Zero to Monero
Privacy in the blockchain: First Edition

Kurt M. Alonso
Jordi Herrera Joancomartí
Koe[1]

[1] Forked from original paper's github 03/20/2018. https://github.com/kurtmagnus/Monero-RCT-report

Abstract

A cryptocurrency blockchain is commonly understood as a public distributed ledger containing transactions verifiable by third parties, be it a mining community or the public in general. It might seem at first glance that transactions need to be sent and stored in clear text format in order to make them publicly verifiable.

As we will show, it is possible to conceal participants of transactions, as well as the amounts involved, using cryptographic artifacts that nevertheless allow transactions to be verified and consensuated by the mining community.

Contents

CHAPTER 1

Introduction

The purpose of blockchains is to furnish trust for operations between unrelated parties, without requiring the collaboration of any trusted third party.

Trust is attained through the use of cryptographic artifacts, which allow data registered in an easily accessible database – the blockchain - to be virtually immutable and non-falsifiable. In other words, a blockchain is a public, distributed database containing data whose legitimacy cannot be disputed by any party.

Cryptocurrencies store transactions in the blockchain, which acts as a public ledger of all the verified currency operations. Most cryptocurrencies store transactions in clear text, to facilitate verification of transactions by the community.

Clearly, an open blockchain defies any basic understanding of privacy, since it virtually *publicizes* the complete transaction histories of its users.

To address the lack of privacy, users of cryptocurrencies such as Bitcoin can obfuscate transactions by using temporary intermediate addresses [20]. However, with appropriate tools it is possible to analyze flows and to a large extent link true senders with receivers [29, 10, 24].

In contrast, the cryptocurrency Monero attempts to tackle the issue of privacy by storing only stealth, single-use addresses for receipt of funds in the blockchain, and by authenticating the dispersal of funds in each transaction with ring signatures. With these methods there are no effective ways to link senders with receivers or trace the origin of funds [1].

Additionally, transaction amounts in the Monero blockchain are concealed behind cryptographic constructions, rendering currency flows opaque.

The result is a cryptocurrency with a high level of privacy.

1.1 Objectives

Monero is a cryptocurrency of recent creation, yet it displays a steady growth in popularity[1]. Unfortunately, there is little comprehensive documentation describing the mechanisms it uses. Even worse, important parts of its theoretical framework have been published in non peer-reviewed papers which are incomplete and/or contain errors. For significant parts of the theoretical framework of Monero, only the source code is reliable as a source of information.

We intend to palliate this situation by collecting in-depth information about Monero's inner workings, reviewing algorithms and cryptographic schemes, and discussing the degree to which they might afford sufficient transaction privacy and security to its users.

We have centered our attention on release 0.11.1.0 of the Monero software suite, the most recent release at the moment this is written. All transaction related mechanisms described here belong to this version. Deprecated transaction schemes have not been explored to any extent, even if they may be partially supported for backward compatibility reasons.

1.2 Readership

We expect the reader to possess a basic understanding of discrete mathematics and algebraic structures, but possibly only fundamental insights in the field of cryptography. We also expect the user to have a basic understanding of how a cryptocurrency like Bitcoin works. For technically oriented laymen we have tried to fill potential knowledge gaps in the footnotes.

A reader with this background should be able to follow our constructive, step-by-step description of the elements of the Monero cryptocurrency.

We have purposefully omitted, or delegated to footnotes, some mathematical technicalities, when they would be in the way of clarity. We have also omitted concrete implementation details where we thought they were not essential. Our objective has been to present the subject halfway between mathematical cryptography and computer programming, aiming at completeness and conceptual clarity.

1.3 Origins of the Monero cryptocurrency

The cryptocurrency Monero, originally known as BitMonero, was created in April, 2014 as a derivative of the proof-of-concept currency CryptoNote.

CryptoNote is a cryptocurrency devised by various individuals. A landmark whitepaper describing it was published under the pseudonym of Nicolas van Saberhagen in October 2013 [30]. It

[1] As of December 28th, 2017, Monero occupies the 10th position as regards market capitalization, see https://coinmarketcap.com/

offered sender and receiver anonymity through the use of one-time addresses, and untraceability of flows by means of ring signatures.

Since its inception, Monero has further strengthened its privacy aspects by implementing amount hiding, as described by Greg Maxwell (among others) in [19], as well as Shen Noether's improvements on ring signatures [23].

1.4 Outline

As hinted earlier, our aim is to deliver a self-contained and step-by-step description of the Monero cryptocurrency. This report has been structured to fulfill this objective, leading the reader through all elements needed to describe the currency's inner workings.

In our quest for comprehensiveness, we have chosen to present all the basic elements of cryptography needed to understand the complexities of Monero. In Chapter 2 we develop essential aspects of Elliptic Curve cryptography.

Chapter 3 outlines the ring signature related algorithms that will be applied to achieve confidential transactions while preventing double-spending attacks.

In Chapter 4 we introduce the cryptographic mechanisms used to conceal amounts.

Finally, with all the components in place, we will be able to expose the transaction schemes used in Monero in Chapter 5.

Appendices A and B explain the structure of sample transactions from the blockchain, providing a connection between the theoretical elements described in earlier sections with their real-life implementation.

CHAPTER 2

Basic concepts

2.1 A few words about notation

One focal objective of this report was to collect, review, correct and homogenize all existing information concerning the inner workings of the Monero cryptocurrency. And, at the same time supply all the necessary details to present the material in a constructive and single-threaded manner.

An important instrument to achieve this was to settle for a number of notational conventions. Among others, we have used:

- lower case letters to denote simple values, integers, strings, bit representations, etc
- upper case letters to denote curve points and complex constructs

For items with a special meaning, we have tried to use as much as possible the same symbols throughout the document. For instance, a curve generator is always denoted by G, its order is l, private/public keys are denoted whenever possible by k/K respectively, etc.

Beyond that, we have aimed at being *conceptual* in our presentation of algorithms and schemes. A reader with a computer science background may feel that we have neglected questions like the bit representation of items, or, in some cases, how to carry out concrete operations.

However, we don't see this as a loss. A simple object such as an integer or a string can always be represented by a bit string. So-called *endianness* is rarely relevant, and is mostly a matter of convention for our algorithms.

Elliptic curve points are normally denoted by pairs (x, y), and can therefore be represented with two integers. However, in the world of cryptography it is common to apply *point compression* techniques, which allow representing a point using only the space of one coordinate. For our conceptual approach it is often accessory whether point compression is used or not, but most of the time it is implicitly assumed.

We have also used hash functions freely without specifying any concrete algorithms. In the case of Monero it will typically be a *Keccak*[1] variant, but if not explicitly mentioned then it is not important to the theory.[2]

These hash functions will be applied to integers, strings, curve points, or combinations of these objects. These occurrences should be interpreted as hashes of bit representations, or the concatenation of such representations. Depending on context, the result of a hash will be numeric, a bit string, or even a curve point. Further details in this respect will be given as needed.

2.2 Elliptic curve cryptography

2.2.1 What are elliptic curves

A finite field $\mathbb{F}_q$, where q is a prime number greater than 3, is the field formed by the set $\{0, 1, 2, ..., q-1\}$. Arithmetic operations $(+, \cdot)$ and the unary operation $(-)$ are calculated $(\text{mod } q)$.[3]

Typically, elliptic curves are defined as the set of points (x, y) satisfying a *Weierstraß* equation for a given (a, b) pair[4]:

$$y^2 = x^3 + ax + b \quad \text{where} \quad a, b, x, y \in \mathbb{F}_q$$

However, the cryptocurrency Monero uses a special curve known to offer improved security over other commonly used *NIST* curves, as well as cryptographic primitives with excellent performance. The curve used belongs to the category of so-called *Twisted Edwards* curves, which are commonly expressed as:

$$ax^2 + y^2 = 1 + dx^2y^2 \quad \text{where} \quad a, d, x, y \in \mathbb{F}_q$$

In what follows we will prefer this second form. The advantage it offers over the previously mentioned Weierstraß form is that basic cryptographic primitives require less arithmetic operations, resulting in faster cryptographic algorithms. See Bernstein et al. in [8] for details.

[1] The Keccak hashing algorithm forms the basis for the NIST standard *SHA-3*.

[2] A hash function takes in some message $\mathfrak{m}$ of arbitrary length and returns a hash h (or *message digest*) of fixed length. Cryptographic hash functions are difficult to reverse, have an interesting feature known as the *large avalanche effect* which causes very similar messages to produce very dissimilar hashes, and make it hard to find two messages with the same message digest.

[3] "calculated $(\text{mod } q)$" means $(\text{mod } q)$ is performed on any instance of an arithmetic operation between two field elements, or negation of a single field element. For example, given a prime field $\mathbb{F}_p$ with $p = 29$, $17 + 20 = 8$ because $37 \ (\text{mod } 29) = 8$. Also, $-13 = (-13) \ (\text{mod } 29) = 16$.

The (positive) modulus is here defined for $a \ (\text{mod } b) = c$ as $a = bx + c$, where $0 \leq c < b$ and x is a signed integer which gets discarded. Imagine a number line. Stand at point a. Walk toward zero with each step $= b$ until you reach an integer ≥ 0 and $< b$. That is c.

[4] Notation: the phrase $a \in \mathbb{F}$ means a is some element in the field $\mathbb{F}$.

Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be 2 points belonging to a Twisted Edwards elliptic curve (henceforth known simply as an EC). We can proceed to define the addition operation $P_1 + P_2 = P_3$ in the following manner[5]

$$x_3 = \frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2} \pmod q$$
$$y_3 = \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2} \pmod q$$

These formulas for addition also apply for point doubling, that is, when $P_1 = P_2$. To subtract a point, invert its coordinates $(x, y) \to (x, -y)$ and do point addition. Whenever a 'negative' element $-x$ of $\mathbb{F}_q$ appears in this report, it is really $-x \pmod q$.

It turns out that elliptic curves have *abelian group* structure[6] under the addition operation described. Each time the operation is performed P_3 is a point on the 'original' elliptic curve, or in other words all $x_3, y_3 \in \mathbb{F}_q$.

Each point P in EC can generate a subgroup of order (size) u out of some of the other points in EC using multiples of itself. For example, some point P's subgroup might have order 5 and contain the points $(0P, P, 2P, 3P, 4P)$, each of which is in EC. At $5P$ the so-called *point-at-infinity* appears, which is like the 'zero' position on an EC and has coordinates $(0, 1)$.

Conveniently, $0P = 5P$ and $5P + P = P$. This means the subgroup is *cyclic*[7]. All P in EC generate a cyclic subgroup. If P generates a subgroup whose order[8] is prime then all the included points (except for the point-at-infinity) also generate that subgroup.

Each EC has an order N equal to the total number of points in the curve including the point-at-infinity, and the orders of all subgroups generated by points are divisors of N (by *Lagrange's theorem*).

[5] $x_3 = (1 + dx_1x_2y_1y_2)^{-1}(x_1y_2 + x_1y_2) \pmod q$ can be calculated using the modulus multiplication and addition properties $(A \circ B) \pmod C = [A \pmod C] \circ [B \pmod C] \pmod C$, and the modular multiplicative inverse. Start inside the parens.

The modular multiplicative inverse is defined as an integer x such that, for $x = a^{-1} \pmod n$, $ax \equiv 1 \pmod n$ for $0 \leq x < n$ and for a and n relatively prime. The extended Euclidean algorithm finds x like this:

$Q = 0$; $newQ = 1$; $R = n$; $newR = a$
while $newR \neq 0$
  $quotient = integer(R/newR)$
  $(Q, newQ) = (newQ, Q - quotient * newQ)$
  $(R, newR) = (newR, R - quotient * newR)$
if $R \leq 1$ then (if $Q < 0$ *return* $Q + n$ else *return* Q) else no solution

Note on the term *congruence*: in the equation $a \equiv b \pmod c$, a is congruent to $b \pmod c$, which just means $a \pmod c = b \pmod c$.

[6] A concise definition of this notion can be found under https://brilliant.org/wiki/abelian-group/

[7] Cyclic subgroup means, for P's subgroup with order u and any integer n, $nP = [n \pmod u]P$.

[8] To find the order, u, of P's subgroup:

1. Find N (e.g. use *Schoof's algorithm*)
2. Find all the divisors of N
3. For every divisor n of N, compute nP.
4. The smallest n such that $nP = 0$ is the order u of the subgroup.

ECs selected for cryptography typically have $N = hl$, where l is some sufficiently large prime number (such as 160 bits). One point in the subgroup of size l (note: h is called the *cofactor*) is usually selected to be the generator G as a convention. For every other point P in the subgroup there exists an integer n satisfying $P = nG$.[9]

Calculating the scalar product nP is not difficult[10], whereas finding n such that $P_1 = nP_2$ is known to be computationally hard. By analogy to modular arithmetic, this problem is often called the *discrete logarithm problem* (DLP). In other words, scalar multiplication can be seen as a *one-way function*, which paves the way for using elliptic curves for cryptography.

2.2.2 Public key cryptography with elliptic curves

Public key cryptography algorithms can be devised in a way analogous to modular arithmetic.

Let k be a randomly selected number satisfying $1 < k < l$, and call it a *private key*. Calculate the corresponding *public key* $K = kG$.

Due to the *discrete logarithm problem* (DLP) we can not easily deduce k from K alone. This property allows us to use the values (k, K) in common public key cryptography algorithms.

[9] Say there is a point P' with order N ($N = hl$). Any other point in EC can be found with $P_i = n_i P'$. If $P_1 = n_1 P'$ has order l, any $P_2 = n_2 P'$ with order l must be in the same subgroup as P_1 because $lP_1 = 0 = lP_2$, and if $l(n_1 P') \equiv l(n_2 P') \equiv NP = 0$, then n_1 & n_2 must both be multiples of h. In other words, the subgroup formed by multiples of (hP') always contains P_1 and P_2. Furthermore, $h(n'P') = 0$ when n' is a multiple of l, and such an n' multiplied by P' can only make h points before $n' = hl$, which cycles back to 0: $hlP' = 0P' = 0$. So, there are only h points in EC where hP will equal 0.

To find a suitable G:

1. Find N of EC, choose subgroup order l, compute $h = N/l$
2. Choose a random point P in EC
3. Compute $G = hP$
4. If $G = 0$ return to step 3, else G generates a subgroup of order l

[10] The scalar product nP is equivalent to $(((P + P) + P)\dots)$. One basic yet powerful algorithm to shorten the calculation of nP is known as *double-and-add*. Let us demonstrate by example. Say $n = 7$, so $nP = P + P + P + P + P + P + P$. Now break this into groups of two. $(P + P) + (P + P) + (P + P) + P$. And again, by groups of two. $[(P + P) + (P + P)] + (P + P) + P$. The total number of + point operations falls from 6 to 4 because $(P + P)$ only needs to be found once.

Double-and-add is implemented by first converting n to binary, then looping through the resultant array to get the sum $Q = nP$. Remember to use the + point operation discussed in Section 2.2.1. This algorithm assumes big-endianness:

$n_{scalar} \rightarrow n_{binary}$; $A = [n_{binary}]$; $Q = 0$, the point-at-infinity; $R = P$
for $k = (A_{size} - 1)...0$
  if A[k] == 1
    Q += R
  R += R
return Q

2.2.3 Diffie-Hellman key exchange with elliptic curves

A basic *Diffie-Hellman* exchange of a shared secret between *Alice* and *Bob* could take place in the following manner:

1. Alice and Bob generate their own private/public keys (k_A, K_A) and (k_B, K_B). Both publish or exchange their public keys, and keep the private keys for themselves.
2. Clearly, it holds that
$$S = k_A K_B = k_A k_B G = k_B k_A G = k_B K_A$$
Alice could privately calculate $S = k_A K_B$, and Bob $S = k_B K_A$, allowing them to use this single value as a shared secret.

An external observer would not be able to easily calculate the shared secret due to the DLP, which prevents them from finding k_A or k_B.

2.2.4 DSA signatures with elliptic curves (ECDSA)[11]

Typically, a cryptographic signature is performed on a cryptographic hash of a message rather than the message itself.[12] However, in this report we will loosely use the term *message* to refer to the message properly speaking and/or its hash value.

Signature

Assume that Alice has the private/public key pair (k, K). To unequivocally sign an arbitrary message $\mathfrak{m}$, she could execute the following steps [13]:

1. Calculate a hash of the message using a cryptographically secure hash function, $h = \mathcal{H}(\mathfrak{m})$
2. Let h' be the leftmost (big endianness) L_l bits of h, where L_l is the bit length of l
3. Generate a random integer r such that $1 < r < l$ and compute $P = (x, y) = rG$.
If $x' = x \pmod{l} = 0$ generate another random integer.
4. Calculate $s = r^{-1}(h' + x'k) \pmod{l}$.[13] If $s = 0$ then go to previous step and repeat
5. The signature is (x', s)

[11] See [14] and ANSI X9.62 for details not mentioned in this report.
[12] Signing a message hash instead of a message facilitates messages with varying size.
[13] Recall footnote 5 for the method to calculate this.

Verification

Any third party who knows the EC domain parameters D, the signature (x', s) and the signing method, $\mathfrak{m}$ and the hash function, and K can verify the signature[14], which means proving that s was created by the owner of k for the message $\mathfrak{m}$, by calculating

$$\begin{aligned} &\text{check that } x' \text{ and } s \text{ are in the interval } [1, q-1] \\ u_1 &= s^{-1}h' \pmod{l} \\ u_2 &= s^{-1}x' \pmod{l} \\ Q &= u_1 G + u_2 K; \text{ if } Q = 0 \text{ reject the signature} \end{aligned}$$

The signature will be valid if and only if the first coordinate of $Q = (x_Q, y_Q)$ satisfies

$$x_Q \equiv x' \pmod{l}$$

Why it works

This stems from the fact that

$$\begin{aligned} Q &= u_1 G + u_2 K \\ &= s^{-1}h'G + s^{-1}x'kG \\ &= s^{-1}(h + x'k)G \end{aligned}$$

Since $s = r^{-1}(h' + x'k) \pmod{l}$, it follows[15] that $r \equiv s^{-1}(h' + x'k) \pmod{l}$, so[16]

$$Q = rG$$

Therefore the owner of k created s for $\mathfrak{m}$: he signed the message.

2.3 Curve Ed25519

Monero uses a particular Twisted Edwards elliptic curve for cryptographic operations, *Ed25519*, the *birational equivalent*[17] of the Montogomery curve *Curve25519*.

Both Curve25519 and Ed25519 were released by Bernstein *et al.* [6, 7, 9].

The curve is defined over the prime field $\mathbb{F}_{2^{255}-19}$ by means of the following equation:

$$-x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2$$

[14] The paper on ECDSA [14] recommends validating D and K are legitimate before verifying a signature. See [28] for an overview of D.

[15] The modular multiplicative inverse has a rule stating:
If $ax \equiv b \pmod{n}$ with a and n relatively prime, the solution to this linear congruence is given by $x = a^{-1}b \pmod{n}$.[2]
Therefore since l is prime, $s = r^{-1}z \pmod{l} \rightarrow sr \equiv z \pmod{l} \rightarrow r \equiv s^{-1}z \pmod{l}$ with $z = (h' + x'k)$.

[16] Recalling footnote 7: $nP = n \pmod{u}P$

[17] Without giving further details, birational equivalence can be thought of as an isomorphism that can be expressed using rational terms.

This curve addresses many concerns raised by the cryptography community. It is well known that *NIST*[18] standard algorithms have issues. For example, it has recently become clear that the random number generation algorithm *PNRG* is flawed and contains a potential backdoor [12]. Seen from a broader perspective, curves endorsed by the NIST are also indirectly endorsed by the NSA, something that the cryptography community sees with suspicion The NSA is notorious for using its power over the NIST to weaken cryptographic algorithms [3].

Curve Ed25519 is not subject to any patents (see [16] for a discussion on this subject), and the team behind it has developed and adapted basic cryptographic algorithms with efficiency in mind [9]. More importantly, it is currently thought to be secure.

Twisted Edwards curves have order expressable as $N = 2^c l$, where l is a prime number and c a positive integer. In the case of curve Ed25519, its order is a 76 decimal digit number:
$2^3 \cdot 7237005577332262213973186563042994240857116359379907606001950938285454250989$

2.3.1 Binary representation

Elements of $\mathbb{F}_{2^{255}-19}$ are 256-bit integers. In other words, they can be represented using 32 bytes. Since each element only requires 255 bits, the most significant bit is always zero.

Consequently, any point in Ed25519 could be expressed using 64 bytes. By applying *point compression* techniques, described here below, however, it is possible to reduce this amount by half, to 32 bytes.

2.3.2 Point compression

The Ed25519 curve has the property that its points can be easily compressed, so that representing a point will consume only the space of one coordinate. We will not delve into the mathematics necessary to justify this, but we can give a brief insight into how it works [7].

This point compression scheme follows from a transformation of the Twisted Edwards curve equation: $x^2 = (1 - y^2)/(a - dy^2)$, which indicates there are two possible x values (+ or −) for each y. Field elements x and y are calculated $\pmod q$, so there are no actual negative values. However, taking $\pmod q$ of $-x$ will change the value between odd and even since q is odd. For example: $-3 \pmod 5 = 2$, $-6 \pmod 9 = 3$. In other words, the field elements x and $-x$ have different odd/even assignments.

If we know x is even, but given its y value the transformed curve equation outputs an odd number (the 'positive' x), then we know negating that value will give us the right x. One bit can convey this information, and conveniently the y coordinate has an extra bit.

Assume that we want to compress a point (x, y). We will employ a little-endian representation of integers. Note: $q = 2^{255} - 19$ is congruent to 5 $\pmod 8$.

[18] National Institute of Standards and Technology, https://www.nist.gov/

Encoding We set the most significant bit of y to 0 if x is even, and 1 if it is odd. The resulting value y' will represent the curve point.

Decoding Retrieve the compressed point y', then copy its most significant bit to the parity bit b before setting it to 0. That will be y.
Let $u = y^2 - 1$ and $v = dy^2 + 1$.
Compute[19] $x = uv^3(uv^7)^{(q-5)/8}$

1. If $vx^2 = u \pmod q$ then keep x
2. Else set $x = x2^{(q-1)/4} \pmod q$
3. Using the parity bit b retrieved in the first step, if $b \neq$ the least significant bit of x then return $q - x$ (the same as $-x \pmod q$), otherwise return x

2.3.3 EdDSA signature algorithm

Bernstein and his team have developed a number of basic algorithms based on curve Ed25519.[20] For illustration purposes we will describe a highly optimized and secure alternative to the ECDSA signature scheme which, according to the authors, allows producing over 100 000 signatures per second using a commodity Intel Xeon processor [7]. The algorithm can also be found described in Internet RFC8032 [15].

Among other things, instead of generating random integers every time, it uses a hash value derived from the private key of the signer and the message itself. This circumvents security flaws related to the implementation of random number generators. Also, another goal of the algorithm is to avoid accessing secret or unpredictable memory locations to prevent so-called *cache timing attacks*.

We provide here an outline of the steps performed by the algorithm for illustration purposes only. A complete description and sample implementation in the Python language can be found in [15].

[19] Similar to the double-and-add algorithm in footnote 10, we could find $a^e \pmod n$ like this:
$e_{scalar} \rightarrow e_{binary}$; $A = [e_{binary}]$; $Q = 1$; $R = a$
for $k = (size_A - 1) \ldots 0$
  if $A[k] == 1$
    $Q = Q * R \pmod n$
  $R = R * R \pmod n$
return Q
Note: each modular scalar multiplication can be calculated using the algorithm in footnote 10 by replacing EC point addition with modular addition.
This also provides an intuitive but moderately slower alternative to the extended Euclidean algorithm for the modular multiplicative inverse (footnote 5). When n is a prime number we can use *Fermats' little theorem*:

$$a^{n-1} \equiv 1 \pmod n \rightarrow a^{-1} \equiv a^{n-2} \pmod n$$

[20] See [9] for efficient group operations in Twisted Edwards EC (ie point addition, doubling, mixed addition, etc).

Signature

1. Let h_k be a hash $\mathcal{H}(k)$ of the signer's private key k. Compute r as a hash $r = \mathcal{H}(h_k, \mathfrak{m})$ of the hashed private key and message.
2. Calculate $R = rG$ and $s = (r + \mathcal{H}(R, K, \mathfrak{m}) \cdot k))$
3. the signature is the pair (R, s)

Verification

Verification is performed as follows

1. Compute $h = \mathcal{H}(R, K, \mathfrak{m})$
2. If the equality $2^c sG = 2^c R + 2^c hK$ holds then the signature is valid[21]

Why it works

$$2^c sG = 2^c((r + \mathcal{H}(R, K, \mathfrak{m}) \cdot k) \cdot G = 2^c R + 2^c \mathcal{H}(R, K, \mathfrak{m}) \cdot K)$$

Binary representation

By default, an EdDSA signature would need $64+32$ bytes to be represented. However, RFC8032 assumes that point R is compressed, which reduces space requirements to only $32 + 32$ bytes.

[21] The 2^c term comes from Bernstein et al.'s general form of the EdDSA algorithm [9]. According to that paper, though it isn't required for adequate verification, removing 2^c provides stronger equations.

CHAPTER 3

Ring signatures

Ring signatures are composed of a ring and a signature. Each *signature* is generated with a single private key and a set of unrelated public keys. Each *ring* is the set of public keys comprising the private key's public key, and the set of unrelated public keys. Somebody verifying the signature would not be able to tell which of the ring's members corresponds to the private key that created it.

Ring signatures were originally called *Group Signatures* because they were thought of as a way to prove that a signer belongs to a group, without necessarily identifying him. In the context of Monero transactions they will help make currency flows untraceable, without opening attack vectors on the money supply.

Ring signature schemes display a number of properties that will be useful for producing confidential transactions:

Signer Ambiguity An observer should be able to determine the signer is a member of the ring, but not which member.[1] Monero uses this to obfuscate the origin of funds in each transaction.

Linkability If a private key is used to sign two different messages then the messages will become linked[2]. As we will show, this property helps prevent double-spending attacks in Monero.

[1] Anonymity for an action is usually in terms of an 'anonymity set', which is 'all the people who could have possibly taken that action'. The largest anonymity set is 'humanity', and for Monero it is the so-called *mixin level* v. Mixin refers to how many fake members each ring signature has. If the mixin is $v = 4$ then there are 5 possible signers. Expanding anonymity sets makes it progressively harder to find real actors.

[2] The linkability property does not apply to non-signing public keys. That is, a ring member whose public key has been mixed into different signatures will not be linked.

Unforgeability No attacker can forge a signature.[3] This helps prevent Monero counterfeiting.

3.1 Linkable Spontaneous Anonymous Group Signatures (LSAG)

Originally (Chaum in [11]), group signature schemes required the system be set up, and in some cases managed, by a trusted person in order to prevent illegitimate signatures and, in a few schemes, adjudicate disputes. Relying on a *group secret* is not desirable since it creates a disclosure risk that could undermine anonymity. Moreover, requiring coordination between group members (i.e. for setup and management) is not scalable beyond small groups or within companies.

Liu *et al.* presented a more interesting scheme in [17] building on the work of Rivest *et al.* in [27]. The authors detailed a group signature algorithm characterized by three properties: *anonymity, linkability,* and *spontaneity.* In other words, the owner of a private key could produce one anonymous signature by selecting any set of co-signers from a list of candidate public keys, without needing to collaborate with anyone.[4]

Signature

Let $\mathfrak{m}$ be the message to sign, $\mathcal{R} = \{K_1, K_2, ..., K_n\}$ a set of distinct public keys (a group/ring), and k_π the signer's private key corresponding to his public key[5] $K_\pi \in \mathcal{R}$, where π is a secret index. Assume the existence of two hash functions, $\mathcal{H}_n$ and $\mathcal{H}_p$, mapping to integers from 1 to l,[6] and curve points in EC[7,8] respectively.

1. Compute key image $\tilde{K} = k_\pi \mathcal{H}_p(\mathcal{R})$
2. Generate random number[9] $\alpha \in_R \mathbb{Z}_l$ and random numbers $r_i \in_R \mathbb{Z}_l$ for $i \in \{0, 1, ..., n\}$ but excluding $i = \pi$.
3. Calculate
$$c_{\pi+1} = \mathcal{H}_n(\mathcal{R}, \tilde{K}, \mathfrak{m}, \alpha G, \alpha \mathcal{H}_p(\mathcal{R}))$$
4. For $i = \pi + 1, \pi + 2, ..., n, 1, 2, ..., \pi - 1$ calculate, replacing $n + 1 \to 1$,
$$c_{i+1} = \mathcal{H}_n(\mathcal{R}, \tilde{K}, \mathfrak{m}, [r_i G + c_i K_i], [r_i \mathcal{H}_p(R) + c_i \tilde{K}])$$

[3] Certain ring signature schemes, including the one in Monero, are strong against adaptive chosen-message and adaptive chosen-public-key attacks. An attacker who can obtain valid signatures for messages and corresponding to specific public keys in rings of his choice cannot discover how to forge the signature of even one message. This is called *existential unforgeability*, see [23] and [17].

[4] In the LSAG scheme linkability only applies to signatures using rings with the same members and in the same order, the 'exact same ring'. It is really "one anonymous signature per ring". Linked signatures can be attached to different messages.

[5] Notation: $K_\pi \in \mathcal{R}$ means K_π is a term in the set $\mathcal{R}$.

[6] In Monero, the hash function $\mathcal{H}_n(x) = \text{sc\_reduce32}(Keccak(x))$ where *Keccak* is the basis of SHA3 and sc_reduce32() puts the 256 bit result in the range 1 to l.

[7] It doesn't matter if points from $\mathcal{H}_p$ are compressed or not. They can always be decompressed.

[8] Monero uses a hash function that returns curve points directly, rather than computing some integer that is then multiplied by G. $\mathcal{H}_p$ would be broken if someone discovered a way to find n_x s.t. $n_x G = \mathcal{H}_p(x)$.

[9] Notation: $\alpha \in_R \mathbb{Z}_l$ means α is randomly selected from $\{1, 2, ..., l\}$.

5. Define r_π such that $\alpha = r_\pi + c_\pi k_\pi \pmod{l}$

The ring signature contains the signature $\sigma(\mathfrak{m}) = (c_1, r_1, r_2, ..., r_n, \tilde{K})$ and the ring $\mathcal{R}$.

Verification

Verification means proving $\sigma(\mathfrak{m})$ is a valid signature created by a private key corresponding to a public key in $\mathcal{R}$, and is done in the following manner

1. For $i = 1, 2, ..., n$ iteratively compute, replacing $n + 1 \to 1$,
$$z_i' = r_i G + c_i K_i$$
$$z_i'' = r_i \mathcal{H}_p(\mathcal{R}) + c_i \tilde{K}$$
$$c_{i+1}' = \mathcal{H}_n(\mathcal{R}, \tilde{K}, \mathfrak{m}, z_i', z_i'')$$
2. If $c_1' = c_1$ then the signature is valid. Note that c_1' is the last term calculated.

Why it works

We can convince ourselves the algorithm works by going through an example. Consider ring $R = \{K_1, K_2, K_3\}$ with $k_\pi = k_2$. First the signature:

1. Create key image: $\tilde{K} = k_\pi \mathcal{H}_p(\mathcal{R})$
2. Generate random numbers: α, r_1, r_3
3. Seed the encryption:
$$c_3 = \mathcal{H}_n(..., \alpha G, \alpha \mathcal{H}_p(\mathcal{R}))$$
4. Iterate:
$$c_1 = \mathcal{H}_n(..., [r_3 G + c_3 K_3], [r_3 \mathcal{H}_p(\mathcal{R}) + c_3 \tilde{K}])$$
$$c_2 = \mathcal{H}_n(..., [r_1 G + c_1 K_1], [r_1 \mathcal{H}_p(\mathcal{R}) + c_1 \tilde{K}])$$
5. Close the loop: $r_2 = \alpha - c_2 k_2 \pmod{l}$

We can substitute α into c_3 to see where the word 'ring' comes from:
$$c_3 = \mathcal{H}_n(..., [(r_2 + c_2 k_2) G], [(r_2 + c_2 k_2) \mathcal{H}_p(\mathcal{R})])$$
$$c_3 = \mathcal{H}_n(..., [r_2 G + c_2 K_2\ ], [r_2 \mathcal{H}_p(\mathcal{R}) + c_2 \tilde{K}\ \ ])$$

Then verification using $\mathcal{R}$ and $\sigma(\mathfrak{m}) = (c_1, r_1, r_2, r_3, \tilde{K})$:

1. r_1
$$z_1' = r_1 G + c_1 K_1 \qquad z"_1 = r_1 \mathcal{H}_p(\mathcal{R}) + c_1 \tilde{K}$$
$$c_2' = \mathcal{H}_n(..., [r_1 G + c_1 K_1], [r_1 \mathcal{H}_p(\mathcal{R}) + c_1 \tilde{K}])$$
2. r_2
$$z_2' = r_2 G + c_2 K_2 \qquad z_2" = r_2 \mathcal{H}_p(\mathcal{R}) + c_2 \tilde{K}$$
$$c_3' = \mathcal{H}_n(..., [r_2 G + c_2 K_2], [r_2 \mathcal{H}_p(\mathcal{R}) + c_2 \tilde{K}])$$

3. r_3 $\quad z'_3 = r_3 G + c_3 K_3 \quad z_3\text{"} = r_3 \mathcal{H}_p(\mathcal{R}) + c_3 \tilde{K}$
$$c'_1 = \mathcal{H}_n(..., [r_3 G + c_3 K_3], [r_3 \mathcal{H}_p(\mathcal{R}) + c_3 \tilde{K}])$$

and so, $c'_1 = c_1$.

Linkability

Given a fixed set of public keys $\mathcal{R}$, and two valid signatures for different messages,
$$\sigma = (c_1, s_1, ..., s_n, \tilde{K})$$
$$\sigma' = (c'_1, s'_1, ..., s'_n, \tilde{K}')$$
if $\tilde{K} = \tilde{K}'$ then clearly both signatures come from the same signing ring and private key because $\tilde{K} = k_\pi \mathcal{H}_p(\mathcal{R})$.

While an observer could link σ and σ', he wouldn't know which K_i in $\mathcal{R}$ was the culprit without solving the DLP or auditing $\mathcal{R}$ in some way (such as learning all k_i with $i \neq \pi$, or learning k_π).[10]

3.2 Back Linkable Spontaneous Anonymous Group Signatures (bLSAG)

In the LSAG signature scheme, linkability of signatures using the same private key can only be guaranteed if the ring is constant. This is obvious from the definition $\tilde{K} = k_\pi \mathcal{H}_p(\mathcal{R})$.

In this section we present an enhanced version of the LSAG algorithm where linkability is independent of the ring's co-signers.

The modification was unraveled in [22] based on a publication by A. Back [5] regarding the CryptoNote [30] ring signature algorithm.

Signature

1. Calculate key image $\tilde{K} = k_\pi \mathcal{H}_p(K_\pi)$
2. Generate random number $\alpha \in_R \mathbb{Z}_l$ and random numbers $r_i \in_R \mathbb{Z}_l$ for $i \in \{0, 1, ..., n\}$ but excluding $i = \pi$
3. Compute
$$c_{\pi+1} = \mathcal{H}_n(\mathfrak{m}, \alpha G, \alpha \mathcal{H}_p(K_\pi))$$
4. For $i = \pi + 1, \pi + 2, ..., n, 1, 2, ..., \pi - 1$ calculate, replacing $n + 1 \to 1$,
$$c_{i+1} = \mathcal{H}_n(\mathfrak{m}, [r_i G + c_i K_i], [r_i \mathcal{H}_p(K_i) + c_i \tilde{K}])$$

[10] LSAG is unforgeable, meaning no attacker could make a valid ring signature without knowing a private key. If he invents a fake $\tilde{K}$ and seeds his signature computation with $c_{\pi+1}$, then, not knowing k_π, he can't calculate a number $r_\pi = \alpha - c_\pi k_\pi$ that would produce $[r_\pi G + c_\pi K_\pi] = \alpha G$. A verifier would reject his signature. Liu *et al.* prove forgeries that manage to pass verification are extremely improbable [17].

5. Define $r_\pi = \alpha - k_\pi c_\pi \pmod{l}$

The signature will be $\sigma(\mathfrak{m}) = (c_1, r_1, ..., r_n, \tilde{K})$.

As in the original LSAG scheme, verification takes place by recalculating the value c_1.

Correctness can also be demonstrated (i.e. 'how it works') in a way similar to the LSAG scheme.

The alert reader will no doubt notice that the key image $\tilde{K}$ depends only on the keys of the true signer. In other words, two signatures will now be linkable if and only if the same private key was used to create the signature. In bLSAG, rings are just involved in obfuscating each signer's identity. bLSAG also reduces the time it takes to sign and verify by removing $\mathcal{R}$ and $\tilde{K}$ from the hash that calculates c_i.

This approach to linkability will prove to be more useful for Monero than the one offered by the LSAG algorithm, as it will allow detecting double-spending attempts without putting constraints on the ring members used.

3.3 Multilayer Linkable Spontaneous Anonymous Group Signatures (MLSAG)

In order to be able to sign multi-input transactions, one has to be able to sign with m private keys. In [22, 23], Noether S. *et al.* describe a multi-layered generalization of the bLSAG signature scheme applicable when we have a set of $n \cdot m$ keys, that is, the set

$$\mathcal{R} = \{K_{i,j}\} \quad \text{for} \quad i \in \{1, 2, ..., n\} \quad \text{and} \quad j \in \{1, 2, ..., m\}$$

for which we know the private keys $\{k_{\pi,j}\}$ corresponding to the subset $\{K_{\pi,j}\}$ for some index $i = \pi$.[11]

Such an algorithm would address our multi-input needs provided we generalize the notion of linkability.

Linkability: if any of private keys $k_{\pi,j}$ is used in 2 different signatures, then these signatures will be automatically linked.

[11] Another way to think about MLSAG is that there are m sub-rings of size n, and in each sub-ring we know a private key at index $i = \pi$ ($m \cdot n$ total public keys). The signature algorithm encrypts a 'stack' of keys at each stage c, composed of one key from each sub-ring. bLSAG is the special case where $m = 1$.

Signature

1. Calculate key images $\tilde{K_j} = k_{\pi,j}\mathcal{H}_p(K_{\pi,j})$ for all $j \in \{1, 2, ..., m\}$
2. Generate random numbers $\alpha_j \in_R \mathbb{Z}_l$, and $r_{i,j} \in_R \mathbb{Z}_l$ for $i \in \{1, 2, ..., n\}$ (except $i = \pi$) and $j \in \{1, 2, ..., m\}$
3. Compute
$$c_{\pi+1} = \mathcal{H}_n(\mathfrak{m}, \alpha_1 G, \alpha_1 \mathcal{H}_p(K_{\pi,1}), ..., \alpha_m G, \alpha_m \mathcal{H}_p(K_{\pi,m}))$$
4. For $i = \pi + 1, \pi + 2, ..., n, 1, 2, ..., \pi - 1$ calculate, replacing $n + 1 \to 1$,
$$c_{i+1} = \mathcal{H}_n(\mathfrak{m}, [r_{i,1}G + c_i K_{i,1}], [r_{i,1}\mathcal{H}_p(K_{i,1}) + c_i \tilde{K_1}], ..., [r_{i,m}G + c_i K_{i,m}], [r_{i,m}\mathcal{H}_p(K_{i,m}) + c_i \tilde{K_m}])$$
5. Define $r_{\pi,j} = \alpha_j - k_{\pi,j}c_\pi \pmod{l}$

The signature will be $\sigma(\mathfrak{m}) = (c_1, r_{1,1}, ..., r_{1,m}, ..., r_{n,1}, ..., r_{n,m}, \tilde{K_1}, ..., \tilde{K_m})$.

Verification

The verification of a signature is done in the following manner

1. For $i = 1, ..., n$ compute, replacing $n + 1 \to 1$,
$$c'_{i+1} = \mathcal{H}_n(\mathfrak{m}, [r_{i,1}G + c_i K_{i,1}], [r_{i,1}\mathcal{H}_p(K_{i,1}) + c_i \tilde{K_1}], ..., [r_{i,m}G + c_i K_{i,m}], [r_{i,m}\mathcal{H}_p(K_{i,m}) + c_i \tilde{K_m}])$$
2. If $c'_1 = c_1$ then the signature is valid

Why it works

Just as with the original LSAG algorithm, we can readily observe that

> if $i \neq \pi$ then clearly the values c'_{i+1} are calculated as described in the signature algorithm
>
> if $i = \pi$ then, since $r_{\pi,j} = \alpha_j - k_{\pi,j}c_\pi$
>
> $$\begin{aligned} r_{\pi,j}G &+ c_\pi K_{\pi,j} &&= (\alpha_j - k_{\pi,j}c_\pi)G &&+ c_\pi K_{\pi,j} &&= \alpha_j G \\ r_{\pi,j}\mathcal{H}_p(K_{\pi,j}) &+ c_\pi \tilde{K_j} &&= (\alpha_j - k_{\pi,j}c_\pi)\mathcal{H}_p(K_{\pi,j}) &&+ c_\pi \tilde{K_j} &&= \alpha_j \mathcal{H}_p(K_{\pi,j}) \end{aligned}$$
>
> In other words, it holds also that $c'_{\pi+1} = c_{\pi+1}$

Linkability

If a private key $k_{\pi,j}$ is re-used to make any signature, the corresponding key image $\tilde{K_j}$ supplied in the signature will reveal it. This observation matches our generalized definition of linkability.[12]

[12] As with LSAG, linked MLSAG signatures do not indicate which public key was used to sign it. However, if each ring has only one key in common, the culprit is obvious.

Space requirements

Assuming point compression, an MLSAG signature would clearly consume a total of

$$(1 + nm + m) \cdot 32 \quad \text{bytes}$$

3.4 Borromean Ring Signatures

We will see in later sections of this report that it will be necessary to prove transaction amounts are within an expected range. This can be accomplished with ring signatures. However, to this particular end it is not necessary that signatures be linkable, which allows us to select more efficient algorithms in terms of space consumed.

In this context, and for the *singular purpose* of proving amount ranges, Monero uses[13] a signature scheme developed by G. Maxwell, which he described in [19]. We present here a complete version of the scheme for educational purposes. In Monero, range proofs require sub-rings with exactly 2 keys corresponding to each digit of an amount represented in binary. This means all $m_i = 2$, so the Borromean scheme can be implemented in a simpler form.

Assume we have a set $\mathcal{R}$ of public keys $\{K_{i,j_i}\}$ for $i \in \{1, 2, ..., n\}$ and $j_i \in \{1, 2, ..., m_i\}$. In other words, $\{K_{i,j_i}\}$ is like a bookshelf of public keys with n shelves, and on each i^{th} shelf are m_i public keys ordered from 1 to m_i. m_i can be different for each i, hence the subscript.

Furthermore, assume for each i there is an index π_i such that the signer knows private key k_{i,π_i} corresponding to K_{i,π_i}. Following the analogy, a signer knows one private key for the $j_i = \pi_i{}^{\text{th}}$ public key on each shelf i in the bookshelf.

In what follows, $\mathfrak{m}$ is a hash of the message to be signed with keys $\{K_{i,j}\}$.

Signature

1. For each shelf $i = 1, ..., n$:
   (a) generate a random value $\alpha_i \in_R \mathbb{Z}_l$
   (b) seed the shelf's sub-ring: set $c_{i,\pi_i+1} = \mathcal{H}_n(\mathfrak{m}, \alpha_i G, i, \pi_i)$
   (c) build first half of sub-ring from seed: for $j_i = \pi_i + 1, ..., m_i - 1$ generate random numbers $r_{i,j_i} \in_R \mathbb{Z}_l$ and compute

$$c_{i,j_i+1} = \mathcal{H}_n(\mathfrak{m}, [r_{i,j_i}G - c_{i,j_i}K_{i,j_i}], i, j_i)$$

2. Use last public key on each shelf to connect all sub-rings together: for $i = 1, ..., n$ generate random numbers $r_{i,m_i} \in_R \mathbb{Z}_l$ and compute

$$c_1 = \mathcal{H}_n([r_{1,m_1}G - c_{1,m_1}K_{1,m_1}], ..., [r_{n,m_n}G - c_{n,m_n}K_{n,m_n}])$$

   Note: if any $\pi_i = m_i$ put the seed term $\alpha_i G$ in the connector c_1.

[13] Monero's first iteration of 'range proofs' used Aggregate Schnorr Non-Linkable Ring Signatures (ASNL) [23]. According to the author of ASNL it reduces to a kind of Borromean ring signature [4], but since the latter is more general and secure it was chosen for final implementation in November 2016.

3. For each $i = 1, .., n$:

   (a) build second half of sub-ring from connector: for $j_i = 1, ..., \pi_i - 1$ generate random numbers $r_{i,j_i} \in_R \mathbb{Z}_l$ and compute [we interpret references to $c_{i,1}$ as c_1]
   $$c_{i,j_i+1} = \mathcal{H}_n(\mathfrak{m}, [r_{i,j_i}G - c_{i,j_i}K_{i,j_i}], i, j_i)$$
   (b) tie sub-ring ends together: set r_{i,π_i} such that $\alpha_i = r_{i,\pi_i} - c_{i,\pi_i}k_{i,\pi_i}$

The signature is
$$\sigma = (c_1, r_{1,1}, ..., r_{1,m_1}, r_{2,1}..., r_{2,m_2}, ..., r_{n,m_n})$$

Verification

Given $\mathfrak{m}$, $\mathcal{R}$, and σ, verification is performed as follows:

1. For $i = 1, ..., n$ and $j_i = 1, ..., m_i$ build each ring:
   $$L'_{i,j_i+1} = r_{i,j_i}G - c'_{i,j_i}K_{i,j_i}$$
   $$c'_{i,j_i+1} = \mathcal{H}_n(\mathfrak{m}, L'_{i,j_i+1}, i, j_i)$$
   Interpret any $c'_{i,1}$ as c_1
2. Compute the connector $c'_1 = \mathcal{H}_n(L'_{1,m_1}, ..., L'_{n,m_n})$

The signature is valid if $c'_1 = c_1$.

Why it works

1. For $j \neq \pi_i$ and for all i we can readily see that $c'_{i,j_i+1} = c_{i,j_i+1}$
2. When $j_i = \pi_i$, for all i
   $$\begin{aligned} L'_{i,\pi_i+1} &= r_{i,\pi_i}G - c'_{i,\pi_i}K_{i,\pi_i} \\ &= (\alpha_i + k_{i,\pi_i}c_{i,\pi_i})G - c'_{i,\pi_i}K_{i,\pi_i} \\ &= \alpha_i G + k_{i,\pi_i}c_{i,\pi_i}G - c'_{i,\pi_i}k_{i,\pi_i}G \\ &= \alpha_i G \end{aligned}$$
   In other words, $c'_{i,\pi_i+1} = \mathcal{H}_n(\mathfrak{m}, \alpha_i G, i, \pi_i) = c_{i,\pi_i+1}$.

Therefore we can conclude the verification step identifies valid signatures.

CHAPTER 4

Pedersen commitments

Generally speaking, a cryptographic *commitment scheme* is a way of publishing a commitment to a value without revealing the value itself.

For example, in a coin-flipping game Alice could privately commit to one outcome (i.e. 'call it') before Bob flips the coin by publishing her committed value hashed with secret data. After he flips the coin, Alice could declare which value she committed to and prove it by publishing her secret data. Bob could then verify her claim.

In other words, assume that Alice has a secret string b and the value she wants to commit to is v. She could simply hash $h = \mathcal{H}(b, v)$ and give h to Bob. Bob flips a coin, Alice gives b to Bob and tells him she committed to v, and then Bob calculates $h' = \mathcal{H}(b, v)$. If $h' = h$, then he knows Alice committed to v before the coin flip.

4.1 Pedersen commitments

A *Pedersen commitment* [25] is a commitment that has the property of being *additive*. If $C(a)$ and $C(b)$ denote the commitments for values a and b respectively, then $C(a+b) = C(a) + C(b)$. This property is useful when committing transaction amounts, as one could prove, for instance, that inputs equal outputs, without revealing the amounts at hand.

Fortunately, Pedersen commitments are easy to implement with elliptic curve cryptography, as the following holds trivially

$$aG + bG = (a + b)G$$

Clearly, by defining a commitment as simply $C(a) = aG$, we would immediately recognize commitments to 0 (because $0G = 0$). We could also create cheat tables of commitments to help us recognize common amounts a.

To attain information-theoretic[1] privacy, one needs to add a secret *blinding factor* and another generator H, such that it is unknown for which value of γ the following holds: $H = \gamma G$. The hardness of the discrete logarithm problem ensures calculating γ from H is infeasible.

We can then define the commitment to an amount a as $C(x, a) = xG + aH$, where x is a blinding factor that prevents observers from guessing a (for example: if you commit $C(a = 1)$, it is trivial to guess and check).

Commitment $C(x, a)$ is information-theoretically private because there are many possible combinations of x and a that would output the same C.[2] If x is truly random, an attacker would have literally no way to figure out a [18].

In the case of Monero, $H = \mathcal{H}_p(G)$.[3]

4.2 Monero commitments

Owning cryptocurrency is not like a bank account, where a person's balance exists as a single value in a database. Rather, a person owns a bunch of transaction *outputs*. Each output has an 'amount', and the sum of all outputs owned is considered a person's balance.

To send cryptocurrency to someone else, we create a transaction. A transaction takes old outputs as *inputs* and addresses new outputs to recipients. Since it is rare for inputs to equal intended outputs, most transactions include 'change', an output that sends excess back to the sender. We will elaborate on these topics in Chapter 5.

In Monero, transaction amounts are hidden using a technique called RingCT, first implemented in January 2017. While transaction verifiers don't know how much Moneroj (plural form of Monero, which means money in Esperanto) is contained in each input and output, they still need to prove the sum of inputs equals the sum of outputs.

In other words, if we had a transaction with inputs containing amounts $a_1, ..., a_m$ and outputs with amounts $b_1, ..., b_p$, then an observer would justifiably expect that:

$$\sum_j a_j - \sum_t b_t = 0$$

[1] Information-theoretic security means even an adversary with infinite computing power could not break an encryption, because they wouldn't have enough information.

[2] Basically, there are many x' and a' such that $x' + a'\gamma = x + a\gamma$. A committer knows one combination, but an attacker has no way to know which one. Furthermore, even the committer can't find another combination without solving the DLP for γ.

[3] The Monero codebase has a function $HashToPoint()$ that maps scalars to EC points. For commitments, $H = HashToPoint(SHA3(G))$, where SHA3 stands for the novel *Keccak* hashing algorithm.

Since commitments are additive, the sum of commitments to inputs and outputs should also equal zero[4]:

$$\sum_j C_{j,in} - \sum_t C_{t,out} = 0$$

To avoid sender identifiability, Shen Noether proposes [22] verifying that commitments sum to a certain non-zero value:

$$\sum_j C_{j,in} - \sum_t C_{t,out} = zG$$
$$\sum_j (x_j G + a_j H) - \sum_t (y_t G + b_t H) = zG$$
$$\sum_j x_j - \sum_t y_t = z$$

The reasons why this is useful will become clear in Chapter 5, when we discuss the structure of transactions.

4.3 Range proofs

One problem with additive commitments is that, if we have commitments $C(a_1)$, $C(a_2)$, $C(b_1)$, and $C(b_2)$ and we intend to use them to prove that $(a_1 + a_2) - (b_1 + b_2) = 0$, then those commitments would apply if one value in the equation were negative.

For instance, we could have $a_1 = 6$, $a_2 = 5$, $b_1 = 21$, and $b_2 = -10$.

$$(6 + 5) - (21 + -10) = 0$$

where

$$21G + -10G = 21G + (l - 10)G = (l + 11)G = 11G$$

We could keep the 21 output and throw away the -10 output, effectively creating 10 more Moneroj than we put in.

The solution addressing this issue in Monero is to prove each output amount is in a certain range using the Borromean signature scheme described in Section 3.4.

Given a commitment $C(b)$ with blinding factor y_b for amount b, use the binary representation $(b_0, b_1, ..., b_{k-1})$ such that

$$b = b_0 2^0 + b_1 2^1 + ... + b_{k-1} 2^{k-1}$$

Generate random numbers $y_0, ..., y_{k-1} \in_R \mathbb{Z}_l$ to be used as blinding factors. Define also Pedersen commitments for each b_i, $C_i = y_i G + b_i 2^i H$, and derive public keys $\{C_i, C_i - 2^i H\}$.

[4] Recall from Section 2.2.1 we can subtract a point by inverting its coordinates. If $P = (x, y)$, $-P = (x, -y)$. Recall also that negations of field elements are calculated $\pmod q$, so $(-y \pmod q)$.

Clearly one of those public keys will equal $y_i G$:

$$\begin{aligned} &\text{if } b_i = 0 \text{ then} \quad C_i &&= y_i G + 0H &&= y_i G \\ &\text{if } b_i = 1 \text{ then} \quad C_i - 2^i H &&= y_i G + 2^i H - 2^i H &&= y_i G \end{aligned}$$

In other words, a blinding factor y_i will always be the private key corresponding to one of the points $\{C_i, C_i - 2^i H\}$. One of these points is a *commitment to zero*, because either $b_i 2^i = 0$ or $b_i 2^i - 2^i = 0$. We can prove a transaction output's amount b is in the range $[0, ..., 2^k - 1]$ by signing it using the Borromean Ring Signature scheme of Section 3.4 with the ring of public keys:

$$\{\{C_0, C_0 - 2^0 H\}, ..., \{C_{k-1}, C_{k-1} - 2^{k-1} H\}\}$$

where we know the private keys $\{y_0, ..., y_{k-1}\}$ corresponding to each pair.

$$\text{Resulting in a signature } \sigma = (c_1, r_{0,1}, r_{0,2}, r_{1,1} ..., r_{k-1,2})$$

4.4 Range proofs in a blockchain

In the context of Monero we will use range proofs to prove there are valid amounts in the outputs of each transaction.

Transaction verifiers will have to check that the sum of each output's range proof commitments C_i equals its amount commitment C_b. For this to work we need to modify our definition of the blinding factors y_i: set $y_0, ..., y_{k-2} \in_R \mathbb{Z}_l$ and, given y_b, define $y_{k-1} = y_b - \sum_{i=0}^{k-2} y_i$. The following equation now holds

$$\sum_{i=0}^{k-1} C_i = C_b$$

We will store only the range proof commitments/keys C_i, the output commitment C_b, and the signature σ's terms in the blockchain. The mining community can easily calculate $C_i - 2^i H$ and verify the Borromean ring signature for each output.

It will not be necessary for the receiver nor any other party to know the blinding factors y_i, as their sole purpose is proving a new output's amount is in range.

Since the Borromean signature scheme requires knowledge of y_i to produce a signature, any third party who verifies one can convince himself that each sub-ring contains a commitment to zero, so total amounts must fall within range and money is not being artificially created.

CHAPTER 5

Monero Transactions

5.1 User keys

Unlike Bitcoin, Monero users have two sets of private/public keys, (k^v, K^v) and (k^s, K^s), generated as described in Section 2.2.2.[1]

The *address* of a user is the pair of public keys (K^v, K^s). Her private keys will be the corresponding pair (k^v, k^s).

Using two sets of keys allows function segregation. The rationale will become clear later in this chapter, but for the moment let us call private key k^v the *view key*, and k^s the *spend key*. A person can use their view key to determine if an output is addressed to them, and their spend key will allow them to send that output in a transaction.

5.1.1 Subaddresses

Monero users can generate subaddresses from each address [21]. Funds sent to a subaddress can be viewed and spent using the address' view and spend keys. By analogy: an online bank account may have multiple balances corresponding to credit cards and deposits, yet they are all accessible and spendable from the same point of view – the account holder. Wallets may

[1] It is currently most common that the view key k^v is simply equal to $\mathcal{H}_n(k^s)$. This means a person only needs to save their spend key k^s in order to access all of the outputs they own. The spend key is typically represented as a series of 25 words (where the 25th word is a checksum). Other, less popular methods include: generating k^v and k^s as separate random numbers, or generating a random 12-word mnemonic a, where $k^s = sc\_reduce32(Keccak(a))$ and $k^v = sc\_reduce32(Keccak(Keccak(a)))$.

aggregate or segregate subaddress balances, and aggregate or segregate spending from those balances.

Subaddress are convenient for receiving funds to the same place when a user doesn't want to link his activities together by publishing/using the same address. Note that due to the uniformly distributed output of $\mathcal{H}_n$, an adversary would have to solve the DLP in order to determine a given subaddress is associated with any particular address [21].

Bob generates his i^{th} subaddress[2] $(i = 1, 2, ...)$ from his address as a pair of public keys $(K^{v,i}, K^{s,i})$:

$$K^{s,i} = K^s + \mathcal{H}_n(k^v, i)G$$
$$K^{v,i} = k^v K^{s,i}$$

So,

$$K^{v,i} = k^v(k^s + \mathcal{H}_n(k^v, i))G$$
$$K^{s,i} = \quad (k^s + \mathcal{H}_n(k^v, i))G$$

Subaddresses are prefixed with an '8', differentiating them from addresses, which are prefixed with '4'. This allows sending wallets to use the correct procedure when constructing transactions. It is slightly different between recipients using addresses and subaddresses (see footnote 3).

5.2 One-time (stealth) addresses

Every Monero user has a public address, which they may distribute to other users in order to be used in transaction outputs. This address is never used directly. Instead, a Diffie-Hellman-like exchange is applied to it, creating a unique *stealth address* for each transaction output to be paid to the user. In this way, even external observers who know all users' public addresses will not be able to identify which user received any given transaction output.

Let's bring this concept into focus with a very simple transaction, containing exactly one input and one output — a payment from Alice to Bob.

Bob has private/public keys (k_B^v, k_B^s) and (K_B^v, K_B^s), and Alice knows his public keys. A transaction could proceed as follows (see [30]):

1. Alice generates a random number r such that $1 < r < l$, and calculates the one-time public key $K^o = \mathcal{H}_n(rK_B^v)G + K_B^s$
2. Alice sets K^o as the addressee of the payment, adds the value rG to the transaction data and submits it to the network.

[2] Note that the index i could just as easily be some password-generated hash $\mathcal{H}_n(x)$. This would allow an address owner to view his subaddress funds from his main address view key, but only be able to spend those funds by inputting a password. There are currently no wallet implementations password protecting subaddresses, nor any known plans to develop a wallet with that feature.

3. Bob receives the data and sees the values rG and K^o. He can calculate $k_B^v rG = rK_B^v$. He can then calculate $K_B'^s = K^o - \mathcal{H}_n(rK_B^v)G$. When he sees that $K_B'^s = K_B^s$, he knows the transaction is addressed to him.[3]

   The private key k_B^v is called the *view key* because anyone who has it (and Bob's public spend key K_B^s) can calculate K'^o for every transaction output in the network, and 'view' which ones are addressed to Bob.

4. The one-time keys for the output are

$$K^o = \mathcal{H}_n(rK_B^v)G + k_B^s G = (\mathcal{H}_n(rK_B^v) + k_B^s)G$$
$$k^o = \mathcal{H}_n(rK_B^v) + k_B^s$$

While Alice can calculate the public key K^o for the address, she can not compute the corresponding private key k^o, since it would require either knowing Bob's spend key k_B^s, or solving the discrete logarithm problem for $K_B^s = k_B^s G$, which we assume to be hard. As will become clear later in this chapter, without k^o Alice can't compute the output's key image, so she can never know for sure if Bob spends the output she sent him.

While a third party with Bob's *view key* can verify an output is addressed to Bob, without knowledge of the *spend key* this third party would not be able to spend that output nor know when it has been spent. They would not be able to sign with the private key k^o of the one-time address, nor create that address's key image.

Such a third party could be a trusted custodian, an auditor, a tax authority, etc. Somebody who could have read access to the user's transaction history, without any further rights. This third party would also be able to decrypt the amounts of Section 5.4.1.

5.2.1 Multi-output transactions

Most transactions will contain more than one output. If nothing else, to transfer 'change' back to the sender.

Monero senders generate only one random value r. The value rG (or, in the case that outputs are directed to subaddresses, $r_t K_t^{s,i}$) is normally known as the *transaction public key* and is published in the blockchain.

To ensure that all output addresses in a transaction with p outputs are different even in cases where the same addressee is used twice, Monero uses an output index. Every output from a transaction has an index $t \in 1, ..., p$. By appending this value to the shared secret before hashing it, one can ensure the resulting stealth addresses are unique:

$$K_t^o = \mathcal{H}_n(rK_t^v, t)G + K_t^s G = (\mathcal{H}_n(rK_t^v, t) + k_t^s)G$$
$$k_t^o = \mathcal{H}_n(rK_t^v, t) + k_t^s$$

[3] If Bob gave Alice his subaddress $(K_B^{v,1}, K_B^{s,1})$, then Alice would need to add $rK_B^{s,1}$ instead of rG to the transaction data so Bob can properly calculate $k_B^v rK_B^{s,1} = rK_B^{v,1}$, which allows him to calculate $K_B'^s$ and then find it corresponds to $K_B^{s,1}$ in his record of address and subaddress key pairs. This also means transactions with multiple outputs to subaddresses need to add unique *transaction public keys* to the data for each and every output (Sections 5.2.1, 5.6 for more on this).

5.3 Transaction types

Monero is a cryptocurrency under steady development. Transaction structures, protocols, and cryptographic schemes are always prone to evolving as new objectives or threats are found.

In this report we have focused our attention on *Ring Confidential Transactions*, a.k.a. *RingCT*, as they are implemented in the current version of Monero. RingCT is mandatory for all new Monero transactions, so we will not describe any legacy transaction types, even if they are still partially supported.

The transaction types we will describe in this chapter are `RCTTypeFull` and `RCTTypeSimple`. The former category (Section 5.4) closely follows the ideas exposed by S. Noether *at al.* in [23]. At the time that paper was written, the authors most likely intended to fully replace the original CryptoNote transaction scheme.

However, for multi-input transactions, the signature scheme formulated in that paper was thought to entail a risk on traceability. This will become clear when we supply technical details, but in short: if one spent output became identifiable, the rest of the spent outputs would also become identifiable. This would have an impact on the traceability of currency flows, not only for the transaction originator affected, but also for the rest of the blockchain.

To mitigate this risk, the Monero Research Lab decided to use a related, yet different signature scheme for multi-input transactions. The transaction type `RCTTypeSimple` (Section 5.5) is the one used in these occasions. The main difference, as we will see later, is that each input is signed independently.

We present a conceptual summary of transaction data in Section 5.6.

5.4 Ring Confidential Transactions of type `RCTTypeFull`

By default, the current code base applies this type of signature scheme when transactions have only one input. The scheme itself allows multi-input transactions, but when it was introduced, the Monero Research Lab decided that it would be advisable to use it only on single-input transactions. For multi-input transactions, existing Monero wallets use the `RCTTypeSimple` scheme described later.

Our perception is that the decision to limit `RCTTypeFull` transactions to one input was rather hastily taken, and that it might change in the future, perhaps if the algorithm to select additional mix-in outputs is improved and ring sizes are increased. Also, S. Noether's original description in [23] did not envision constraints of this type. At any rate, it is not a hard constraint. An alternative wallet might choose to sign transactions using either scheme, independently of the number of inputs involved.

We have therefore chosen to describe the scheme as if it were meant for multi-input transactions.

An actual example of a `RCTTypeFull` transaction, with all its components, can be inspected in Appendix A.

5.4.1 Amount Commitments

Recall from Section 4.2 that we had defined a commitment to an output's amount b as:

$$C(b) = yG + bH$$

In the context of Monero, output recipients should be able to view their outputs' amounts. This means the blinding factor yG must be communicated to the receiver.

The solution adopted in Monero is a Diffie-Hellman shared secret rK_B^v. For any given transaction in the blockchain, each of its outputs $t \in 1, ..., p$ has 2 associated values called *mask* and *amount* satisfying[4]

$$mask_t = y_t + \mathcal{H}_n(rK_B^v, t)$$
$$amount_t = b_t + \mathcal{H}_n(\mathcal{H}_n(rK_B^v, t))$$

The receiver, Bob, will be able to calculate the blinding factor y_t and the amount b_t using the *transaction public key* rG and his *view key* k_B^v. He can also check that the commitment $C(y_t, b_t)$ provided in the transaction data, henceforth denoted C_t^b, corresponds to the amount at hand.

More generally, any third party with access to Bob's *view key* would be able to decrypt his output amounts, and make sure they agree with their associated commitments.

5.4.2 Commitments to zero

Assume a transaction sender has previously received amounts $a_1, ..., a_m$ from various outputs, addressed to one-time addresses $K_{\pi,1}^o, ..., K_{\pi,m}^o$ and with amount commitments $C_{\pi,1}^a, ..., C_{\pi,m}^a$.

This sender knows the private keys $k_{\pi,1}^o, ..., k_{\pi,m}^o$ corresponding to the one-time addresses (Section 5.2). The sender also knows the blinding factors x_j used in commitments $C_{\pi,j}^a$ (Section 5.4.1).

A transaction consists of inputs $a_1, ..., a_m$ and outputs $b_1, ..., b_p$ such that $\sum_{j=1}^{m} a_j - \sum_{t=1}^{p} b_t = 0$.

The sender re-uses the commitments from the previous outputs, $C_{\pi,1}^a, ..., C_{\pi,m}^a$, and creates commitments for $b_1, ..., b_p$. Let these commitments be $C_1^b, ..., C_p^b$.

As hinted in Section 4.2, the sum of the commitments will not be truly 0, but a curve point zG:

$$\sum_j C_{\pi,j}^a - \sum_t C_{\pi,t}^b = zG$$

The sender will know z, allowing him to create a signature on this *commitment to zero*.

[4] As with the stealth address K^o, the output index t is appended to the hash in each mask/amount pair. This ensures outputs directed to the same address are secure. Furthermore, the amount term contains an extra hash $\mathcal{H}_n$ to prevent statistical analysis of blockchain data, which might provide insight into currency flows, and to more generally make the mask/amount values unrelated.

Indeed, z follows from the blinding factors if and only if inputs equal outputs (recalling Section 4.1, we don't know γ in $H = \gamma G$):

$$\begin{aligned}
&\sum_{j=1}^{m} C^a_{\pi,j} - \sum_{t=1}^{p} C^b_{\pi,t} \\
=&\sum_j x_j G - \sum_t y_t G + (\sum_j a_j - \sum_t b_t) H \\
=&\sum_j x_j G - \sum_t y_t G \\
=&zG
\end{aligned}$$

5.4.3 Signature

The sender selects v sets of size m, of additional unrelated addresses and their commitments from the blockchain, corresponding to apparently unspent outputs. She mixes the addresses in a *ring* with her own m unspent outputs' addresses, adding false commitments to zero, as follows:

$$\begin{aligned}
\mathcal{R} = \{&\{K^o_{1,1}, ..., K^o_{1,m}, (\sum_j C_{1,j} - \sum_t C^b_t)\}, \\
&... \\
&\{K^o_{\pi,1}, ..., K^o_{\pi,m}, (\sum_j C^a_{\pi,j} - \sum_t C^b_t)\}, \\
&... \\
&\{K^o_{v+1,1}, ..., K^o_{v+1,m}, (\sum_j C_{v+1,j} - \sum_t C^b_t)\}\}
\end{aligned}$$

Looking at the structure of the key ring, we see that if

$$\sum_j C^a_{\pi,j} - \sum_t C^b_t = 0$$

then any observer would recognize the set of addresses $\{K^o_{\pi,1}, ..., K^o_{\pi,m}\}$ as the ones in use as inputs, and therefore currency flows would be traceable.

With this observation made we can see the utility of zG. All commitment terms in $\mathcal{R}$ return some EC point, and the π^{th} such term is zG. This allows us to create an MLSAG signature (Section 3.3) on $\mathcal{R}$:

MLSAG signature for inputs The private keys for $\{K^o_{\pi,1}, ..., K^o_{\pi,m}, (\sum_j C^a_{\pi,j} - \sum_t C^b_t)\}$ are $k^o_{\pi,1}, ..., k^o_{\pi,m}, z$, which are known to the sender. MLSAG in this scenario does not use a key image for the commitment to zero zG. This means calculating and verifying the signature excludes the term $r_{i,m+1}\mathcal{H}_p(K_{i,m+1}) + c_i\tilde{K}_z$.

The message $\mathfrak{m}$ signed in the input MLSAG is essentially a hash of all transaction information *except* for the MLSAG signature itself.[5] This ensures transactions are tamper-proof from the perspective of both transaction authors and verifiers.

Range proofs for outputs To avoid the amount ambiguity of outputs described in Section 4.3, the sender must also employ the Borromean signature scheme of Section 3.4 to sign amount ranges for each output $t \in 1, ..., p$.

Range proofs are not needed for input amounts because they are either expressed clearly (as with transaction fees and block rewards), or were proven in range when first created as outputs.

In the current version of the Monero software, each amount is expressed as a fixed point number of 64 bits. This means the data for each range proof will contain 64 commitments and $2 \cdot 64 + 1$ signature terms.

5.4.4 Transaction fees

Typically transaction outputs are *lower* than transaction inputs, in order to provide a fee for miners. Transaction fee amounts are stored in clear text (not encrypted) in the transaction data transmitted to the network. Miners can create an additional output for themselves with the fee. In turn, this fee amount must also be converted into a commitment.

The solution is to calculate the commitment of the fee f without the masking effect of any blinding factor. That is, $C(f) = fH$, where f is communicated in clear text.

The network verifies the MLSAG signature on $\mathcal{R}$ by including fH as follows:

$$(\sum_j C_{i,j} - \sum_t C_t^b) - fH$$

Which works because this is a commitment to zero:

$$(\sum_j C_{\pi,j} - \sum_t C_t^b) - fH = zG$$

5.4.5 Avoiding double-spending

An MLSAG signature (Section 3.3) contains images $\tilde{K}_j$ of private keys $k_{\pi,j}$. An important property in any cryptographic signature scheme is that it should be unforgeable with non-negligible probability. Therefore, to all practical effects, we can assume a signature's key images must have been deterministically produced from legitimate private keys.

[5] The actual message is $\mathfrak{m} = \mathcal{H}(\mathcal{H}(tx\_prefix), \mathcal{H}(ss), \mathcal{H}(\text{range proof signatures}))$ where:
tx_prefix ={transaction version (i.e. ringCT = 2), inputs {key offsets, key image}, outputs {one-time addresses}}
ss ={signature type (simple vs full), transaction fee, pseudo output commitments for inputs, ecdhInfo (masks and amounts), output commitments}. See Appendices A & B regarding this terminology.

The network need only verify that key images included in MLSAG signatures (corresponding to inputs and calculated as $\tilde{K}^o_j = k^o_{\pi,j}\mathcal{H}_p(K^o_{\pi,j})$) have not appeared before in other transactions.[6] If they have, then we can be sure we are witnessing an attempt to spend twice a previously received output $C^a_{\pi,j}$, addressed to $K^o_{\pi,j}$.

If someone tries to spend $C^a_{\pi,j}$ twice, they will reveal the index π for both transactions where it appears. This has two effects: 1) all outputs at index π in the first transaction are revealed as its real inputs, and 2) all outputs at index π in the second transaction are revealed as not having been spent before. The second is a problem even considering miners would reject the double-spend transaction.

These effects could weaken the network benefits of ring signatures, and are part of the reason `RCTTypeFull` is only used for single-input transactions. The other main reason is that a cryptanalyst would know that in general, all real inputs share an index.

5.4.6 Space requirements

MLSAG signature (inputs)

From Section 3.3 we recall that an MLSAG signature in this context would be expressed as

$$\sigma(\mathfrak{m}) = (c_1, r_{1,1}, ..., r_{1,m+1}, ..., r_{v+1,1}, ..., r_{v+1,m+1}, \tilde{K}^o_1, ..., \tilde{K}^o_m)$$

As a result of the heritage from CryptoNote, the values $\tilde{K}^o_j$ are not referred to as part of the signature, but rather as *images* of the private keys $k^o_{\pi,j}$. These *key images* are normally stored separately in the transaction structure as they are used to detect double-spending attacks.

With this in mind and assuming point compression, an MLSAG signature will require $((v+1)\cdot(m+1)+1)\cdot 32$ bytes of storage where v is the mixin level and m is the number of inputs. In other words, a transaction with 1 input and a total ring size of 32 would consume $(32\cdot 2+1)\cdot 32 = 2080$ bytes.

To this value we would add 32 bytes to store the key image of each input, for $m \cdot 32$ bytes of storage, and additional space to store the ring member offsets in the blockchain (see Appendix A). These offsets are used by verifiers to find each MLSAG signature's ring members' output keys and commitments in the blockchain, and are stored as variable length integers, hence we can not exactly quantify the space needed.

Range proofs (outputs)

From Section 3.4, Section 4.3, and Section 5.4.3 we know that a Monero Borromean signature for range proofs takes the form of an n-tuple

$$\sigma = (c_1, r_{0,1}, r_{0,2}, r_{1,1}..., r_{63,2})$$

[6] Verifiers must also check the key image is a member of the generator's subgroup (recall Section 2.2.1) by seeing if $l\tilde{K}^o_j = 0$, as it is sometimes possible to add an EC point with subgroup order equal to a multiple of l to the key image and still produce a verifiable signature. See [26] for more details.

Ring keys are considered part of ring signatures. However, in this case it is only necessary to store the commitments C_j, as the ring key counterparts $C_j - 2^j H$ can be easily derived (for verification purposes).

Respecting this convention, a range proof will require $(1+64\cdot 2+64)32 = 6176$ bytes per output.

5.5 Ring Confidential Transactions of type `RCTTypeSimple`

In the current Monero code base, transactions having more than one input are signed using a different scheme, referred to as `RCTTypeSimple`.

The main characteristic of this approach is that instead of signing the entire set of inputs as a whole, the sender signs each of the inputs individually.

Among other things, this means one can't use commitments to zero in the same way as for `RCTTypeFull` transactions. A public key zG is a commitment to zero if and only if the sender knows the corresponding private key z. If the amounts alone do not sum to zero, then due to the hardness of determining γ such that $H = \gamma G$, it would not be possible to know z.

In more detail, assume that Alice wants to sign input j. Imagine for a moment we could sign an expression like this

$$C_j^a - \sum_t C_t^b = x_j G - \sum_t y_t G + (a_j - \sum_t b_t)H$$

Since $a_j - \sum_t b_t \neq 0$, Alice would have to solve the DLP for $H = \gamma G$ in order to obtain the private key of the expression, something we have assumed to be a computationally difficult problem.

5.5.1 Amount Commitments

As explained, if inputs are decoupled from each other the sender is not able to sign an aggregate commitment to zero against the outputs of the current transaction. On the other hand, signing each input individually implies an intermediate approach. The sender could create new commitments to the input amounts and commit to zero with respect to each of the previous outputs being spent. In this way, the sender could prove the transaction takes as input only the outputs of previous transactions.

In other words, assume that the amounts being spent are $a_1, ..., a_m$. These amounts were outputs in previous transactions, in which they had commitments

$$C_j^a = x_j G + a_j H$$

The sender can create new commitments to the same amounts but using different blinding factors, that is

$$C_j'^a = x_j' G + a_j H$$

Clearly, she would know the private key of the difference between the two commitments:

$$C_j^a - C_j'^a = (x_j - x_j')G$$

hence, she would be able to use this value as a *commitment to zero* for each input. Let us say $(x_j - x_j') = z_j$, and call each $C_j'^a$ a *pseudo output commitment.*

Similarly to `RCTTypeFull` transactions, the sender can include each output's encoded blinding factor (mask) for y_t and amount for b_t in the transaction (see Section 5.4.1), which will allow each receiver t to decode y_t and b_t using the shared secret rK_t^v.

Before committing a transaction to the blockchain, the network will want to verify that the transaction balances. In the case of `RCTTypeFull` transactions, this was simple, as the MLSAG signature scheme implies each sender has signed with the private key of a commitment to zero.

For `RCTTypeSimple` transactions, blinding factors for input and output commitments are selected such that

$$\sum_j x_j - \sum_t y_t = 0$$

This will have the effect that

$$(\sum_j C_j'^a - \sum_t C_t^b) - fH = 0$$

Fortunately, choosing such blinding factors is simple. In the current version of Monero, all blinding factors are random except x_m, which is simply set to

$$x_m = \sum_t y_t - \sum_{j=1}^{m-1} x_j$$

5.5.2 Signature

As we mentioned earlier, in transactions of type `RCTTypeSimple` each input is signed individually. We use the same MLSAG signature scheme as for `RCTTypeFull` transactions, except with different signing keys.

Assume that Alice is signing input j. This input spends a previous output with key $K_{\pi,j}^o$ that had commitment $C_{\pi,j}^a$. Let $C_{\pi,j}'^a$ be a new commitment for the same amount but with a different blinding factor.

Similarly to the previous scheme, the sender selects v unrelated outputs and their respective commitments from the blockchain, to mix with the real j^{th} output

$$K_{1,j}^o, ..., K_{\pi-1,j}^o, K_{\pi+1,j}^o, ..., K_{v+1,j}^o$$
$$C_{1,j}, ..., C_{\pi-1,j}, C_{\pi+1,j}, ..., C_{v+1,j}$$

She can then sign using the following ring:

$$\begin{aligned}\mathcal{R}_j = \{&\{K^o_{1,j}, (C_{1,j} - C'^a_{\pi,j})\},\\ &\dots\\ &\{K^o_{\pi,j}, (C^a_{\pi,j} - C'^a_{\pi,j})\},\\ &\dots\\ &\{K^o_{v+1,j}, (C_{v+1,j} - C'^a_{\pi,j})\}\}\end{aligned}$$

Alice will know the private key $k^o_{\pi,j}$ for $K^o_{\pi,j}$ as well as the one for the commitment to zero ($C^a_{\pi,j}$ - $C'^a_{\pi,j}$), which is z_j. Therefore she can sign the j^{th} input with an MLSAG signature on $\mathcal{R}_j$. Recalling Section 5.4.3, there is no key image for the commitments to zero z_jG, and consequently no corresponding key image term in each input's signature's construction.

Each input in `RCTTypeSimple` transactions is signed individually, applying the scheme described in Section 5.4.3, but using rings like $\mathcal{R}_j$ as defined above.

The advantage of signing inputs individually is that the set of real inputs and commitments to zero need not be placed at the same index π, as they are in the MLSAG algorithm. This means even if one input's origin became identifiable, the other inputs' origins would not.

The message $\mathtt{m}$ signed by each input is essentially the same as for `RCTTypeFull` transactions (see Footnote 5), except it includes pseudo-output commitments for the inputs. Only one message is produced, and each input MLSAG signs it.

5.5.3 Space Requirements

MLSAG signature (inputs)

Each ring $\mathcal{R}_j$ contains $(v+1)\cdot 2$ keys. Using the point compression technique from Section 2.3.2, an input signature σ will require $(2(v+1)+1)\cdot 32$ bytes. On top of this is, the key image $\tilde{K}^o_{\pi,j}$ and the pseudo output commitment $C'^a_{\pi,j}$ leave a total of $(2(v+1)+3)\cdot 32$ bytes per input.

A transaction with 20 inputs using rings with 32 total members will need $((32\cdot 2+3)\cdot 32)20 = 42880$ bytes.

For the sake of comparison, if we were to apply the `RCTTypeFull` scheme to the same transaction, the MLSAG signature and key images would require $(32\cdot 21+1)\cdot 32 + 20\cdot 32 = 22176$ bytes.

Range proofs (outputs)

The size of range proofs remains the same for `RCTTypeSimple` transactions. As we calculated for `RCTTypeFull` transactions, each output will require 6176 bytes of storage.

5.6 Concept summary

To summarize this chapter we present the main content of a transaction, reorganized for conceptual clarity. Real examples can be found in Appendices A and B.

- Type: '1' is `RCTTypeFull`, and '2' is `RCTTypeSimple`
- Inputs: for each input $j \in 1, ..., m$ spent by the transaction author
  - **Ring member offsets**: a list of 'offsets' indicating where a verifier can find input j's ring members $i \in 1, ..., v+1$ in the blockchain (includes the real input)
  - **MLSAG Signature**: σ terms c_1, and $r_{i,j}$ for $i \in 1, ..., v+1$ and input j
  - **Key image**: the key image $\tilde{K}_j^{o,a}$ for input j
  - **Pseudo output commitment** [`RCTTypeSimple` only]: $C_j'^a$ for input j
- Outputs: for each output $t \in 1, ..., p$ to address or subaddress (K_t^v, K_t^s)
  - **One-time (stealth) address**: $K_t^{o,b}$
  - **Output commitment**: C_t^b for output t
  - **Diffie-Hellman terms**: so receivers can compute C_t^b and b_t for output t
    - *Mask*: $y_t + \mathcal{H}_n(rK_t^v, t)$
    - *Amount*: $b_t + \mathcal{H}_n(\mathcal{H}_n(rK_t^v, t))$
  - **Range proof** for output t using a Borromean ring signature
    - *Signature*: σ terms c_1, and $r_{i,j}$ for $i \in 0, ..., 63$ and $j \in 1, 2$
    - *Bit commitments*: C_i for $i \in 0, ..., 63$
- Transaction fee: communicated in clear text multiplied by 10^{12}, so a fee of 1.0 would be recorded as 1000000000000
- Extra: includes the transaction public key rG, or, if at least one output is directed to a subaddress, $r_t K_t^{s,i}$ for each subaddress'd output t and $r_t G$ for each normal address'd output t

Bibliography

[1] How does monero's privacy work? https://www.monero.how/how-does-monero-privacy-work [Online; accessed 04/04/2018].

[2] Modular arithmetic. https://en.wikipedia.org/wiki/Modular_arithmetic [Online; accessed 04/16/2018].

[3] Trust the math? an update. http://www.math.columbia.edu/~woit/wordpress/?p=6522 [Online; accessed 04/04/2018].

[4] Issue with proof of unforgeability of asnl, 2016. https://github.com/monero-project/research-lab/issues/4 [Online; accessed 04/04/2018].

[5] Adam Back. Ring signature efficiency. BitcoinTalk, 2015. https://bitcointalk.org/index.php?topic=972541.msg10619684#msg10619684 [Online; accessed 04/04/2018].

[6] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. *Twisted Edwards Curves*, pages 389–405. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[7] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, Sep 2012.

[8] Daniel J. Bernstein and Tanja Lange. *Faster Addition and Doubling on Elliptic Curves*, pages 29–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[9] Daniel J. Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. Cryptology ePrint Archive, Report 2007/286, 2007. http://eprint.iacr.org/2007/286 [Online; accessed 04/04/2018].

[10] Karina Bjørnholdt. Dansk politi har knækket bitcoin-koden, May 2017. http://www.dansk-politi.dk/artikler/2017/maj/dansk-politi-har-knaekket-bitcoin-koden [Online; accessed 04/04/2018].

[11] David Chaum and Eugène Van Heyst. Group signatures. In *Proceedings of the 10th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'91, pages 257–265, Berlin, Heidelberg, 1991. Springer-Verlag.

[12] Thomas C Hales. The NSA back door to NIST. *Notices of the AMS*, 61(2):190–192.

[13] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[14] Don Johnson and Alfred Menezes. The elliptic curve digital signature algorithm (ecdsa). Technical Report CORR 99-34, Dept. of C&O, University of Waterloo, Canada, 1999. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.472.9475&rep=rep1&type=pdf [Online; accessed 04/04/2018].

[15] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, January 2017.

[16] Alexander Klimov. ECC patents?, October 2005. http://article.gmane.org/gmane.comp.encryption.general/7522 [Online; accessed 04/04/2018].

[17] Joseph K. Liu, Victor K. Wei, and Duncan S. Wong. *Linkable Spontaneous Anonymous Group Signature for Ad Hoc Groups*, pages 325–335. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[18] Greg Maxwell. Confidential transactions. https://people.xiph.org/~greg/confidential_values.txt [Online; accessed 04/04/2018].

[19] Gregory Maxwell and Andrew Poelstra. Borromean ring signatures. 2015. https://pdfs.semanticscholar.org/4160/470c7f6cf05ffc81a98e8fd67fb0c84836ea.pdf [Online; accessed 04/04/2018].

[20] Arvind Narayanan and Malte Möser. Obfuscation in bitcoin: Techniques and politics. *CoRR*, abs/1706.05432, 2017.

[21] Sarang Noether and Brandon Goodell. An efficient implementation of monero subaddresses, mrl-0006, October 2017. https://lab.getmonero.org/pubs/MRL-0006.pdf [Online; accessed 04/04/2018].

[22] Shen Noether. Ring signature confidential transactions for monero. Cryptology ePrint Archive, Report 2015/1098, 2015. http://eprint.iacr.org/2015/1098 [Online; accessed 04/04/2018].

[23] Shen Noether, Adam Mackenzie, and the Monero Research Lab. Ring confidential transactions. *Ledger*, 1(0):1–18, 2016.

[24] Michael Padilla. Beating bitcoin bad guys, August 2016. http://www.sandia.gov/news/publications/labnews/articles/2016/19-08/bitcoin.html [Online; accessed 04/04/2018].

[25] Torben Pryds Pedersen. *Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing*, pages 129–140. Springer Berlin Heidelberg, Berlin, Heidelberg, 1992.

[26] luigi1111 Riccardo "fluffypony" Spagni. Disclosure of a major bug in cryptonote based currencies, May 2017. https://getmonero.org/2017/05/17/disclosure-of-a-major-bug-in-cryptonote-based-currencies.html [Online; accessed 04/10/2018].

[27] Adi Shamir Ronald L. Rivest and Yael Tauman. How to leak a secret. C. Boyd (Ed.): ASIACRYPT 2001, LNCS 2248, pp. 552-565, 2001. https://people.csail.mit.edu/rivest/pubs/RST01.pdf [Online; accessed 04/04/2018].

[28] SJD AB S. Josefsson and I. Liusvaara. Edwards-curve digital signature algorithm (eddsa). Internet Research Task Force (IRTF), 2017. https://tools.ietf.org/pdf/rfc8032.pdf [Online; accessed 04/04/2018].

[29] QingChun ShenTu and Jianping Yu. Research on anonymization and de-anonymization in the bitcoin system. *CoRR*, abs/1510.07782, 2015.

[30] Nicolas van Saberhagen. Cryptonote v2.0. [Online; accessed 04/04/2018].

Appendices

APPENDIX A

`RCTTypeFull` Transaction structure

We present in this chapter a dump from a real Monero transaction of type `RCTTypeFull`, together with explanatory notes for relevant fields.

The dump was obtained executing command `print_tx <TransactionID>` in the `monerod` daemon run in non-detached mode. The first line printed shows the actual command run, which the interested reader can use to replicate our results.

For editorial reasons we have shortened long hexadecimal chains, presenting only the beginning and end as in `0200010c7f[...]409`.

Component `rctsig_prunable`, as indicated by its name, is in theory *prunable* from the blockchain. That is, once a block has been consensuated and the current chain length rules out all possibilities of double-spending attacks, this whole field could be pruned. This is something that has not yet been done in the Monero blockchain, but it is nevertheless a possibility. This would yield considerable space savings.

Key images and ring keys are stored separately, in the non-prunable area of transactions. These components are essential for detecting double-spend attacks and can't be pruned away.

Our sample transaction has 1 input and 2 outputs.

```
1 print_tx  b43a7ac21e1b60ad748ec905d6e03cf3165e5d8c9e1c61c263d328118c42eaa6
2 Found in blockchain at height 1467685
3 0200010c7f[...]409
```

```
{
   "version": 2,
   "unlock_time": 0,
   "vin": [ {
      "key": {
         "amount": 0,
         "key_offsets": [ 799048, 782511, 1197717, 216704, 841722
         ],
         "k_image": "595a612d0df27181c46a8af70a9bd682f2a000124b873ba5d2b9f4b4e4efd672"
      }
   }
   ],
   "vout": [ {
      "amount": 0,
      "target": {
         "key": "aa9595f55f2cfaed3bd2a67453bb064dc7fd454a09c2418d7338782790185fe3"
      }
   }, {
      "amount": 0,
      "target": {
         "key": "0ccb48ed2ebbcaa8e8831111029f3300069cff0d1408acffbfc3810b362ea217"
      }
   }
   ],
   "extra": [ 2, 33, 0, 129, 70, 77, 194, 248, 93, 24, 94, 15, 107, 233, 0, 229, 82,
   175, 243, 123, 58, 204, 135, 171, 100, 101, 192, 42, 187, 157, 168, 222, 98, 192,
   110, 1, 1, 185, 87, 22, 38, 116, 81, 124, 85, 68, 36, 44, 229, 235, 46, 159, 139,
   114, 234, 211, 50, 41, 28, 92, 26, 249, 184, 228, 197, 64, 139, 5
   ],
   "rct_signatures": {
      "type": 1,
      "txnFee": 26000000000,
      "ecdhInfo": [ {
         "mask": "68f508c5515694ce5a33b316b990e8b67a944725c93d806767e61b2e0b13d300",
         "amount": "913372a2424b22bd9712183f5a7c8027c8d9af89b52d1e7d06fd1f87a1e5d20d"
      }, {
         "mask": "fbc3e5bdb36fc58e5800ffc549ab7bd533fadb7e6b64898c82ea620d749fc80e",
         "amount": "b9335c3dc0afb774f812f9f58a412c849f3c828d873f1c16ab102963799d9809"
      }],
      "outPk": [ "cf141f5dfe04df14afad6b451d600aa5826a9be44a76a1630850c1d5951d482e",
                 "e10bb69b66af5dabec765c7f5f7528926088877fa36746833828a0575896ae57"]
   },
   "rctsig_prunable": {
      "rangeSigs": [ {
```

```
48            "asig": "b9b544a7[...]d4c5726e81c4c4b6205dacc05208",
49            "Ci": "bc7ae457[...]fe490458"
50         }, {
51            "asig": "9c457b41[...]545b60c",
52            "Ci": "ce9b4d8e[...]03a6752"
53         }],
54         "MGs": [ {
55            "ss": [[ "a8120b96f5f2ac5bceab37f7d6bf8d86554d87c4af3441007cad92f54a24d908",
56             "2e6bc016297a5d398936c9f45e7a80215138f69e55179b337922e2d51c1a9f00"],
57             ["1e1052a68c38bb88b6e8f257d999c13f1d5f4fa219cc23479ccbfa6b14b5960a",
58              "e914d35eed0d27344fbc3a89b91bd445d433b561efc844c9f466a61ebb5f6d09"],
59             ["e04d011f515461fdbd8d13536c23143dc365d87dd323defb1af834e540a8fc0e",
60              "f9b41a117a1415fec54f1cc16aeef859b2cab1494b9e26a95fc9eaf4f571fa00"],
61             ["de7a7b30795cab310b632f708c6c2546847a5cbcc27ff48e75c1556c3f6f180c",
62              "6218695558359d115e308b008d9aa368c38672732d2fc21c6317ad7d15918c05"],
63             ["0ca70bbdea0e391b1e24e2540f33b48dd9dc554c61ebf23bb3691aab5094e40f",
64              "dafecd436b2448504c0a3a1997b356c141f1d4b5977cc66e5f55592f13731501"]],
65            "cc": "5059757cf06216215955aaa108e8dd40be157856749a9d883bcac611e395a409"
66         }]
67      }
68   }
69
```

Transaction components

- `version` (line 3) - Transaction format version, '2' corresponds to RingCT.
- `vin` (line 7-14) - List of inputs (there's only one)
- `amount` (line 9) - Deprecated (legacy) amount field for type 1 transactions
- `key_offset` (line 10) - Relative offsets with respect to most recent block for ring components: spent output, and mixin outputs. As an illustration, 799048 is to be interpreted as the 799048^{th} transaction counting backwards, starting from within the most recent block. This allows verifiers to find ring member keys and commitments in the blockchain, and makes it obvious those members are legitimate.
- `k_image` (line 12) - Key image $\tilde{K_j}$ from Section 3.3, where $m = j = 1$ for 1 input.
- `vout` (lines 16-27) - List of outputs (there are two)
- `amount` (line 17) - Deprecated amount field for type 1 transactions
- `key` (line 19) - One-time destination key for output t as described in Section 5.2 Also known as the stealth address.

- `extra` (lines 28-32) - Miscellaneous data, including the *transaction public key*, i.e. the share secret rG of Section 5.2
- `rct_signatures` (lines 33-45) - First part of signature data
- `type` (line 34) - Signature type; `RCTTypeFull` is type 1.
- `txnFee` (line 35) - Transaction fee in clear, in this case 0.026 XMR
- `ecdhInfo` (lines 36-42) - 'elliptic curve diffie-hellman info': Encrypted mask and amount of each of the outputs $t \in 1, ..., p$, here $p = 2$
- `mask` (line 37) - Field *mask* at $t = 1$ as described in Section 5.4.1
- `amount` (line 38) - Field *amount* at $t = 1$ as described in Section 5.4.1
- `outPk` (lines 43-44) - Commitments for each output, Section 5.4.2
- `rctsig_prunable` (lines 46-67) - Second part of signature data
- `rangeSigs` (lines 47-53) - Range proofs for output $t \in 1, ..., p$ commitments
- `asig` (line 48) - Borromean signature terms for the range proof of output $t = 1$ (includes c_1 and all r), see Section 5.4.6
- `Ci` (line 49) - Borromean commitments (ring keys) for the range proof on output $t = 1$, as described in Section 5.4.3. As explained in Section 5.4.6 only the commitments C_i need to be stored, as the values $C_i - 2^i H$ can be easily derived by verifiers.
- `MGs` (lines 54-66) - Remaining elements of the MLSAG signature
- `ss` (lines 55-64) - Components $r_{i,j}$ from the MLSAG signature
$$\sigma(\mathfrak{m}) = (c_1, r_{1,1}, ..., r_{1,m+1}, ..., r_{v+1,1}, ..., r_{v+1,m+1}, \tilde{K}_1, ..., \tilde{K}_m)$$
- `cc` (line 65) - Component c_1 from aforementioned MLSAG signature

APPENDIX B

RCTTypeSimple Transaction structure

In this section we show the structure of a sample transaction of type `RCTTypeSimple`. The transaction has 4 inputs and 2 outputs.

```
print_tx 3ebf45fc5f8fd6830378073841228 17d5debfa762c7a7845cb7ccfe9ee20940b
Found in blockchain at height 1469563
020004[...]923b3d70d
{
  "version": 2,
  "unlock_time": 0,
  "vin": [ {
      "key": {
        "amount": 0,
        "key_offsets": [ 1567249, 1991110, 349235, 15551, 3620
        ],
        "k_image": "9661119b4b54529e1be14ef97fbdc0504d17a6c8dfedd55d2455b93a6336bb41"
      }
    }, {
      "key": {
        "amount": 0,
        "key_offsets": [ 2502375, 650851, 337433, 396459, 39529
        ],
        "k_image": "2102414d8edfa229f9ebf32ab90acd9cf23963a8c3b6ba0e181fc1d5782c046c"
      }
    }, {
      "key": {
```

```
        "amount": 0,
        "key_offsets": [ 1907097, 696508, 806254, 510195, 6709
        ],
        "k_image": "de14ec8958b311bd38a05aa3fb08fdd360001f1b9c060264eecdd8c08c9e83c4"
      }
    }, {
      "key": {
        "amount": 0,
        "key_offsets": [ 1150236, 1943388, 788506, 37175, 7462
        ],
        "k_image": "e470f77dd5a4149210cb61ee107e73caea1ef9f61d05384e3bd4372fdc85bf17"
      }
    }
  ],
  "vout": [ {
      "amount": 0,
      "target": {
        "key": "787cad1ebb181e1fc04b24d4d06c3d2882c38b262a7635de8ad487c536e40a12"
      }
    }, {
      "amount": 0,
      "target": {
        "key": "faf4137928392b39ccf0a830c0261573009959787697f9d4fb769c25781fb911"
      }
    }
  ],
  "extra": [ 1, 20, 56, 120, 111, 89, 89, 64, 10, 98, 96, 255, 202, 235, 203,
             255, 2, 197, 176, 147, 61, 60, 41, 145, 207, 178, 212, 71, 37, 69, 19,
             147, 205],
  "rct_signatures": {
    "type": 2,
    "txnFee": 558805800000,
    "pseudoOuts": [ "64dea29ac5560f93773240d58ca5768b879fd3c95e0b3b50a80ec36a6ff3a6da",
                    "a60e7a00e65ff2a6299b92b166a629e9b0d62f6df50e40535140716757efe4c0",
                    "4c67403adbc9dc0ca5a1a6abc846ab6d232dc3fa295099b3c7a9d005bac60eba",
                    "635b26d78117d77899859ecb61e10125c3956a5c113b932f33c92c561acddaa3"],
    "ecdhInfo": [ {
        "mask": "ccffac42a86bec7b36ce9957cbdfe481d419bc5353335d0c236c347aea758d0c",
        "amount": "c0d6cf3e1db55dd459b73faf34d7339c3fa1b3d3356cfb2adc3faf798264b00e"
      }, {
        "mask": "62cc846003d9c5425c6cf33b30754a4c044f5d9d02621460e45664b886673109",
        "amount": "726dbacad62022bf0f5af05c72482b3f040d631d3f576b5e2615ea72f84c5f06"
      }],
    "outPk": [ "cb3b729b4fca6e66736666201633e3f905c367a2f3d18e31fe3d3c18d2be93fd",
```

```
                "1e8c86b7f211a99e1762bf62254efe65ea5c5328b62b0ea8d679b2e52800f633"]
  },
  "rctsig_prunable": {
    "rangeSigs": [ {
        "asig": "9bb7cce09[...]61de7ce0a",
        "Ci": "50dfd2e8[...]b3a7c8fca1b1"
      }, {
        "asig": "e3905fa5c[...]b5213444908",
        "Ci": "595c2cec5f2[...]72a628ab5c"
      }],
    "MGs": [ {
        "ss": [ [ "3b2d26ea7628015fd8317e4e298ceda6b534ac894b83f7b6190a353cee6ec702",
                  "2d772ad7b7ff2ba8a1a66c8d69c0a0d49d72808eaf803c59f13c3d78b653440c"],
                [ "c188891fb37d76305f0209222f52d22ede43018facfe91f949ecb8dcf709b30a",
                  "303896ca67ea7969544641d5bb94a436558bcf6522bb9bc77bd1abb5f2146c08"],
                [ "e01c88b7308403a9dd023d9eacf1ade17ab0fa54250148431b5a33c98e636100",
                  "ba36e34e5245e89c7c21af845b949cf3e82188df639390f094e31c9ba773060c"],
                [ "37175d72d2bef3f8bd9e65fb2861f7bce91e3b1e30278b2dcf26112831ac9405",
                  "8e77a5dd641a89e86dbe0708e8f59d0e2dc9fe4ddfd9b367c3a93522198a4706"],
                [ "fbb4f94f9ce0f081421e63677a63d5914f0536a481d57b6e5fc5379c84dfcb05",
                  "1ec40aa3c8a94c6b1915b7796423b0d7d6011aa2d6af636aff309b832f193408"]],
        "cc": "a03119e4257cca37f89ac3e97f0598b712c79162c73932d58ab4ce08c4ad6709"
      }, {
        "ss": [ [ "f7aedeec462d7588330c71589fde5f0f234a627a6e5ed72cff34825a04d41707",
                  "31d7a5ba4e782db5c0704ab751a2ef8c4732f3cf699bc8f9994e79a97cd3190e"],
                [ "00e1d1ecdf31fdf7d57661f2234bfc859cfdc4dbfdfd0f5eec0576ef22592203",
                  "fdf08b803fa6de18bf0e0dc6855e877877bda0101eceb81e2223fe0175606300"],
                [ "3397ba3f9e8db066e3c4911b896debefbc73efbac4988e6aff5731ff8db15405",
                  "43d2b03d5263de99f56c256e646be503edd63dd03d377a469379fbf487e8600e"],
                [ "31afd1d5c3b07170ac127605fc35cbcc19cf963a35b2ff8f804e17e3b804000d",
                  "3a3bc124a10b0cf416656f8f682a427445140895440cca644c6aa38966399f0c"],
                [ "f499f0e4922d5cba35e3cc033489b60ec7ea26ff19cc9dd29357670f4bf8790b",
                  "1a4732b31f0f1a7d3322be5c4baca098f0a032c192bf9f8a6b5fd83cbdd9d401"]],
        "cc": "095fcab7ebf64c2ffecdacf11b70f97f0e709de0a84b3a13abca627f9df2c901"
      }, {
        "ss": [ [ "1808924b4154118c48f0b305562b6ffba86f38c64d4d8a087823f3383cddd006",
                  "4b18544be50aee8c4594b568d6be741c155a132cb83392d9b1a4cf35c3d5760c"],
                [ "9a95eeeecf3c3a48c43873a372c263357ff5f258a7bf8ed29a767237b0b0f202",
                  "4f1ea4d9d4b56db780dd078a3e8219d0f54eaccc197901671002a206f063cb0e"],
                [ "2c800017cab2b8388f58fed0d61a46570f64cacd8fabc4e84ddee735b3135f0a",
                  "458016f6fdf58b329fae0f929226ee2b8e410a14db8c6ede9b74fb718de71507"],
                [ "ea0fcaf793602dce25c8b2c4d17163f4298933b3fb09874307d8cde9a63c2c0c",
                  "df76fbcd336c07f37e90e1a0d0db1ba49519ba4325062228bc9242af2c525703"],
                [ "e3a7a0477eeb602a9a8203a6a496cca90c4769d57410246c4c8d665df34df900",
```

```
111                   "44d206154f0ca85e12a92eefdbc3784e17e701a32ff93b550467679f67500c0d"]],
112           "cc": "6f80de1c1d566776d2831f15c9a85fb1d8e8cecfd0d2753b318f0e84d89d3b08"
113         }, {
114           "ss": [ [ "5b82b4644b57e3d623de7c72c6ebd52959815c12c80b479e4cbe5437cf67640c",
115                     "b70e0b69c75faba6a1630429f9f497db351347c210467f69e1b1c5f1a72afe02"],
116                   [ "f25a93a98f980cf489eb8f69369f4ec63eaea91fd677decab9b6ca0fe2feb606",
117                     "124536c374cb6023a6aa6f22ae6e115a1ba12cb36c48f5f5ad43ce90f471da02"],
118                   [ "151ddc82322456d7f31b8b4b2290098c3bf2428370c7ef325660b5463ff26404",
119                     "2fb9d2979e16c2b1131686bb85068ec559f9c6c64581e609b451bb2cd9d5740d"],
120                   [ "c03bed01d6ad60b3da5d2c88cf2e5023b51133c37e4917511715a11f09d8740d",
121                     "432e01c2075ab6361af8636cc1c9254e12db98f5c323088792dfb42a1c894401"],
122                   [ "52206d801214e70d20ee7ca53823c143aa06c3d1b22b118cc8a15c9f861f0102",
123                     "563c134f56f7a290e0980877e93bc4b08651e53dade079b1e6c066b70fb81406"]],
124           "cc": "4102cd245db3e0d7c0e2280cfdba38b9b7a7ad8715b8fe68c1170cf923b3d70d"
125         }]
126     }
127 }
128
129
130
```


Transaction components

What follows is a short explanation of the most important elements in this type of transaction. We only mention components that are specific to, or differ from, the previous `RCTTypeFull` transaction type.

- `type` (line 53) - Signature type, in this case the value 2, corresponding to `RCTTypeSimple` transactions
- `pseudoOuts` (lines 55-58) - Pseudo-output commitments $C_j'^a$, as described in Section 5.5.1. Please recall that the sum of these commitments will equal the sum of the 2 output commitments of this transaction (plus the transaction fee fH).