when to run the multiplication i.e. the 'prep_mul' flag.

Further in the process we have that the "Preprocessing setup files" run the preprocessing funciton from preprocessing.hpp and the sign_benchmark function from sign.hpp.

The preprocessing function implements the offline phase of our MPC scheme of the whitepaper, which would correspond to step 1,2 and 3 of protocol 8. Our implementation is not a "true" offline phase as it also consumes some multiplication triples to do the equality testing.

The online phase of our MPC is implemented in the sign.hpp file and referencing protocol 8 it would correspond to step 4 and 5.

When considering the parameters for the equality testing in the offline phase then we have set the value of k to be 40 and we have the value of κ being 0 / not set.

Running our code It should be noted that for running some of the malicious protocols SSL keys are required therefore run to test the code one should be in the MP-SPDZ directory and run "Scripts/setup-ssl.sh [nparties]"

After compiling there will be generated some executables. Some of them requires 2 parties to be run and others require 3 parties. Table **reference** shows the mapping from protocol to executable. As an example to run mascot which is a 2 party implementation you would have to run with the following: "./mascot-rsig-party.x -p 0" and "./mascot-rsig-party.x -p 1" in two different shells.

In general the idea for MP-SPDZ is that each party requires one TCP port to be open to other parties. By default party 0 opens on port 5000, party 1 opens on port 5001 and so on as the party number increases. If wanted then this default behaviour can be overwritten using the –portnumbase flag to set a new default base port or individually for parties with the –my-port flag.

Protocol	Executeable
FAKE	fake-spdz-rsig-party.x
Mascot	mascot-rsig-party.x
Semi-Honest OT	semi-rsig-party.x
Malicious Shamir	mal-shamir-rsig-party.x
Semi-Hoenst Shamir	shamir-rsig-party.x
Malicious Replicated	mal-rep-rsig-party.x
Semi-Honst Replicated	rep-rsig-party.x

Table 1: Protocol to executable mapping

10.3 Evaluation

By using this specific library we had the option to evaluate our code with different protocols. Below we are listing some benchmarks for different parts of our implementation over different protocols. We have split it up into two tables. 3 presents the results for the offline phase and 2 presents the online phase. All of the results are acquired from running our code 1000 times for each protocol and the data shown is in milliseconds.

For the offline phase we have measured most of the equality testing protocol as well as the generation of q,w,L and R values to find the possible bottlenecks in the implementation. For the equality testing we measured PRANDM, the KOR and the total time for the equality testing. For the online phase we measured the verification time, signing times, MAC checking time after signing and verfication and the amount of data sent in bytes.

Offline Benchmarking We have testing it locally on a PC where we got the following results:

Share Type	PRANDM	KOR	EQ Total	q, w, L, R
FAKE	0.188	28.686	29.358	4.024
MASCOT	547.315	503.571	1051.363	24.791
Semi-Honest OT	84.057	78.716	163.035	5.735
Malicious Shamir	2.354	37.114	39.829	2.322
Semi-Honest Shamir	0.986	35.281	36.618	2.215
Malicious Replicated	1.923	40.891	43.307	4.045
Semi-Honest Replicated	0.986	17.766	19.158	4.061

Table 2: Average time in milliseconds for offline phase

As we can see from the results the biggest bottleneck seems to be the place where we consume the multiplication triples in the equality testing, namely the KOR.

If one tries to compare the time in the EQ Total column and the two other EQ related columns then we can see it doesn't add up i.e. $EQTotal \neq KOR + PRANDM$. This is because we have not included measurements of the fast / non interesting parts of the equality testing protocol as we are specifically looking for bottlenecks.

It can also be seen that we have some outliers primarily OT where both PRANDM and the KOR takes around the same time. For these results it should be noted that they used the "fix" (buffer_bits) we made to the library and not something that existed already. Therefore, we strongly believe that our "fix" is very inefficient and not representative of the true time a good implementation would take.

It is also apparent that the implementation which only runs the online phase is the fastest which makes sense as it would only have the communication rounds for the KOR.

If we look at the best performing share type across the columns, ignoring fake, it is semi-honest replicated that is the best one.

Online Benchmarking When looking at the online phase we have the results in the following table:

Share Type	Verf	Sign	Online Checking	Online Check Data Sent
FAKE	1.571	1.974	0.064	120 bytes
MASCOT	1.225	34.016	0.021	120 bytes
Semi-Honest OT	1.736	5.63	0	0 bytes
Malicious Shamir	2.102	6.512	0	0 bytes
Semi-Honest Shamir	2.063	3.01	0	0 bytes
Malicious Replicated	1.915	1.403	0.013	64 bytes
Semi-Honest Replicated	1.806	0.252	0	0 bytes

Table 3: Average time in milliseconds for online phase

We see that most signing and verf times take from 1-6 milliseconds on average. However there are some that lie outside this range such as semi-honest replicated signing which on average took 0.252 milliseconds. Then we have malicious shamir which is 6.512 milliseconds and then we have Mascot which lies way outside the range with an average signing time of 34.016 milliseconds.

The only thing we can contribute to this is the fact that it uses the buffer_bits that we put in and then an access method of getters and setters. Besides this all the protocols do run the exact same code for the online phase.

It can also be seen that for the MAC checking after sign / verf some protocols have that it takes 0 milliseconds, this is not entirely true as individual measures are around 0.006 milliseconds. It is probably reported as 0 as when we in the end take the average the measurements become insignificant. However we also see that there is sent 0 bytes which is indeed true as the code would output something like, "Online checking took 0.0063 ms and sending 0 bytes", for those protocols.

Between Machines We have limited benchmarking results between machines. We only have a successful benchmarking of the fake offline phase. We will just describe the result for the fake. PRANDM ended up taking 0.63 milliseconds, KOR took 636.7557 and EQ in total took 639.9355 milliseconds. The generation of q, w, L and R took 5.688 millisecond. The average signing took 31.656 milliseconds and the average verification took 4.537 milliseconds. The average online checking took 7.263 milliseconds and for the data sent in the online checking was the same. We note that the results are as expected in the sense that the computation does take a longer time.

We have not had the option of testing the other protocols. The reason for this is that we at the point of having access to a network that did not directly deny the traffic the code sends we only had working implementations of Mascot and FAKE.

When we would try to run MASCOT we would receive MAC check failure and we attribute this to the buffer_bits implementation that we had for OT.

Later on in the project we ended up having working implementations of the other protocols as well however at that point Martin moved back home and therefore no longer had access to that network.

The networks we have tried to run the code on is the university network, Martin's current network, mobile data, Martin's old network, Michael's network and Michael's parent's network. Out of all of these we only had success with the network in place in Martin's old apartment.

Missing from implementation

As mentioned earlier the biggest bottleneck that we had in the code was the consumption of multiplication triples in the equality testing protocol. The obvious solution to this is to multi-thread our implementation. However, we were not able to figure out how to do so with the library. We asked Marcel, the contributor of the library, about how we ought go about this as we couldn't make it work with the standard thread library. He said:

> There are two aspects to consider:Most instances have to be separate per thread such as *Player, *Prep, SubProcessor. If using OT, you have to set BaseMachine::thread_num differently for every thread and populate BaseMachine::ot_setups such that BaseMachine::ot_setups[thread_num] is available.

However since the population happens automatically we would only need to set thread_num. However, even using this approach we could not manage to get the multi-threading to work. We had various attempts just to mention some then we tried to create two basemachine giving each a unique thread_num matching the thread_num set for the ArithmeticProcessor and passing that along. We also tried to pass the beaver protocol by reference into regular threading. These are just some of the things we tried. Mostly, we would encounter MAC Check failures or insufficient data errors from our attempts.

In general we also tried to play around with the library ourselves and we found code bits that made us somewhat confused as to whether or not we would have to comment out code bits as some functions that were called had comments saying "* is not suitable in threaded programs". This was mostly for the timing features in the library however, when we tried to comment out the code bit we ended up getting other errors and so on all the way down to the virtual machine's instruction handling so we gave up on these approaches.

It would also have been interesting to try to run the code with share types of Soho and HighGear as well. The reason for this is they should be using the BGV encryption briefly explained earlier. We tried to play around with the

Soho share but when we tried to include say "Protocols/SohoPrep.h" the compilation would throw 17 errors related to SohoPrep.h with some of them being: "./Protocols/SohoPrep.h:13:25: error: no template named 'SemiHonestRingPrep' class SohoPrep : public SemiHonestRingPrep<T>"
"./Protocols/SohoPrep.h:25:27: error: use of undeclared identifier 'usage' BufferPrep<T>(usage),".
We are quite sure that we did not introduce these and therefore we chose to spend our efforts elsewhere.

We also had the issue of how to actually share the secret signer index. For this purpose we should be able to use secret input sharing features of the library however when this was initialised it would generate its own global MAC key which would be different from the one used in the rest of the protocol which in turn would cause MAC check failures. As we did not find any way of actually getting the global MAC key used in either part we choose to share the secret signer index using the constant() function. We also had to open the signer's secret key in the start of the implementation, meaning everyone learns the secret key, to be able to compute the corresponding public key and the related key image.

We have chosen to use this approach for the secret signer index and the secret as we wanted to do benchmarking and in a real setting the actual signer would be able to compute the public key and key image before secret sharing his/her public key.

11 Future works

Probably the most relevant current work would be to figure out how to provide working multi-threading for our code using the library. If this would end up working we could get way better benchmarks especially for the measurements of KOR execution. Besides this it would also be great to have included the share types of Soho and Highgear as they closely relate to the theory from [3] that we have included. These two parts mention here, also mentioned in the missing form implementation paragraph, would be the most obvious way to go with the thesis. However, there are also other aspects to consider. We could also take what we have implemented so far and try to implement more functions of Monero or another cryptocurrency and try to benchmark how a fully decentralised threshold cryptocurrency implementation would perform using the library.

We could also take the direction as where we now have an implemented signature scheme it could be interesting to implement the same signature scheme in different libraries and compare things such as ease of the library as well the performance of the different libraries. From this the future work of the project could also potentially help others by having a working example of how to use the library including considerations of why things are implemented as they are.

References

[1] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. Cryptology ePrint Archive, Report 2017/1066, 2017. `https://ia.cr/2017/1066`.

[2] Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. In Juan A. Garay and Roberto De Prisco, editors, *Security and Cryptography for Networks*, pages 182–199. Springer Berlin Heidelberg, 2010.

[3] Ivan Damgard, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure mpc for dishonest majority – or: Breaking the spdz limits. Cryptology ePrint Archive, Report 2012/642, 2012. `https://ia.cr/2012/642`.

[4] Ivan Damgård. On σ-protocols. *Lecture Notes, University of Aarhus, Department for Computer Science*, 2010. `https://www.cs.au.dk/~ivan/Sigma.pdf`.

[5] Ivan Damgård and Jesper Buus Nielsen. Commitment schemes and zero-knowledge protocols. pages 1 – 15, 2017.

[6] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[7] Sarang Noether koe, Kurt M. Alonso. Zero to monero, 2020.

[8] Russell W. F. Lai, Viktoria Ronge, Tim Ruffing, Dominique Schröder, Sri Aravinda Krishnan Thyagarajan, and Jiafan Wang. Omniring: Scaling up private payments without trusted setup - formal foundations and constructions of ring confidential transactions with log-size proofs. Cryptology ePrint Archive, Report 2019/580, 2019. `https://ia.cr/2019/580`.

[9] Torben Pryds Pederson. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology — CRYPTO '91*, pages 129–140, 1992.

[10] Nicolas van Saberhagen. Cryptonote v 2 - bytecoin, Oct 2013.

[11] Shi-Feng Sun, Man Ho Au, Joseph K. Liu, Tsz Hon Yuen, and Dawu Gu. Ringct 2.0: A compact accumulator-based (linkable ring signature) protocol for blockchain cryptocurrency monero. Cryptology ePrint Archive, Paper 2017/921, 2017. `https://eprint.iacr.org/2017/921`.

[12] Tsz Hon Yuen, Shi feng Sun, Joseph K. Liu, Man Ho Au, Muhammed F. Esgin, Qingzhao Zhang, and Dawu Gu. Ringct 3.0 for blockchain confidential transaction: Shorter size and stronger security. Cryptology ePrint Archive, Paper 2019/508, 2019. `https://eprint.iacr.org/2019/508`.

A Inner-product argument

Let $\mathcal{P}$ be the prover and $\mathcal{V}$ be the verifier. Inner product arguments protocol is:

Protocol 9 ImprovedInner − ProductArgument

Input: $(\mathbf{g}, \mathbf{h} \in \mathbb{G}^n, u, P \in \mathbb{G}, c \in \mathbb{Z}_p; \mathbf{a}, \mathbf{b} \in \mathbb{Z}_p^n)$
$\mathcal{P}$ input: $(\mathbf{g}, \mathbf{h}, u, P, \mathbf{a}, \mathbf{b})$
$\mathcal{V}$ input: $(\mathbf{g}, \mathbf{h}, u, P)$
Output: $\mathcal{V}$ accepts or rejects

$$if\ n = 1 \text{ do steps (2 - 4) else do steps(5 - 11)} \tag{1}$$

$$\mathcal{P} \text{ sends } a, b \text{ to } \mathcal{V} \tag{2}$$

$$\mathcal{V} \text{ computes } c = a \cdot b \tag{3}$$

$$\mathcal{V} \text{ checks if } P = g^a h^b u^c \text{ and accepts if true, otherwise reject} \tag{4}$$

$$\mathcal{P}\text{computes} : n' = \frac{n}{2}, L = \mathbf{g}_{[n':]}^{\mathbf{a}_{[:n']}} \mathbf{h}_{[:n']}^{\mathbf{b}_{[n':]}} u^{\langle \mathbf{a}_{[:n']}, \mathbf{b}_{[n':]} \rangle} \text{ and } R = \mathbf{g}_{[:n']}^{\mathbf{a}_{[n':]}} \mathbf{h}_{[n':]}^{\mathbf{b}_{[:n']}} u^{\langle \mathbf{a}_{[n':]}, \mathbf{b}_{[:n']} \rangle} \tag{5}$$

$$\mathcal{P} \text{ sends } L, R \text{ to } \mathcal{V} \tag{6}$$

$$\mathcal{V} \text{ Chooses x at random } \in \mathbb{Z}_p^* \tag{7}$$

$$\mathcal{V} \text{ sends } x \text{ to } \mathcal{P} \tag{8}$$

$$\mathcal{P} \text{ and } \mathcal{V} \text{ computes} : \mathbf{g'} = \mathbf{g}_{[:n']}^{x^{-1}} \circ \mathbf{g}_{[n':]}^{x}, \mathbf{h'} = \mathbf{h}_{[:n']}^{x} \circ \mathbf{g}_{[n':]}^{x^{-1}} \text{ and } P' = L^{x^2} P R^{x^{-2}} \tag{9}$$

$$\mathcal{P} \text{ computes} : \mathbf{a}' = \mathbf{a}_{[:n']} x + \mathbf{a}_{[n':]} x^{-1}, \mathbf{b}' = \mathbf{b}_{[:n']} x^{-1} + \mathbf{b}_{[n':]} x \tag{10}$$

$$\text{Recursively run protocol 9 on input} : (\mathbf{g'}, \mathbf{h'}, u, P', \mathbf{a'}, \mathbf{b'}) \tag{11}$$

B Range Proof

B.1 Range proof protocol

Let $\mathcal{P}$ be the prover and $\mathcal{V}$ be the verifier. Inner product arguments protocol is:

Protocol 10 RangeProof

Input: $\mathcal{P}$ gets input v, γ:

$$\mathcal{P} \text{ computes the following:} \mathbf{a}_L \in \{0,1\}^n s.t \langle \mathbf{a}_L, 2^n = v \rangle, \mathbf{a}_R = \mathbf{a}_L - 1^n \quad (1)$$
$$\mathcal{P} \text{ chooses } \alpha \in \mathbb{Z}_p \text{ and computes: } A = h^\alpha \mathbf{g}^{\mathbf{a}_L} \mathbf{h}^{\mathbf{a}_R} \quad (2)$$
$$\mathcal{P} \text{ chooses } \mathbf{s}_L, \mathbf{S}_R \in \mathbb{Z}_p^n, \rho \in \mathbb{Z}_p \text{ and computes: } S = h^\rho \mathbf{g}^{\mathbf{s}_L} \mathbf{h}^{\mathbf{s}_R} \quad (3)$$
$$\mathcal{P} \text{ sends } A, S \text{ to } \mathcal{V} \quad (4)$$
$$\mathcal{V} \text{ chooses } y, z \in \mathbb{Z}_p^* \quad (5)$$
$$\mathcal{V} \text{ sends } y, z \text{ to } \mathcal{P} \quad (6)$$
$$\mathcal{P} \text{ chooses } \tau_1, \tau_2 \in \mathbb{Z}_p \quad (7)$$
$$\mathcal{P} \text{ commits to } t_1, t_2 : T_i = g^{t_i} h^{\tau_i} \ i \in \{1,2\} \quad (8)$$
$$\mathcal{P} \text{ sends } T_1, T_2 \text{ to } \mathcal{V} \quad (9)$$
$$\mathcal{V} \text{ Chooses x at random } \in \mathbb{Z}_p^* \quad (10)$$
$$\mathcal{V} \text{ sends } x \text{ to } \mathcal{P} \quad (11)$$
$$\mathcal{P} \text{ computes the following: } \quad (12)$$
$$\mathbf{l} = l(x) = \mathbf{a}_L - z \cdot \mathbf{1}^n + \mathbf{s}_L \cdot x \quad (13)$$
$$\mathbf{r} = r(x) = \mathbf{y}^n \circ (\mathbf{a}_R + z \cdot \mathbf{1}^n + \mathbf{s}_R \cdot x) + z^2 \cdot \mathbf{2}^n \quad (14)$$
$$t = \langle \mathbf{l}, \mathbf{r} \rangle \quad (15)$$
$$\tau_x = \tau_1 \cdot x + \tau_2 \cdot x^2 + z^2 \cdot \gamma \quad (16)$$
$$\mu = \alpha + \rho \cdot x \quad (17)$$
$$\mathcal{P} \text{ sends } t_x, \mu, t, \mathbf{l}, \mathbf{r} \text{ to } \mathcal{V} \quad (18)$$
$$\mathcal{V} \text{ computes } : \quad (19)$$
$$h_i' = h_i^{y-i+1} \forall i \in [1, n] \quad (20)$$
$$P = AS^x \cdot \mathbf{g}^{-z} \cdot \mathbf{h'}^{z \cdot \mathbf{y}^n + z^2 \cdot \mathbf{2}^n} \quad (21)$$
$$\mathcal{V} \text{ checks that } : \quad (22)$$
$$t = \langle \mathbf{l}, \mathbf{r} \rangle \quad (23)$$
$$g^t h^{\tau_x} = V^{z^2} \cdot g^{\delta(y,z)} \cdot T_1^x \cdot T_2^{x^2} \quad (24)$$
$$P = h^\mu \cdot \mathbf{g}^{\mathbf{l}} \cdot (\mathbf{h'}^{\mathbf{r}}) \quad (25)$$

C Correctness proof for BGV encryption and decryption

Correctness proof which shows decryption works and it recovers the message m both in the cases where no one knows the shared secret key and where one party knows it:

$$\begin{aligned}\sum_{i=1}^{n} t_i \bmod p &= \sum_{i=1}^{n}(v_i + p\cdot r_i) \bmod p\\ &= (v_1 + p\cdot r_1) + \sum_{i=2}^{n}(v_i + p\cdot r_i) \bmod p\\ &= (c_0 - s_1\cdot c_1 + p\cdot r_1) + \sum_{i=2}^{n}(-s_i\cdot c_1 + p\cdot r_i) \bmod p\\ &= (((a\cdot s + p\cdot \epsilon)\cdot v + p\cdot e_0 + m) - s_i\cdot(a\cdot v + p\cdot e_1) + p\cdot r_1)\\ &\quad + \sum_{i=2}^{n}(-s_i\cdot(a\cdot v + p\cdot e_1) + p\cdot r_i) \bmod p\\ &= (asv + p\epsilon v + pe_0 + m - s_1av - s_1pe_1 + pr_1) + \sum_{i=2}^{n}(-s_iav - s_ipe_1 + pr_i) \bmod p\\ &= m\end{aligned}$$

In the case where the shared secret key is known:

$$\begin{aligned}m' \bmod p &= [c_0 - s\cdot c_1]_{ql} \bmod p\\ &= [(b\cdot v + p\cdot e_0 + m) - s\cdot(a\cdot v + p\cdot e_1)]_{ql} \bmod p\\ &= [((a\cdot s + p\cdot\epsilon)\cdot v + p\cdot e_0 + m) - s\cdot(a\cdot v + p\cdot e_1)]_{ql} \bmod p\\ &= [asv + p\epsilon v + pe_0 + m - sav - pe_1]_{ql} \bmod p\\ &= p\epsilon v + pe_0 + m - pe_1 \bmod p\\ &= m\end{aligned}$$