Zero to Monero

Privacy in the blockchain: First Edition
June 4, 2018 Draft 0.20.0

Kurt M. Alonso[1,2]

koe[1,3]

[1] Dear reader,
Welcome to our report, we hope you enjoy it!

Do you feel Monero has enough documentation?

If you find this report valuable you can donate Monero (XMR), empowering us to keep this report up-to-date with new developments, to assemble new reports, and basically to survive, here:

43sHzpng7oFAUMrRzg5RSg2XoYQbCSRYBRt4PV61ByqwY9ovfRGqMenj3ZkEQEaXsf7edQtTitH5xKG3t27kkKafKX4oFzY

We can keep going with your support!

Please note this report is released into the public domain.

[2] kurt@oktav.se
[3] ukoe@protonmail.com; for questions and comments, we recommend CC'ing both authors.

Abstract

Cryptography. It may seem like only mathematicians and computer scientists have access to this obscure, esoteric, powerful, elegant topic. In fact, many kinds of cryptography are simple enough that anyone can learn their fundamental concepts.

Many people know cryptography is used to secure communications, whether they be coded letters or private digital interactions. Another application is in so-called cryptocurrencies. These digital moneys use cryptography to ensure, first and foremost, that no piece of money can be duplicated or created at will. To that end, cryptocurrencies typically rely on 'blockchains', creating public, distributed ledgers containing records of currency transactions that can be verified by third parties.

It might seem at first glance that transactions need to be sent and stored in plain text format in order to make them publicly verifiable. In fact, it is possible to conceal participants of transactions, as well as the amounts involved, using cryptographic tools that nevertheless allow transactions to be verified and agreed upon by observers [59]. This is exemplified in the cryptocurrency Monero.

We endeavor here to teach anyone who knows basic algebra and simple computer science concepts like the 'bit representation' of a number not only how Monero works at a deep and comprehensive level, but also how useful and beautiful cryptography can be.

For our experienced readers: Monero is a standard one-dimensional distributed acyclic graph (DAG) cryptocurrency blockchain where transactions are based on elliptic curve cryptography using curve Ed25519, transaction inputs are signed with Schnorr-style multilayered linkable spontaneous anonymous group signatures (MLSAG), and output amounts (communicated to recipients via ECDH) are concealed with Pedersen commitments and Schnorr-style Borromean ring signatures. Much of this report is spent explaining these ideas.

Contents

1 Introduction **1**

1.1 Objectives 2

1.2 Readership 2

1.3 Origins of the Monero cryptocurrency 3

1.4 Outline 3

1.5 Disclaimer 4

2 Basic Concepts **5**

2.1 A few words about notation 5

2.2 Modular arithmetic 6

2.2.1 Modular addition and multiplication 7

2.2.2 Modular exponentiation 8

2.2.3 Modular multiplicative inverse 8

2.2.4 Modular polynomials 9

2.3 Elliptic curve cryptography 10

2.3.1 What are elliptic curves? 10

2.3.2 Public key cryptography with elliptic curves 12

2.3.3 Diffie-Hellman key exchange with elliptic curves . . . 12
2.3.4 Schnorr signatures and the Fiat-Shamir transform . . . 13
2.3.5 Signing messages . . . 15
2.4 Curve Ed25519 . . . 16
2.4.1 Binary representation . . . 17
2.4.2 Point compression . . . 17
2.4.3 EdDSA signature algorithm . . . 18

3 Ring Signatures **20**
3.1 Linkable Spontaneous Anonymous Group (LSAG) signatures . . . 21
3.2 Back's Linkable Spontaneous Anonymous Group (bLSAG) signatures . . . 24
3.3 Multilayer Linkable Spontaneous Anonymous Group (MLSAG) signatures . . . 25
3.4 Borromean ring signatures . . . 27

4 Pedersen Commitments **29**
4.1 Pedersen commitments . . . 29
4.2 Amount commitments . . . 30
4.3 Range proofs . . . 31
4.4 Range proofs in a blockchain . . . 32

5 Monero Transactions **34**
5.1 User keys . . . 34
5.2 One-time (stealth) addresses . . . 34
5.2.1 Multi-output transactions . . . 36
5.3 Subaddresses . . . 36
5.3.1 Sending to a subaddress . . . 37
5.4 Integrated addresses . . . 37
5.5 Transaction types . . . 39
5.6 Ring Confidential Transactions of type `RCTTypeFull` . . . 40
5.6.1 Amount commitments . . . 40

5.6.2 Commitments to zero . . . 41
5.6.3 Signature . . . 42
5.6.4 Transaction fees . . . 43
5.6.5 Avoiding double-spending . . . 44
5.6.6 Space and verification requirements . . . 44
5.7 Ring Confidential Transactions of type `RCTTypeSimple` . . . 45
5.7.1 Amount commitments . . . 46
5.7.2 Signature . . . 47
5.7.3 Space and verification requirements . . . 48
5.8 Concept summary: Monero transactions . . . 49
5.8.1 Storage and verification requirements . . . 50

6 Multisignatures in Monero **51**
6.1 Communicating with co-signers . . . 52
6.2 Key aggregation for addresses . . . 52
6.2.1 Naive approach . . . 52
6.2.2 Drawbacks to naive approach . . . 53
6.2.3 Robust key aggregation . . . 54
6.3 Thresholded Schnorr-like signatures . . . 55
6.3.1 Simple threshold Schnorr-like signatures . . . 55
6.3.2 Back's Linkable Spontaneous Threshold Anonymous Group (bLSTAG) signatures . . . 57
6.3.3 Multilayer Linkable Spontaneous Threshold Anonymous Group (MLSTAG) signatures . . . 58
6.4 MLSTAG Ring Confidential signatures for Monero . . . 58
6.4.1 `RCTTypeFull` N-of-N multisig . . . 59
6.4.2 `RCTTypeSimple` N-of-N multisig . . . 61
6.4.3 Simplified communication . . . 62
6.5 Recalculating key images . . . 63

6.6 Smaller thresholds . . . 64
6.6.1 1-of-N key aggregation . . . 65
6.6.2 (N-1)-of-N key aggregation . . . 65
6.6.3 M-of-N key aggregation . . . 67
6.7 Key families . . . 68
6.7.1 Family trees . . . 69
6.7.2 Nesting multisig keys . . . 69
6.7.3 Implications for Monero . . . 71

7 The Monero Blockchain **72**
7.1 Digital currency . . . 72
7.1.1 Shared version of events . . . 73
7.1.2 Simple blockchain . . . 74
7.2 Difficulty . . . 74
7.2.1 Mining a block . . . 75
7.2.2 Mining speed . . . 76
7.2.3 Consensus: largest cumulative difficulty . . . 76
7.2.4 Mining in Monero . . . 77
7.3 Money supply . . . 78
7.3.1 Block reward . . . 78
7.3.2 Block size penalty . . . 80
7.3.3 Dynamic minimum fee . . . 80
7.3.4 Emission tail . . . 81
7.4 Blockchain structure . . . 81
7.4.1 Transaction ID . . . 82
7.4.2 Miner transaction . . . 82
7.4.3 Merkle tree . . . 83
7.4.4 Blocks . . . 84

Bibliography **85**

Appendices 87

A `RCTTypeFull` **Transaction structure** 89

B `RCTTypeSimple` **Transaction structure** 93

C Block content 97

D Genesis block 100

CHAPTER 1

Introduction

In the digital realm it is often trivial to make endless copies of information, with equally endless alterations. For a currency to exist digitally and be widely adopted, its users must believe its supply is strictly limited. A money recipient must be able to verify they are not receiving counterfeit coins, or coins that have already been sent to someone else. To accomplish that without requiring the collaboration of any third party like a central authority, its supply and complete transaction history must be publicly verifiable.

We can use crytographic tools to allow data registered in an easily accessible database - the blockchain - to be virtually immutable and unforgeable, with legitimacy that cannot be disputed by any party.

Cryptocurrencies store transactions in the blockchain, which acts as a public ledger[1] of all the currency operations that have been verified. Most cryptocurrencies store transactions in clear text, to facilitate verification of transactions by the community of users.

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 [45]. However, with appropriate tools it is possible to analyze flows and to a large extent link true senders with receivers [57, 26, 49].

[1] In this context ledger just means a record of all currency creation and exchange events. Specifically, how much money was transferred in each event and to whom.

In contrast, the cryptocurrency Monero (Moe-neh-row) 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 [8].

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 [58] [11], yet it displays a steady growth in popularity[2]. 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.

Moreover, for those without a background in mathematics, learning the basics of elliptic curve cryptography, which Monero uses extensively, can be a haphazard and frustrating endeavor.

We intend to palliate this situation by introducing the fundamental concepts necessary to understand elliptic curve cryptography, reviewing algorithms and cryptographic schemes, and collecting in-depth information about Monero's inner workings.

To provide the best experience for our readers, we have taken care to build a constructive, step-by-step description of the Monero cryptocurrency.

In the first edition of this report we have centered our attention on version 7 of the Monero protocol, corresponding to version 0.12.x.x of the Monero software suite. All transaction related mechanisms described here belong to those versions. Deprecated transaction schemes have not been explored to any extent, even if they may be partially supported for backward compatibility.

1.2 Readership

We anticipate many readers will encounter this report with little to no understanding of discrete mathematics, algebraic structures, cryptography, and blockchains. We have tried to be thorough enough that laymen from all perspectives may learn Monero without needing external research.

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

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 half-way 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 [58]. Monero means 'money' in the language Esperanto, and its plural form is Moneroj (Moe-neh-rowje, similar to Moneros but using the -ge from orange).

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 [59]. It offered receiver anonymity through the use of one-time addresses, and sender ambiguity 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 [40] and integrated into ring signatures based on Shen Noether's recommendations in [48].

1.4 Outline

As mentioned, 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 parts of 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, and their mathematical antecedents. In Chapter 2 we develop essential aspects of elliptic curve cryptography.

Chapter 3 outlines the ring signature algorithms that will be applied to achieve confidential transactions.

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

With all the components in place, we explain the transaction schemes used in Monero in Chapter 5.

We use Chapter 6 to describe a multisignature method that allows multiple people to send and receive money collaboratively.

Finally, we unfold the Monero blockchain in Chapter 7.

Appendices A and B explain the structure of sample transactions from the blockchain. Appendix C explains the structure of blocks (including block headers and miner transactions) in Monero's blockchain, while Appendix D brings our report to a close by explaining the structure of Monero's genesis block. These provide a connection between the theoretical elements described in earlier sections with their real-life implementation.

We use margin notes to indicate where Monero implementation details can be found in the source code. There is usually a file path, such as src/ringct/rctOpps.cpp, and a function, such as signMultisig(). Note: any instance of '-' indicates split text, such as crypto- note → cryptonote.

Isn't this useful?

1.5 Disclaimer

All signature schemes, applications of elliptic curves, and Monero implementation details should be considered descriptive only. Readers considering serious practical applications (as opposed to a hobbyist's explorations) should consult original sources and technical specifications (which we have cited where possible). Signature schemes need well-vetted security proofs, and Monero implementation details can be found in Monero's source code.

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 complicated 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 we have neglected questions like the bit representation of items, or, in some cases, how to carry out concrete operations. Moreover, students of mathematics may find we disregarded explanations of abstract algebra.

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.

src/crypto/keccak.c

A hash function takes in some message $\mathfrak{m}$ of arbitrary length and returns a hash h (or *message digest*) of fixed length, with each possible output equiprobable for a given input. Cryptographic hash functions are difficult to reverse, have an interesting feature known as the *large avalanche effect* which can cause very similar messages to produce very dissimilar hashes, and it is hard to find two messages with the same message digest.

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 Modular arithmetic

Most modern cryptography begins with modular arithmetic, which in turn begins with the modulus operation (denoted 'mod'). We only care about the positive modulus, which always returns a positive integer.

The positive modulus is here defined for $a \pmod{b} = c$ as $a = bx + c$, where $0 \leq c < b$ and x is a signed integer which gets discarded (note that b is an integer bigger than zero).

Let's imagine a number line. We stand at point a, then walk toward zero with each step $= b$ until we reach an integer ≥ 0 and $< b$. That is c. For example, $4 \pmod{3} = 1$, $-5 \pmod{4} = 3$, and so on. Readers may recognize c is similar to the 'remainder' after dividing.

Note that, if $a \leq n$, $-a \pmod{n}$ is the same as $n - a$.

[1] The Keccak hashing algorithm forms the basis for the NIST standard *SHA-3* [19].

2.2.1 Modular addition and multiplication

In computer science it is important to avoid large numbers when doing modular arithmetic. For example, if we have $29 + 87 \pmod{99}$ and we aren't allowed variables with three or more digits (such as $116 = 29 + 87$), then we can't compute $116 \pmod{99} = 17$ directly.

To perform $c = a + b \pmod{n}$, where a and b are each less than the modulus n, we can do this:

- Compute $x = n - a$. If $x > b$ then $c = a + b$, otherwise $c = b - x$.

We can use modular addition to achieve modular multiplication with an algorithm called 'double-and-add'. Let us demonstrate by example. Say we want to perform $7 * 8 \pmod{9} = 2$. This is the same as

$$7 * 8 = 8 + 8 + 8 + 8 + 8 + 8 + 8 \pmod{9}$$

Now break this into groups of two.

$$(8 + 8) + (8 + 8) + (8 + 8) + 8$$

And again, by groups of two.

$$[(8 + 8) + (8 + 8)] + (8 + 8) + 8$$

The total number of + point operations falls from 6 to 4 because we only need to find $(8+8)$ once.

Double-and-add is implemented by converting the first number (the 'multiplicand') to binary (in our example, $7 \rightarrow [0111]$), then going through the binary array and doubling and adding.

Let's make an array $A = [0111]$ and index it 0,1,2,3. A[0] = 0 is the first element of A and is the most significant bit. We set a result variable to be initially $s = 0$, and set a sum variable to be initially $s = 8$ (more generally, we start with $S = b$). We follow this algorithm:

1. Iterate through: $i = (A_{size} - 1, ..., 0)$
   (a) If A[i] == 1, then $r = r + s \pmod{n}$.
   (b) Compute $s = s + s \pmod{n}$.
2. Use the final r as result.

In our example, this sequence appears:

1. $i = 3$
   (a) A[3] = 1, so $r = 0 + 8 \pmod{9} = 8$
   (b) $s = 8 + 8 \pmod{9} = 7$
2. $i = 2$

   (a) A[2] $= 1$, so $r = 8 + 7 \pmod 9 = 6$
   (b) $s = 7 + 7 \pmod 9 = 5$

3. $i = 1$
   (a) A[1] $= 1$, so $r = 6 + 5 \pmod 9 = 2$
   (b) $s = 5 + 5 \pmod 9 = 1$

4. $i = 0$
   (a) A[0] $= 0$, so r stays the same
   (b) $s = 1 + 1 \pmod 9 = 2$

5. $r = 2$ is the result

2.2.2 Modular exponentiation

It's obvious that $8^7 \pmod 9$ is the same as $8*8*8*8*8*8*8 \pmod 9$. Just like double-and-add, we can do square-and-multiply. For $a^e \pmod n$:

1. Define $e_{scalar} \rightarrow e_{binary}$; $A = [e_{binary}]$; $r = 1$; $m = a$
2. Iterate through: $i = (A_{size} - 1, ..., 0)$
   (a) If A[i] == 1, then $r = r * m \pmod n$.
   (b) Compute $m = m * m \pmod n$.
3. Use the final r as result.

2.2.3 Modular multiplicative inverse

Sometimes we need to calculate $1/b \pmod n$, or in other words $b^{-1} \pmod n$.

In the equation $a \equiv b \pmod n$, a is *congruent* to $b \pmod n$, which just means $a \pmod n = b \pmod n$.

The modular multiplicative inverse is defined as an integer c such that, for $c = a^{-1} \pmod n$, $ac \equiv 1 \pmod n$ for $0 \leq x < n$ and for a and n relatively prime. Relatively prime just means they don't share any divisors except 1. We could also say the fraction a/n can't be reduced/simplified.[2]

[2] We want to point out the modular multiplicative inverse has a rule stating:
If $ac \equiv b \pmod n$ with a and n relatively prime, the solution to this linear congruence is given by $c = a^{-1}b \pmod n$.[10]
This means we can do $c = a^{-1}b \pmod n \rightarrow ca \equiv b \pmod n \rightarrow a \equiv c^{-1}b \pmod n$.

We can use square-and-multiply to compute the modular multiplicative inverse when n is a prime number because of *Fermats' little theorem*:

$$\begin{aligned} a^{n-1} &\equiv 1 \pmod{n} \\ a * a^{n-2} &\equiv 1 \pmod{n} \\ c \equiv a^{n-2} &\equiv a^{-1} \pmod{n} \end{aligned}$$

More generally (and more rapidly), the so-called extended Euclidean algorithm can find c like this (short and sweet just to expose it):

1. Set $r = 0$; $r' = 1$; $q = n$; $q' = a$.
2. While $r' \neq 0$, iterate through these steps:
   - $quotient = integer(q/q')$
   - $(r, r') = (r', r - quotient * r')$ [3]
   - $(q, q') = (q', q - quotient * q')$
3. If $q \leq 1$ then (if $r < 0$ then $c = r + n$, else $c = r$), else no solution.

2.2.4 Modular polynomials

Suppose we have an equation $c = 3 * 4 * 5 \pmod{9}$. Computing this is straightforward. Given some operation $\circ$ (for example, $\circ = *$) between two variables A and B:

$$(A \circ B) \pmod{n} = [A \pmod{n}] \circ [B \pmod{n}] \pmod{n}$$

In our example, we set $A = 3 * 4$, $B = 5$, and $n = 9$:

$$\begin{aligned} (3 * 4 * 5) \pmod{9} &= [3 * 4 \pmod{9}] * [5 \pmod{9}] \pmod{9} \\ &= [3] * [5] \pmod{9} \\ &= 6 \end{aligned}$$

Now we have a way to do modular subtraction.

$$\begin{aligned} A - B \pmod{C} &\rightarrow A + (-B) \pmod{n} \\ &\rightarrow [A \pmod{n}] + [-B \pmod{n}] \pmod{n} \end{aligned}$$

The same principle would apply to something like $x = (a - b * c * d)^{-1}(e * f + g^h) \pmod{n}$.

[3] We use r and r' from the previous round to compute the current round's r and r' at the same time.

2.3 Elliptic curve cryptography

2.3.1 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 $\{0, 1, 2, ..., q-1\}$. Addition and multiplication $(+, \cdot)$ and negation $(-)$ are calculated $\pmod q$.

"Calculated $\pmod q$" means $\pmod q$ 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 $37 \pmod{29} = 8$. Also, $-13 = (-13) \pmod{29} = 16$.

Typically, an elliptic curve with a given (a, b) pair is defined as the set of points (x, y) satisfying a *Weierstraß* equation:[4]

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

The cryptocurrency Monero uses a special curve belonging to the category of so-called *Twisted Edwards* curves [23], which are commonly expressed as (for a given (a, d) pair):

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

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 fewer arithmetic operations, resulting in faster cryptographic algorithms (see Bernstein *et al.* in [25] for details).

Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be two points belonging to a Twisted Edwards elliptic curve (henceforth known simply as an EC). We define addition on points by defining $P_1 + P_2 = (x_1, y_1) + (x_2, y_2)$ as the point $P_3 = (x_3, y_3)$ where

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

These formulas for addition also apply for point doubling; that is, when $P_1 = P_2$. To subtract a point, invert its coordinates $(x, y) \to (-x, y)$ [23] and use 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[5] under the addition operation described. Each time two curve points are added together, P_3 is a point on the 'original' elliptic curve, or in other words all $x_3, y_3 \in \mathbb{F}_q$.

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

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

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*[6]. All P in EC generate a cyclic subgroup. If P generates a subgroup whose order is prime then all the included points (except for the point-at-infinity) generate that same 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*).

To find the order, u, of any given point 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) and h is the so-called *cofactor* which could be as small as 1 or 2. One point in the subgroup of size l is usually selected to be the generator G as a convention.[7] For every other point P in that subgroup there exists an integer $0 \leq n < l$ satisfying $P = nG$.

Let's expand our understanding of subgroups. Say there is a point P' with order N, where $N = hl$. Any other point P_i can be found with some integer n_i such that $P_i = n_i P'$. 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 there are only h variations of $n' \pmod N$ (including the point at infinite for $n' = 0$) because when $n' = hl$ it cycles back to 0: $hlP' = 0P' = 0$. So, there are only h points in EC where hP will equal 0.

With this new understanding it is clear we can use the following algorithm to find points in the subgroup of order l:

1. Find N of the elliptic curve EC, choose subgroup order l, compute $h = N/l$.
2. Choose a random point P' in EC.

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

[7] In Monero $G = (x, 4/5)$ [24], where x is the 'even', or $b = 0$, variant based on point decompression of $y = 4/5 \pmod q$. See Section 2.4.2.

3. Compute $P = hP'$.
4. If $P = 0$ return to step 2, else P is in the subgroup of order l.

Calculating the scalar product between any integer n and any point P, nP, is not difficult, whereas finding n such that $P_1 = nP_2$ is known to be computationally hard. By analogy to modular arithmetic, this 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.

The scalar product nP is equivalent to $(((P + P) + P)\dots)$. Though not always the most efficient approach, we can use double-and-add like in Section 2.2.1. To get the sum $R = nP$, remember we use the + point operation discussed in Section 2.3.1.

1. Define $n_{scalar} \rightarrow n_{binary}$; $A = [n_{binary}]$; $R = 0$, the point-at-infinity; $S = P$
2. Iterate through: $i = (A_{size} - 1, ..., 0)$
   (a) If A[i] == 1, then R += S.
   (b) Compute S += S.
3. Use the final R as result.

2.3.2 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.

2.3.3 Diffie-Hellman key exchange with elliptic curves

A basic *Diffie-Hellman* [28] 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.

2.3.4 Schnorr signatures and the Fiat-Shamir transform

In 1989 Charles Schnorr published a now famous interactive authentication protocol [54], generalized by Maurer in 2009 [38], that allowed someone to prove they know the private key k of a given public key K without revealing any information about it [40]. It goes something like this:

1. The prover generates a random integer $\alpha \in_R \mathbb{Z}_l$,[8] computes αG, and sends αG to the verifier.
2. The verifier generates a random *challenge* $c \in_R \mathbb{Z}_l$ and sends c to the prover.
3. The prover computes $r = \alpha + c * k$ and sends r to the verifier.
4. The verifier computes $R' = rG$ and $R = \alpha G + c * K$, and checks $R' \stackrel{?}{=} R$.

The verifier can compute $R = \alpha G + c * K$ before the prover, so providing c is like saying "I challenge you to find the discrete logarithm of R". A challenge the prover can only overcome by knowing k (except with negligible probability).

If α was chosen randomly by the verifier then r is randomly distributed [55] and k is information-theoretically secure.[9] However, if the prover reuses α to prove his knowledge of k, anyone who knows both challenges in $r = \alpha + c * k$ and $r' = \alpha + c' * k$ can compute k (two equations, two unknowns).[10]

$$k = \frac{r - r'}{c - c'}$$

If the prover knew c from the beginning (i.e. if the verifier secretly gave it to her), she could generate a random r and compute $\alpha G = rG - cK$. When she later sends r to the verifier, she 'proves' knowledge of k without ever having known it. Someone observing the transcript of events between prover and verifier would be none the wiser. The scheme is not *publicly verifiable*. [40]

In his role as challenger, the verifier spits out a random number after receiving αG, making him equivalent to a *random function*. Random functions, such as hash functions, are known as random oracles because computing one is like requesting a random number from someone [40].[11]

[8] Notation: $\alpha \in_R \mathbb{Z}_l$ means α is randomly selected from $\{0, 1, 2, ..., l - 1\}$. In other words, $\mathbb{Z}_l$ is all integers (mod l).

[9] A cryptosystem with information-theoretic security is one where even an adversary with infinite computing power could not break it, because they simply wouldn't have enough information.

[10] If the prover is a computer, you could imagine someone 'cloning'/copying the computer after it generates α, then presenting each copy with a different challenge. In security proofs, phrases like 'forking lemma' and 'rewind on success' are analagous to this cloning attack. See for example [37].

[11] More generally, "[i]n cryptography... an oracle is any system which can give some extra information on a system, which otherwise would not be available."[2]

Using a hash function, instead of the verifier, to generate challenges is known as a *Fiat-Shamir transform* [29], because it makes an interactive proof non-interactive and publicly verifiable [40].

Non-interactive proof

1. Generate random number $\alpha \in_R \mathbb{Z}_l$, and compute αG.
2. Calculate the challenge using a cryptographically secure hash function, $c = \mathcal{H}(\alpha G)$.
3. Define $r = \alpha + c * k$.
4. Publish the proof pair $(\alpha G, r)$.

Verification

1. Calculate the challenge: $c = \mathcal{H}(\alpha G)$.
2. Compute $R = rG$ and $R' = \alpha G + c * K$.
3. If $R = R'$ then the prover must know k (except with negligible probability).

Why it works

$$\begin{aligned} rG &= (\alpha + c * k)G \\ &= (\alpha G) + (c * kG) \\ &= \alpha G + c * K \\ R &= R' \end{aligned}$$

An important part of any proof/signature scheme is the resources required to verify them. This includes space to store proofs, and time spent verifying. In this scheme we store one EC point and one integer, and need to know the public key - another EC point.

Since hash functions are comparatively fast to compute, verification time is mostly a function of elliptic curve operations. For this proof scheme there are: src/ringct/rctOps.cpp

PA Point addition for some points A, B: $A + B$ [1]
PS Point subtraction for some points A, B : $A - B$ [0]
KBSM Known-base scalar multiplications for some integer a: aG [1]
VBSM Variable-base scalar multiplications for some integer a, and point P: aP [1]
SVBA Single-variable-base addition for some integer a, and point B: $aG + B$ [0]
DVBA Double-variable-base addition for some integers a, b, and point B: $aG + bB$ [0]
VBA Variable-base additions for some integers a, b, and points A, B: $aA + bB$ [0]

2.3.5 Signing messages

Typically, a cryptographic signature is performed on a cryptographic hash of a message rather than the message itself, which facilitates signing messages of varying size. However, in this report we will loosely use the term *message*, and its symbol $\mathfrak{m}$, to refer to the message properly speaking and/or its hash value, unless specified.

Signing messages is a staple of Internet security. One common signature scheme is called ECDSA. See [34], ANSI X9.62, and [32].

The signature scheme we present here is an alternative formulation of the transformed Schnorr proof from before. Thinking of signatures in this way prepares us for exploring ring signatures in the next chapter.

Signature

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

1. Generate random number $\alpha \in_R \mathbb{Z}_l$, and compute αG.
2. Calculate the challenge using a cryptographically secure hash function, $c = \mathcal{H}(\mathfrak{m}, [\alpha G])$.
3. Define r such that $\alpha = r + c * k_A$.
4. Publish the signature (c, r).

Verification

Any third party who knows the EC domain parameters D (specifying which elliptic curve was used), the signature (c, r) and the signing method, $\mathfrak{m}$ and the hash function, and K_A can verify the signature:

1. Calculate the challenge: $c' = \mathcal{H}(\mathfrak{m}, [rG + c * K_A])$.
2. If $c = c'$ then the signature passes.

In this signature scheme we store two scalars, need one public EC key, and verify with: src/ringct/rctOps.cpp

DVBA Double-variable-base addition for some integers a, b, and point B: $aG + bB$ [1]

Why it works

This stems from the fact that

$$\begin{aligned} rG &= (\alpha - c * k_A)G \\ &= \alpha G - c * K_A \\ \alpha G &= rG + c * K_A \\ \mathcal{H}_n(\mathfrak{m}, [\alpha G]) &= \mathcal{H}_n(\mathfrak{m}, [rG + c * K_A]) \\ c &= c' \end{aligned}$$

Therefore the owner of k_A (Alice) created (c, r) for $\mathfrak{m}$: she signed the message. The probability someone, a forger, without k_A could have made (c, r) is negligible.

2.4 Curve Ed25519

Monero uses a particular Twisted Edwards elliptic curve for cryptographic operations, *Ed25519*, the *birational equivalent*[12] of the Montogomery curve *Curve25519*.

Both Curve25519 and Ed25519 were released by Bernstein *et al.* [23, 24, 25].

The curve is defined over the prime field $\mathbb{F}_{2^{255}-19}$ (i.e. $q = 2^{255} - 19$) by means of the following equation:

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

This curve addresses many concerns raised by the cryptography community. It is well known that NIST[13] standard algorithms have issues. For example, it has recently become clear the random number generation algorithm PNRG is flawed and contains a potential backdoor [31]. Seen from a broader perspective, standardization authorities like NIST lead to a cryptographic monoculture, introducing a point of centralization. This was illustrated when the NSA used its influence over NIST to weaken an international cryptographic standard [14].

Curve Ed25519 is not subject to any patents (see [36] for a discussion on this subject), and the team behind it has developed and adapted basic cryptographic algorithms with efficiency in mind [25].

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:

$2^3 \cdot 7237005577332262213973186563042994240857116359379907606001950938285454250989$

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

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

2.4.1 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.

2.4.2 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 [53]. Point compression for the Ed25519 curve was first described in [24], while the concept was first introduced in [43].

This point compression scheme follows from a transformation of the Twisted Edwards curve equation (assuming $a = -1$, which is true for Monero): $x^2 = (y^2-1)/(dy^2+1)$,[14] 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 have a curve point and know its x is even, but given its y value the transformed curve equation outputs an odd number, then we know negating that number 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).

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

1. 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.
2. Let $u = y^2 - 1 \pmod q$ and $v = dy^2 + 1 \pmod q$. This means $x^2 = u/v \pmod q$.
3. Compute[15] $z = uv^3(uv^7)^{(q-5)/8} \pmod q$.
   (a) If $vz^2 = u \pmod q$ then $x' = z$.
   (b) If $vz^2 = -u \pmod q$ then calculate $x' = z * 2^{(q-1)/4} \pmod q$.
4. Using the parity bit b from the first step, if $b \neq$ the least significant bit of x' then $x = -x' \pmod q$, otherwise $x = x'$.
5. Return the decompressed point (x, y).

[14] Here $d = -\frac{121665}{121666}$.
[15] Since $q = 2^{255} - 19 \equiv 5 \pmod 8$, $(q-5)/8$ and $(q-1)/4$ are integers.

2.4.3 EdDSA signature algorithm

Bernstein and his team have developed a number of basic algorithms based on curve Ed25519.[16] 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 [24]. The algorithm can also be found described in Internet RFC8032 [35]. Note this is a very Schnorr-like signature scheme.

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* [24].

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 [35].

Signature

1. Let h_k be a hash $\mathcal{H}(k)$ of the signer's private key k. Compute α as a hash $\alpha = \mathcal{H}(h_k, \mathfrak{m})$ of the hashed private key and message. Depending on implementation, $\mathfrak{m}$ could be the actual message or its hash [35].
2. Calculate αG and the challenge $ch = \mathcal{H}([\alpha G], K, \mathfrak{m})$.
3. Calculate $r = \alpha + ch \cdot k$.
4. The signature is the pair $(\alpha G, r)$.

Verification

Verification is performed as follows

1. Compute $ch' = \mathcal{H}([\alpha G], K, \mathfrak{m})$.
2. If the equality $2^c rG = 2^c \alpha G + 2^c ch' * K$ holds then the signature is valid.

The 2^c term comes from Bernstein *et al.*'s general form of the EdDSA algorithm [25]. According to that paper, though it isn't required for adequate verification, removing 2^c provides stronger equations.

The public key K can be any EC point, but we only want to use points in the generator G's subgroup. Multiplying by the cofactor 2^c ensures all points are in that subgroup. Alternatively,

[16] See [25] for efficient group operations in Twisted Edwards EC (i.e. point addition, doubling, mixed addition, etc).

verifiers could check $lK \stackrel{?}{=} 0$, which only works if K is in the subgroup. We do not know of any weaknesses caught by these precautions, though as we will see the later method is important in Monero.

In this signature scheme we store one EC point and one scalar, have one public EC key, and verify with: src/ringct/rctOps.cpp

PA Point addition for some points A, B: $A + B$ [1]
KBSM Known-base scalar multiplications for some integer a: aG [1]
VBSM Variable-base scalar multiplications for some integer a, and point P: aP [1]

Why it works

$$\begin{aligned} 2^c rG &= 2^c(\alpha + \mathcal{H}([\alpha G], K, \mathfrak{m}) \cdot k) \cdot G \\ &= 2^c \alpha G + 2^c \mathcal{H}([\alpha G], K, \mathfrak{m}) \cdot K \end{aligned}$$

Binary representation

By default, an EdDSA signature would need $64 + 32$ bytes to be represented. However, RFC8032 assumes that point αG is compressed, which reduces space requirements to only $32 + 32$ bytes. Including the public key K implies $32 + 32 + 32$ bytes.

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 a signature would not be able to tell which ring member 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 a signer belongs to a group, without necessarily identifying him. In the context of Monero they will allow for unforgeable, signer-ambiguous transactions that leave currency flows largely untraceable.

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 must be a member of the ring (except with negligible probability), but not which member.[1] Monero uses this to obfuscate the origin of funds in each transaction.

[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 plus the real signer. 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.

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 is used to prevent double-spending attacks in Monero (except with negligible probability).

Unforgeability No attacker can forge a signature except with negligible probability.[3] This is used to prevent theft of Monero funds by those not in possession of the appropriate private keys.

3.1 Linkable Spontaneous Anonymous Group (LSAG) signatures

Originally (Chaum in [27]), 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 [37] building on the work of Rivest *et al.* in [52]. 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

Ring signature schemes are very Schnorr-like. In fact, our signature scheme in Section 2.3.5 can be considered a one-key ring signature. We get to two keys by, instead of defining r right away, generating a decoy r' and creating a new challenge to define r with.

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.

[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.

[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 [48] and [37].

[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". Linkable 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 such that $n_x G = \mathcal{H}_p(x)$.

1. Compute key image $\tilde{K} = k_\pi \mathcal{H}_p(\mathcal{R})$.
2. Generate random number $\alpha \in_R \mathbb{Z}_l$ and random numbers $r_i \in_R \mathbb{Z}_l$ for $i \in \{1, 2, ..., n\}$ 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 $n + 1 \to 1$,
$$c_{i+1} = \mathcal{H}_n(\mathcal{R}, \tilde{K}, \mathfrak{m}, [r_i G + c_i K_i], [r_i \mathcal{H}_p(R) + c_i \tilde{K}])$$
5. Define r_π such that $\alpha = r_\pi + c_\pi k_\pi \pmod{l}$.

The ring signature contains the signature $\sigma(\mathfrak{m}) = (c_1, r_1, r_2, ..., r_n)$, the key image $\tilde{K}$, 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. Check $l\tilde{K} \stackrel{?}{=} 0$.
2. For $i = 1, 2, ..., n$ iteratively compute, replacing $n + 1 \to 1$,
$$c'_{i+1} = \mathcal{H}_n(\mathcal{R}, \tilde{K}, \mathfrak{m}, [r_i G + c_i K_i], [r_i \mathcal{H}_p(\mathcal{R}) + c_i \tilde{K}])$$
3. If $c'_1 = c_1$ then the signature is valid. Note that c'_1 is the last term calculated.

We must check $l\tilde{K} \stackrel{?}{=} 0$ because it is possible to add an EC point from the subgroup of size h (the cofactor) and, if all c_i are multiples of h (which we could achieve with automated trial and error using different α and r_i values), make h unlinked valid signatures using the same ring and signing key.[9] This is because an EC point multiplied by its subgroup's order is zero.[10]

To be clear, given some point K in the subgroup of order l, some point K^h with order h, and an integer c divisible by h:
$$\begin{aligned} c * (K + K^h) &= cK + cK^h \\ &= cK + 0 \end{aligned}$$

In this scheme we store $(1+n)$ integers, have one EC key image, use n public keys, and verify with: src/ringct/rctOps.cpp

VBSM Variable-base scalar multiplications for some integer a, and point P: aP [1]
DVBA Double-variable-base addition for some integers a, b, and point B: $aG + bB$ [n]
VBA Variable-base additions for some integers a, b, and points A, B: $aA + bB$ [n]

[9] We are not concerned with points from other subgroups because the output of $\mathcal{H}_n$ is confined to $\mathbb{Z}_l$. For EC order $N = hl$, all divisors of N are either multiples of l or divisors of h.

[10] In Monero's early history this was not checked for. Fortunately, it was not exploited before a fix was implemented in April, 2017 (v5 of the protocol) [51].

Why it works

We can informally 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 signature loop:
$$c_3 = \mathcal{H}_n(..., [\alpha G], [\alpha \mathcal{H}_p(\mathcal{R})])$$
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 $\mathcal{R}$, $\tilde{K}$, and $\sigma(\mathfrak{m}) = (c_1, r_1, r_2, r_3)$:

1. We use r_1 and c_1 to compute
$$c_2' = \mathcal{H}_n(..., [r_1 G + c_1 K_1], [r_1 \mathcal{H}_p(\mathcal{R}) + c_1 \tilde{K}])$$
2. From when we made the signature, we see $c_2' = c_2$. With r_2 and c_2' we compute
$$c_3' = \mathcal{H}_n(..., [r_2 G + c_2' K_2], [r_2 \mathcal{H}_p(\mathcal{R}) + c_2' \tilde{K}])$$
3. We can easily see that $c_3' = c_3$ by substituting c_2 for c_2'. Using r_3 and c_3' we get
$$c_1' = \mathcal{H}_n(..., [r_3 G + c_3' K_3], [r_3 \mathcal{H}_p(\mathcal{R}) + c_3' \tilde{K}])$$

No surprises here: $c_1' = c_1$ if we substitute c_3 for c_3'.

Linkability

Given a fixed set of public keys $\mathcal{R}$ and two valid signatures for different messages,
$$\sigma(\mathfrak{m}) = (c_1, s_1, ..., s_n) \text{ with } \tilde{K}, \text{ and}$$
$$\sigma'(\mathfrak{m}') = (c_1', s_1', ..., s_n') \text{ with } \tilde{K}',$$
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).[11]

[11] LSAG is unforgeable, meaning no attacker could make a valid ring signature without knowing a private key.

3.2 Back's Linkable Spontaneous Anonymous Group (bLSAG) signatures

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 [47] based on a publication by A. Back [22] regarding the CryptoNote [59] ring signature algorithm (previously used in Monero, and now deprecated), which was inspired by Fujisaki and Suzuki's work in [30].

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 $i \in \{1, 2, ..., n\}$ 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 $n + 1 \to 1$,
$$c_{i+1} = \mathcal{H}_n(\mathfrak{m}, [r_i G + c_i K_i], [r_i \mathcal{H}_p(K_i) + c_i \tilde{K}])$$
5. Define $r_\pi = \alpha - c_\pi k_\pi \pmod{l}$.

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

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

Just like LSAG, in this scheme we store $(1+n)$ integers, have one EC key image, use n public keys, and verify with: src/ringct/rctOps.cpp

VBSM Variable-base scalar multiplications for some integer a, and point P: aP [1]
DVBA Double-variable-base addition for some integers a, b, and point B: $aG + bB$ [n]
VBA Variable-base additions for some integers a, b, and points A, B: $aA + bB$ [n]

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 the key image $\tilde{K}$ only depends on the true signer's keys. In other words, two signatures will now be linkable if and only if the same private key was used to

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 [37].

create them. In bLSAG, rings are just involved in obfuscating each signer's identity. bLSAG also simplifies the scheme 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 (MLSAG) signatures

In order to sign multi-input transactions, one has to sign with m private keys. In [47, 48], 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

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

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

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

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

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 $r_{\pi,j} = \alpha_j - c_\pi k_{\pi,j} \pmod{l}$.

The signature will be $\sigma(\mathfrak{m}) = (c_1, r_{1,1}, ..., r_{1,m}, ..., r_{n,1}, ..., r_{n,m})$, with key images $(\tilde{K}_1, ..., \tilde{K}_m)$.

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

Verification

The verification of a signature is done in the following manner

1. For all $j \in \{1, ..., m\}$ check $l\tilde{K}_j \stackrel{?}{=} 0$.
2. For $i = 1, ..., n$ compute, replacing $n + 1 \to 1$,

$$c'_{i+1} = \mathcal{H}_n(\mathfrak{m}, [r_{i,1}G + c_i K_{i,1}], [r_{i,1}\mathcal{H}_p(K_{i,1}) + c_i \tilde{K}_1], ..., [r_{i,m}G + c_i K_{i,m}], [r_{i,m}\mathcal{H}_p(K_{i,m}) + c_i \tilde{K}_m])$$

3. 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 $r_{\pi,j} = \alpha_j - c_\pi k_{\pi,j}$ closes the loop,

$$\begin{aligned} r_{\pi,j}G &+ c_\pi K_{\pi,j} = (\alpha_j - c_\pi k_{\pi,j})G &+ c_\pi K_{\pi,j} = \alpha_j G \\ r_{\pi,j}\mathcal{H}_p(K_{\pi,j}) + c_\pi \tilde{K}_j &= (\alpha_j - c_\pi k_{\pi,j})\mathcal{H}_p(K_{\pi,j}) + c_\pi \tilde{K}_j &= \alpha_j \mathcal{H}_p(K_{\pi,j}) \end{aligned}$$

  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]

Space and verification requirements

In this scheme we store $(1+m*n)$ integers, have m EC key images, use $m*n$ public keys, and verify with: src/ringct/rctOps.cpp

VBSM Variable-base scalar multiplications for some integer a, and point P: aP [m]
DVBA Double-variable-base addition for some integers a, b, and point B: $aG + bB$ $[m*n]$
VBA Variable-base additions for some integers a, b, and points A, B: $aA + bB$ $[m*n]$

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

3.4 Borromean ring signatures

We will see in later sections of this report it is necessary to prove transaction amounts are within an expected range. This can be accomplished with ring signatures. However, linkable signatures here would allow observers to link outputs using common amounts, or guess and check the entire transaction history. Fortunately, excluding linkability allows us to select more efficient algorithms in terms of space consumed and verification speed.

In this context, and for the *singular purpose* of proving amount ranges, Monero uses a signature scheme developed by G. Maxwell, which he described in [40].[13] We present here a complete version of the scheme for educational purposes. In Monero, range proofs require loops 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 ith 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 $i = 1, ..., n$ [skip this entire step for any i where $\pi_i = m_i$]:
   (a) generate a random value $\alpha_i \in_R \mathbb{Z}_l$,
   (b) seed the shelf's loop: set $c_{i,\pi_i+1} = \mathcal{H}_n(\mathfrak{m}, [\alpha_i G])$,
   (c) build first half of loop from seed: if $\pi_i + 1 = m_i$ skip this step, otherwise for $j_i = \pi_i + 1, ..., m_i - 1$ generate random numbers $r_{i,j_i} \in_R \mathbb{Z}_l$ and compute
   $$c_{i,j_i+1} = \mathcal{H}_n(\mathfrak{m}, [r_{i,j_i} G + c_{i,j_i} K_{i,j_i}])$$
2. Use last public key on each shelf to connect all loops together: for $i = 1, ..., n$ generate random numbers $r_{i,m_i} \in_R \mathbb{Z}_l$ and compute
   $$c_1 = \mathcal{H}_n(\mathfrak{m}, [r_{1,m_1} G + c_{1,m_1} K_{1,m_1}], ..., [r_{n,m_n} G + c_{n,m_n} K_{n,m_n}])$$
   Note: if any $\pi_i = m_i$, generate α_i instead of r_{i,m_i} and put $\alpha_i G$ in c_1.
3. For each shelf $i = 1, .., n$:

[13] Monero's first iteration of 'range proofs' used Aggregate Schnorr Non-Linkable Ring Signatures (ASNL) [48]. According to the author of ASNL it reduces to a kind of Borromean ring signature [20], but since the latter is more general and secure it was chosen for final implementation in November 2016.

(a) build second half of loop from connector: if $\pi_i = 1$ skip this, otherwise 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]
$$c_{i,j_i+1} = \mathcal{H}_n(\mathfrak{m}, [r_{i,j_i}G + c_{i,j_i}K_{i,j_i}])$$

(b) tie loop 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
$$\sigma = (c_1, r_{1,1}, ..., r_{1,m_1}, r_{2,1}..., r_{2,m_2}, ..., r_{n,m_n})$$

Note this signature scheme reduces to our Schnorr-like signature from Section 2.3.4 when $\mathcal{R}$ only contains one key.

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 loop:
$$L'_{i,j_i+1} = r_{i,j_i}G + c'_{i,j_i}K_{i,j_i}$$
$$c'_{i,j_i+1} = \mathcal{H}_n(\mathfrak{m}, L'_{i,j_i+1})$$
Interpret any $c'_{i,1}$ as c_1. Also, it is not necessary to compute c'_{i,m_i+1}.
2. Compute the connector $c'_1 = \mathcal{H}_n(\mathfrak{m}, L'_{1,m_1}, ..., L'_{n,m_n})$.

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

In this scheme we store $(1+\sum_{i=1}^n m_i)$ integers, use $\sum_{i=1}^n m_i$ public keys, and verify with: src/ringct/rctOps.cpp

DVBA Double-variable-base addition for some integers a, b, and point B: $aG + bB$ $[\sum_{i=1}^n m_i]$

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:
$$\begin{aligned} L'_{i,\pi_i+1} &= r_{i,\pi_i}G + c'_{i,\pi_i}K_{i,\pi_i} \\ &= (\alpha_i - c_{i,\pi_i}k_{i,\pi_i})G + c'_{i,\pi_i}K_{i,\pi_i} \\ &= \alpha_i G - c_{i,\pi_i}k_{i,\pi_i}G + c'_{i,\pi_i}k_{i,\pi_i}G \\ &= \alpha_i G \end{aligned}$$
In other words, $c'_{i,\pi_i+1} = \mathcal{H}_n(\mathfrak{m}, [\alpha_i G]) = c_{i,\pi_i+1}$.

Therefore we can conclude the verification step identifies valid signatures (except in the case of a forgery, which can only occur with negligible probability).

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 Bob 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 *blah* and the value she wants to commit to is *heads*. She could simply hash $h = \mathcal{H}(blah, heads)$ and give h to Bob. Bob flips a coin, Alice tells Bob about *blah* and informs him she committed to *heads*, and then Bob calculates $h' = \mathcal{H}(blah, heads)$. If $h' = h$, then he knows Alice called *heads* before the coin flip.

Alice uses the so-called 'salt' *blah* so Bob can't just guess $\mathcal{H}(heads)$ and $\mathcal{H}(tails)$ before his coin flip, and figure out she committed to *heads*.

4.1 Pedersen commitments

A *Pedersen commitment* [50] is a commitment that has the property of being *additively homomorphic*. If $C(a)$ and $C(b)$ denote the commitments for values a and b respectively, then $C(a + b) = C(a) + C(b)$.[1] This property is useful when committing transaction amounts, as one

[1] Additively homomorphic in this context means addition is preserved when you transform scalars into EC points by applying, for scalar x, $x \to xG$.

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 zero (because $0G = 0$). We could also create cheat tables of commitments to help us recognize common amounts a.

To attain information-theoretic 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. In the case of Monero, $H = \mathcal{H}_p(G)$.[2]

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 just 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.[3] If x is truly random, an attacker would have literally no way to figure out a [39] [55].

4.2 Amount 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 amounts in all outputs owned is considered a person's balance.

To send cryptocurrency to someone else, we create a transaction. A transaction references old outputs as *inputs* and addresses new outputs to recipients. Since it is rare for input amounts to equal intended output amounts, 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 (v4 of the protocol). While transaction verifiers don't know how much Moneroj is contained in each input and output, they still need to prove the sum of input amounts equals the sum of output amounts.

[2] The Monero codebase has a function *to_point*() that maps scalars to EC points. For commitments, $H = to_point(Keccak(G))$. src/ringct/rctTypes.h

[3] 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.

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:

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

Since commitments are additive and we don't know H, we could easily prove our inputs equal outputs to observers by making the sum of commitments to input and output amounts equal zero:[4]

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

To avoid sender identifiability, Shen Noether proposed [47] 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.

Note that $C = zG$ is called a *commitment to zero* because we can make a signature with z, which proves that there is no H component to C (assuming γ is unknown). In other words $C = zG + 0H$.

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 still 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 direct the 21 output to a friend and keep the -10 output (then ignore it), 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. This method treats each bit in

[4] Recall from Section 2.3.1 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 $(-y \pmod q)$.

the binary representation of the amount as a separate amount which we can commit to zero.

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 $\{C_i, C_i - 2^i H\}$.

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 $\{C_i, C_i - 2^i H\}$. The point that equals $y_i G$ 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.

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.

src/ringct /rctSigs.cpp genBorromean()

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 amount's 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 loop contains a commitment to zero, so total amounts must fall within range and money is not being artificially created.

In this scheme we store $(1+2*k)$ integers, use k public keys (commitments), and verify with:

DVBA Double-variable-base addition for some integers a, b, and point B: $aG + bB$ $[2*k]$

Verifiers need to calculate $C_i - 2^i H$, so verification times include:

PS Point subtraction for some points $A, B : A - B$ $[k]$
KBSM Known-base scalar multiplications for some integer a: aG $[k]$

For a total verification requirement:

src/ringct/rctOps.cpp

PS Point subtraction for some points $A, B : A - B$ $[k]$
KBSM Known-base scalar multiplications for some integer a: aG $[k]$
DVBA Double-variable-base addition for some integers a, b, and point B: $aG + bB$ $[2*k]$

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 2.3.2.

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 (and retroactively figure out it has been spent).[1]

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

[1] It is currently most common for the view key k^v to equal $\mathcal{H}_n(k^s)$. 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 $k^v = sc_reduce32(Keccak(Keccak(a)))$. [4]

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^v_B, k^s_B) and (K^v_B, K^s_B), and Alice knows his public keys. A transaction could proceed as follows (see [59]):

1. Alice generates a random number r such that $1 < r < l$, and calculates the one-time public key $K^o = \mathcal{H}_n(rK^v_B)G + K^s_B$.

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.

3. Bob receives the data and sees the values rG and K^o. He can calculate $k^v_B rG = rK^v_B$. He can then calculate $K'^s_B = K^o - \mathcal{H}_n(rK^v_B)G$. When he sees that $K'^s_B = K^s_B$, he knows the transaction is addressed to him.

   The private key k^v_B is called the *view key* because anyone who has it (and Bob's public spend key K^s_B) can calculate K'^o for every transaction output in the blockchain (record of transactions), and 'view' which ones are addressed to Bob.

4. The one-time keys for the output are:
$$K^o = \mathcal{H}_n(rK^v_B)G + k^s_B G = (\mathcal{H}_n(rK^v_B) + k^s_B)G$$
$$k^o = \mathcal{H}_n(rK^v_B) + k^s_B$$

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^s_B, or solving the discrete logarithm problem for $K^s_B = k^s_B G$, 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.[2]

A third party with Bob's *view key* can verify an output is addressed to Bob, yet 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' key image.

[2] Imagine Alice produces two transactions, each containing the same one-time output address K^o that Bob can spend. Since K^o only depends on r and K^v_B, there is no reason she can't do it. Bob can only spend one of those outputs because each stealth address only has one key image, so if he isn't careful Alice might trick him. She could make transaction 1 with a lot of money for Bob, and later transaction 2 with a small amount for Bob. If he spends the money in 2, he can never spend the money in 1. In fact, no one could spend the money in 1, effectively 'burning' it. Monero wallets have been designed to ignore the smaller amount in this scenario.

Such a third party could be a trusted custodian, an auditor, a tax authority, etc. Somebody who could be allowed 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 5.6.1.

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 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:

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

5.3 Subaddresses

Monero users can generate subaddresses from each address [46]. Funds sent to a subaddress can be viewed and spent using its main 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.

Subaddresses 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. As we will see, an observer would have to solve the DLP in order to determine a given subaddress is derived from any particular address [46].

They are also useful for differentiating between received outputs. For example, if Alice wants to buy an apple from Bob on a Tuesday, Bob could write a receipt describing the purchase and make a subaddress for that receipt, then ask Alice to use that subaddress when she sends him the money. This way Bob can associate the money he receives with the apple he sold. We explore another way to distinguish between received outputs in the next section.

Bob generates his i^{th} subaddress ($i = 1, 2, ...$) 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,

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

5.3.1 Sending to a subaddress

Let's say Alice is going to send Bob money via his subaddress $(K_B^{v,1}, K_B^{s,1})$ with a simple one-input, one-output transaction.

1. Alice generates a random number r such that $1 < r < l$, and calculates the one-time public key $K^o = \mathcal{H}_n(rK_B^{v,1})G + K_B^{s,1}$.

2. Alice sets K^o as the addressee of the payment, **adds the value $rK_B^{s,1}$ to the transaction data** and submits it to the network.

3. Bob receives the data and sees the values $rK_B^{s,1}$ and K^o. **He can calculate** $k_B^v rK_B^{s,1} = rK_B^{v,1}$. He can then calculate $K_B'^{s,1} = K^o - \mathcal{H}_n(rK_B^{v,1})G$. When he sees that $K_B'^{s,1} = K_B^{s,1}$, he knows the transaction is addressed to him.

   Bob only needs his private view key k_B^v and subaddress public spend key $K_B^{s,1}$ to find transaction outputs sent to his subaddress.

4. The one-time keys for the output are:
$$\begin{aligned} K^o &= \mathcal{H}_n(rK_B^{v,1})G + K_B^{s,1} = (\mathcal{H}_n(rK_B^{v,1}) + k_B^{s,1})G \\ k^o &= \mathcal{H}_n(rK_B^{v,1}) + k_B^{s,1} \end{aligned}$$

Now, Alice' transaction public key is particular to Bob ($rK_B^{s,1}$ instead of rG). If she creates a p-output transaction with at least one output intended for a subaddress, Alice needs to make a unique transaction public key for each output $t \in 1, ..., p$. In other words, if Alice is sending to Bob's subaddress $(K_B^{v,1}, K_B^{s,1})$ and Carol's address (K_C^v, K_C^s), she will put two transaction public keys $\{r_1 K_B^{s,1}, r_2 G\}$ in the transaction data.[3]

5.4 Integrated addresses

In order to differentiate between the outputs they receive, a recipient can request senders include a *payment ID* in transaction data.[4] For example, if Alice wants to buy an apple from Bob on a Tuesday, Bob could write a receipt describing the purchase and ask Alice to include the receipt's ID number when she sends him the money. This way Bob can associate the money he receives with the apple he sold.

[3] In Monero subaddresses are prefixed with an '8', separating them from addresses, which are prefixed with '4'. This helps senders choose the correct procedure when constructing transactions.

[4] Currently, Monero only supports one payment ID per transaction.

Senders can communicate payment IDs in clear text, but manually including the IDs in transactions is inconvenient, and a privacy hazard for recipients, who might inadvertently expose their activities. In Monero, recipients can integrate payment IDs into their addresses, and provide those *integrated addresses*, containing (K^v, K^s, payment ID), to senders. Payment IDs can technically be integrated into any kind of address, including normal addresses, subaddresses, and multisignature addresses.[5]

Senders addressing outputs to integrated addresses can encode payment IDs using the shared secret rK_t^v, the output index t, and a XOR operation, which recipients can then decode with the appropriate transaction public key and another XOR procedure [7]. Encoding payment IDs in this way allows senders to prove they made particular transactions (i.e. for audits, refunds, etc.).

Binary operator XOR

The binary operator XOR evaluates two arguments and returns true if one, but not both, of the arguments is true [17]. Here is its truth table:

A	B	A XOR B
T	T	F
T	F	T
F	T	T
F	F	F

In the context of computer science, XOR is equivalent to bit addition modulo 2. For example, the XOR of two bit pairs:

$$\begin{aligned}\mathrm{XOR}(\{1,1\},\{1,0\}) &= \{1+1, 1+0\} \pmod 2 \\ &= \{0,1\}\end{aligned}$$

Examining the previous example, each of these produce the same output: $\mathrm{XOR}(\{1,1\},\{1,0\})$, $\mathrm{XOR}(\{0,0\},\{0,1\})$, $\mathrm{XOR}(\{1,0\},\{1,1\})$, or $\mathrm{XOR}(\{0,1\},\{0,0\})$. There are 2^{b} combinations of XOR inputs (with b bits each) for the same output, so if input $A \in_R \{1,...,2^{\mathrm{b}}\}$, an observer who learned $C = \mathrm{XOR}(A,B)$ could not gain any information about B.

At the same time, anyone who knows two of the elements A, B, C, where $C = \mathrm{XOR}(A,B)$, can calculate the third element, such as $A = \mathrm{XOR}(B,C)$. XOR indicates if two elements are different or the same, so knowing C and B is enough to reveal A. A careful examination of the truth table reveals this to be true.

[5] Since an observer can recognize the difference between transactions with and without payment IDs of any kind, using them is thought to make the Monero transaction history less uniform. Subaddresses ('disposable addresses' more generally) can be used for the same purpose, so payment IDs may be superfluous. Based on these ideas, there is some effort in the Monero development community to deprecate all forms of payment ID [5]. However, cryptocurrency exchanges find them useful and are slow to change, so payment IDs might remain a feature indefinitely. Also note integrated addresses have only ever been implemented for normal addresses.

Encoding

The sender encodes each payment ID for inclusion in transaction data[6]

$$\begin{aligned} k_{\text{stealth}} &= \mathcal{H}_n(rK_t^v, t) \\ k_{\text{payment ID}} &= k_{\text{stealth}} \rightarrow \text{reduced to bit length of payment ID} \\ \text{encoded payment ID} &= \text{XOR}(k_{\text{payment ID}}, \text{payment ID}) \end{aligned}$$

The output of a hash function $\mathcal{H}$ is uniformly distributed across the range of possible outputs. In other words, for some input A, $\mathcal{H}(A) \in_R^D \mathbb{S}_H$ where $\mathbb{S}_H$ is the set of possible outputs from $\mathcal{H}$. We use $\in_R^D$ to indicate the function is deterministically random. $\mathcal{H}(A)$ produces the same thing every time, but its output is equivalent to a random number.

Decoding

A receiver t can find the tth payment ID using his view key and the transaction public key rG:

$$\begin{aligned} k_{\text{stealth}} &= \mathcal{H}_n(k_t^v rG, t) \\ k_{\text{payment ID}} &= k_{\text{stealth}} \rightarrow \text{reduced to bit length of payment ID} \\ \text{payment ID} &= \text{XOR}(k_{\text{payment ID}}, \text{encoded payment ID}) \end{aligned}$$

Similarly, senders can decode payment IDs they had previously encoded by recalculating the shared secret $k_{\text{stealth}} = \mathcal{H}_n(rK_t^v, t)$.

5.5 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 deprecated 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 5.6) closely follows the ideas explained by S. Noether *at al.* in [48]. At the time that paper was written, the authors most likely intended to fully replace the original CryptoNote transaction scheme.

[6] In Monero payment IDs for integrated addresses are conventionally 64 bits long, while independent, clear text payment IDs are usually 256 bits long.

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 5.7) is used in those situations. The main difference, as we will see later, is that each input is signed independently.

We present a conceptual summary of transactions in Section 5.8.

5.6 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 [48] 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.

5.6.1 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 reconstruct the amount commitments. This means the blinding factor y and amount b must be communicated to the receiver.

The solution adopted in Monero is a Diffie-Hellman shared secret rK^v_B. For any given transaction in the blockchain, each of its outputs $t \in \{1, ..., p\}$ has two associated values called *mask* and *amount* satisfying[7]

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

src/ringct/rctOps.cpp ecdhEncode()

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^v_B. He can also check that the commitment $C(y_t, b_t)$ provided in the transaction data, henceforth denoted C^b_t, corresponds to the amount at hand.

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

5.6.2 Commitments to zero

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

This sender knows the private keys $k^o_{\pi,1}, ..., k^o_{\pi,m}$ corresponding to the one-time addresses (Section 5.2). The sender also knows the blinding factors x_j used in commitments $C^a_{\pi,j}$ (Section 5.6.1).

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, $C^a_{\pi,1}, ..., C^a_{\pi,m}$, and creates commitments for $b_1, ..., b_p$. Let these new commitments be $C^b_1, ..., C^b_p$.

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

$$\sum_j C^a_{\pi,j} - \sum_t C^b_{\pi,t} = zG$$

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

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

[7] 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 do not have similar *masks* and *amounts*, except with negligible probability. 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.

src/cryptonote_core/cryptonote_tx_utils.cpp construct_tx_with_tx_key() calls derivation_to_scalar

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

5.6.3 Signature

The sender selects v sets of size m, of additional unrelated addresses and their commitments from the blockchain, corresponding to apparently unspent outputs.[8] 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 πth such term is zG. This allows us to create an MLSAG signature (Section 3.3) on $\mathcal{R}$.

[8] In Monero it is standard for the set of 'additional unrelated addresses' to be selected from this distribution: 50% from 1.8 days ago up to the default transaction spendable age, which is usually 10 blocks, 50% from the remaining blockchain. Each segment has a triangle probability aimed at the 1.8 day mark, where an output 1.8 days old is twice as likely to be chosen as an output 0.9 days old. To spend an output of type X, we find all other outputs of type X (i.e. RingCT outputs) and choose its ring members from that set based on the distribution. The triangle probabilities are achieved by rolling a random number for each decoy ring member, normalizing it, taking the square root, multiplying it by the number of eligible type X outputs, and using the output at that index in the group (flip the index if the triangle faces backward by computing [#eligible - index]).

CRYPTONOTE_DEFAULT_TX_SPENDABLE_AGE

src/wallet/wallet2.cpp get_outs()

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 building 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.[9] 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 $t \in \{1, ..., p\}$. No message $\mathfrak{m}$ is signed by the Borromean signatures.

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.

5.6.4 Transaction fees

Typically transaction outputs are *lower* in total than transaction inputs, in order to provide a fee for miners.[10] Transaction fee amounts are stored in clear text in the transaction data transmitted to the network. Miners can create an additional output for themselves with the fee. This fee amount must also be converted into a commitment so verifiers can confirm transactions sum to zero.

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:

$$(\sum_j C_{i,j} - \sum_t C^b_t) - fH$$

[9] 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 era version (i.e. ringCT = 2), inputs {key offsets, key image}, outputs {one-time addresses}, extra {transaction public key, payment ID or encoded payment ID, misc.}}
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.

src/ringct/rctSigs.cpp get_pre_mlsag_hash()

[10] In Monero there is a minimum base fee per kB of transaction data. It is semi-mandatory because while you can create new blocks with tiny-fee transactions, most Monero nodes won't relay such transactions to other nodes. We go into more detail on this in Section 7.3.3.

Which works because this is a commitment to zero:

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

5.6.5 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.

The network need only verify that key images included in MLSAG signatures (corresponding to inputs and calculated as $\tilde{K}_j^o = k_{\pi,j}^o \mathcal{H}_p(K_{\pi,j}^o)$) have not appeared before in other transactions.[11] If they have, then we can be sure we are witnessing an attempt to re-spend an output $C_{\pi,j}^a$ addressed to $K_{\pi,j}^o$.

If someone tries to spend $C_{\pi,j}^a$ 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.

5.6.6 Space and verification requirements

MLSAG signature (inputs)

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

$$\sigma(\mathfrak{m}) = (c_1, r_{1,1}, ..., r_{1,m+1}, ..., r_{v+1,1}, ..., r_{v+1,m+1}) \text{ with } (\tilde{K}_1^o, ..., \tilde{K}_m^o)$$

As a result of the heritage from CryptoNote, the values $\tilde{K}_j^o$ are not referred to as part of the signature, but rather as *images* of the private keys $k_{\pi,j}^o$. 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.

[11] Verifiers must also check the key image is a member of the generator's subgroup (recall Section 3.1).

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.

Including the computation of $(\sum_j C_{i,j} - \sum_t C_t^b) - fH$, and verifying key images are in G's subgroup with $l\tilde{K}$, we verify a `RCTTypeFull` transaction's MLSAG (Section 3.3) with:

src/ringct/rctOps.cpp

PA Point addition for some points A, B: $A + B$ $[(m-1)*(v+1)+p]$
PS Point subtraction for some points A, B : $A - B$ $[v+1]$
VBSM Variable-base scalar multiplications for some integer a, and point P: aP $[m+1]$
DVBA Double-variable-base addition for some integers a, b, and point B: $aG+bB$ $[(m+1)*(v+1)]$
VBA Variable-base additions for some integers a, b, and points A, B: $aA + bB$ $[m*(v+1)]$

Range proofs (outputs)

From Section 3.4, Section 4.3, and Section 5.6.3 we know that a Monero Borromean signature for range proofs takes the form of an n-tuple

src/ringct/ringCT.cpp genBorromean()

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

Ring keys are considered part of ring signatures. However, in this case it is only necessary to store the commitments C_i, as the ring key counterparts $C_i - 2^i H$ 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.

Including $C_i - 2^i H$ and checking $\sum_i C_i = C$, range proofs for p outputs will require this to verify (Section 4.4):

PS Point subtraction for some points A, B : $A - B$ $[p*64]$
KBSM Known-base scalar multiplications for some integer a: aG $[p*64]$
DVBA Double-variable-base addition for some integers a, b, and point B: $aG + bB$ $[p*2*64+63]$

5.7 Ring Confidential Transactions of type `RCTTypeSimple`

In the current Monero code base, transactions with 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 at once, the sender signs each input separately.

Among other things, this means we can't use commitments to zero in the same way as for `RCTTypeFull` transactions. It's really unlikely that we could match an input amount to an output amount, and in most cases each input will be less than the sum of outputs. So, we cannot directly commit $inputs - outputs = 0$.

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

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

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 computationally difficult.

5.7.1 Amount commitments

As explained, if inputs are decoupled from each other then the sender can't sign an aggregate commitment to zero. 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 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 5.6.1), 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

$$(\sum_j C'^a_j - \sum_t C^b_t) - fH = 0$$

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

$$x'_m = \sum_t y_t - \sum_{j=1}^{m-1} x'_j$$

5.7.2 Signature

As we mentioned, 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 $K^o_{\pi,j}$ that had commitment $C^a_{\pi,j}$. Let $C'^a_{\pi,j}$ be a new commitment for the same amount but with a different blinding factor.

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

$$K^o_{1,j}, ..., K^o_{\pi-1,j}, K^o_{\pi+1,j}, ..., K^o_{v+1,j}$$
$$C_{1,j}, ..., C_{\pi-1,j}, C_{\pi+1,j}, ..., C_{v+1,j}$$

She can then sign using the following ring:

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

Alice will know the private keys $k^o_{\pi,j}$ for $K^o_{\pi,j}$, and z_j for the commitment to zero ($C^a_{\pi,j}$ - $C'^a_{\pi,j}$). She can sign the jth input with an MLSAG signature on $\mathcal{R}_j$. Recalling Section 5.6.3, there is no key

image for the commitments to zero $z_j G$, and consequently no corresponding key image component in each input's signature's construction.

Each input in `RCTTypeSimple` transactions is signed individually, applying the scheme described in Section 5.6.3, 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 aggregated case. This means even if one input's origin became identifiable, the other inputs' origins would not.

The message $\mathfrak{m}$ signed by each input is essentially the same as for `RCTTypeFull` transactions (see Footnote 9), except it includes pseudo output commitments for the inputs. Only one message is produced, and each input MLSAG signs it.

5.7.3 Space and verification requirements

MLSAG signature (inputs)

Each ring $\mathcal{R}_j$ contains $(v+1) \cdot 2$ keys. Using the point compression technique from Section 2.4.2, 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.

Including the computation of $(C_{i,j} - C'^a_{\pi,j})$ and $(\sum_j C'^a_j - \sum_t C^b_t) - fH$, and verifying key images are in G's subgroup with $l\tilde{K}$, we verify all of a `RCTTypeSimple` transaction's MLSAGs with: src/ringct/rctOps.cpp

PA Point addition for some points A, B: $A + B$ $[(m-1)+p]$
PS Point subtraction for some points A, B : $A - B$ $[m * (v+1)]$
VBSM Variable-base scalar multiplications for some integer a, and point P: aP $[m+1]$
DVBA Double-variable-base addition for some integers a, b, and point B: $aG + bB$ $[2 * m * (v+1)]$
VBA Variable-base additions for some integers a, b, and points A, B: $aA + bB$ $[m * (v+1)]$

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.

As before, each range proof will require this to verify:

PS Point subtraction for some points A, B : $A - B$ $[p * 64]$
KBSM Known-base scalar multiplications for some integer a: aG $[p * 64]$
DVBA Double-variable-base addition for some integers a, b, and point B: $aG + bB$ $[p * 2 * 64 + 63]$

5.8 Concept summary: Monero transactions

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

- Type: '0' is `Miner Transaction`, '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 $r_{i,j}$ & $r^z_{i,j}$ for $i \in \{1, ..., v+1\}$ and input j
  - **Key image**: the key image $\tilde{K}^{o,a}_j$ for input j
  - **Pseudo output commitment** [`RCTTypeSimple` only]: C'^a_j for input j
- Outputs: for each output $t \in \{1, ..., p\}$ to address or subaddress (K^v_t, K^s_t)
  - **One-time (stealth) address**: $K^{o,b}_t$
  - **Output commitment**: C^b_t for output t
  - **Diffie-Hellman terms**: so receivers can compute C^b_t and b_t for output t
    - *Mask*: $y_t + \mathcal{H}_n(rK^v_t, t)$
    - *Amount*: $b_t + \mathcal{H}_n(\mathcal{H}_n(rK^v_t, t))$
  - **Range proof** for output amount b_t using a Borromean ring signature
    - *Signatures*: 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, $r_t K^{s,i}_t$ for each subaddress'd output t and $r_t G$ for each normal address'd output t, and a payment ID or encoded payment ID (max one per transaction)

5.8.1 Storage and verification requirements

For `RCTTypeFull` we need $((v+1)\cdot(m+1)+m+1)\cdot 32$ bytes of storage, and verify with: src/ringct/rctOps.cpp

PA Point addition for some points A, B: $A+B$ $[(m-1)*(v+1)+p]$
PS Point subtraction for some points A, B : $A-B$ $[v+1]$
VBSM Variable-base scalar multiplications for some integer a, and point P: aP $[m+1]$
DVBA Double-variable-base addition for some integers a, b, and point B: $aG+bB$ $[(m+1)*(v+1)]$
VBA Variable-base additions for some integers a, b, and points A, B: $aA+bB$ $[m*(v+1)]$

For `RCTTypeSimple` we need $(2(v+1)+3)\cdot m\cdot 32$ bytes of storage, and verify with:

PA Point addition for some points A, B: $A+B$ $[(m-1)+p]$
PS Point subtraction for some points A, B : $A-B$ $[m*(v+1)]$
VBSM Variable-base scalar multiplications for some integer a, and point P: aP $[m+1]$
DVBA Double-variable-base addition for some integers a, b, and point B: $aG+bB$ $[2*m*(v+1)]$
VBA Variable-base additions for some integers a, b, and points A, B: $aA+bB$ $[m*(v+1)]$

Output range proofs need $p\cdot 6176$ bytes of storage, and verify with:

PS Point subtraction for some points A, B : $A-B$ $[p*64]$
KBSM Known-base scalar multiplications for some integer a: aG $[p*64]$
DVBA Double-variable-base addition for some integers a, b, and point B: $aG+bB$ $[p*2*64+63]$

Miscellaneous storage requirements:

- One-time output addresses: $p*32$ bytes
- Output commitments: $p*32$ bytes
- Diffie-Hellman terms for outputs: $2*p*32$ bytes
- Transaction public key: 32 bytes normally, $p*32$ bytes if sending to at least one subaddress.
- Payment ID: 8 bytes for an integrated address, or 32 bytes for a plain payment ID. Max one per tx.
- Transaction fee: stored as variable length integer, so ≤ 64 bits
- Input offsets: stored as variable length integer, so ≤ 64 bits per input

CHAPTER 6

Multisignatures in Monero

Cryptocurrency transactions are not recoverable. If someone steals your private keys or scams you, the money lost could be gone forever. Dividing signing power between people can weaken the potential danger of a miscreant.

For example, say you deposit money into a joint account with a security company that monitors for suspicious activity related to your account. Transactions can only be signed if both you and the company cooperate. If someone steals your keys, you can notify the company there is a problem, and the company will stop signing transactions for your account. This is usually called an 'escrow' service.[1]

Cryptocurrencies use a 'multisignature' technique to achieve so-called 'M-of-N multisig'. In M-of-N, N people cooperate to make a joint key, and only M people (for $M \leq N$) are needed to sign with that key. We begin this chapter by introducing the basics of N-of-N multisig, progress into N-of-N Monero multisig, generalize for M-of-N multisig, and then explain how to nest multisig keys inside other multisig keys.

This chapter is based on a pre-release draft of a Monero Research Lab paper written by pseudonymous Surae Noether. Our only contributions are formally describing M-of-N multisig, a novel approach to nesting multisig keys, and a number of conventions for multisig implementation.

[1] Multisignatures have a diversity of applications, from corporate accounts to newspaper subscriptions.

6.1 Communicating with co-signers

Building joint keys and joint transactions requires communicating secret information between people who could be located all around the globe. To keep that information secure from unwanted observers, co-signers need to encrypt the messages they send each other.

A very simple way to encrypt messages using elliptic curve cryptography is via Diffie-Hellman exchange (ECDH). We already mentioned this in Section 5.6.1, where Monero output commitment mask and amount are communicated to receivers via the shared secret rK^v. It looked like this:

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

We could easily extend this to any message. First encode the message as a series of bits, then break it into chunks equal in size to the output of $\mathcal{H}_n$. Generate a random number $r \in \mathbb{Z}_l$ and perform a Diffie-Hellman exchange on all the message chunks using the recipient's public key K. Send those encrypted chunks to the intended recipient along with the public key rG, who can then decrypt the message with the shared secret krG. Message senders should also sign encrypted messages (usually the encrypted message's hash) so they can't be easily tampered with.

Since encryption is not essential to the operation of a cryptocurrency like Monero, we do not feel it necessary to go into more detail. Curious readers can look at this excellent conceptual overview [3], or see a technical description of the popular AES encryption scheme here [18].

6.2 Key aggregation for addresses

6.2.1 Naive approach

Let's say there are N people who want to create a shared multisignature address, which we denote $(K^{v,sh}, K^{s,sh})$. Funds can be sent to that address just like any normal address, but, as we will see later, to spend those funds all N people have to work together to sign transactions.

Since all N participants should be able to view funds received by the shared address, we can let everyone know the shared view key $k^{v,sh}$ (recall Sections 5.1, 5.2). To give all participants equal power, the view key can be a sum of view key components that all participants send each other securely. For participant $e \in \{1, ..., N\}$, his view key component is $k_e^{v,sh,c} \in_R \mathbb{Z}_l$, and all participants can compute the shared private view key

$$k^{v,sh} = \sum_{e=1}^{N} k_e^{v,sh,c}$$

In a similar fashion, the shared spend key $K^{s,sh} = k^{s,sh}G$ could be a sum of private spend key components. However, if someone knows all the private spend key components then they know the total private spend key. Add in the private view key from earlier, and he can sign transactions on his own. It wouldn't be multisignature, just a plain old signature.

Instead, we get the same effect if the shared spend key is a sum of public spend keys. Say the participants have public spend keys $K_e^{s,sh,c}$ which they send to each other securely. Now let them each compute

$$K^{s,sh} = \sum_e K_e^{s,sh,c}$$

Clearly this is the same as

$$K^{s,sh} = (\sum_e k_e^{s,sh,c})G$$

6.2.2 Drawbacks to naive approach

Using a sum of public spend keys is intuitive and seemingly straightforward, but leads to a few issues.

Key aggregation test

An outside adversary who knows all the public spend keys $K_e^{s,sh,c}$ can trivially test a given public address (K^v, K^s) for key aggregation by computing $K^{s,sh} = \sum_e K_e^{s,sh,c}$ and checking $K^s \stackrel{?}{=} K^{s,sh}$. This ties in with a broader requirement that aggregated keys be indistinguishable from normal keys so observers can't gain any insight into users' activities based on the kind of address they publish.[2]

We can get around this by creating brand new spend keys for each multisignature address, or by masking old keys. The former case is easy, but may be inconvenient.

The second case proceeds like this: given participant e's old key pair (K_e^v, K_e^s) with private keys (k_e^v, k_e^s) and random masks μ_e^v, μ_e^s,[3] let his new private key components for the shared address be

$$k_e^{v,sh,c} = \mathcal{H}_n(k_e^v, \mu_e^v)$$
$$k_e^{s,sh,c} = \mathcal{H}_n(k_e^s, \mu_e^s)$$

If participants don't want observers to gather the new keys and test for key aggregation, they would have to communicate their new key components to each other securely.

If key aggregation tests are not a concern, they could publish their public key components $(K_e^{v,sh,c}, K_e^{s,sh,c})$ as normal addresses. Any third party could then compute the shared address from those individual addresses and send funds to it, without interacting with any of the joint recipients [41].

[2] If at least one honest participant uses share components selected randomly from a uniform distribution, then keys aggregated by a simple sum are indistinguishable [55].

[3] The random masks could easily be derived from some password. For example, $\mu^s = \mathcal{H}_n(password)$ and $\mu^v = \mathcal{H}_n(\mu_s)$.

Key cancelation

If the shared spend key is a sum of public keys, a dishonest participant who learns his collaborators' spend key components ahead of time can cancel them.

For example, say there are two people, Alice and Bob, who want to make a shared address. Alice, in good faith, tells Bob her key components $(k_A^{v,sh,c}, K_A^{s,sh,c})$. Bob privately makes his key components $(k_B^{v,sh,c}, K_B^{s,sh,c})$ but doesn't tell Alice right away. Instead, he computes $K_B'^{s,sh,c} = K_B^{s,sh,c} - K_A^{s,sh,c}$ and tells Alice $(k_B^{v,sh,c}, K_B'^{s,sh,c})$. The shared address is:

$$\begin{aligned} K^{v,sh} &= (k_A^{v,sh,c} + k_B^{v,sh,c})G \\ &= k^{v,sh}G \\ K^{s,sh} &= K_A^{s,sh,c} + K_B'^{s,sh,c} \\ &= K_A^{s,sh,c} + K_B^{s,sh,c} - K_A^{s,sh,c} \\ &= K_B^{s,sh,c} \end{aligned}$$

This leaves a shared address $(k^{v,sh}G, K_B^{s,sh,c})$ where Alice knows the private shared view key, and Bob knows both the private view key *and* private spend key! Bob can sign transactions on his own, fooling Alice, who might believe funds sent to the address can only be spent with her permission.

We could solve this issue by requiring each participant, before aggregating keys, make a signature proving they know the private key to their spend key component.[4] This is inconvenient and vulnerable to implementation mistakes. Fortunately, a solid alternative is available.

6.2.3 Robust key aggregation

We can easily resist key cancellation with a small change to spend key aggregation (leaving view key aggregation the same). Let the set of N signers' spend key components be $\mathbb{S} = \{K_1^{s,sh,c}, ..., K_N^{s,sh,c}\}$ ordered according to some convention (such as smallest to largest numerically). The robust aggregated spend key is

$$K^{s,sh} = \sum_e \mathcal{H}_n(K_e^{s,sh,c}, \mathbb{S}) K_e^{s,sh,c}$$

Now if Bob tries to cancel Alice' spend key, he gets stuck with a very difficult problem.

$$\begin{aligned} K^{s,sh} &= \mathcal{H}_n(K_A^{s,sh,c}, \mathbb{S}) K_A^{s,sh,c} + \mathcal{H}_n(K_B'^{s,sh,c}, \mathbb{S}) K_B'^{s,sh,c} \\ &= \mathcal{H}_n(K_A^{s,sh,c}, \mathbb{S}) K_A^{s,sh,c} + \mathcal{H}_n(K_B'^{s,sh,c}, \mathbb{S}) K_B^{s,sh,c} - \mathcal{H}_n(K_B'^{s,sh,c}, \mathbb{S}) K_A^{s,sh,c} \\ &= [\mathcal{H}_n(K_A^{s,sh,c}, \mathbb{S}) - \mathcal{H}_n(K_B'^{s,sh,c}, \mathbb{S})] K_A^{s,sh,c} + \mathcal{H}_n(K_B'^{s,sh,c}, \mathbb{S}) K_B^{s,sh,c} \end{aligned}$$

We leave Bob's frustration to the reader's imagination.

Just like with the naive approach, any third party who knows $\mathbb{S}$ and the corresponding public view keys can compute the shared address.

[4] Monero's first iteration of multisignature, made available in April 2018, used this naive key aggregation, and required users sign their spend key components.

Since participants don't need to prove they know their private spend keys, or really interact at all prior to signing transactions, our robust key aggregation meets the so-called *plain public-key model*, where "the only requirement is that each potential signer has a public key"[41].[5]

6.3 Thresholded Schnorr-like signatures

It takes a certain amount of signers for a multisignature to work, so we say there is a 'threshold' of signers below which the signature can't be produced. A multisignature with N participants that requires all N people to build a signature, usually referred to as *N-of-N multisig*, would have a threshold of N. Later we will generalize this to M-of-N (M $\leq$ N) multisig where N participants create the shared address but only M people are needed to make signatures.

Let's take a step back from Monero and explore general N-of-N thresholded signatures. Say there are N people who each have a public key in the set $\mathbb{S} = \{K_1^{sh,c}, ..., K_N^{sh,c}\}$, where each person $e \in \{1, ..., N\}$ knows the private key $k_e^{sh,c}$. Their shared public key, which they will use to sign messages, is

$$K^{sh} = \sum_e \mathcal{H}_n(K_e^{sh,c}, \mathbb{S}) K_e^{sh,c}$$

6.3.1 Simple threshold Schnorr-like signatures

All signatures in this document lead from Maurer's general zero-knowledge proof of knowledge [38], so we can demonstrate the essential form of thresholded signatures using a simple Schnorr-like signature (recall Section 2.3.5).

Signature

Suppose the people of set $\mathbb{S}$ want to jointly sign a message $\mathfrak{m}$ using their shared public key K^{sh}. They could collaborate on a basic Schnorr-like signature like this

1. Each participant $e \in \{1, ..., N\}$ does the following:
   (a) picks random component $\alpha_e^{sh,c} \in_R \mathbb{Z}_l$,
   (b) computes $\alpha_e^{sh,c} G$,
   (c) and sends $\alpha_e^{sh,c} G$ to the other participants securely.
2. Each participant computes
$$\alpha^{sh} G = \sum_e \alpha_e^{sh,c} G$$
3. Each participant $e \in \{1, ..., N\}$ does the following:

[5] As we will see later, key aggregation only meets the plain public-key model for N-of-N and 1-of-N multisig.

(a) computes the challenge $c = \mathcal{H}_n(\mathfrak{m}, [\alpha^{sh}G])$,

(b) defines $r_e^{sh,c} = \alpha_e^{sh,c} - c * \mathcal{H}_n(K_e^{sh,c}, \mathbb{S})k_e^{sh,c} \pmod{l}$,

(c) and sends $r_e^{sh,c}$ to the other participants securely.

4. Each participant computes
$$r^{sh} = \sum_e r_e^{sh,c}$$

5. Any participant can publish the signature $\sigma(\mathfrak{m}) = (c, r^{sh})$.

Verification

Given K^{sh}, $\mathfrak{m}$, and $\sigma(\mathfrak{m}) = (c, r^{sh})$:

1. Compute the challenge $c' = \mathcal{H}_n(\mathfrak{m}, [r^{sh}G + c * K^{sh}])$.

2. If $c' = c$ then the signature is legitimate except with negligible probability.

We included the superscript sh for clarity, but in reality the verifier has no way to tell K^{sh} is aggregated unless a participant tells him, or unless he knows the key components $K_e^{sh,c}$.

Why it works

Secret component r^{sh} is the core of this signature. Participant e knows two secrets in $r_e^{sh,c}$, so his private key $k_e^{sh,c}$ is information-theoretically secure from other participants (assuming he never reuses $\alpha_e^{sh,c}$). Moreover, verifiers use the shared public key K^{sh}, so all key components are needed to build signatures.

$$\begin{aligned}
r^{sh}G &= (\sum_e r_e^{sh,c})G \\
&= (\sum_e (\alpha_e^{sh,c} - c * \mathcal{H}_n(K_e^{sh,c}, \mathbb{S})k_e^{sh,c}))G \\
&= (\sum_e \alpha_e^{sh,c})G - c * (\sum_e \mathcal{H}_n(K_e^{sh,c}, \mathbb{S})k_e^{sh,c})G \\
&= \alpha^{sh}G - c * K^{sh} \\
\alpha^{sh}G &= r^{sh}G + c * K^{sh} \\
\mathcal{H}_n(\mathfrak{m}, [\alpha^{sh}G]) &= \mathcal{H}_n(\mathfrak{m}, [r^{sh}G + c * K^{sh}]) \\
c &= c'
\end{aligned}$$

6.3.2 Back's Linkable Spontaneous Threshold Anonymous Group (bLSTAG) signatures

Naturally, bLSTAG follows after simple Schnorr-like signatures. The concept remains the same, so we can jump right in. Recall Section 3.2.

Signature

Let $\mathfrak{m}$ be the message to sign, $\mathcal{R} = \{K_1, K_2, ..., K_n\}$ a ring of n distinct public keys and K_π^{sh} the N-of-N shared public key to be signed with, where π is a secret index in $\mathcal{R}$.

1. Each participant $e \in \{1, ..., N\}$ does the following:
   (a) computes key image component $\tilde{K}_e^{sh,c} = \mathcal{H}_n(K_{\pi,e}^{sh,c}, \mathbb{S})k_{\pi,e}^{sh,c}\mathcal{H}_p(K_\pi^{sh})$,
   (b) sends $\tilde{K}_e^{sh,c}$ to the other participants securely.
2. Each participant computes the shared key image
$$\tilde{K}^{sh} = \sum_e \tilde{K}_e^{sh,c}$$
3. Each participant $e \in \{1, ..., N\}$ does the following:
   (a) generates seed component $\alpha_e^{sh,c} \in_R \mathbb{Z}_l$ and computes $\alpha_e^{sh,c}G$ and $\alpha_e^{sh,c}\mathcal{H}_p(K_\pi^{sh})$,
   (b) generates random numbers $r_{i,e}^{sh,c} \in_R \mathbb{Z}_l$ for $i \in \{1, 2, ..., n\}$ but excluding $i = \pi$,
   (c) and sends $\alpha_e^{sh,c}G$, $\alpha_e^{sh,c}\mathcal{H}_p(K_\pi^{sh})$, and all $r_{i,e}^{sh,c}$ to the other participants securely.
4. Each participant computes
$$\alpha^{sh}G = \sum_e \alpha_e^{sh,c}G$$
$$\alpha^{sh}\mathcal{H}_p(K_\pi^{sh}) = \sum_e \alpha_e^{sh,c}\mathcal{H}_p(K_\pi^{sh})$$
   and all
$$r_i^{sh} = \sum_e r_{i,e}^{sh,c}$$
5. Each participant $e \in \{1, ..., N\}$ can now build the signature:
   (a) Compute
$$c_{\pi+1} = \mathcal{H}_n(\mathfrak{m}, [\alpha^{sh}G], [\alpha^{sh}\mathcal{H}_p(K_\pi^{sh})])$$
   (b) For $i = \pi + 1, \pi + 2, ..., n, 1, 2, ..., \pi - 1$ calculate, replacing $n + 1 \to 1$,
$$c_{i+1} = \mathcal{H}_n(\mathfrak{m}, [r_i^{sh}G + c_iK_i], [r_i^{sh}\mathcal{H}_p(K_i) + c_i\tilde{K}^{sh}])$$
6. To close the signature, each participant $e \in \{1, ..., N\}$ does the following:

(a) defines $r^{sh,c}_{\pi,e} = \alpha^{sh,c}_e - c_\pi * \mathcal{H}_n(K^{sh,c}_{\pi,e}, \mathbb{S})k^{sh,c}_{\pi,e} \pmod l$,

(b) sends $r^{sh,c}_{\pi,e}$ to the other participants.

7. Each participant can now compute
$$r^{sh}_\pi = \sum_e r^{sh,c}_{\pi,e}$$

The signature will be $\sigma(\mathfrak{m}) = (c_1, r^{sh}_1, ..., r^{sh}_n)$, with key image $\tilde{K}^{sh}$.

Verification

Verification does not change, so we will not repeat ourselves. See Section 3.2.

Why it works

Opening and closing the signature loop requires all key component owners to participate. Furthermore, the key image depends on each participants' private key components.

6.3.3 Multilayer Linkable Spontaneous Threshold Anonymous Group (MLSTAG) signatures

Expanding bLSTAG to MLSTAG is fairly trivial. Instead of one layer, there are m layers each containing a shared public key at the same secret index π. It does not matter if one group of people owns all the shared keys or if separate groups own them. In the end, each step of signature generation requires components from everyone. Recall Section 3.3.

6.4 MLSTAG Ring Confidential signatures for Monero

Monero thresholded ring confidential transactions add some complexity because MLSTAG signing keys are one-time (stealth) addresses and commitments to zero (for input amounts).

Recalling Section 5.2: one-time (stealth) addresses assigning ownership of a transaction's t^{th} output to whoever has public address (K^v_t, K^s_t) go like this

$$K^o_t = \mathcal{H}_n(rK^v_t, t)G + K^s_t = (\mathcal{H}_n(rK^v_t, t) + k^s_t)G$$
$$k^o_t = \mathcal{H}_n(rK^v_t, t) + k^s_t$$

We can update our notation for shared address $(K^{v,sh}_t, K^{s,sh}_t)$:
$$K^{o,sh}_t = \mathcal{H}_n(rK^{v,sh}_t, t)G + K^{s,sh}_t$$
$$k^{o,sh}_t = \mathcal{H}_n(rK^{v,sh}_t, t) + k^{s,sh}_t$$

Just as before, anyone with $k_t^{v,sh}$ and $K_t^{s,sh}$ can discover $K_t^{o,sh}$ is their address' output, and can decode the Diffie-Hellman terms for output amount and corresponding commitment mask (Section 5.6.1).

This also means multisig subaddresses are possible (Section 5.3). Multisig transactions using funds received to a subaddress require some fairly straightforward modifications to the following algorithms, which we mention in footnotes.

6.4.1 `RCTTypeFull` N-of-N multisig

Let's say the owners of a shared address $(K^{v,sh}, K^{s,sh})$ have received $j \in \{1, ..., m\}$ outputs with one-time addresses $K_j^{o,sh}$ and amounts $a_1, ..., a_m$, and now want to spend them with $t \in \{1, ..., p\}$ new outputs with amounts $b_1, ..., b_p$. Recall Section 5.6.

Most parts of a multisig transaction can be completed by whoever initiated it. Only the MLSTAG signatures require collaboration. An initiator needs to do these things:

1. Generate a transaction private key $r^{sh} \in_R \mathbb{Z}_l$ and communicate it to his fellow signers securely (Section 5.2).
2. Select a random mask for output t, $y_t^{sh} \in_R \mathbb{Z}_l$, and send the ECDH terms $mask_t$ and $amount_t$ to his fellow signers securely (Section 5.6.1).
3. Produce range proofs for each output t, and send the signatures to his fellow signers securely. Recall Sections 3.4, 4.3, 4.4.

Now the group of signers is ready to build input signatures.

Preparing transaction inputs

For each input j the participants select v sets of size m of additional unrelated addresses and their commitments from the blockchain, corresponding to apparently unspent outputs. They mix the addresses in a ring with their own m unspent outputs' addresses, adding fake commitments to zero, as follows (Section 5.6.3):

$$\begin{aligned}\mathcal{R} = \{&\{K_{1,1}^o, ..., K_{1,m}^o, (\sum_j C_{1,j} - \sum_t C_t^b)\},\\ &...\\ &\{K_{\pi,1}^{o,sh}, ..., K_{\pi,m}^{o,sh}, (\sum_j C_{\pi,j}^a - \sum_t C_t^b)\},\\ &...\\ &\{K_{v+1,1}^o, ..., K_{v+1,m}^o, (\sum_j C_{v+1,j} - \sum_t C_t^b)\}\}\end{aligned}$$

Here the private keys for $\{K^{o,sh}_{\pi,1}, ..., K^{o,sh}_{\pi,m}, (\sum_j C^a_{\pi,j} - \sum_t C^b_t)\}$ are $k^{o,sh}_{\pi,1}, ..., k^{o,sh}_{\pi,m}, z$, where, using u_j as the output index in the transaction where $K^{o,sh}_{\pi,j}$ was sent to the multisig address, the private key

$$k^{o,sh}_{\pi,j} = \mathcal{H}_n(rK^{v,sh}, u_j) + \sum_e \mathcal{H}_n(K^{s,sh,c}_e, \mathbb{S})k^{s,sh,c}_e$$

is a composite, which no single participant knows.[6]

Note that each participant can store the private key $k'^{s,sh,c}_e = \mathcal{H}_n(K^{s,sh,c}_e, \mathbb{S})k^{s,sh,c}_e$ instead of recomputing it every time. This is useful for separating the key merge and threshold signing processes.

Commitment to zero z is, assuming input amounts equal output amounts plus transaction fee, simply input commitment masks (obtained by Diffie-Hellman from received inputs, using the shared view key) minus output commitment masks (non-initiator participants can compute these from $mask_t$), which all participants can compute:

$$z = \sum_j x^{sh}_j - \sum_t y^{sh}_t$$

MLSTAG RingCT

First they construct the shared key images for all inputs $j \in \{1, ..., m\}$ with one-time addresses $K^{o,sh}_{\pi,j}$.

1. For each input j each participant e does the following:
   (a) computes $\tilde{K}^{o,sh,c}_{j,e} = \mathcal{H}_n(K^{s,sh,c}_e, \mathbb{S})k^{s,sh,c}_e \mathcal{H}_p(K^{o,sh}_{\pi,j})$,
   (b) sends $\tilde{K}^{o,sh,c}_{j,e}$ to the other participants securely.
2. Each participant can now compute[7]
$$\tilde{K}^{o,sh}_j = \mathcal{H}_n(rK^{v,sh}, u_j)\mathcal{H}_p(K^{o,sh}_{\pi,j}) + \sum_e \tilde{K}^{o,sh,c}_{j,e}$$

Then they build the MLSTAG signature.

1. Each participant e does the following for $j \in \{1, ..., m+1\}$:

[6] If $K^{o,sh}_{\pi,j}$ is built from an i-indexed multisig subaddress, then (from Section 5.3) it is a composite:

$$k^{o,sh}_{\pi,j} = \mathcal{H}_n(k^{v,sh} r_{u_j} K^{s,sh}, u_j) + \sum_e \mathcal{H}_n(K^{s,sh,c}_e, \mathbb{S})k^{s,sh,c}_e + \mathcal{H}_n(k^{v,sh}, i)$$

[7] If the one-time address corresponds to an i-indexed multisig subaddress, add

$$\tilde{K}^{o,sh}_j = ... + \mathcal{H}_n(k^{v,sh}, i)\mathcal{H}_p(K^{o,sh}_{\pi,j})$$

(a) generates seed components $\alpha^{sh,c}_{j,e} \in_R \mathbb{Z}_l$ and computes $\alpha^{sh,c}_{j,e} G$ and $\alpha^{sh,c}_{j,e} \mathcal{H}_p(K^{o,sh}_{\pi,j})$,

(b) generates, for $i \in \{1, ..., v+1\}$ except $i = \pi$, random components $r^{sh,c}_{i,j,e}$,

(c) and sends all $\alpha^{sh,c}_{j,e} G$, $\alpha^{sh,c}_{j,e} \mathcal{H}_p(K^{o,sh}_{\pi,j})$, and $r^{sh,c}_{i,j,e}$ to the other participants securely. Since all participants know commitment to zero z, they can send $\alpha^{sh,c}_{m+1,e}$ directly.

2. Each participant can compute all

$$\alpha^{sh}_j G = \sum_e \alpha^{sh,c}_{j,e} G$$
$$\alpha^{sh}_j \mathcal{H}_p(K^{o,sh}_{\pi,j}) = \sum_e \alpha^{sh,c}_{j,e} \mathcal{H}_p(K^{o,sh}_{\pi,j})$$
$$\alpha^{sh}_{m+1} = \sum_e \alpha^{sh,c}_{m+1,e}$$
$$r^{sh}_{i,j} = \sum_e r^{sh,c}_{i,j,e}$$

3. Each participant can build the signature loop (see Section 3.3), and close the commitment to zero layer by defining $r^{sh}_{\pi,m+1} = \alpha^{sh}_{m+1} - c_\pi z \pmod{l}$.

4. To finish closing the signature, each participant e does the following for $j \in \{1, ..., m\}$:

   (a) defines $r^{sh,c}_{\pi,j,e} = \alpha^{sh,c}_{j,e} - c_\pi \mathcal{H}_n(K^{s,sh,c}_e, \mathbb{S}) k^{s,sh,c}_e \pmod{l}$,

   (b) sends $r^{sh,c}_{\pi,j,e}$ to the other participants securely.

5. Everyone can compute[8]

$$r^{sh}_{\pi,j} = \sum_e r^{sh,c}_{\pi,j,e} - c_\pi * \mathcal{H}_n(rK^{v,sh}, u_j)$$

The signature is $\sigma(\mathfrak{m}) = (c_1, r^{sh}_{1,1}, ..., r^{sh}_{1,m+1}, ..., r^{sh}_{v+1,m+1})$ with $(\tilde{K}^{o,sh}_1, ..., \tilde{K}^{o,sh}_m)$.

Since in Monero the message $\mathfrak{m}$ and the challenge c_π depend on all other parts of the transaction, any participant who tries to cheat by sending the wrong value, at any point in the whole process, to his fellow signers will cause the signature to fail. $r^{sh}_{\pi,j}$ is only useful for the $\mathfrak{m}$ and c_π it is defined for.

6.4.2 `RCTTypeSimple` N-of-N multisig

`RCTTypeSimple` uses pseudo-output commitments for inputs, and signs inputs separately. To accommodate these differences from `RCTTypeFull`, we need one more sequence of events prior to building input MLSTAGs. The initiator must construct pseudo-output commitments (Section 5.7.1).

[8] If the one-time address $K^{o,sh}_{\pi,j}$ corresponds to an i-indexed multisig subaddress, include

$$r^{sh}_{\pi,j} = ... - c_\pi * \mathcal{H}_n(k^{v,sh}, i)$$

He selects, for each input $j \in \{1, ..., m-1\}$, mask components $x'^{sh}_j \in_R \mathbb{Z}_l$, then computes the mth mask as

$$x'^{sh}_m = \sum_t y^{sh}_t - \sum_{j=1}^{m-1} x'^{sh}_j$$

He communicates all x'^{sh}_j to the other participants securely.

With pseudo-output commitments $C'^a_j = x'^{sh}_j G + a_j H$, all participants will know the commitments to zero $z_j = x_j - x'_j$. MLSTAG signatures use just one input at a time with distinct secret indices π_j, but otherwise proceed the same as in Section 6.4.1.

6.4.3 Simplified communication

It takes a lot of steps to build a multisignature Monero transaction. We can reduce interactions between signers by consolidating communication into two parts with four total rounds.

1. Key aggregation for a multisig public address. Anyone with a set of public addresses can merge an N-of-N address, but no participant will know the shared view key unless they learn all the components, so the group starts by sending $k^{v,sh,c}_e$ and $K^{s,sh,c}_e$ to each other securely. Any participant can merge and publish $(K^{v,sh}, K^{s,sh})$, allowing the group to receive funds to the shared address. M-of-N aggregation requires more steps, which we describe in the next section.

2. Transactions:

   (a) Some participant or sub-coalition (the initiator) decides to write a transaction. They choose m inputs with one-time addresses $K^{o,sh}_j$ and amount commitments C^a_j, v sets of m additional one-time addresses and commitments to be used as mixins, pick p output recipients with public addresses (K^v_t, K^s_t) and amounts b_t to send them, decide a transaction fee f,[9] pick a transaction private key r^{sh},[10] construct the ECDH terms $mask_t$ and $amount_t$ for each input, produce range proofs for each input, and, if using `RCTTypeSimple`, generate pseudo-output commitments x'^{sh}_j with $j \neq m$. They also prepare their contribution for the next communication round.

   The initiator sends all this information to the other participants securely.[11] The other participants can signal agreement by sending their part of the next communication round, or negotiate for changes.

   (b) Each participant chooses their components of the MLSTAG signature(s), and sends all of it to other participants securely.

[9] The transaction fee does not have to be explicit, because anyone can infer it from $\sum_j a_j - \sum_t b_t = f$.
[10] Or transaction private keys r^{sh}_t if sending to at least one subaddress.
[11] He doesn't need to send the output amounts b_t directly, as they can be computed from $mask_t$.

MLSTAG Signature(s): key image $\tilde{K}^{o,sh,c}_{j,e}$, signature randomness $\alpha^{sh,c}_{j,e}G$, $\alpha^{sh,c}_{j,e}\mathcal{H}_p(K^{o,sh}_{\pi,j})$, $\alpha^{sh,c}_{m+1,e}$, and $r^{sh,c}_{i,j,e}$ with $i \neq \pi_j$.

(c) Each participant closes their part of the MLSTAG signature(s), sending all $r^{sh,c}_{\pi_j,j,e}$ to the other participants securely.

Assuming the process went well, all participants can finish writing the transaction and broadcast it on their own. Transactions authored by a multisig coalition are indistinguishable from those authored by individuals.

6.5 Recalculating key images

If someone loses their records and wants to calculate their address' balance (received minus spent funds), they need to check the blockchain for spent outputs. View keys are only useful for reading received funds; users need to calculate key images for all received outputs to see if they have been spent, by comparing with key images stored in the blockchain. Since members to a shared address can't compute key images on their own, they need to enlist the other participants' help.

Calculating key images from a simple sum of components might fail if dishonest participants report false keys. Given a received output with one-time address $K^{o,sh}$, the group can produce a simple 'linkable' Schnorr-like signature (basically single-key bLSTAG) to prove the key image $\tilde{K}^{o,sh}$ is legitimate without revealing their private spend key components or needing to trust each other.

Signature

1. Each participant e does the following:
   (a) computes $\tilde{K}^{o,sh,c}_e = \mathcal{H}_n(K^{s,sh,c}_e, \mathbb{S})k^{s,sh,c}_e\mathcal{H}_p(K^{o,sh})$,
   (b) generates seed component $\alpha^{sh,c}_e \in_R \mathbb{Z}_l$ and computes $\alpha^{sh,c}_e G$ and $\alpha^{sh,c}_e\mathcal{H}_p(K^{o,sh})$,
   (c) and sends $\tilde{K}^{o,sh,c}_e$, $\alpha^{sh,c}_e G$, and $\alpha^{sh,c}_e\mathcal{H}_p(K^{o,sh})$ to the other participants.
2. Each participant can compute:[12]

$$\tilde{K}^{o,sh} = \mathcal{H}_n(rK^{v,sh}, u)\mathcal{H}_p(K^{o,sh}) + \sum_e \tilde{K}^{o,sh,c}_e$$

$$\alpha^{sh}G = \sum_e \alpha^{sh,c}_e G$$

$$\alpha^{sh}\mathcal{H}_p(K^{o,sh}) = \sum_e \alpha^{sh,c}_e\mathcal{H}_p(K^{o,sh})$$

[12] If the one-time address corresponds to an i-indexed multisig subaddress, add

$$\tilde{K}^{o,sh} = ... + \mathcal{H}_n(k^{v,sh}, i)\mathcal{H}_p(K^{o,sh})$$

3. Each participant computes the challenge
$$c = \mathcal{H}_n([\alpha^{sh} G], [\alpha^{sh} \mathcal{H}_p(K^{o,sh})])$$

4. Each participant does the following:
   (a) defines $r_e^{sh,c} = \alpha_e^{sh,c} - c * \mathcal{H}_n(K_e^{s,sh,c}, \mathbb{S}) k_e^{s,sh,c} \pmod{l}$,
   (b) sends $r_e^{sh,c}$ to the other participants securely.

5. Each participant can compute[13]
$$r^{sh} = \sum_e r_e^{sh,c} - c * \mathcal{H}_n(r K^{v,sh}, u)$$

The signature is (c, r^{sh}) with $\tilde{K}^{o,sh}$.

Verification

1. Check $l\tilde{K}^{o,sh} \stackrel{?}{=} 0$.
2. Compute $c' = \mathcal{H}_n([r^{sh} G + c * K^{o,sh}], [r^{sh} \mathcal{H}_p(K^{o,sh}) + c * \tilde{K}^{o,sh}])$
3. If $c' = c$ then the key image $\tilde{K}^{o,sh}$ corresponds to one-time address $K^{o,sh}$ (except with negligible probability).

6.6 Smaller thresholds

At the beginning of this chapter we discussed escrow services, which used 2-of-2 multisig to split signing power between a user and a security company. That setup isn't ideal, because if the security company is compromised, or refuses to cooperate, your funds may get stuck.

We can get around that potentiality with a 2-of-3 multisig address, which has three participants but only needs two of them to sign transactions. Escrow services can offer 2-of-3 multisig where they provide one key and users provide two keys. Users can store one key in a secure location (like a safety deposit box), and use the other for day-to-day purchases. If the escrow service fails, a user can use the (secure key + day key) to withdraw funds.

Multisignatures with sub-N thresholds have a wide range of uses.

[13] If the one-time address $K^{o,sh}$ corresponds to an i-indexed multisig subaddress, include
$$r^{sh} = ... - c * \mathcal{H}_n(k^{v,sh}, i)$$

6.6.1 1-of-N key aggregation

Suppose a group of people want to make a multisig key K^{sh} they can all sign with. The solution is trivial: let everyone know the private key k^{sh}. There are three ways to do this.[14]

1. One participant or sub-coalition selects a key and sends it to everyone else securely.
2. All participants compute private key components and send them securely, using the simple sum as merged key.
3. Participants extend M-of-N multisig to 1-of-N. This might be useful if an adversary has access to the group's communications.

In this case, for Monero, everyone would know the private keys $(k^{v,sh,\text{1xN}}, k^{s,sh,\text{1xN}})$. Before this section all shared keys were N-of-N, but now we use superscript 1xN to denote 1-of-N merged keys.

6.6.2 (N-1)-of-N key aggregation

In an (N-1)-of-N shared key, such as 2-of-3 or 4-of-5, any set of (N-1) participants can sign. We achieve this with Diffie-Hellman shared secrets. Lets say there are participants $e \in \{1, ..., N\}$, with public keys K_e which they are all aware of.

Each participant e computes, for $w \in \{1, ..., N\}$ but $w \neq e$,

$$k_{e,w}^{sh,c,\text{(N-1)xN}} = \mathcal{H}_n(k_e K_w)$$

Then he computes $K_{e,w}^{sh,c,\text{(N-1)xN}} = k_{e,w}^{sh,c,\text{(N-1)xN}} G$ and sends it to all other participants securely.

Each participant will have (N-1) private key components corresponding to each of the other participants, making N(N-1) total keys between everyone. All keys are shared by two Diffie-Hellman partners, so there are really N(N-1)/2 unique keys. Those unique keys compose the set $\mathbb{S}$.

All N(N-1)/2 key components can be merged together using the robust key aggregation approach from Section 6.2.3. Furthermore, all N(N-1)/2 private key components can be assembled with just (N-1) participants since each participant shares one Diffie-Hellman secret with the Nth guy.

For example, say there are three people with public keys $\{K_1, K_2, K_3\}$, to which they each know a private key, who want to make a 2-of-3 multisig key. After Diffie-Hellman and sending each other the public keys they know the following:

1. Person 1: $k_{1,2}^{sh,c,\text{2x3}}$, $k_{1,3}^{sh,c,\text{2x3}}$, $K_{2,3}^{sh,c,\text{2x3}}$
2. Person 2: $k_{2,1}^{sh,c,\text{2x3}}$, $k_{2,3}^{sh,c,\text{2x3}}$, $K_{1,3}^{sh,c,\text{2x3}}$
3. Person 3: $k_{3,1}^{sh,c,\text{2x3}}$, $k_{3,2}^{sh,c,\text{2x3}}$, $K_{1,2}^{sh,c,\text{2x3}}$

[14] Note that key cancellation is largely meaningless here because everyone knows the full private key.

Where $k_{1,2}^{sh,c,2\text{x}3} = k_{2,1}^{sh,c,2\text{x}3}$, and so on. The set $\mathbb{S} = \{K_{1,2}^{sh,c,2\text{x}3}, K_{1,3}^{sh,c,2\text{x}3}, K_{2,3}^{sh,c,2\text{x}3}\}$.

The merged key is:

$$\begin{aligned} K^{sh,2\text{x}3} =& \mathcal{H}_n(K_{1,2}^{sh,c,2\text{x}3}, \mathbb{S}) K_{1,2}^{sh,c,2\text{x}3} + \\ & \mathcal{H}_n(K_{1,3}^{sh,c,2\text{x}3}, \mathbb{S}) K_{1,3}^{sh,c,2\text{x}3} + \\ & \mathcal{H}_n(K_{2,3}^{sh,c,2\text{x}3}, \mathbb{S}) K_{2,3}^{sh,c,2\text{x}3} \end{aligned}$$

Now let's say persons 1 and 2 want to sign a message $\mathfrak{m}$. We will use a basic Schnorr-like signature to demonstrate.

1. Each participant $e \in \{1, 2\}$ does the following:
   (a) picks random component $\alpha_e^{sh,c} \in_R \mathbb{Z}_l$,
   (b) computes $\alpha_e^{sh,c} G$,
   (c) and sends $\alpha_e^{sh,c} G$ to the other participants securely.
2. Each participant computes
$$\alpha^{sh} G = \sum_e \alpha_e^{sh,c} G$$
3. Participant 1 does the following:
   (a) computes the challenge $c = \mathcal{H}_n(\mathfrak{m}, [\alpha^{sh} G])$,
   (b) computes $r_1^{sh,c} = \alpha_1^{sh,c} - c * [\mathcal{H}_n(K_{1,3}^{sh,c,2\text{x}3}, \mathbb{S}) k_{1,3}^{sh,c,2\text{x}3} + \mathcal{H}_n(K_{1,2}^{sh,c,2\text{x}3}, \mathbb{S}) k_{1,2}^{sh,c,2\text{x}3}]$,
   (c) and sends $r_1^{sh,c}$ to participant 2 securely.
4. Participant 2 does the following:
   (a) computes the challenge $c = \mathcal{H}_n(\mathfrak{m}, [\alpha^{sh} G])$,
   (b) computes $r_2^{sh,c} = \alpha_2^{sh,c} - c * \mathcal{H}_n(K_{2,3}^{sh,c,2\text{x}3}, \mathbb{S}) k_{2,3}^{sh,c,2\text{x}3}$,
   (c) and sends $r_2^{sh,c}$ to participant 1 securely.
5. Each participant computes
$$r^{sh} = \sum_e r_e^{sh,c}$$
6. Any participant can publish the signature $\sigma(\mathfrak{m}) = (c, r^{sh})$.

The only change with sub-N threshold signatures is how to 'close the loop' by defining $r_{\pi,e}^{sh,c}$ (in the case of ring signatures, with secret index π). Each participant must include their shared secret corresponding to the 'missing person', but since all the other shared secrets are doubled up there is a trick. Given the set $\mathbb{S}_o$ of all participants' original keys, only the *first person* - ordered by index in $\mathbb{S}_o$ - with the copy of a shared secret uses it to calculate his $r_{\pi,e}^{sh,c}$.

In the previous example, participant 1 computes

$$r_1^{sh,c} = \alpha_1^{sh,c} - c * [\mathcal{H}_n(K_{1,3}^{sh,c,2\text{x}3}, \mathbb{S}) k_{1,3}^{sh,c,2\text{x}3} + \mathcal{H}_n(K_{1,2}^{sh,c,2\text{x}3}, \mathbb{S}) k_{1,2}^{sh,c,2\text{x}3}]$$

while participant 2 only computes

$$r_2^{sh,c} = \alpha_2^{sh,c} - c * \mathcal{H}_n(K_{2,3}^{sh,c,2\text{x}3}, \mathbb{S})k_{2,3}^{sh,c,2\text{x}3}$$

The same principle applies to computing the shared key image in sub-N threshold Monero multisig transactions.

6.6.3 M-of-N key aggregation

We can understand M-of-N by adjusting our perspective on (N-1)-of-N. In (N-1)-of-N every shared secret between two public keys, such as K_1 and K_2, contains two private keys, k_1k_2G. It's a secret because only person 1 can compute k_1K_2, and only person 2 can compute k_2K_1.

What if there is a third person with K_3, there exists shared secrets k_1k_2G, k_1k_3G, and k_2k_3G, and the participants send these public keys to each other (making them no longer secret)? They each contributed a private key to two of the public keys. Now say they make a new shared secret with that third public key.

Person 1 computes shared secret $k_1 * (k_2k_3G)$, person 2 computes $k_2 * (k_1k_3G)$, and person 3 computes $k_3 * (k_1k_2G)$. Now they all know $k_1k_2k_3G$, making a three-way shared secret (so long as no one publishes it).

The group could use $k^{sh} = \mathcal{H}_n(k_1k_2k_3G)$ as a shared private key, and publish $k^{sh}G$ as a 1-of-3 multisig address.

In a 3-of-3 multisig every 1 person has a secret, in a 2-of-3 multisig, every group of 2 people has a shared secret, and in 1-of-3 every group of 3 people has a shared secret.

Now we can generalize to M-of-N: every possible group of (N-M+1) people have a shared secret. If (N-M) people are missing, all their shared secrets are owned by at least 1 person who remains. There are M people remaining, and they can collaborate to sign with the group's aggregated key.

M-of-N algorithm

Given participants $e \in \{1, ..., N\}$ with initial private keys $k_1, .., k_N$ who wish to produce an M-of-N merged key (M $\leq$ N; M and N $\geq$ 2 since 1-of-N is trivial), we can use an interactive algorithm.

We will use $\mathbb{S}_s$ to denote all the *unique* public keys at stage $s \in \{0, ..., (N-M)\}$. Each set is indexed with $g_s \in \{1, ..., (\text{size of } \mathbb{S}_s)\}$, and the set $\mathbb{S}_{N-M}$ is ordered according to a sorting convention (such as smallest to largest numerically).[15]

We will use $\mathbb{S}_{s,e}^K$ to denote the set of public keys each participant created at stage s of the algorithm. In the beginning $\mathbb{S}_{0,e}^K = \{K_e\}$.

The set $\mathbb{S}_e^k$ will contain each participant's private keys at the end, when they hash their shared secrets. Participants use these keys to collaborate on signatures.

[15] Notation: we use $\mathbb{S}[n]$ to denote the nth element of the set.

1. Each participant e sends their original public key set $\mathbb{S}^K_{0,e} = \{K_e\}$ to the other participants securely.

2. Each participant builds $\mathbb{S}_0$ by collecting all $\mathbb{S}^K_{0,e}$ and removing duplicates.

3. For merge stage $s \in \{1, ..., (N-M)\}$ (skip if M = N)

   (a) Each participant e does the following:

      i. For each element of $\mathbb{S}_{s-1} \notin \mathbb{S}^K_{s-1,e}$, compute new shared secret
      $$k_e * \mathbb{S}_{s-1}[g_{s-1}]$$
      ii. Put all new shared secrets in $\mathbb{S}^K_{s,e}$.
      iii. If $s = (N-M)$, compute the shared private key for each element x in $\mathbb{S}^K_{N-M,e}$
      $$\mathbb{S}^k_e[x] = \mathcal{H}_n(\mathbb{S}^K_{N-M,e}[x])$$
      and overwrite the public key by setting $\mathbb{S}^K_{N-M,e}[x] = \mathbb{S}^k_e[x] * G$.
      iv. Send the other participants $\mathbb{S}^K_{s,e}$.

   (b) Each participant builds $\mathbb{S}_s$ by collecting all $\mathbb{S}^K_{s,e}$ and removing duplicates.[16]

4. Each participant sorts $\mathbb{S}_{N-M}$ according to the convention.

5. The merged key is computed like this
$$K^{sh,\mathrm{MxN}} = \sum_{g_{(N-M)}} \mathcal{H}_n(\mathbb{S}_{(N-M)}[g_{(N-M)}], \mathbb{S}_{(N-M)}) * \mathbb{S}_{(N-M)}[g_{(N-M)}]$$

6. Each participant e overwrites each element x in $\mathbb{S}^k_e$ with
$$\mathbb{S}^k_e[x] = \mathcal{H}_n(\mathbb{S}^k_e[x]G, \mathbb{S}_{(N-M)}) * \mathbb{S}^k_e[x]$$

Note: if users want to have unequal signing power in a multisig, like 2 shares in a 3-of-4, they should use multiple key components instead of reusing the same one.

6.7 Key families

Up to this point we have considered key aggregation between a simple group of signers. For example, Alice, Bob, and Carol each contributing key components to a 2-of-3 multisig address.

Now imagine Alice wants to sign all transactions from that address, but doesn't want Bob and Carol to sign without her. In other words, (Alice + Bob) or (Alice + Carol) are acceptable, but not (Bob + Carol).

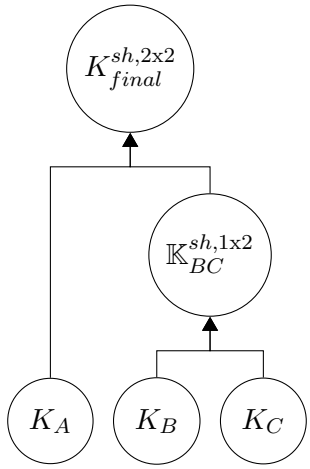
We can achieve that scenario with two layers of key aggregation: first a 1-of-2 multisig key $\mathbb{K}^{sh,1\mathrm{x}2}_{BC}$ between Bob and Carol, then a 2-of-2 multisig key $K^{sh,2\mathrm{x}2}_{final}$ between Alice and $\mathbb{K}^{sh,1\mathrm{x}2}_{BC}$. Basically, a (1+(1-of-2))-of-2 multisig address. We will explain what $\mathbb{K}$ means in the next section.

This implies access structures to signing rights can be fairly open-ended.

[16] Participants should keep track of who has which keys at the last stage ($s = N - M$), to facilitate collaborative signing, where only the first person in $\mathbb{S}_0$ with a certain private key uses it to sign. See Section 6.6.2.

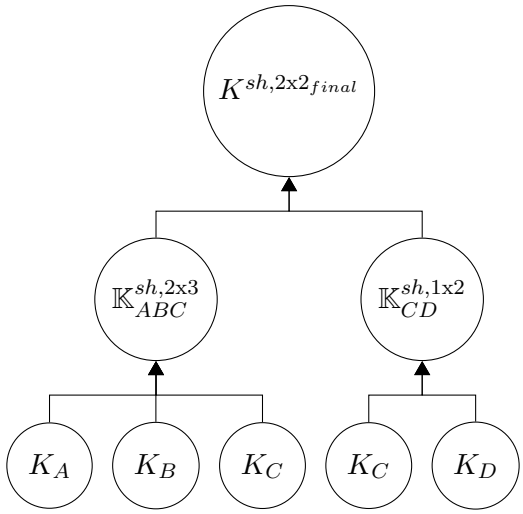
6.7.1 Family trees

We can diagram the (1+(1-of-2))-of-2 multisig address like this:



The keys K_A, K_B, K_C are considered *root ancestors*, while $\mathbb{K}^{sh,1x2}_{BC}$ is the *child* of *parents* K_B and K_C. Parents can have more than one child, though for conceptual clarity we consider each copy of a parent as distinct. This means there can be multiple root ancestors that are the same key.

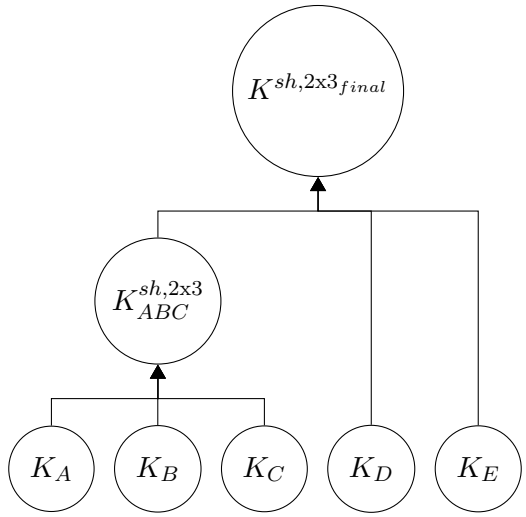
For example, in this 2-of-3 and 1-of-2 joined in a 2-of-2, Carol's key K_C is used twice and displayed twice:



Separate sets $\mathbb{S}$ are defined for each multisig sub-coalition. There are three sets in the previous example: $\mathbb{S}_{ABC} = \{K_{AB}, K_{BC}, K_{AC}\}$, $\mathbb{S}_{CD} = \{K_{CD}\}$, and $\mathbb{S}_{final} = \{\mathbb{K}^{sh,2x3}_{ABC}, \mathbb{K}^{sh,1x2}_{CD}\}$.

6.7.2 Nesting multisig keys

Suppose we have the following key family



If we merge the keys in $\mathbb{S}_{ABC}$ corresponding to the first 2-of-3, we run into an issue at the next level. Let's take just one shared secret, between K^{sh}_{ABC} and K_D, to demonstrate:

$$k_{ABC,D} = \mathcal{H}_n(k^{sh}_{ABC} K_D)$$

Now, two people from ABC could easily contribute key components so the sub-coalition can compute

$$k^{sh}_{ABC} K_D = \sum k^{sh,c} K_D$$

The problem is everyone from ABC can compute $k_{ABC,D} = \mathcal{H}_n(k^{sh}_{ABC} K_D)$! If everyone from a lower-tier multisig knows all its private keys for a higher-tier multisig, then the lower-tier multisig might as well be 1-of-N.

We get around this by not completely merging keys until the final child key. Instead, we just do the first part $\mathcal{H}_n(K, \mathbb{S})K$ for all keys output by low-tier multisigs, and put those in a key set

$$\mathbb{K}^{sh,out} = \{[\mathcal{H}_n(K_1, \mathbb{S})K_1], [\mathcal{H}_n(K_2, \mathbb{S})K_2], ...\}$$

To use $\mathbb{K}$ in a new multisig, we pass it around just like a normal key, with one change. Operations involving $\mathbb{K}$ use each of its member keys, instead of the whole merged key. For example, the public 'key' of a shared secret between $\mathbb{K}_x$ and K_A would produce a new key set that looks like

$$\mathbb{K}^{sh}_{x,A} = \{[\mathcal{H}_n(k_A \mathbb{K}_x[1])G], [\mathcal{H}_n(k_A \mathbb{K}_x[2])G], ...\}$$

This way all members of $\mathbb{K}_x$ only know shared secrets corresponding to their private keys from their lower-tier multisig. An operation between a keyset of size two ${}^2\mathbb{K}_A$ and keyset of size three ${}^3\mathbb{K}_B$ would produce a keyset of size six ${}^6\mathbb{K}_{AB}$. We can generalize all keys in a key family as keysets, where single keys are denoted ${}^1\mathbb{K}$. Elements of a keyset are ordered according to some convention (i.e. smallest to largest numerically), and sets containing keysets are ordered by the first element in each keyset, according to some convention.

We let the key sets propagate through the family structure until the final child appears, at which point every single key in all key sets is summed together.

More formally, we can say there is an operation `premerge` which takes in a set $\mathbb{S}$ and outputs a keyset $\mathbb{K}$ of equal size, containing each keyset $\mathbb{K}_i$ from $\mathbb{S}$ transformed into $\mathcal{H}_n(\mathbb{K}_i, \mathbb{S})\mathbb{K}_i$. There is

another operation `merge` which takes in a set $\mathbb{S}$, `premerges` it into $\mathbb{K}$, and then outputs a keyset ${}^1\mathbb{K}$ that is the sum of all keysets of size one in $\mathbb{K}$. `Premerge` is used on the output sets of nested multisigs, and `merge` is used on the final child multisig's output set.[17]

While this may seem very complicated, a well-designed algorithm can easily keep track of everything.

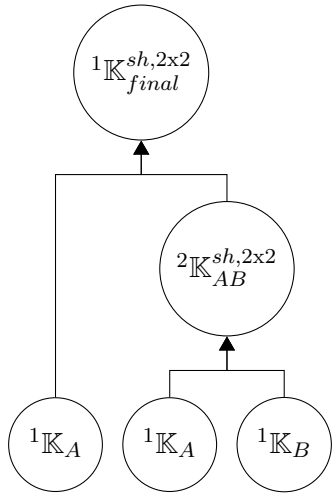
Note that the operation $\mathcal{H}_n(\mathbb{K}, \mathbb{S})\mathbb{K}$ (an abbreviated notation - the key set's members need to be separated out) needs to be done to the outputs of *all* nested multisigs, even when an N'-of-N' multisig is nested into an N-of-N, because the set $\mathbb{S}$ will change.

6.7.3 Implications for Monero

Each sub-coalition contributing to the final key needs to contribute components to Monero transactions, and so every sub-sub-coalition needs to contribute to its child sub-coalition.

This means every root ancestor, even when there are multiple copies of the same key in the family structure, must contribute one root component to their child, and each child one component to its child and so on. We use simple sums at each level.

For example, let's take this family



Say they need to compute some shared value x for a signature. Root ancestors contribute: $x_{A,1}$, $x_{A,2}$, x_B. The total is $x^{sh} = x_{A,1} + x_{A,2} + x_B$.

[17] `Merge` can also be used on the output set of a nested multisig, if the sub-coalition composing it wants to use their shared address for other purposes.

CHAPTER 7

The Monero Blockchain

The Internet Age has brought a new dimension to the human experience. We can correspond with people on every corner of the planet, and an unimaginable wealth of information is at our fingertips. Exchanging goods and services is fundamental to a peaceful and prosperous society, and in the digital realm we can offer our productivity to the whole world.

Media of exchange (moneys) are essential, giving us a point of reference to an immense diversity of economic goods that would otherwise be impossible to evaluate, and enabling mutually beneficial interactions between people with nothing in common. Throughout history there have been many kinds of money, from sea shells to paper to gold. Those were exchanged by hand, and now money can be exchanged electronically.

In the current, by far most pervasive, model, electronic transactions are handled by third party financial institutions. These institutions are given custody of money and trusted to transfer it upon request. Such institutions must mediate disputes, their payments are reversible, and they can be censored or controlled by powerful organizations. [44]

It seems appropriate for the Internet Age to have its own unique currencies.

Note: this chapter includes more implementation details than previous chapters, as a blockchain's nature depends heavily on its parameters and specific structure.

7.1 Digital currency

Let's try to make a digital currency from scratch.

Suppose two pals Jim and Dwight need a currency for their Secret Society of Stealthy Sleuths. Jim sends an email to Dwight saying "As co-founder, I hereby conjure 5 Stealthbucks for myself and 5 Stealthbucks for you." Later that day Dwight discovers Kevin had made 69 Stealthbucks for himself and used them to buy Dwight's bobblehead from Jim. There should only be 10 Stealthbucks, what went wrong?

In the **email model** anyone can make Stealthbucks, and anyone can send their Stealthbucks over and over. It does not have a limited supply, nor is it 'double spend proof'.

Jim proposes a new system, Stealthbucks 2.0, where he keeps a record on his computer of who owns all the Stealthbucks. To exchange them, people need to talk to Jim. Since Jim is the sole custodian, there is no question the Stealthbucks on Kevin's computer are bogus. Jim promises Dwight they will start off with 100 Stealthbucks each and that's it - no more. Dwight tries to buy his bobblehead back and Jim says "You need to prove your identity!" What went wrong?

In the **video game model**, where the entire currency is stored on one central database, users rely on the custodian to be honest. The currency's supply is unverifiable for observers, and the custodian can change the rules at any time, or be censored by powerful outsiders.

7.1.1 Shared version of events

Since Jim can't be trusted to manage Stealthbucks, Dwight suggests setting up a bunch of computers that each have a record of every Stealthbuck transaction. When a new transaction is made on one computer, it is broadcast to the other computers, which only accept it if it follows the rules.

Rule 1: Money can only be created in clearly defined scenarios.

Rule 2: Transactions spend money that already exists.

Rule 3: Transactions output money equal to the money spent.

Rule 4: Transactions are formatted correctly.

Rule 5: Only the person who owns a piece of money can spend it.

Rule 6: A person can only spend a piece of money once.

Rules 2-6 are already covered by the transaction schemes discussed in Chapter 5, which add the benefits of ambiguous signing, anonymous receipt of funds, and unreadable amount transfers. We explain Rule 1 later in this chapter.

What if one of the computers goes rogue and starts making a bunch of Stealthbucks for itself? No one will accept transactions spending counterfeit Stealthbucks because they know *other* users only want legitimate coins. Users only benefit from Stealthbucks when other users accept it in exchange. Each user must act honestly and follow the rules if they want to spend their money.

We call this situation a 'social minimum', 'Schelling point', or 'social contract'.

Simply storing all transactions has a problem. If two computers receive legitimate transactions spending the same money, before they have a chance to send the information to each other, how do they decide which is correct? There is a 'fork' in the currency, because two different copies that follow the same rules exist.

It seems obvious the earliest legitimate transaction spending a piece of money should be canonical. This is easier said than done. As we will see, obtaining consensus for transaction histories constitutes the raison d'être of blockchain technology.

7.1.2 Simple blockchain

First, we need all of the computers, henceforth referred to as *nodes*, to agree on the order of transactions.

Let's say Stealthbucks started with a 'genesis' declaration by Jim and Dwight: "Let the Stealthbucks begin!". We call this message a 'block', and its block hash is

$$BH_G = \mathcal{H}_n(\text{Let the Stealthbucks begin!})$$

Every time a node receives some transactions, they use the transaction hashes, TH, as messages, along with the previous block's hash, and compute new block hashes

$$BH_1 = \mathcal{H}_n(BH_G, TH_1, TH_2, ...)$$
$$BH_2 = \mathcal{H}_n(BH_1, TH_1, TH_2, ...)$$

And so on, publishing each new block of messages as it's made. Each new block references the previous, most recently published block. In this way a clear order of events extends all the way back to the genesis message. We have a chain of blocks: a very simple 'blockchain'.[1]

Nodes can include a timestamp in their blocks to aid record keeping. If most nodes are honest with timestamps then the blockchain provides a decent picture of when each transaction was recorded.

What if different transactions spending the same money are added to blocks referencing the same previous block, which are published at the same time? The network of nodes will fork again, as each node receives one of the new blocks before the other (for simplicity, imagine about half the nodes have each side of the fork). Let's keep improving our blockchain.

7.2 Difficulty

If nodes can publish new blocks whenever they want, the network will tend to fracture and diverge into many different, equally legitimate, chains. Say it takes 30 seconds to make sure everyone in

[1] A blockchain is technically a 'directed acyclic graph' (DAG), with Bitcoin-style blockchains a one-dimensional variant. DAGs contain a finite number of nodes and one-directional edges (vectors) connecting nodes. If you start at one node, you will never loop back to it no matter what path you take. [6]

the network gets a new block. What if transactions are made every 31 seconds? New blocks would just barely make it everywhere before another one gets sent out.

Now what if new blocks are every 15 seconds, 10 seconds, etc? Since message transmission time is a function of distance, the network would fracture into small clumps circumscribed by the time it takes for a new block to propagate before a new one is produced.

We can avoid this by controlling how fast the entire network makes new blocks. If the time it takes to make a new block is much higher than the time for the previous block to reach every node, the network will tend to remain intact.

7.2.1 Mining a block

The output of a hash function is uniformly distributed. This means, for any given input, its hash is equally likely to be every single possible output. Furthermore, it takes a certain amount of time to compute a single hash.

Let's imagine a hash function $\mathcal{H}_i(x)$ which outputs a number from 1 to 100: $\mathcal{H}_i(x) \in_R^D \{1, ..., 100\}$. We use $\in_R^D$ to say the output is deterministically random. Given some x, $\mathcal{H}_i(x)$ selects the same 'random' number from {1,...,100} every time you calculate it. It takes 1 minute to calculate $\mathcal{H}_i(x)$.

Say we are given a message $\mathfrak{m}$ and so-called 'nonce' $n = 1$, and told to find an n such that $\mathcal{H}_i(\mathfrak{m}, n) \in \{1, ..., 10\}$. Guessing and checking by incrementing n for each new hash, how many hashes will it probably take?

It should be around 10 hashes, for 10 minutes of hashing, because there is only a $1/10^{\text{th}}$ chance any given input will output a good answer. This isn't to say $n = 10$ is the right answer, just that if we take a lot of messages and do this process for each one, $n = 10$ will be the *average* value.

We call searching for a useful nonce *mining*, and publishing the message with its nonce is a *proof of work* because it proves we looked for a useful nonce (even if we were lucky and it needed just one hash to find), which anyone can verify by computing $\mathcal{H}_i(\mathfrak{m}, n)$.

Now say we have a hash function for generating proofs of work $\mathcal{H}_{PoW} \in_R^D \{0, ..., m\}$, where m is its maximum possible output. Given a message $\mathfrak{m}$ (a block of information), and a nonce n to mine, we can define the *difficulty* d like this: if $\mathcal{H}_{PoW}(\mathfrak{m}, n) * d < m$, then n is accepted.

In reality we will look for a *target* hash t such that $\mathcal{H}_{PoW}(\mathfrak{m}, n) < t$. The target $t = m/d$. It should take about d hashes to mine a useful n, because the probability of $\mathcal{H}_{PoW}(\mathfrak{m}, n) < t$ working is $t/m = 1/d$ (it was 10/100 in the example above).

As difficulty goes up, it takes a computer more and more hashes, and therefore longer and longer periods of time, to find useful nonces.

7.2.2 Mining speed

Assume all nodes are mining nonces at the same time, but quit on their 'current' block when they receive a new one from the network. They immediately start mining a fresh block that references the new one.

Suppose we let $t = m$ so any hash works (the difficulty is $d = 1$), and collect a bunch b of blocks B from the blockchain (say, $u = 1, ..., b$). For now, assume the nodes who mined them were honest, so each block timestamp TS_u is accurate. The total time between the earliest block and most recent block is $totalTime = TS_b - TS_1$. The approximate number of hashes to compute all the block hashes is $totalDifficulty = \sum_u d_u$, in this case b hashes.

Now we can guess how fast the network, with all its nodes, can compute hashes. If the actual speed didn't change much while the bunch of blocks was being produced, it should be effectively[2]

$$hashSpeed \approx totalDifficulty/totalTime$$

If we want to set the target time to mine new blocks, so blocks are produced at a rate (one block)/(target time), then from the hash speed we can calculate how many hashes it should take for the network to spend that amount of time mining.

$$miningHashes = hashSpeed * targetTime$$

Since difficulty is approximately how many hashes it takes to generate a proof of work, we can compute a new difficulty to ensure it takes around *miningHashes* to produce the next block. Note: we round up so difficulty never equals zero.

$$newDifficulty = (totalDifficulty/totalTime) * targetTime$$

There is no guarantee the next block will take *newDifficulty* amount of hashes to mine, but over time and many blocks and constantly re-calibrating, the difficulty will track with the network's real hash speed and blocks will tend to take *targetTime*.[3]

7.2.3 Consensus: largest cumulative difficulty

Now we have a mechanism to resolve conflicts between chain forks. Since difficulty represents how much work was spent to mine a block, higher difficulty means more work performed.

By convention, the chain with highest cumulative difficulty (from all blocks in the chain), and therefore with most work spent constructing, is considered the real, legitimate version. If a chain splits and each fork has the same cumulative difficulty, nodes continue mining on their fork until one branch gets ahead of the other, at which point the weaker branch is discarded ('orphaned').

[2] If node 1 tries nonce $n = 23$ and later node 2 also tries $n = 23$, node 2's effort is wasted because the network already 'knows' $n = 23$ doesn't work (otherwise node 1 would have published that block). The network's *effective* hash rate depends on how fast it hashes *unique* nonces.

[3] If we assume network hash rate is constantly, gradually, increasing, then since new difficulties depend on *past* hashes (i.e. before the hash rate increased a tiny bit) we should expect actual block times to, on average, be slightly less than *targetTime*. The effect of this on the emission schedule (Section 7.3.1) could be canceled out by penalties from increasing block sizes, which we explore in Section 7.3.2.

If nodes wish to change or upgrade the basic protocol, i.e. the set of rules a node considers when deciding if a blockchain copy or new block is legitimate, they may easily do so by forking the chain. Whether the new branch has any impact on users depends on how many nodes switch and how much software infrastructure is modified.

For an attacker to convince honest nodes to alter the transaction history, perhaps in order to respend/unspend funds, he must create a chain fork (on the current protocol) with higher total difficulty than the main chain. This is very hard to do unless you control over 50% of the network hash speed. [44]

7.2.4 Mining in Monero

To make sure chain forks are on an even footing, we don't sample the most recent blocks (for calculating new difficulties), instead lagging our bunch b by l. For example, if there are 29 blocks in the chain (blocks 1,...,29), $b = 10$, and $l = 5$, we sample blocks 15-24 in order to compute block 30's difficulty.

If mining nodes are dishonest they can manipulate timestamps so new difficulties don't match the network's real hash speed. We get around this by sorting timestamps chronologically, then chopping off the first o outliers and last o outliers. Now we have a 'window' of blocks $w = b - 2 * o$. From the previous example, if $o = 3$ and timestamps are honest then we would chop blocks 15-17 and 22-24, leaving blocks 18-21 to compute block 30's difficulty from.

Monero is somewhat bizarre. Instead of sorting block difficulties so they correspond with their blocks' sorted timestamps, we use an array of the original bunch's blocks' cumulative difficulties, leave it unsorted, and chop off the o outliers from that. Cumulative difficulty for a block is that block's difficulty plus the difficulty of all previous blocks in the chain.

src/cryptonote_core_blockchain.cpp get_difficulty_for_next_block()

Using the chopped arrays of w timestamps and cumulative difficulties (indexed from 1,...,w), we define

$$totalTime = choppedSortedTimestamps[w] - choppedSortedTimestamps[1]$$
$$totalDifficulty = choppedCumulativeDifficulties[w] - choppedCumulativeDifficulties[1]$$

In Monero the target time is 120 seconds (2 minutes), $l = 15$ (30 mins), $b = 720$ (one day), and $o = 60$.[4]

src/cryptonote_config.h

Block difficulties are not stored in the blockchain, so someone downloading a copy of the blockchain and verifying all blocks are legitimate needs to recalculate difficulties from recorded timestamps. There are a few rules to consider for the first $b + l = 735$ blocks.

src/cryptonote_basic_difficulty.cpp next_difficulty()

> **Rule 1**: Ignore the genesis block (block 0, with $d = 1$) completely. Blocks 1 and 2 have $d = 1$.
>
> **Rule 2**: Before chopping off outliers, try to get the window w to compute totals from.

[4] In March, 2016, (v2 of the protocol) Monero changed from 1 minute target block times to 2 minute target block times [12]. Other difficulty parameters have always been the same.

Rule 3: After w blocks, chop off high and low outliers, scaling the amount chopped until b blocks. If the amount of previous blocks is odd, remove one more low outlier than high.

Rule 4: After b blocks, sample the earliest b blocks until $b+l$ blocks, after which everything proceeds normally - lagging by l.

Monero proof of work (PoW)

Monero uses a proof of work hash algorithm known as Cryptonight, designed to be relatively inefficient on GPU, FPGA, and ASIC architectures [56] compared to standard hash functions like SHA256. In April, 2018, (v7 of the protocol) it was slightly modified to counter the advent of Cryptonight ASICs [21].

7.3 Money supply

Obviously a digital currency needs a supply of money for users to transact with. There are two basic mechanisms for creating money in a blockchain-based cryptographic currency (a.k.a. cryptocurrency).

First, the currency's creators can simply conjure a set amount and distribute it to people in the genesis message. This is often called an 'airdrop'. Sometimes cryptocurrency creators give themselves a large amount of money in a so-called 'pre-mine'.

Second, the currency can be automatically distributed as reward for mining a block, much like mining for gold. There are two types here. In the Bitcoin model the total possible supply is capped. Block rewards slowly decline to zero, after which no more money is ever made. In the inflation model the supply continues to rise indefinitely.

Some cryptocurrencies employ both mechanisms for money creation. In fact, Monero is based on a currency known as Bytecoin that had a large pre-mine, followed by block rewards [11]. Monero had no pre-mine, and as we will see, its block rewards slowly decline to a small amount after which all new blocks reward that same amount, making Monero an inflationary currency.

7.3.1 Block reward

The block reward concept is straightforward. Block miners, before mining for a nonce, make a 'miner transaction' with no inputs and one output. The output amount is equal to the block reward, plus transaction fees from all transactions to be included in the block, and is communicated in clear text. Nodes who receive a mined block must verify the block reward is correct, and can calculate the current money supply by summing all past block rewards together.

Bit shifting

Bit shifting is used for calculating the base block reward (as we will see in Section 7.3.2, the block reward can sometimes be reduced below the base amount).

Suppose we have an integer A = 13, which has a bit representation [1101]. If we shift the bits of A down by 2, denoted A >> 2, we get [0011].01, which equals 3.25. In reality that last .01 outside the array, equal to 0.25, gets thrown away - 'shifted' into oblivion, leaving us with [0011] = 3.

This operation is equivalent to *floor*(13/4) = 3, or *floor*(A/2ˆ2), where *floor* rounds the number down - chopping off the 01 part of [0011].01.

Calculating base block reward for Monero

Let's call the current total money supply M, and the 'limit' of the money supply L = $2^{64} - 1$ (in binary it is [11....11], with 64 bits). In the beginning of Monero, the base block reward B = (L-M) >> 20, or in other words *floor*((L-M)/2ˆ20). If M = 0, then, in decimal format,

$$\mathrm{L} = 18,446,744,073,709,551,615$$
$$\mathrm{B}_0 = (L - 0) >> 20 = 17,592,186,044,415$$

These numbers are in 'atomic units' - 1 atomic unit of Moneroj can't be divided (there is no 0.5 atomic units). Clearly atomic units are ridiculous to look at - L is over 18 quintillion! We can divide everything by 10ˆ12 to move the decimal point over, giving us the standard units of Monero (a.k.a. XMR, Monero's so-called 'stock ticker').

$$\frac{\mathrm{L}}{10^{12}} = 18,446,744.073709551615$$
$$\mathrm{B}_0 = \frac{(L - 0) >> 20}{10^{12}} = 17.592186044415$$

And there it is, the very first block reward, dispersed to pseudonymous thankful_for_today (who was responsible for starting the Monero Project) in Monero's genesis block [58], was about 17.6 Moneroj! See Appendix D to confirm this for yourself.[5]

As more and more blocks are mined block rewards accumulate and M grows, continuously lowering future block rewards. Initially (since the genesis block in April, 2014) Monero blocks were mined once per minute, but in March, 2016, it became two minutes per block [12]. To keep the 'emission schedule', i.e. the rate of money creation,[6] the same, block rewards were doubled. This just means, after the change, we use (L-M) >> 19 instead of >> 20 for new blocks. Currently the base block reward is

$$\mathrm{B} = \frac{(L - M) >> 19}{10^{12}}$$

[5] Monero amounts are stored in atomic-unit format in the blockchain.

[6] See here for an interesting comparison of Monero and Bitcoin's emission schedules: [15].

7.3.2 Block size penalty

It would be nice to mine every new transaction into a block right away. However, what if someone submits a lot of transactions maliciously? The blockchain, storing every transaction, would quickly grow unpleasantly massive.

One mitigation is a fixed block size, so the number of transactions per block is limited. What if honest transaction volume rises? Each transaction author would bid for a spot in new blocks by offering fees to miners. Miners would mine transactions with the highest fees in order to earn maximum money. As transaction volume increases, fees would become prohibitively large for transactions of small amounts (such as Alice buying an apple from Bob). Only people willing to outbid everyone else would get their transactions into the blockchain.

Monero avoids those extremes (limited vs unlimited block size) with a dynamic block size. Miners can make blocks bigger than typical blocks from the recent past, but they have to pay a penalty in the form of reduced block reward. It's the price of making a bigger block.

To calculate a new block's size penalty, we sample the most recent 100 blocks in the blockchain and find the *median* block size, *median_100blocks*. We set the variable M100 to the larger value in the set {*median_100blocks*, 300kB}.[7] Only blocks larger than M100 pay a penalty, but the median can slowly rise, allowing progressively bigger blocks with no penalty. The maximum block size is 2*M100.

If the intended block size is greater than M100, then, given base block reward B, the block reward penalty is

$$\mathrm{P} = \mathrm{B} * ((\mathrm{block_size}/\mathrm{M100}) - 1)^2$$

The actual block reward is therefore

$$\mathrm{B}^{\mathrm{actual}} = \mathrm{B} - \mathrm{P}$$

$$\mathrm{B}^{\mathrm{actual}} = \mathrm{B} * (1 - ((\mathrm{block_size}/\mathrm{M100}) - 1)^2)$$

src/cryptonote_basic/cryptonote_basic_impl.cpp get_block_reward()

Using the ˆ2 operation means penalties are sub-proportional to block size. A block size 10% larger than M100 has just a 1% penalty, and so on. [15]

We can expect miners to create blocks larger than M100 when the fee from adding another transaction is bigger than the penalty incurred.

7.3.3 Dynamic minimum fee

To prevent malicious actors from flooding the blockchain with transactions, which could be used to pollute ring signatures, and generally bloat it unnecessarily, Monero requires a minimum fee per kB of transaction data. Originally this was simply 0.002 XMR/kB, then in January, 2017, (v4 of the protocol) a formula was added to mitigate some security risks from transaction fees higher

[7] In the beginning of Monero it was 20kB, then increased to 60kB in March, 2016, (v2 of the protocol) [12], and has been 300kB since April, 2017 (v5 of the protocol) [1].

than block rewards, and to deincentivize miners from pushing M100 above 300kB during transient high volumes of submitted transactions [9].

First we say the base dynamic fee is $f_b^{kB} = 0.0004/\text{kB}$,[8] then we compute B^{actual} and M100 from the previous sections.

The minimum fee per kB is[9]

$$f^{kB} = f_b^{kB} * (300\text{kB}/\text{M100}) * (\text{B}^{\text{actual}}/10)$$

src/cryptonote_core blockchain.cpp get_dynamic_per_kb_fee()

Note: we round transaction size up to the nearest kB.

7.3.4 Emission tail

Let's suppose a cryptocurrency with fixed supply and dynamic block size. After a while its block rewards fall to zero. With no more penalty on increasing block size, miners will simply add any transaction with a non-zero fee to their blocks.

Block sizes will stabilize around the average rate of transactions submitted to the network, and transaction authors will have no compelling reason to use transaction fees above the minimum, which would be zero according to Section 7.3.3.

This introduces an unstable, insecure situation. Miners will have little to no incentive to mine new blocks, leading to a fall in network hash rate as returns on investment decline. Block times will remain the same as difficulties adjust, but the cost of performing a double-spend attack may become feasible for malicious actors.

Monero mitigates this by not allowing the block reward to fall below 0.6 XMR (0.3 XMR per minute). This means when the following condition is met,

$$\begin{aligned} 0.6 &> ((L - M) >> 19)/10^{12} \\ \text{M} &> \text{L} - 0.6 * 2^{19} * 10^{12} \\ \text{M}/10^{12} &> \text{L}/10^{12} - 0.6 * 2^{19} \\ \text{M}/10^{12} &> 18,132,171.273709551615 \end{aligned}$$

then the Monero chain will enter a so-called 'emission tail', with constant 0.6 XMR (0.3 XMR/minute) block rewards forever after.[10]

7.4 Blockchain structure

The style of blockchain Monero uses is simple.

[8] The base fee was changed from 0.002 XMR/kB to 0.0004 XMR/kB in April, 2017 (v5 of the protocol) [1].

[9] To check if a given fee is correct, we allow a 2% buffer on f^{kB} in case of integer overflow. This means the effective minimum fee is $0.98 * f^{kB}$.

[10] The Monero emission tail's estimated arrival is May, 2022 [13].

It starts with a genesis message of some kind (in our case a miner transaction dispersing the first block reward), which constitutes the genesis block. The next block contains a reference to the previous block, in the form of block ID. A Block ID is simply a hash of the block's header (a list of information about a block), a so-called 'Merkle root' that attaches all the block's transaction IDs (which are hashes of each transaction), and the number of transactions (including a miner transaction containing the block reward and transaction fees).

To produce a new block, one must do proof of work hashes by changing a nonce value stored in the block header until the difficulty target condition is met. The proof of work hash is identical to the block ID, except uses a different, tailor-made proof of work hash function.

We use this section to fill in missing concepts: transaction ID, Merkle root, miner transaction, and block header and general block format.

7.4.1 Transaction ID

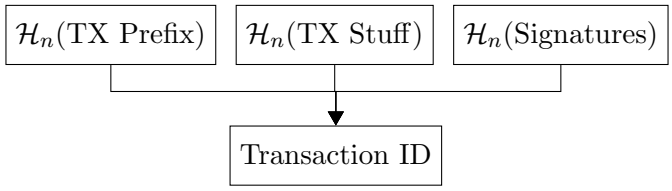
Transaction IDs are similar to the message signed by input MLSAG signatures (Section 5.6.3), but include the MLSAG signatures too.

The following information is hashed:

src/cryptonote_basic/cryptonote_format_utils.cpp calculate_transaction_hash()

- TX Prefix = {transaction era version (i.e. ringCT = 2), inputs {key offsets, key images}, outputs {one-time addresses}, extra {transaction public key, payment ID or encoded payment ID, misc.}}
- TX Stuff = {signature type (simple vs full), transaction fee, pseudo output commitments for inputs, ecdhInfo (masks and amounts), output commitments}
- Signatures = {MLSAGs, range proofs}

In this tree diagram, we use a black arrow to indicate a hash of inputs.



7.4.2 Miner transaction

Each block has a miner transaction that permits whoever mined a block to send himself the block reward and any transaction fees from transactions included in the block (summed into one

src/cryptonote_core/cryptonote_tx_utils.cpp construct_miner_tx()

output).[11] The output of a miner transaction is locked, unspendable, for 60 blocks after it is published [16].[12]

In place of an 'input', miner transactions record the block height of its block. This ensures the miner transaction's ID, which is simply a normal transaction ID except with $\mathcal{H}_n(\text{Signatures}) \rightarrow \mathcal{H}_n(0)$, is always unique (i.e. in case some malicious actor makes the miner transaction's amount and output one-time address the same for different blocks).

7.4.3 Merkle tree

Some users may want to discard unnecessary data from their copy of the blockchain. For example, once you verify some transaction's range proofs and input signatures, the only reason to keep that signature information is so users who obtain it from you can verify it for themselves.

To facilitate 'pruning' transaction data, and to more generally organize it within a block, we use a Merkle tree [42], which is just a binary hash tree of transaction IDs. The Merkle root results from all transactions, allowing us to easily include them in block hashes.[13] Any branch in a Merkle tree can be pruned if you keep its root hash.

src/crypto/tree-hash.c tree_hash()

An example Merkle tree based on four transactions and a miner transaction is diagrammed in Figure 7.1.[14]

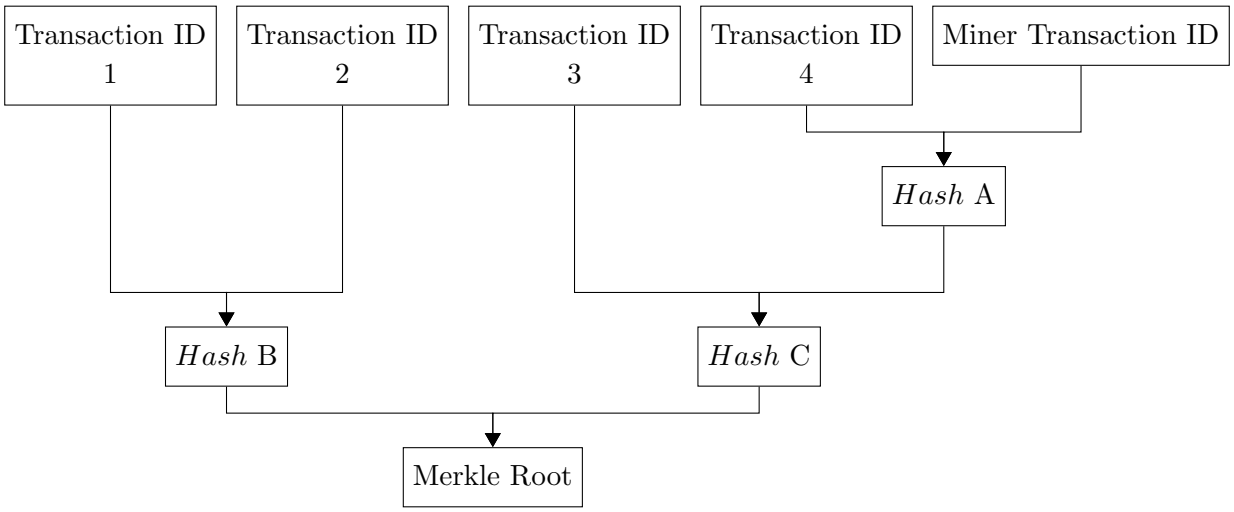


Figure 7.1: Merkle Tree

[11] The miner transaction output can theoretically be sent to a subaddress and/or use multisig and/or encoded (or not) payment ID. We don't know if any implementations have any of those features. Note that we need a transaction public key as usual.

[12] Any transaction's author can lock its outputs, rendering them unspendable until after a specified block height. He only has the option to lock all outputs to the same block height. It is only mandatory for miner transactions.

[13] We do not know of any Monero users who prune their blockchain, nor of any software able to do it. If pruning *were* implemented, it would probably involve deleting all signature data after verification, and keeping $\mathcal{H}_n(\text{Signatures})$ for computing transaction IDs. Signatures constitute most of a block's data, so this could allow a substantial reduction in blockchain size.

[14] A bug in the code for computing Merkle trees led to a serious attack on Monero on September 4, 2014 [33].

7.4.4 Blocks

A block is basically a block header and some transactions. Block headers record important information about each block. A block's transactions can be referenced with their Merkle root. We present here the outline of a block's content. Our readers can find a real block example in Appendix C.

- Block header:
  - Major version - Used to track hard forks (changes to protocol).
  - Minor version - Used for voting.
  - Timestamp - UTC (Coordinated Universal Time) time of block. Added by miners, timestamps are unverified but they won't be accepted if lower than the median timestamp of the last 60 blocks.
  - Previous block's ID - Referencing the previous block, this is the essential feature of a blockchain.
  - nonce - A 32 byte integer that miners change over and over until the PoW hash meets the difficulty target. Other people verifying a block can easily recalculate the PoW hash.
- Miner transaction - Disperses block reward and transaction fees to the block's miner.
- Transaction IDs - References to non-miner transactions (a.k.a. tx) added to the blockchain by this block. Tx IDs can (in combination with the miner tx ID), be used to calculate the Merkle root, and to find the actual transactions wherever they are stored.

src/cryptonote_core/blockchain.cpp check_block_timestamp()

src/cryptonote_basic/cryptonote_format_utils.cpp get_block_hashing_blob()

Block IDs are computed like this:[15]

$$\text{Block ID} = \mathcal{H}_n(\text{Block header}, \text{Merkle root}, \#\text{transactions} + 1)$$

calculate_block_hash()

And block mining is performed like this:
- While $PoW_{output} > target$, keep changing the nonce and recalculating

$$PoW_{output} = \mathcal{H}_{PoW}(\text{Block header}, \text{Merkle root}, \#\text{transactions} + 1)$$

get_block_longhash()

In addition to the data in each transaction (Section 5.8), we store the following information:

- Major and minor versions: variable integers $\leq$ 64 bits
- Timestamp: variable integer $\leq$ 64 bits
- Previous block's ID: 32 bytes
- nonce: 32 bytes
- Miner transaction: 32 bytes for one-time address, 32 bytes for transaction public key, and variable integers for unlock time, height, and amount.
- Transaction IDs: 32 bytes each

[15] We add 1 to the number of transactions to account for the miner tx.

Bibliography

[1] Add intervening v5 fork for increased min block size. https://github.com/monero-project/monero/pull/1869 [Online; accessed 05/24/2018].

[2] cryptography - what is a cryptographic oracle? https://security.stackexchange.com/questions/10617/what-is-a-cryptographic-oracle [Online; accessed 04/22/2018].

[3] Cryptography tutorial. https://www.tutorialspoint.com/cryptography/index.htm [Online; accessed 05/19/2018].

[4] Cryptonote address tests. https://xmr.llcoins.net/addresstests.html [Online; accessed 04/19/2018].

[5] Devmeeting 2018-05-06. https://monerobase.com/wiki/DevMeeting_2018-05-06 [Online; accessed 05/06/2018].

[6] Directed acyclic graph. https://en.wikipedia.org/wiki/Directed_acyclic_graph [Online; accessed 05/27/2018].

[7] How do payment ids work? https://monero.stackexchange.com/questions/1910/how-do-payment-ids-work [Online; accessed 04/21/2018].

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

[9] How does the dynamic fee calculation work? https://monero.stackexchange.com/questions/2531/how-does-the-dynamic-fee-calculation-work [Online; accessed 05/25/2018].

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

[11] Monero inception and history. https://monero.stackexchange.com/questions/475/monero-inception-and-history-how-did-monero-get-started-what-are-its-origins-a/476#476 [Online; accessed 05/23/2018].

[12] Monero v0.9.3 - hydrogen helix - released! https://www.reddit.com/r/Monero/comments/4bgw4z/monero_v093_hydrogen_helix_released_urgent_and/ [Online; accessed 05/24/2018].

[13] Tail emission. https://getmonero.org/resources/moneropedia/tail-emission.html [Online; accessed 05/24/2018].

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

[15] Useful for learning about monero coin emission. https://www.reddit.com/r/Monero/comments/512kwh/useful_for_learning_about_monero_coin_emission/d78tpgi/ [Online; accessed 05/25/2018].

[16] What is the block maturity value seen in many pool interfaces? https://monero.stackexchange.com/questions/2251/what-is-the-block-maturity-value-seen-in-many-pool-interfaces [Online; accessed 05/26/2018].

[17] Xor – from wolfram mathworld. http://mathworld.wolfram.com/XOR.html [Online; accessed 04/21/2018].

[18] Federal information processing standards publication (FIPS 197). Advanced Encryption Standard (AES), 2001.

[19] Nist releases sha-3 cryptographic hash standard, August 2015. https://www.nist.gov/news-events/news/2015/08/nist-releases-sha-3-cryptographic-hash-standard [Online; accessed 06/02/2018].

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

[21] Monero cryptonight variants, and add one for v7, April 2018. https://github.com/monero-project/monero/pull/3253 [Online; accessed 05/23/2018].

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

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

[24] 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.

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

[26] 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].

[27] 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.

[28] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, September 2006.

[29] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[30] Eiichiro Fujisaki and Koutarou Suzuki. *Traceable Ring Signature*, pages 181–200. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

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

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

[33] Surae Noether Jan Macheta, Sarang Noether and Javier Smooth. Counterfeiting via merkle tree exploits within virtual currencies employing the cryptonote protocol, mrl-0002, September 2014. https://lab.getmonero.org/pubs/MRL-0002.pdf [Online; accessed 05/27/2018].

[34] 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].

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

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

[37] 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.

[38] Ueli Maurer. Unifying zero-knowledge proofs of knowledge. In Bart Preneel, editor, *Progress in Cryptology – AFRICACRYPT 2009*, pages 272–286, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

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

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

[41] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple schnorr multi-signatures with applications to bitcoin. 2018.

[42] R. C. Merkle. Protocols for public key cryptosystems. In *1980 IEEE Symposium on Security and Privacy*, pages 122–122, April 1980.

[43] Victor S Miller. Use of elliptic curves in cryptography. In *Lecture Notes in Computer Sciences; 218 on Advances in cryptology—CRYPTO 85*, pages 417–426, Berlin, Heidelberg, 1986. Springer-Verlag.

[44] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf, 2008.

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

[46] 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].

[47] 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].

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

[49] 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].

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

[51] 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].

[52] 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].

[53] SJD AB S. Josefsson and N. Moeller. Eddsa and ed25519. Internet Research Task Force (IRTF), 2015. https://tools.ietf.org/html/draft-josefsson-eddsa-ed25519-03 [Online; accessed 05/11/2018].

[54] C. P. Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 239–252, New York, NY, 1990. Springer New York.

[55] Paola Scozzafava. Uniform distribution and sum modulo m of independent random variables. *Statistics & Probability Letters*, 18(4):313 – 314, 1993.

[56] Seigen, Max Jameson, Tuomo Nieminen, Neocortex, and Antonio M. Juarez. Cryptonight hash function. CryptoNote, March 2013. https://cryptonote.org/cns/cns008.txt [Online; accessed 04/04/2018].

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

[58] thankful_for_today. [ann][bmr] bitmonero - a new coin based on cryptonote technology - launched, April 2014. Monero's actual launch date was April 18th, 2014. https://bitcointalk.org/index.php?topic=563821.0 [Online; accessed 05/24/2018].

[59] 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. `<TransactionID>` is a hash of the transaction. 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 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, and was added to the blockchain at timestamp 2017-12-18 20:28:11 UTC (as reported by its block's miner).

```
monerod print_tx  b43a7ac21e1b60ad748ec905d6e03cf3165e5d8c9e1c61c263d328118c42eaa6
Found in blockchain at height 1467685
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"
      }],
```

```
43        "outPk": [ "cf141f5dfe04df14afad6b451d600aa5826a9be44a76a1630850c1d5951d482e",
44                   "e10bb69b66af5dabec765c7f5f7528926088877fa36746833828a0575896ae57"]
45     },
46     "rctsig_prunable": {
47        "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

- (line 2) - `print_tx` reports where it found the transaction.
- (line 3) - Raw transaction data with no readability formatting.
- `version` (line 5) - Transaction format/era version; '2' corresponds to RingCT.
- `unlock_time` (line 6) - Prevents a transactions outputs from being spent until the time has past. It is either a block height or a UNIX timestamp if the number is larger than the beginning of UNIX time.
- `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 799048th 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 the 1 input here.
- `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, and payment ID or encoded payment ID from Section 5.4
- `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': Obscured mask and amount for each of the outputs $t \in 1, ..., p$; here $p = 2$
- `mask` (line 37) - Field *mask* at $t = 1$ as described in Section 5.6.1
- `amount` (line 38) - Field *amount* at $t = 1$ as described in Section 5.6.1
- `outPk` (lines 43-44) - Commitments for each output, Section 5.6.2
- `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 all c and r), see Section 5.6.6
- `Ci` (line 49) - Commitments (ring keys) for the range proof on output $t = 1$, as described in Section 5.6.3. As explained in Section 5.6.6 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
$$\sigma(\mathfrak{m}) = (c_1, r_{1,1}, ..., r_{1,m+1}, ..., r_{v+1,1}, ..., r_{v+1,m+1})$$
- `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, and was added to the blockchain at timestamp 2017-12-21 11:36:20 UTC (as reported by its block's miner).

```
monerod print_tx 3ebf45fc5f8fd683037807384122817d5debfa762c7a7845cb7ccfe9ee20940b
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"
```

```
20        }
21      }, {
22        "key": {
23          "amount": 0,
24          "key_offsets": [ 1907097, 696508, 806254, 510195, 6709
25          ],
26          "k_image": "de14ec8958b311bd38a05aa3fb08fdd360001f1b9c060264eecdd8c08c9e83c4"
27        }
28      }, {
29        "key": {
30          "amount": 0,
31          "key_offsets": [ 1150236, 1943388, 788506, 37175, 7462
32          ],
33          "k_image": "e470f77dd5a4149210cb61ee107e73caea1ef9f61d05384e3bd4372fdc85bf17"
34        }
35      }
36    ],
37    "vout": [ {
38        "amount": 0,
39        "target": {
40          "key": "787cad1ebb181e1fc04b24d4d06c3d2882c38b262a7635de8ad487c536e40a12"
41        }
42      }, {
43        "amount": 0,
44        "target": {
45          "key": "faf4137928392b39ccf0a830c026157300995978769 7f9d4fb769c25781fb911"
46        }
47      }
48    ],
49    "extra": [ 1, 20, 56, 120, 111, 89, 89, 64, 10, 98, 96, 255, 202, 235, 203,
50               255, 2, 197, 176, 147, 61, 60, 41, 145, 207, 178, 212, 71, 37, 69, 19,
51               147, 205],
52    "rct_signatures": {
53      "type": 2,
54      "txnFee": 558805800000,
55      "pseudoOuts": [ "64dea29ac5560f93773240d58ca5768b879fd3c95e0b3b50a80ec36a6ff3a6da",
56                      "a60e7a00e65ff2a6299b92b166a629e9b0d62f6df50e40535140716757efe4c0",
57                      "4c67403adbc9dc0ca5a1a6abc846ab6d232dc3fa295099b3c7a9d005bac60eba",
58                      "635b26d78117d77899859ecb61e10125c3956a5c113b932f33c92c561acddaa3"],
59      "ecdhInfo": [ {
60          "mask": "ccffac42a86bec7b36ce9957cbdfe481d419bc5353335d0c236c347aea758d0c",
61          "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"],
```

```
104                 [ "9a95eeeecf3c3a48c43873a372c263357ff5f258a7bf8ed29a767237b0b0f202",
105                   "4f1ea4d9d4b56db780dd078a3e8219d0f54eaccc197901671002a206f063cb0e"],
106                 [ "2c800017cab2b8388f58fed0d61a46570f64cacd8fabc4e84ddee735b3135f0a",
107                   "458016f6fdf58b329fae0f929226ee2b8e410a14db8c6ede9b74fb718de71507"],
108                 [ "ea0fcaf793602dce25c8b2c4d17163f4298933b3fb09874307d8cde9a63c2c0c",
109                   "df76fbcd336c07f37e90e1a0d0db1ba49519ba4325062228bc9242af2c525703"],
110                 [ "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'^a_j, as described in Section 5.7.1. Please recall that the sum of these commitments will equal the sum of the 2 output commitments of this transaction (plus the transaction fee commitment fH).

APPENDIX C

Block content

In this section we show the structure of a sample block, namely the 1582196th block after the genesis block. The block has 5 transactions, and was added to the blockchain at timestamp 2018-05-27 21:56:01 UTC (as reported by the block's miner).

```
monerod print_block 1582196
timestamp: 1527458161
previous hash: 30bb9b475a08f2ea6fe07a1fd591ea209a7f633a400b2673b8835a975348b0eb
nonce: 2147489363
is orphan: 0
height: 1582196
depth: 2
hash: 50c8e5e51453c2ab85ef99d817e166540b40ef5fd2ed15ebc863091ca2a04594
difficulty: 51333809600
reward: 4634817937431
{
  "major_version": 7,
  "minor_version": 7,
  "timestamp": 1527458161,
  "prev_id": "30bb9b475a08f2ea6fe07a1fd591ea209a7f633a400b2673b8835a975348b0eb",
  "nonce": 2147489363,
  "miner_tx": {
    "version": 2,
    "unlock_time": 1582256,
```

```
20      "vin": [ {
21          "gen": {
22            "height": 1582196
23          }
24        }
25      ],
26      "vout": [ {
27          "amount": 4634817937431,
28          "target": {
29            "key": "39abd5f1c13dc6644d1c43db68691996bb3cd4a8619a37a227667cf2bf055401"
30          }
31        }
32      ],
33      "extra": [ 1, 89, 148, 148, 232, 110, 49, 77, 175, 158, 102, 45, 72, 201, 193,
34      18, 142, 202, 224, 47, 73, 31, 207, 236, 251, 94, 179, 190, 71, 72, 251, 110,
35      134, 2, 8, 1, 242, 62, 180, 82, 253, 252, 0
36      ],
37      "rct_signatures": {
38        "type": 0
39      }
40    },
41    "tx_hashes": [ "e9620db41b6b4e9ee675f7bfdeb9b9774b92aca0c53219247b8f8c7aecf773ae",
42                   "6d31593cd5680b849390f09d7ae70527653abb67d8e7fdca9e0154e5712591bf",
43                   "329e9c0caf6c32b0b7bf60d1c03655156bf33c0b09b6a39889c2ff9a24e94a54",
44                   "447c77a67adeb5dbf402183bc79201d6d6c2f65841ce95cf03621da5a6bffefc",
45                   "90a698b0db89bbb0704a4ffa4179dc149f8f8d01269a85f46ccd7f0007167ee4"
46    ]
47  }
```

Block components

- (lines 2-10) - Block information collected by software, not actually part of the block properly speaking.
- `is orphan` (line 5) - Signifies if this block was orphaned. Nodes usually store all branches during a fork situation, and discard unnecessary branches when a cumulative difficulty winner emerges, thereby orphaning the blocks.
- `depth` (line 7) - In a blockchain copy, the depth of any given block is how far back in the chain it is relative to the most recent block.
- `hash` (line 8) - This block's block ID.

- `difficulty` (line 9) - Difficulty isn't stored in a block, since users can compute *all* block difficulties from block timestamps and the rules in Section 7.2.
- `major_version` (line 12) - Corresponds to protocol version used to verify this block.
- `minor_version` (line 13) - Used by miners to signify which protocol version they like. This seems largely meaningless to us.
- `timestamp` (line 14) - An integer representation of this block's UTC timestamp, as reported by the block's miner.
- `prev_id` (line 15) - The previous block's block ID. Herein lies the essence of Monero's blockchain.
- `nonce` (line 16) - The nonce used by this block's miner to pass its difficulty target. Anyone can recompute the proof of work and verify the nonce is valid.
- `miner_tx` (lines 17-38) - This block's miner transaction.
- `version` (line 18) - Transaction format/era version; '2' corresponds to RingCT.
- `unlock_time` (line 19) - The miner transaction's output can't be spent until 60 more blocks have been mined.
- `vin` (lines 20-25) - Inputs to the miner tx. There are none, since the miner tx is used to generate block rewards and collect transaction fees.
- `gen` (line 21) - Short for 'generate'.
- `height` (line 22) - The block height this miner tx's block reward was generated for. Each block height can only generate a block reward once.
- `vout` (lines 26-32) - Outputs of the miner tx.
- `amount` (line 27) - Amount dispersed by the miner tx, containing block reward and fees from this block's transactions. Recorded in atomic units.
- `key` (line 29) - One-time (stealth) address assigning ownership of the miner tx's amount.
- `extra` (lines 33-36) - Extra information for the miner tx, including transaction public key.
- `tx_hashes` (line 37) - All transaction IDs included in this block (but not the miner tx ID).

APPENDIX D

Genesis block

In this section we show the structure of the Monero genesis block. The block has 0 transactions (it just sends the first block reward to thankful_for_today [58]). Monero's founder did not add a timestamp, perhaps as a relic of Bytecoin, the coin Monero's code was forked from, whose creators apparently tried to hide a large pre-mine [11].

Block 1 was added to the blockchain at timestamp 2014-04-18 10:49:53 UTC (as reported by the block's miner), so we can assume the genesis block was created the same day. This corresponds with the launch date announced by thankful_for_today [58].

```
monerod print_block 0
timestamp: 0
previous hash: 0000000000000000000000000000000000000000000000000000000000000000
nonce: 10000
is orphan: 0
height: 0
depth: 1580975
hash: 418015bb9ae982a1975da7d79277c2705727a56894ba0fb246adaabb1f4632e3
difficulty: 1
reward: 17592186044415
{
  "major_version": 1,
  "minor_version": 0,
  "timestamp": 0,
```

```
15    "prev_id": "0000000000000000000000000000000000000000000000000000000000000000",
16    "nonce": 10000,
17    "miner_tx": {
18      "version": 1,
19      "unlock_time": 60,
20      "vin": [ {
21          "gen": {
22            "height": 0
23          }
24        }
25      ],
26      "vout": [ {
27          "amount": 17592186044415,
28          "target": {
29            "key": "9b2e4c0281c0b02e7c53291a94d1d0cbff8883f8024f5142ee494ffbbd088071"
30          }
31        }
32      ],
33      "extra": [ 1, 119, 103, 170, 252, 222, 155, 224, 13, 207, 208, 152, 113, 94, 188,
34      247, 244, 16, 218, 235, 197, 130, 253, 166, 157, 36, 162, 142, 157, 11, 200, 144,
35      209
36      ],
37      "signatures": [ ]
38    },
39    "tx_hashes": [ ]
40  }
```

Genesis block components

Since we used the same software to print the genesis block and the block from Appendix C, the structure appears basically the same. We point out some unique differences.

- `difficulty` (line 9) - While the genesis block's difficulty is reported as 1, it isn't used for anything.
- `timestamp` (line 14) - The genesis block doesn't have a meaningful timestamp.
- `prev_id` (line 15) - We use 64 zeros for the previous ID, by convention.
- `nonce` (line 16) - $n = 10000$ by convention.
- `amount` (line 27) - This exactly corresponds to our calculation for the first block reward (17.592186044415 XMR) in Section 7.3.1.

- `key` (line 29) - The very first Moneroj was dispersed to Monero's founder thankful_for_today.
- `signatures` (line 36) - There are no signatures in the genesis block. This is here as an artifact of the `print_block` function. The same is true for `tx_hashes` in line 38.